ISSN 0190-3918
ISBN 0-8186-0479-4
IEEE Catalog Number 83CH1922-4
Library of Congress Number 79-640377
IEEE Computer Society Order No. 479

INTERNATIONAL
CONFERENCE ON
PARALLEL
PROCESSING

PROCEEDINGS

OF THE

1983 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 23–26, 1983

H. J. Siegel and Leah Siegel
Editors

Co-Sponsored by

Department of Computer and Information Science
OHIO STATE UNIVERSITY
Columbus, Ohio

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

COMPUTER
SOCIETY
PRESS

1983 INTERNATIONAL CONFERENCE
ON PARALLEL PROCESSING

H. J. Siegel and Leah Siegel

ISSN 0190-3918
ISBN 0-8186-0479-4
IEEE Catalog Number 83CH1922-4
Library of Congress Number 79-640377
IEEE Computer Society Order No. 479

COMPUTER
SOCIETY
PRESS

PROCEEDINGS

OF THE

1983 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 23–26, 1983

H. J. Siegel and Leah Siegel

Editors

Co-Sponsored by

Department of Computer and Information Science
OHIO STATE UNIVERSITY
Columbus, Ohio

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

ISSN 0190-3918
ISBN 0-8186-0479-4
IEEE Catalog Number 83CH1922-4
Library of Congress Number 79-640377
IEEE Computer Society Order No. 479

IEEE
COMPUTER
SOCIETY
PRESS

The papers appearing in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.

Published by IEEE Computer Society Press
1109 Spring Street
Suite 300
Silver Spring, MD 20910

ISSN 0190-3918
ISBN 0-8186-0479-4 (paper)
ISBN 0-8186-4479-6 (microfiche)
ISBN 0-8186-8479-8 (casebound)
Library of Congress 79-640377
IEEE Catalog No. 83CH1992-4
IEEE Computer Society Order No. 479

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

The Institute of Electrical and Electronics Engineers, Inc.

PREFACE

This volume is the Proceedings of the 1983 International Conference on Parallel Processing, the twelfth in a series of annual meetings. This year's conference represents the largest yet, both in number of papers submitted and number of papers presented at the conference. The previous records were 136 papers submitted (1981) and 67 papers presented (1982). This year 240 papers were submitted! The final program contains 97 papers from twelve countries. Academia, industry, and government research labs are all represented.

Because of the large number of excellent papers submitted, the task of arriving at a program was an extremely difficult one. For the first time, the Parallel Processing Conference will have parallel sessions in order to accommodate more papers. Even at that, there were many very good papers which it was not possible to include, and there were many papers submitted as regular papers which were accepted as short papers. On the positive side, because of the large number of submissions, all of the papers finally accepted and included in this proceedings are of the highest quality. We sincerely thank all of the authors who submitted papers for their interest in the conference.

Special thanks go to the 379 referees who read and evaluated the manuscripts. Each submitted paper was sent to three referees. Without the efforts of these reviewers, the task of arriving at a program would have been virtually impossible. The names of the referees are listed in these proceedings.

We wish to thank Dr. Ben Coates, Head of the Electrical Engineering School at Purdue, for his support and encouragement. We thank Dee Dee Dexter, Carol Edmundson, and Jenny Hiatt for their excellent job in managing the massive amounts of paperwork involved in handling 240 submissions, 720 reviews, and 97 accepted papers. We also want to thank Wanda Booth, Andy Hughes, Sharon Katz, Mickey Krebs, Nancy Lein, Pat Loomis, Vicky Spence, and Linda Stovall for their assistance. Ken Batcher provided valuable information about the running of the 1982 conference. We thank Ming T. Liu and Chuan-Lin Wu for handling the reviewing of the papers submitted from Purdue.

Finally, we wish to acknowledge the efforts of Tse-Yun Feng. As always, he has been a one-man conference organizing committee, handling publicity, local arrangements, budget, and all other facets of the conference. We thank him for giving us the opportunity to chair this year's technical program.

H. J. Siegel
Leah Siegel
Program Co-Chairs

Purdue University
June 1983

LIST OF REFEREES

Ikram Abdou
M.A. Abidi
Amir Abouelnaga
Shrikant Acharya
W. Ackerman
George B. Adams III
Loyce Adams
Subhash Agarwal
Tilak Agerwala
Dharma Agrawal
Mohan Ahuga
S.J. Allan
Dave Anderson
John B. Anderson
Richard Antony
Bruce Arden
Clifford Arnold
V. Ashok
J.W. Atwood
T.S. Axelrod
Takanobu Baba
Hussein Badr
Jean-Loup Baer
T.R. Bashkow
Kenneth Batcher
Paul Bay
Stephan Bechtolsheim
Jon Bentley
Bruce Berra
Bharat Bhargava
L.N. Bhuyan
Gianfranco Bilardi
John Board, Jr.
Michael Bodner
Andre Bondi
P. Bounds
Faye Briggs
Edward C. Bronson
Mark E. Brown
Jim C. Browne
John Bruner
John Burger
John Burkley
F.J. Burkowski
Steven Butner
Bill Buzbee
D.A. Calahan
James Calhoun
Peter Cappello
Avinash Chandak
Chiou-Min Chang
Su-shing Chen
Ming-Yang Chern
Chung-Yang Chiang
Yetung Chiang
Francis Chin
Y.C. Chow
Carolyn Cline
Fred Cohen
Leonard Cohn
Hank Cook
Ed Coyle
Karel Culik
David Culler
Janice Cuny
Ron Cytron

Scott Danforth
Edward Davidson
Al Davis
Carl Davis
Nat Davis
Doug DeGroot
E. Dekel
Narsingh Deo
B.C. Desai
Sanjay Deshpande
Dave DeWitt
S. Dhar
Daniel Dias
Nikitas Dimopoulos
Karl Doty
Robert Douglass
Edwin Drogin
David Du
P.F. Dubois
Ahmed Elmagarmid
P.G. Eltgroth
P. Enslow
M.D. Ercegovac
Lars Ericson
Adly Fam
Art Feather
Raphael Finkel
Allen Firstenberg
Paul Fisher
Eugene Fiume
Raymond Ford
T.J. Forquer
J.A.B. Fortes
Alain Fournier
M. Franklin
King-Sun Fu
Henry Fuchs
George Fucik
D.S. Fussell
Daniel Gajski
John Gallant
E.J. Gallopoulos
Dennis Gannon
Oscar Garcia
J.L. Gaudiot
Pieter Geerkens
S.A. Ghozati
Gary Gladden
S.M. Goldwasser
Allan Gottlieb
James Graham
Marshall D. Graham
Peter Gregono
William Greiman
Nancy Griffeth
Gao Guang-Rong
John Gustafson
John Guttag
N.R. Hall
Susanne Hambrusch
Sang Han
Robert Haralick
C.D. Harber
Paula Hawthorn
Kye Hedlund
Don Heller

Frank P. Hiner, III
Daniel Hirshberg
Lawrence Ho
Yang-Chang Hong
G.S. Hope
William Hopkins
Edward Horvath
C.E. Houstis
Ching Hsiao
YuHen Hu
Paul Hudak
Michael Huhns
Kai Hwang
Yul Inn
Keki Irani
Mary Jane Irwin
Robert Israel
M.R. Ito
Bijan Jabbari
Bharadwaj Jayaraman
David Jefferson
Roy Jenevein
Lennert Johnsson
Harry Jordan
J. Robert Jump
Thaddeus Kadela
Avinash C. Kak
Gerald Kane
Alejandro Kapauan
Svetlana Kartashev
Krishna Kavipurapu
John Kearns
Robert Keller
T.M. Kinter
Gloria Kissin
Dave Klappholz
Hideaki Kobayashi
Aaron Konstam
Israel Koren
Suraj Kothari
J.S. Kowalik
Clyde Kruskal
Annette Krygiel
James T. Kuehn
J.G. Kuhl
Robert Kuhn
William Kuhn
Ashok Kulkarn
Manoj Kumar
S.P. Kumar
H.T. Kung
S.Y. Kung
Ten-Hwang Lai
S. Lakshmivarahan
C. Lam
Duncan Lawrie
Daryl Lawton
C.W. Lee
Francis Lee
Hykyu Lee
Kyungsook Lee
Manjai Lee
Wong-Hua Lee
William Leler
Bruce Lester
Steven Levitan

Hungwen Li
Pey-yun Peggy Li
Tao Li
Richard Lian
Karl Lieberherr
Gie-Ming Lin
Huai-An Lin
Woei Lin
Gary Lindstrom
G.J. Lipovski
Yury Litvin
Ming T. Liu
David Loendorf
Hubert Love, Jr.
Richard Lyon
Bill MacDonald
Gyula Mago
Srinivas Makam
Miroslaw Malek
O.P. Malik
Creve Maples
Peter Marinos
Gerald Masson
A.D. McAulay
Charlie McDowell
S.D. McEwan
Jim McGraw
Robert J. McMillen
P. Mehrotra
Joseph Mercurio
Dave Meyer
David Middleton
Russ Miller
O. Robert Mitchell
Joseph Mohan
Dan I. Moldovan
Robert Montoye
Karam Mossaad
Fred Mowle
Trevor Mudge
Phil Mueller
Tadao Murata
Barbara Naused
Victor Nelson
Lionel Ni
Wesley Nurden
John O'Donnell
R.R. Oldehoeft
Ibrahim Onyuksel
J. Opartny
Eli Opper
Yavuz Oruc
Krishnan Padmanabhan
Christos Papachristou
Stott Parker
Janak Patel
Dave Paterson
Girish Pathak
A.R. Pleszkun
Jerry L. Potter
Dhiraj Pradhan
Terrence Pratt
S. Preiser
Kendall Preston, Jr.
Noah S. Prywes
Krish Purswani
G.Z. Qadah
Donna Quammen
Michael Quinn

C.S. Raghavendra
T.A. Rahman
Bharat Rathi
S. Reddy
Daniel Reed
Anthony Reeves
R.A. Reynolds
John Rice
Tom Rice
Garry Rodrigue
Gruia-Catalin Roman
Azriel Rosenfeld
J. Rootenberg
Jerry Rothstein
Larry Rudolph
Sartaj Sahni
Ahmed Sameh
Nicola Santoro
Subhash Sarin
John Savage
Prashant Sawkar
Michael Schlansker
J.G. Schwartz
Herb Schwetman
Robert Seban
Zary Segall
Charles Seitz
Matthew Sejnowski
Sowrirajan Seshadri
David Shaw
John Shen
Heonshik Shin
Kang Shin
Howard Sholl
Dan Siewiorek
A. Silberschatz
Harvey Silverman
Bart Sinclair
Dan Slotnick
Bradley Smith
Bruce Smith
Burton Smith
D.R. Smith
Gerald Smith
Kirk Smith
S. Diane Smith
Lawrence Snyder
Mary Soffa
Vason Srini
John A. Stankovic
Kenneth Steiglitz
Stephen Stepoway
Stanley Sternberg
L.J. Stockmeyer
Sal Stolfo
Albert Stone
Harold Stone
Quentin Stout
Philip Swain
Earl Swartzlander
Tsung-Wei Sze
Jiro Tanaka
Steve Tanimoto
Fred Taylor
Suchai Thanawastien
Alexander Thomasian
C.D. Thompson
Kenneth Thurber
Ioannis Tollis

Kishor Trivedi
Roger Tsai
F.M. Tse
Yung Tsin
David Tuomenoksa
Leonard Uhr
Jeff Ullman
L. David Umbaugh
W.K. Van Nurden
John Van Rosendale
Andre van Tilborg
Peter Varman
Alex Veidenbaum
James Vellenga
Charles Vick
Newman Vosbury
Robert Voigt
Robert Wagner
Benjamin Wah
Abraham Waksman
Don Walker
Pong-Sheng Wang
D. Wann
Robert Wedig
Charles Weems
Rich Weiss
David Wells
Charles Wetherell
Andrew Whinston
Jack Wileden
Elizabeth Williams
Tom Williams
David Wilson
Larry Wittie
F.S. Wong
Nam Woo
Chuan-Lin Wu
Steve Wu
Yee-Hong Yang
Phil Yeh
W.C. Yen
Mark Yoder
Matthew Yuschik

Acknowledgment of Prior Work

Part of a chapter in our article "Parallel Simulation by Means of a Prescheduled MIMD-System Featuring Synchronous Pipeline Processors", published in the Proceedings of the 1982 International Conference on Parallel Processing under the subheading "Processor Scheduling Strategy" is quite similar to the work of D'Hollander. While we acknowledged the original work, due to an oversight D'Hollander's paper was not mentioned and we profoundly regret this omission. The references in our article should therefore be augmented by: E.H. D'Hollander "Speedup Bounds for Continuous System Simulation on a Homogeneous Multiprocessor", Int. Conf. on Par. Proc. 1981, pp 176 - 182.

M. Tadjan, R. Bührer, W. Hälg
Swiss Federal Institute of Technology

Author Index

Table of Contents

SESSION 1: PANEL DISCUSSION: PERFORMANCE OF EXISTING SUPERCOMPUTERS ON COMPUTATIONALLY INTENSIVE TASKS

SESSION 2A: MULTISTAGE NETWORK PERFORMANCE

SESSION 2B: NUMERICAL ALGORITHMS I

SESSION 3A: MULTISTAGE NETWORKS

SESSION 3B: NUMERICAL ALGORITHMS II

SESSION 4A: NETWORK CONNECTION CAPABILITIES

SESSION 4B: SPECIAL PURPOSE SYSTEMS

SESSION 5A: NODE-TO-NODE NETWORKS

SESSION 5B: NON-NUMERICAL ALGORITHMS I

SESSION 6A: TREE STRUCTURED SYSTEMS

SESSION 6B: NON-NUMERICAL ALGORITHMS II

SESSION 7A: PARALLEL PROGRAMMING AND LANGUAGES

SESSION 7B: IMAGES AND SPEECH

SESSION 8A: EXPRESSING PARALLELISM

SESSION 8B: DATABASE MACHINES/SIGNAL PROCESSING

SESSION 9A: DATA FLOW

SESSION 9B: SIMULATION/OPERATING SYSTEMS

SESSION 10A: MODELS

SESSION 10B: SCHEDULING RESOURCES

SESSION 11A: SYSTEM PERFORMANCE

SESSION 11B: VLSI PROCESSOR ARRAYS

SESSION 12A: COMPUTER ARCHITECTURES

SESSION 12B: ASSOCIATIVE PROCESSING/DISTRIBUTED SYSTEMS

SESSION 13A: MULTIPROCESSOR SYSTEMS

SESSION 13B: PIPELINING

SESSION 1 - PANEL DISCUSSION

Performance of Existing Supercomputers on Computationally Intensive Tasks

Sidney Fernbach, Chairman

Cray-1	Michael Ess
Cyber 205	Kevin Moriarty
HEP-1	Burton Smith

AN INTERFERENCE ANALYSIS OF INTERCONNECTION NETWORKS

Laxmi N. Bhuyan and C.W. Lee

Department of Electrical Engineering
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

ABSTRACT

An interference analysis of the Interconnection Networks (INs) for a tightly coupled multiprocessor is presented in this paper. The interconnections considered are crossbars and delta networks. Two situations are examined: when a memory module is equally likely to be addressed by a processor and when a processor has a favorite memory. It is shown that for a higher rate of favorite requests, the delta networks perform close to a crossbar.

INTRODUCTION

A multiprocessor architecture can be broadly divided into two categories: loosely coupled and tightly coupled. In a loosely coupled multiprocessor, each processor has a local memory and the communication between the Processing Elements (PEs) is accomplished through an Interconnection Network (IN). A PE essentially consists of a processor and its local memory. In a tightly coupled system, the processors are connected to one side of the IN and the memory modules are connected to the other side. The IN is capable of connecting a processor to any one of the memory modules. The loosely coupled and tightly coupled architectures are illustrated in Fig. 1. In this paper, we consider an interference analysis of the INs for a tightly coupled multiprocessor.

A crossbar interconnection [1] allows all possible one-to-one and simultaneous connections between the processors and the memory modules. When two or more processors try to access the same memory, only one of them will be connected and the rest will be blocked or rejected. Band width (BW) is defined as the expected number of memory requests accepted per cycle or the average number of memory modules remaining busy in a cycle. Clearly, this is a parameter which specifies as to what extent an IN is efficient. The interference analysis of an MxN crossbar for M processors and N memory modules, when a processor is equally likely to address any one of the N common memories, is well known [2-4]. However, in a practical situation, a processor is likely to address a particular memory most of the time except when an interprocessor communication is necessary. If processor i (P_i) communicates more often with a memory module i (MM_i), we will call MM_i as a favorite memory of P_i and P_i as a favorite processor of MM_i. We will assume that we have a prior knowledge of a factor m which is the probability that P_i addresses MM_i provided that P_i generates a request. When $m = \frac{1}{N}$, a processor is equally likely to address any one of the N memory modules and the favorite case reduces to an equally likely case. In this paper, we carry out an analysis for such a favorite memory case for an MxN crossbar switch when $M = N$, $M \geqslant N$ and $M \leqslant N$.

Because of the $O(N^2)$ switch complexity of an NxN crossbar, Multistage Interconnection Networks (MINs) have been proposed recently for large values of N. Several MINs such as Omega [5], Indirect binary n-cube [6], Generalized cube [7] and Baseline [8] are known. An NxN MIN basically employs $\log_2 N$ stages of 2x2 switches with N/2 number of switches per stage. It is capable of performing a subset of one-to-one and simultaneous mappings while reducing the cost to $O(N \log_2 N)$. The mappings or permutations, achieved by one network, may be different than another depending on the interconnection used between the stages. However, these MINs are all functionally equivalent in terms of their BWs and the total number of permutations, achieved. Interference analysis of such MINs have been reported in a few papers [4,9-11] for equally likely cases. The VLSI performance of these networks have also been studied [12,13] when the whole network is fabricated on a single chip. In terms of area*delay characteristics the MINs do not perform that well compared to the crossbars, as they do in an SSI implementation.

Delta network [4] is a self routing interconnection network that connects $M = a^n$ inputs to $N = b^n$ outputs through n stages of a x b crossbar switches. All the MINs form a class of Delta networks with $a = b = 2$. A still braoder class of networks called Radix Shuffle Networks (RSNs) was introduced recently [11] for connecting M processors to N memory modules for arbitrary values of M and N. If M and N can be factored into 'r' components as $M = m_1 \times m_2 \times \ldots \times m_r$ and $N = n_1 \times n_2 \times \ldots \times n_r$, an RSN consists of 'r' stages of switches with the ith stage employing $m_i \times n_i$ crossbar modules. Delta is a special case of the RSN when all m_i's are equal to a and all n_i's are equal to b. All the above cited networks form a part of the Banyan networks [14], introduced for partitioning multiprocessor systems. Interference analysis of the RSNs was also reported [11] when a processor is equally likely to address a memory module. We carry out here an analysis for the RSNs for the favorite memory case. The results derived for a crossbar are successfully applied to the RSNs. Because of the complexity involved, we restrict our analysis to NxN Delta networks only. The theoretical results match with those obtained from simulations.

0190-3918/83/0000/0002$01.00 © 1983 IEEE

ANALYSIS OF CROSSBAR

A crossbar is capable of connecting M processors to N memory modules for any arbitrary values of M and N [1]. The analyses given here are based on the following assumptions.

1. The crossbar operates in a synchronous mode i.e. the requests issued by the processors begin and end simultaneously.
2. The requests are random and the request generated by a processor is independent of the request generated by another processor.
3. Requests which are not accepted are blocked or rejeted.
4. The requests generated in a cycle are independent of the requests generated in the previous cycle.
5. p_o is the probability with which a processor generates a request. Thus p_o is the rate of request of a processor per cycle.
6. m is the probability with which processor P_i addresses memory MM_i given that P_i generates a request. Thus $m \cdot p_o$ is the rate of request of a processor directed to its favorite memory.

In an MIMD [15] operation, the memory requests are asynchronous. Various simulations [3,4] indicate that assumption 1 does not bring in a substantial difference in the results. When asynchronous operation is assumed, buffers should be provided in the switches [9]. Assumption 4 is unrealistic because the requests rejected in a cycle will indeed be resubmitted in the next cycle. This assumption leads to amazingly simpled closed form equations for a crossbar [3] and produces negligible discrepancies in the result [2]. Assumption 5 indicates that a processor need not send a request in every cycle. Assumption 6 considers memory module MM_i as a favorite memory of the processor P_i. In an MxN crossbar for an equally likely case, a processor addresses a memory with a probability of $\frac{1}{N}$. MM_i is considered a favorite memory of P_i only if $m > \frac{1}{N}$. The values of p_o and m are program dependent and can be determined. With a probability p_o of a processor generating a request, the probability $q_{(j)}$ that j requests are generated by M processors is given by:

$$q_{(j)} = \binom{M}{j} p_o^j \cdot (1 - p_o)^{M-j}$$

where $\binom{M}{j}$ is the binomial coefficient.

(a) Equally likely case for MxN crossbar

This is a situation where a processor is equally likely to address any one of the N memory modules. The probability that a processor addresses a particular memory is $\frac{1}{N}$, given that the processor generates a request. Probability that a memory module is addressed by k processors, given that j requests are generated by the processors;

$$p_{e(k,j)} = \binom{j}{k}\left(\frac{1}{N}\right)^k\left(1 - \frac{1}{N}\right)^{j-k} .$$

Subscript 'e' stands for equally likely case.

For various values of k ranging from 1 to j, the rate of request at a memory module;

$$\begin{aligned} p_{e(j)} &= \sum_{1 \leqslant k \leqslant j} \binom{j}{k}\left(\frac{1}{N}\right)^k\left(1 - \frac{1}{N}\right)^{j-k} \\ &= \sum_{0 \leqslant k \leqslant j} \binom{j}{k}\left(\frac{1}{N}\right)^k\left(1 - \frac{1}{N}\right)^{j-k} - \left(1 - \frac{1}{N}\right)^j \\ &= 1 - \left(1 - \frac{1}{N}\right)^j . \end{aligned}$$

With each processor having a probability p_o of generating a request, the total rate of request at a memory module is given by;

$$\begin{aligned} p_e &= \sum_{0 \leqslant j \leqslant M} q_{(j)} \cdot p_{e(j)} \\ &= \sum_{0 \leqslant j \leqslant M} \binom{M}{j} p_o^j (1 - p_o)^{M-j} \left\{1-\left(1 - \frac{1}{N}\right)^j\right\} \\ &= 1 - \left(1 - \frac{p_o}{N}\right)^M . \qquad (1) \end{aligned}$$

BW is the average number of memory modules remaining busy in a cycle.
In other words BW = rate of request at a memory module * the number of memory modules.
Hence, for equally likely case,

$$BW_e = N\left\{1-\left(1 - \frac{p_o}{N}\right)^M\right\} . \qquad (2)$$

(b) Favorite memory case for NxN crossbar

Let m be the prbability that processor P_i requests memory module MM_i given that P_i generates a request. Hence, the probability of P_i requesting MM_i, $P_i \rightarrow MM_i = p_o \cdot m$. Probability that P_i does not request MM_i; $P_i \not\rightarrow M_i = 1 - p_o \cdot m$.

Given another processor P_j for $i \neq j$, which generates a request; $P_j \rightarrow MM_i = (1-m)\frac{1}{N-1}$ = x say and $P_j \not\rightarrow MM_i = 1 - x$.

In a situation when there are a total of j requests of which k requests arrive at MM_i, two distinct possibilities can occur;

$P_i \rightarrow MM_i$ and (k-1) other processors $\rightarrow MM_i$

or $P_i \not\rightarrow MM_i$ and k other processors $\rightarrow MM_i$.

The rate of request at MM_i given j requests at the input side;

$$\begin{aligned} p_{f(j)} &= \sum_{1 \leqslant k \leqslant j} \Big\{p_o m \cdot \binom{j-1}{k-1} \cdot x^{k-1}(1-x)^{j-k} \\ &\qquad +(1-p_o m)\binom{j-1}{k}x^k(1-x)^{j-k-1}\Big\} \\ &= 1 - (1-p_o m)(1-x)^{j-1} . \qquad (3) \end{aligned}$$

The subscript 'f' stands for the favorite memory case.

With a probability p_o of P_j generating a request and with (N-1) other processors besides P_i , the total rate of request at MM_i;

$$p_f = \sum_{0\leqslant j-1\leqslant N-1} \binom{N-1}{j-1} \cdot p_o^{j-1} \cdot (1-p_o)^{N-j} \cdot p_{f(j)}$$

$$= 1 - (1-p_o m)(1-p_o x)^{N-1} .$$

With $x = \frac{1-m}{N-1}$; $p_f = 1 - (1-p_o m)(1-p_o \frac{1-m}{N-1})^{N-1}$ (4)

and $BW_f = N\{1 - (1 - p_o \cdot m)(1-p_o \cdot \frac{1-m}{N-1})^{N-1}\}$. (5)

$\lim_{m\to 1} BW_f = p_o N$, which means that if all the requests were favorite, the BW is equal to the number of requests generated; so all the requests are accepted. Equally likely is a special case of the favorite memory case with $m = \frac{1}{N}$.

The BW of an NxN crossbar are plotted in Fig. 2 both for favorite memory case with m = 0.8 and an equally likely case. With a favorite memory, a processor remains busy with its favorite memory module most of the time. As a result, less conflicts occur which in turn give rise to a higher BW. A favorite case for NxN crossbars can also be visualized as shown in Fig. 3a. Rate of request at MM_i due to P_i ; $p_A = p_o m$. Rate of request at MM_i due to (N-1) other processors;

$p_B = 1 - \{1 - \frac{p_o(1-m)}{N-1}\}^{N-1}$, similar to eqn. (1) for (N-1) processors and (N-1) non-favorite memories. Because of the assumption 2, these two rates are statistically independent of each other. Hence, the total rate of request at MM_i ;

$$p_{AB} = p_A + p_B - p_A \cdot p_B$$

$$= p_o m + 1 - (1- p_o \frac{1-m}{N-1})^{N-1}$$

$$- p_o m + p_o m(1-p_o \frac{1-m}{N-1})^{N-1}$$

$$= 1-(1-p_o m)(1-p_o \frac{1-m}{N-1})^{N-1} = p_f \text{ in eqn. (4).}$$

(c) <u>Favorite memory case for MxN crossbars with M>N</u>

The situation is depicted in Fig. 3b. The processors are divided into two groups. Group A consists of N processors having favorite memories and group B consists of M-N processors that are equally likely to address any one of the memory modules with a probability of $\frac{1}{N}$. The rate of request at MM_i due to the processors belonging to group A ;

$p_A = 1-(1-p_o m)(1-p_o \frac{1-m}{N-1})^{N-1}$; same as in eqn. (4).

The rate of request at MM_i due to the processors belonging to group B;

$p_B = 1 - (1- \frac{p_o}{N})^{M-N}$; from eqn. (1) .

Since the request rates are statistically independent, the overall request rate at MM_i ;

$$p_f = p_A + p_B - p_A \cdot p_B$$

$$= 1-(1- p_o m)(1 - p_o \frac{1-m}{N-1})^{N-1}(1 - \frac{p_o}{N})^{M-N} . \quad (6)$$

$$BW_f = N\{1-(1-p_o m)(1-p_o \frac{1-m}{N-1})^{N-1}(1 - \frac{p_o}{N})^{M-N}\}. \quad (7)$$

When M=N , eqn. (7) reduces to eqn. (5).

When $m = \frac{1}{N}$, $BW = N\{1 - (1 - \frac{p_o}{N})^M\}$ which is same as the equally likely case.

(d) <u>Favorite memory case for MxN crossbars with N>M</u>

The situation is depicted in Fig. 3c. The memory modules are divided into two groups A and B . Group A consists of M favorite memories and group B consists of (N-M) memories that are equally addressed by a processor with a probability $x = (1-m) \cdot \frac{1}{N-1}$, given that the processor generates a request.

Given that there are j requests generated by the processors including processor P_i , the rate of request at MM_i belonging to group A;

$p_{A(j)} = 1 - (1 - p_o m)(1-x)^{j-1}$; from eqn. (3).

With the proessors having a probability of request p_o , the total rate of request at MM_i belonging to group A;

$$p_A = \sum_{0\leqslant j-1\leqslant M-1} \binom{M-1}{j-1} p_o^{j-1} (1 - p_o)^{M-j} p_{A(j)}$$

$$= 1 - (1 - p_o m)(1 - p_o x)^{M-1} .$$

With $x = \frac{1-m}{N-1}$; $p_A = 1-(1 - p_o m)(1 - p_o \frac{1-m}{N-1})^{M-1}$.

A processor addresses a memory module belonging to group B with a probability of $\frac{1-m}{N-1}$. Probability of generation of a request being p_o , the rate of request at MM_j belonging to group B;

$$p_B = 1 - (1 - p_o \frac{1-m}{N-1})^M \text{ ; from eqn. (1)}$$

Then

$$BW_f = p_A \cdot M + p_B \cdot (N-M)$$

$$= M\{1-(1-p_o M)(1-p_o \frac{1-m}{N-1})^{M-1}\}$$

$$+(N-M)\{1-(1-p_o \frac{1-m}{N-1})^M\}$$

$$= N-M(1-p_o M)(1-p_o \frac{1-m}{N-1})^{M-1}-(N-M)(1-p_o \frac{1-m}{N-1})^M . \quad (8)$$

Again with $m = \frac{1}{N}$, $BW_f = N-N(1 - \frac{p_o}{N})^M = BW_e$. BW_f obtained with M = 16 are plotted in Fig. 4

for various values of N. Compared to the BW for equally likely case (BW_e), there is a fast increase in BW_f with increase in N for $N \leqslant 16$. The rate of increase in BW_f in Fig. 5 for N fixed at 16, is also similar to that obtained in Fig. 4. The difference between BW_f and BW_e is maximum when M is equal to N. This is reasonable because the maximum possible BW is limited to $\mathrm{Min}\{M,N\}$ irrespective of whether a favorite case or not.

ANALYSIS OF DELTA NETWORKS

A delta network [4] is a multistage interconnection network that connects $M = a^n$ inputs to $N = b^n$ outputs through n stages of a x b crossbar modules. The ith stage of the delta network consists of $M \frac{b^{i-1}}{a^i}$ number of a x b crossbar switches and produces $M(\frac{b}{a})^i$ outputs. An interference analysis of these networks for equally likely case is presented in [4]. The analysis is based on a recursive computation of the rate of request at a stage. The rate of request on an output line of the ith stage is:

$$p_i = 1 - (1 - \frac{p_{i-1}}{b})^a \qquad (9)$$

where p_o is the probability of generation of a request by a processor. With a given value of p_o, the final rate of request at an output line of the delta network can be computed using the above recursive equation. Then $BW = N \cdot p_n$. We develop here such a recursive analysis for delta networks as applicable to the favorite memory case. Because of the complexity involved, we restrict our analysis to NxN delta networks only. An NxN delta network with $N=a^n$, consists of n stages of axa crossbar modules with $\frac{N}{a}$ such modules per stage. The interconnection between the stages is an a-shuffle of the inputs. S_a, the a-shuffle of an integer j is given by;

$$\begin{aligned} S_a &= a\,j \mod (N-1) \quad \text{for } 0 \leqslant j < N-1 \\ &= j \qquad\qquad\qquad\quad \text{for } j = N-1 \,. \end{aligned} \qquad (10)$$

Omega network [5] is a special case of delta network with a=2. We define that a processor is connected to its favorite module when all the switches are in straight connection as shown in Fig. 6 for an 8x8 omega network. When all the switches are connected straight in a delta network with a-shuffle interconnection before each stage, an identity permutation results. Hence, MM_i is a favorite memory of P_i for $0 \leqslant i \leqslant N-1$. The analyses presented below also hold for delta networks which do not employ an a-shuffle interconnection before each stage. In such networks, the straight connections of the switches may not result in an identity mapping and hence, the favorite memories will be different without any change in the actual performance. In addition to the assumptions spelled out for the crossbar analysis, we make an important additional assumption for delta networks. Whenever a number of requests reach an output line of a switch, a request is randomly accepted with an equal probability. We will call this as an Equal Acceptance (EA) rule.

Consider two switches A and B from two adjacent stages of the delta network as shown in Fig. 7. There can be only one connection from an output of switch A to the input of switch B. The other ouput lines of switch A will be connected to (a-1) other switches of the (i+1)th stage. The number of output lines being same at each stage, the rate of request remains same for all the output lines of a particular stage. However, it may vary from stage to stage. Let p_i be the rate of request on an output line of the ith stage of switches. Clearly, p_o is the input rate of request which is equal to the probability of a processor generating a request. Let m_i be the probability that there is a favorite request on an input line to the (i+1)th stage, given that there is a request on that line. Let $m_i^!$ be the fraction of p_i available due to a favorite request at the input of switch A. So, $\bar{m}_i^!$, the fraction of p_i that comes from other inputs of switch A is $(1 - m_i^!)$.

From eqn. (4),

$$p_i = 1-(1-p_{i-1}m_{i-1})(1-p_{i-1} \cdot \frac{1-m_{i-1}}{a-1})^{a-1}$$
$$\text{for } 1 \leqslant i \leqslant n \qquad (11)$$

The rate of request at the output of switch A due to a favorite request $= m_i^! \, p_i$. Given k requests at an output line of switch A, a request is accepted with a probability of $\frac{1}{k}$ because of EA assumption. All other requests are rejected. In addition to a favorite request, (k-1) other requests arrive at the output line of switch A. If there are a total of j requests at the input of switch A, the rate of request at an output line due to a favorite request is:

$$\frac{1}{k}\left\{p_{i-1}m_{i-1}\binom{j-1}{k-1} x_{i-1}^{k-1} (1 - x_{i-1})^{j-k}\right\} \quad \text{where}$$

$$x_{i-1} = \frac{1-m_{i-1}}{a-1} \,.$$

For k varying between 1 to j, the rate of favorite requests at an output line of switch A is:

$$\sum_{1 \leqslant k \leqslant j} \frac{p_{i-1}m_{i-1}}{k} \binom{j-1}{k-1} x_{i-1}^{k-1}(1-x_{i-1})^{j-k}$$
$$= \frac{p_{i-1}\, m_{i-1}}{j\, x_{i-1}} \{1 - (1 - x_{i-1})^j\} \,.$$

With p_{i-1} being the probability of request generation for (j-1) input lines for an axa crossbar at the ith stage,

$$p_i m_i^! = \sum_{0 \leqslant j-1 \leqslant a-1} \binom{a-1}{j-1} p_{i-1}^{j-1} (1-p_{i-1})^{a-j} \frac{p_{i-}}{j} \frac{1m_{i-1}}{x_{i-1}}$$
$$\{1-(1-x_{i-1})^j\}$$
$$= \frac{m_{i-1}}{a\, x_{i-1}} \{1 - (1 - p_{i-1}\, x_{i-1})^a\} \,.$$

With $x_{i-1} = \frac{1 - m_{i-1}}{a-1}$,

$$p_i \; m_i' = \frac{m_{i-1}(a-1)}{a(1-m_{i-1})} \{1-(1-p_{i-1} \frac{1-m_{i-1}}{a-1})^a\} \quad . \qquad (12)$$

Using l' Hospital's rule it can be shown that $\lim\limits_{m_{i-1} \to 1} p_i m_i' = p_{i-1}$. This means all the requests are accepted if they were favorite. A closer look at the operation of delta networks (Fig. 7) reveals that m_{i-1} consists of two types of favorite requests to switch A . Let $m(f)_{i-1}$ be the fraction of m_{i-1} that consists of requests to memory module MM_i . $m(nf)_{i-1}$ is the fraction of m_{i-1} directed towards other memory modules, but appears as a favorite request to switch A. Assuming the requests due to $m(f)_{i-1}$ and $m(nf)_{i-1}$share m_i' , at the output of switch A, as per their proportion; the fraction of m_i' due to $m(f)_{i-1}$ is $\frac{m(f)_{i-1}}{m_{i-1}} \cdot m_i'$ and the fraction of m_i' due to $m(nf)_{i-1}$ is $\frac{m(nf)_{i-1}}{m_{i-1}} \cdot m_i'$.

Out of the non-favorite part of m_i' a fraction of requests will be directed towards (a-1) other outputs of switch B (Fig. 7) and the rest will appear as a favorite request.

In a delta network, a fraction of $\dfrac{\frac{N}{a^{i+1}} - 1}{\frac{N}{a^i} - 1}$ of this non-favorite rate of request appears as a favorite request to a switch at the (i+1)th stage for $1 \leqslant i \leqslant n-1$. Again, the fraction of the rate of request $\bar{m}_i'$, at the output line of switch A, consists of both favorite and non-favorite requests to switch B . The request rate is equally directed to all the a output lines of the switch B . Out of this, $\frac{a^i}{N}$ is the fraction of the request rate that is directed to favorite memory MM_i and $(\frac{1}{a} - \frac{a^i}{N})$ is the fraction of request rate, directed to other memory modules but appears as a favorite request to switch B.

Hence, at the input of the (i+1)th stage,

$$m(f)_i = \frac{m(f)_{i-1}}{m_{i-1}} \cdot m_i' + \frac{a^i}{N} \bar{m}_i' \quad ; \qquad (13)$$

$$m(nf)_i = \frac{\frac{N}{a^{i+1}} - 1}{\frac{N}{a^i} - 1} \cdot \frac{m(nf)_{i-1}}{m_{i-1}} \cdot m_i' + (\frac{1}{a} - \frac{a^i}{N}) \bar{m}_i' \qquad (14)$$

and

$$m_i = m(f)_i + m(nf)_i \quad \text{for } 1 \leqslant i \leqslant n-1 \; . \qquad (15)$$

At the input of the first stage, $m(f)_o$ is the probability that a processor requests its favorite memory given that it generates a request. Then, $m(nf)_o = \frac{\frac{N}{a} - 1}{N-1} \cdot (1 - m(f)_o)$. (16)

With given values of p_o and $m(f)_o$, p_i's and m_i's can be computed recursively for $1 \leqslant i \leqslant n-1$. The rate of request at the output of the final stage, p_n , decides the BW_f of an NxN delta network.

$$BW_f = p_n \times N \quad . \qquad (17)$$

Favorite Analysis of Omega Networks

An Omega network [5] is a special case of delta network for $N = 2^n$. With $a = 2$, the above set of equations can be simplified as below for $1 \leqslant i \leqslant n-1$.

$$m(nf)_o = \frac{\frac{N}{2} - 1}{N - 1} (1-m(f)_o) \text{ and } m_o = m(f)_o + m(nf)_o . \qquad (18)$$

$$p_i = 1-(1-p_{i-1}m_{i-1})(1-p_{i-1} + p_{i-1} \cdot m_{i-1}) \qquad (19)$$

$$m_i' = \frac{1}{p_i} \{m_{i-1}(p_{i-1} - \frac{1}{2} p_{i-1}^2) + \frac{1}{2} p_{i-1}^2 \cdot m_{i-1}^2\} (20)$$

$$\bar{m}_i' = 1 - m_i' \quad .$$

$$m(f)_i = \frac{m(f)_{i-1}}{m_{i-1}} m_i' + \frac{2^i}{N} \cdot \bar{m}_i' \quad . \qquad (21)$$

$$m(nf)_i = \frac{\frac{N}{2^{i+1}} - 1}{\frac{N}{2^i} - 1} \; \frac{m(nf)_{i-1}}{m_{i-1}} m_i' + (\frac{1}{2} - \frac{2^i}{N})\bar{m}_i' . (22)$$

$$m_i = m(f)_i + m(nf)_i \quad \text{for } 1 \leqslant i \leqslant n-1 \quad (23)$$

$$\text{and } BW_f = P_n N \; . \qquad (24)$$

The BW_f , obtained for an Omega network, for various values of N are plotted in Fig. 8 together with the BW_e for equally likely case. It may be noted that for higher values of $m(f)_o$ the performance of an Omega network is close to that of a crossbar. With a straight connection of 2x2 switches in an Omega network, a favorite case corresponds to an identity permutation. If most of the time an identity permutation is desired, less conflicts will occur which will give rise to an increased Bandwidth. The above set of equations, derived for Omega network, also holds good for MINs like Indirect binary n-cube, Generalized cube and Base line networks, as long as there is one and only one path from a processor to a memory module. The favorite memories may be different depending on the permutations obtained when all the switches are straight connected. Fig. 9 shows a variation of p_i's and m_i's at various stages of a 1024x1024 Omega network. When i=0 , it represents the input side and i=10 represents the output side of the Omega network. Although p_i reduces with increase in i because of more conflicts, m_i goes on increasing. This means that for large i , the rate of request at the output is mainly due to the favorite requests. The limiting value of m_i is unity.

CONCLUSIONS

Crossbar and Delta networks were analyzed in this paper for equally likely and favorite memory cases. Equally likely is shown to be a special case of the favorite memory analysis. With favorite memories, the Bandwidth is much higher because of less conflicts. The Delta networks perform close to crossbars for favorite memory cases, thus increasing the cost effectiveness. In a multistage interconnection network, the rate of request at a stage (p_i) reduces with increase in the stages, but the rate of favorite request goes on increasing, being limited to unity.

The analysis has been restricted to NxN delta networks because of the complexity involved. The analytical results match with those obtained in simulations.

REFERENCES

1. W.A. Wulf and C.G. Bell, "Cmmp - A Multi-miniprocessor", Proc. AFIPS, Fall Joint Computer Conference, Dec. 1972.

2. D.P. Bhandarkar, "Analysis of Memory Interference in Multiprocessor", IEEE Trans. on Computers, C-24, Sept. 1975, pp. 897-908.

3. W.D. Strecker, "Analysis of the Instruction Execution Rate in Certain Computer Structures", Ph.D. dissertation, Carnegie-Mellon University, 1970.

4. J.H. Patel, "Performance of Processor-Memory Inter-connections for Multiprocessors", IEEE Trans. on Computers, C-30, Oct. 1981, pp. 771-780.

5. D.H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Trans. on Computers, C-24, Dec. 1975, pp. 1145-1155.

6. M.C. Pease, "The Indirect Binary N-Cube Microprocessor Array", IEEE Trans. on Computers, C-26, May 1977, pp. 458-473.

7. H.J. Siegel and R.J. McMillan, "The Multistage Cube: A Versatile Interconnection Network", Computer, Vol. 14, no. 12, Dec. 1981, pp. 65-76.

8. C.L. Wu and T.Y. Feng, "On a class of Multi-stage Interconnection Networks", IEEE Trans. on Computers, C-29, Aug. 1980, pp. 694-702.

9. D.M Dias and J.R. Jump, "Analysis and Simulation of Buffered Delta Network", IEEE Trans. on Computers, C-30, April 1981, pp. 273-282.

10. S. Thawawastien and V.P. Nelson, "Interference Analysis of Shuffle/Exchange Network", IEEE Trans. on Computers, C-30, Aug. 1981, pp. 545-556.

11. L.N. Bhuyan and D.P. Agrawal, "Design and Performance of a General Class of Interconnection Networks", Proc. 1982 Int. Conf. on Parallel Processing, Aug. 1982, pp. 2-9. Also to appear in IEEE Trans. on Computers.

12. M.A. Franklin, "VLSI Performance Comparison of Banyan and Crossbar Communication Networks", IEEE Trans. on Computers, C-30, April 1981, pp. 283-291.

13. L.N. Bhuyan and D.P. Agrawal, "VLSI Performance of Multistage Interconnection Networks using 4x4 switches", Proc. 3rd Int. Conf. on Distributed Computing Systems, Oct. 1982, pp. 606-613.

14. L.R. Goke and G.J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems", Proc. 1st Int. Symp. on Computer Architecture, Dec. 1973, pp. 21-28.

15. M.J. Flynn, "Some Computer Organizations and their Effectiveness", IEEE Trans. on Computers, C-21, Sept. 1972, pp. 948-960.

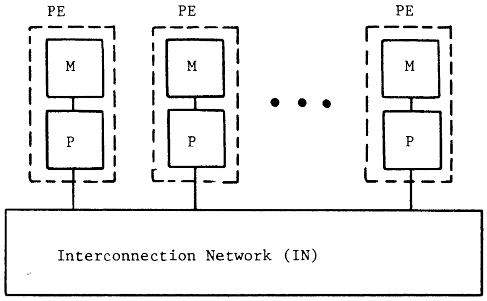


Fig. 1a. A loosely coupled multiprocessor

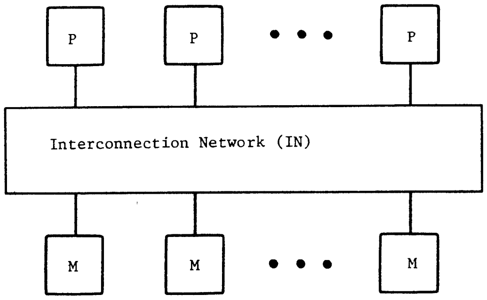


Fig. 1b. A tightly coupled multiprocessor

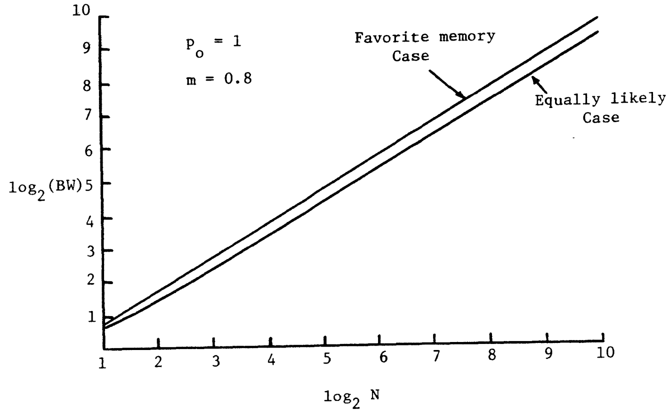


Fig. 2. Bandwidth of N X N crossbars

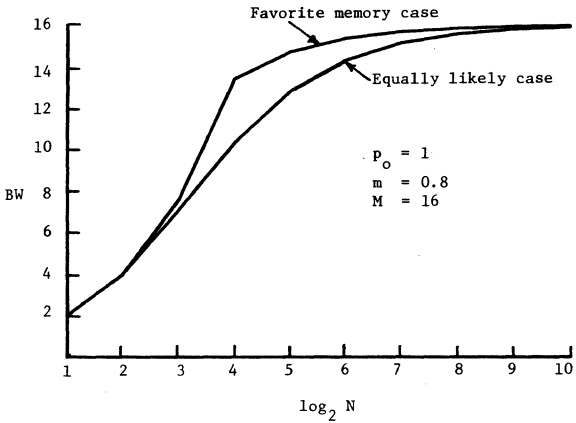


Fig. 4. Variation of BW in an M x N crossbar with M = 16

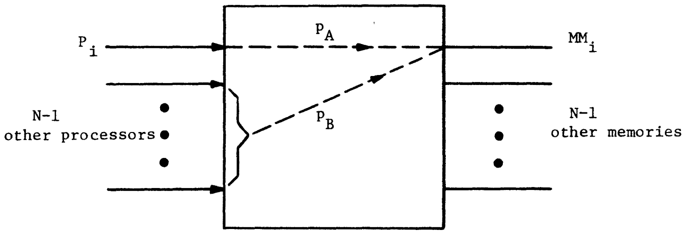


Fig. 3a. Request at MM_i in an N x N crossbar

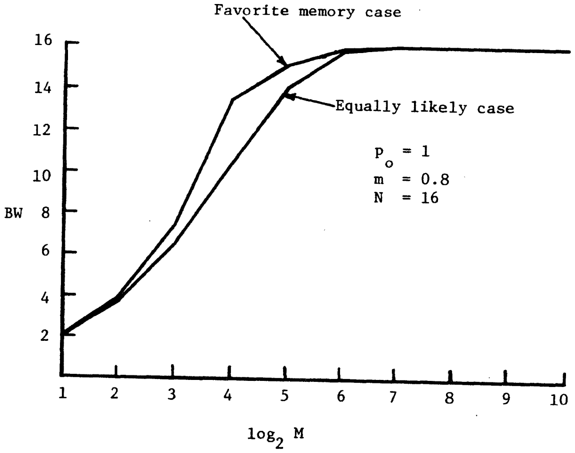


Fig. 5. Variation of BW in an M x N crossbar with N = 16.

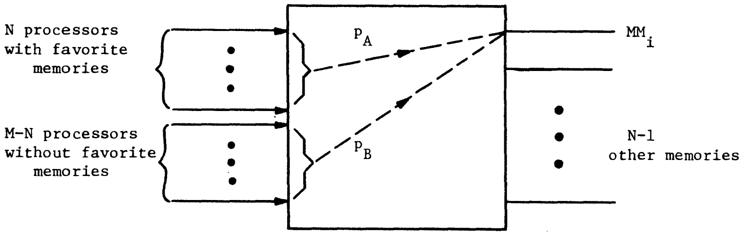

N processors with favorite memories
M-N processors without favorite memories
P_A
P_B
MM_i
N-1
other memories

Fig. 3b. Request at MM_i in an M x N crossbar with M $\geq$ N

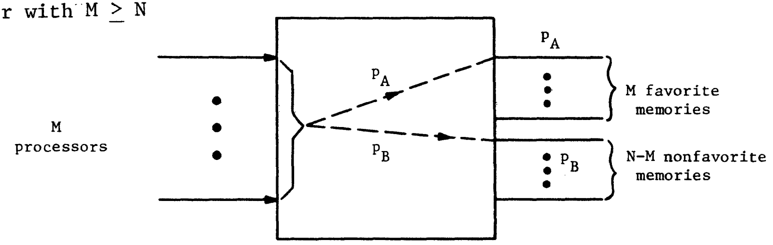


Fig. 3c. Request at memories in an M x N cross bar with N $\geq$ M

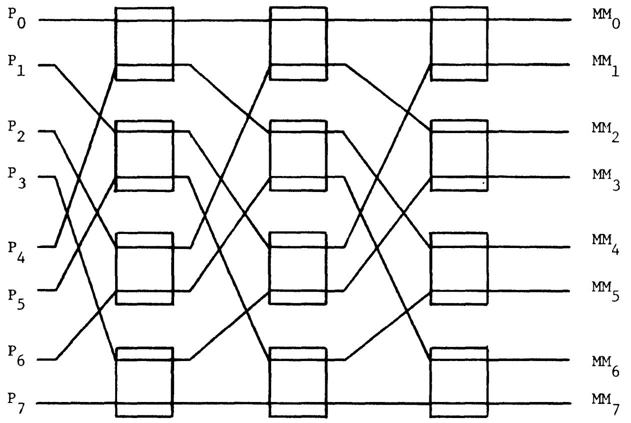


Fig. 6. Favorite memory connection for a 8 x 8 Omega network

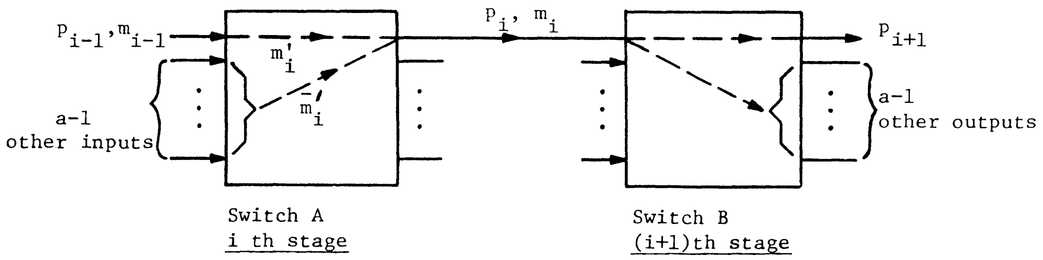


Fig. 7. Requests at two adjacent stages of a Delta network

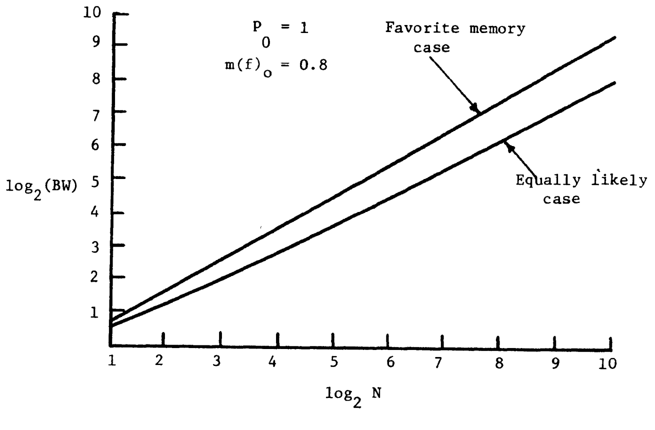


Fig. 8. Bandwidth for N x N Omega networks

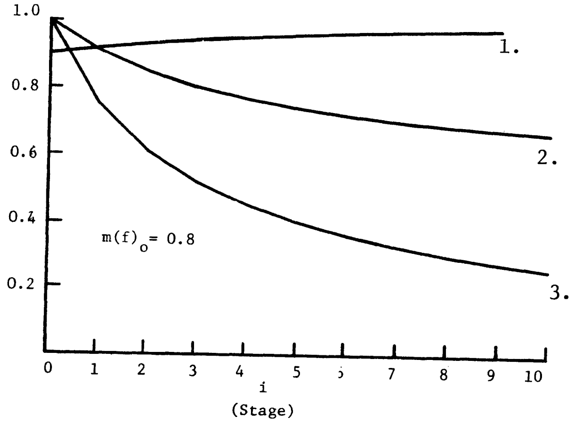


Fig. 9. p_i,m_i at various stages of a 1024 x 1024 Omega network.

1. m_i, favorite memory case; 2. p_i, favorite memory case; 3. p_i, equally likely case.

GENERALIZED DELTA NETWORKS

Manoj Kumar and J. R. Jump

Department of Electrical Engineering, Rice University
Houston, TX 77251

Abstract

The throughput of unbuffered delta networks is related to the arrival rate by a quadratic recurrence relation. Lower and upper bounds on the solution of this recurrence relation are derived in this paper.

Two approaches for improving the throughput of unbuffered delta networks are discussed in this paper. The first approach combines multiple delta subnetworks of size N×N each in parallel, to obtain a network of size N×N. Three distribution policies, used to distribute the incoming packets between the subnetworks, are discussed in this paper and their effect on the throughput is investigated.

The second approach replaces each link of the simple delta networks by K parallel links (K=2,4, ...). The throughput of these networks is analyzed and one possible implementation for the crossbar switches to be used in these networks is discussed. The throughput of such networks with four parallel links is almost equal to the throughput of crossbars.

1. Introduction

Delta networks have been considered frequently for processor-memory and processor-processor interconnection in modular computer systems such as SIMD, MIMD and Data Flow Machines [1, 3, 7, 8, 9, 11, 12, 14]. An N×N ($N = 2^n$) delta network can be constructed from basic switches of size B×B ($B = 2^b$), each capable of connecting its inputs to any of its outputs (see Figure 1.1).

The network has n/b stages (numbered 1,2,...,n/b) and each stage has 2^{n-b} basic switches. The outputs of switches in all stages, except the last, are connected to the inputs of switches in the next stage by the shuffle connection or one of its minor variants. The network shown in Figure 1.1 is an 8×8 delta network constructed from 2×2 basic switches. The N inputs of the switches in the first stage and the N outputs of the switches in the last stage constitute the inputs and the outputs of the network. A truncated delta network is obtained by deleting one or more stages from a regular delta network.

The modules at the network inputs generate fixed sized packets to be transmitted over the network. The arrival of packets at the network inputs are independent and identical Bernoulli process with parameter X_{in} (the arrival rate). These packets are directed equiprobably to all network outputs.

All the switches in the network are synchronized by a single clock. A connection between two switches which is capable of carrying one packet in each clock cycle is called a link.

The switches in a buffered delta network have internal buffers to temporarily store an incoming packet that cannot be forwarded in the current cycle. Unbuffered delta networks have no such internal buffers. In this paper we will investigate the performance of unbuffered delta networks only.

The network control is decentralized and each switch in the network operates autonomously. In addition to data, each packet carries its destination address. A switch in stage i ($1 \leq i \leq n/b$) uses bits $b_{c[(i-1)b+1]},\ldots,b_{c[i*b]}$ of the destination address (expressed as an binary number $b_1 b_2 \ldots b_{n/b}$) to route the packets to the appropriate output port. These bits are called the control bits for stage i. The operation of delta networks is described in detail in [12].

Packets arriving at two distinct network inputs may require the use of a common link between two stages. Since only one packet can use that link in a clock cycle, one of the packets will be ignored in the current cycle and resubmitted at a later time. Because of such conflicts there is a degradation in the throughput (number of packets transmitted/unit cycle) of the network.

The performance of unbuffered delta networks has been investigated by Patel[12] and Dias and Jump [5, 4]. The throughput of an unbuffered delta network has been expressed as a quadratic recurrence relation [12]. Unfortunately, this recurrence relation fails to show the dependence of network throughput on the number of stages in the network, the basic switch size, and the arrival rate.

Kruskal and Snir provide asymptotic solutions for this recurrence relation [10]. In this paper we show that one of these solutions is a strict upper bound on the performance of delta networks. We also derive a strict lower bound on the performance of delta networks, which is much more accurate than the upper bound for networks constructed from 2×2 switches. Both these bounds incorporate the dependence of network throughput on the number of stages, basic switch size and the arrival rate.

The performance of unbuffered delta networks can be improved by either using multiple delta subnetworks in parallel as shown in Figure 1.2a [8], or by replacing each link in the delta network by multiple links as shown in Figure 1.2b [13].

In the first approach, various policies can be used for distributing the incoming packets between the subnetworks. Some of these are discussed in this paper, where the effect of distribution policy on the performance of the network is investigated.

The throughput of multiple link delta networks, constructed from 2×2 switches, can be expressed as a set of coupled nonlinear recurrence relations [10]. These recurrence relations again fail to show the dependence of throughput on the number of stages in the network, the switch size and the arrival rate. In this paper we analyze multiple link delta networks constructed from larger switches. The throughput of these networks is expressed by coupled nonlinear recurrence relations. An approximate solution with a simple

This work was supported by the National Science Foundation under the grant MCS 80-01667.

0190-3918/83/0000/0010$01.00 © 1983 IEEE

functional form is also derived for the throughput.

The above mentioned approaches also improve the fault-tolerance of the network. In the first approach, only one correctly functioning delta subnetwork is required to allow communication between any input, output pair of the network. In the second approach, only one out of each set of links connecting two particular switches, is required to function correctly.

In section two of this paper we will establish fairly tight lower and upper bounds on the throughput of simple delta networks. These bounds have simple functional forms. In section 3 different techniques for combining multiple delta networks in parallel are considered and the improvement in throughput achieved by the use of different distribution policies is compared. In section 4 the use of multiple links is investigated. Switch implementations for supporting multiple links are described. The throughput of these networks is compared with the throughput of crossbars.

In sections 3 and 4, basic switches of size 2×2 only have been considered to keep presentation simple. However, the results can be easily generalized for B×B switches.

2. Performance of Unbuffered Delta Networks

The following expression for the throughput of a B×B crossbar switch was derived by Patel [12]. If the arrival of packets at the inputs of a switch are independent and identical Bernoulli process with the same arrival rate X_{in}, then the arrival of packets at each output (output process) is a Bernoulli process with the parameter X_{out} (output rate). X_{out} is related to X_{in} as follows

$$X_{out} = 1 - (1 - X_{in}/B)^B \qquad (2.1)$$

The output processes at different outputs of a switch are identical but not independent. In an N×N delta network constructed from B×B switches, the output rate of switches in stage i is denoted by x_i. The arrival rate at the input links of the network, X_{in}, is equal to x_0, the arrival rate at the input of switches in stage 1. The throughput of the network, X_{out}, is equal to $x_{n/b}$. It was shown by Dias that the arrival of packets at the B inputs of the same switch in any stage are identical and independent Bernoulli processes [6]. Therefore the output rate x_i of stage i $(1 \le i \le n/b)$ can be expressed by the quadratic recurrence relation

$$x_i = 1 - (1 - x_{i-1}/B)^B \qquad (2.2)$$

$$x_0 = X_{in}, \quad X_{out} = x_{n/b}, \quad \text{and } 0 \le X_{in} \le 1$$

Unfortunately, this recurrence relation provides no insight into either the functional form of X_{out} and its dependence on x_{in}, N and B or the upper and lower bounds on X_{out}. Such functional forms or bounds would give a better idea of the network throughput and allow us to compare the throughputs of various networks without resorting to computationally intensive or graphical techniques. The upper and lower bounds on x_i are derived as follows

Define $y_i = 1 / x_i$ for $0 \le i \le n/b$

then

$$y_{i+1} - y_i = \frac{1}{1 - (1 - x_i/B)^B} - \frac{1}{x_i}$$

$$= \frac{\binom{B}{2}\left(\frac{x_i}{B}\right)^2 - \binom{B}{3}\left(\frac{x_i}{B}\right)^3 + \ldots}{x_i * \left[\binom{B}{1}\left(\frac{x_i}{B}\right) - \binom{B}{2}\left(\frac{x_i}{B}\right)^2 + \ldots\right]} \qquad (2.3)$$

therefore

$$\lim_{y_i \to \infty} \quad y_{i+1} - y_i = \frac{B-1}{2B}$$

Define $e_i = y_{i+1} - y_i - \frac{B-1}{2B}$

then

$$e_i = \frac{2Bx_i - (2B + (B-1)x_i)(1 - (1 - x_i/B)^B)}{x_i(1 - (1 - x_i/B)^B)\,2B} \qquad (2.4)$$

and

$$y_i = y_0 + \frac{i(B-1)}{2B} + \sum_{j=0}^{i-1} e_j \qquad (2.5)$$

Let n_i and d_i denote tne numerator and the denominator in the expression for e_i in equation 2.4.

$$d_i = 2Bx_i * \left[x_i - \binom{B}{2}\left(\frac{x_i}{B}\right)^2 + \left\{\binom{B}{3}\left(\frac{x_i}{B}\right)^3 - \binom{B}{4}\left(\frac{x_i}{B}\right)^4\right\} + \ldots + \left\{ \text{last term} \right\}\right] \qquad (2.6)$$

where the last term is $\{B(x_i/B)^{B-1} - (x_i/B)^B\}$ iff B is odd and it is $\{(x_i/B)^B\}$ otherwise. Since each term witnin the braces is a positive quantity (because x_i is less than 1) we have tne inequaiity

$$d_i \ge 2Bx_i^2 - (B-1)x_i^3 \qquad (2.7)$$

Similarly, n_i can be wrıtten as

$$n_i = 2B * x_i - (2B + (B-1)x_i)\left[x_i - \left\{\binom{B}{2}\left(\frac{x_i}{B}\right)^2 - \binom{B}{3}\left(\frac{x_i}{B}\right)^3\right\} - \left\{\binom{B}{4}\left(\frac{x_i}{B}\right)^4 - \binom{B}{5}\left(\frac{x_i}{B}\right)^5\right\} - \ldots - \left\{\text{last term}\right\}\right] \qquad (2.8)$$

where the lasc term is $\{(x_i/B)_B\}$ itf B is odd and it is $\{B(x_i/B)^{B-1} - (x_i/B)^B\}$ otherwise.

Again, each term within the braces is positive and therefore

$$n_i \leq 2Bx_i - (2B+(B-1)x_i)\left[x_i - \binom{B}{2}\left(\frac{x_i}{B}\right)^2 + \binom{B}{3}\left(\frac{x_i}{B}\right)^3\right]$$

simplifying this expression we have

$$n_i \leq x_i^3 \left[\frac{B^2-1}{6B}\right] + \frac{2\binom{B}{4}}{B^3}x^4$$

since $0 \leq x_i \leq 1$

$$n_i \leq x_i^3 \left[\frac{B^2-1}{6B} + \frac{2\binom{B}{4}}{B^3}\right] \tag{2.9}$$

using the bounds for n_i and d_i we get the upper bound for e_i

$$e_i \leq \frac{x_i^3 \left[\frac{B^2-1}{6B} + \frac{2\binom{B}{4}}{B^3}\right]}{2Bx_i^2 - (B-1)x_i^3}$$

$$\leq \frac{\left[\frac{B^2-1}{6B} + \frac{2\binom{B}{4}}{B^3}\right]}{2B*y_i - (B-1)} \tag{2.10}$$

If x_i is positive then both n_i and d_i are positive (follows from equations 2.7 and 2.9), and therefore x_{i+1} and e_i will be positive too. From this lower bound on e_i the following lower bound on y_i follows easily

$$y_i \geq y_0 + i(B-1)/2B \tag{2.11}$$

In equation 2.10 the occurrence of y_i in the denominator can be replaced by the lower bound for y_i and the following inequality is obtained

$$e_i \leq \frac{\left[\frac{B^2-1}{6B} + \frac{2\binom{B}{4}}{B^3}\right]}{2B*y_0 + (B-1)*i - (B-1)}$$

Simplifying this inequality we have

$$e_i \leq \frac{B^2 - B + 2}{4B^2[i + 2y_0B/(B-1) - 1]}$$

Therefore

$$\sum_{j=0}^{i-1} e_j \leq \frac{B^2 - B + 2}{4B^2} * \log_e\left[\frac{i(B-1)}{(2By_0 - 2B + 2)} + 1\right] \tag{2.12}$$

Denote the right hand side of the above inequality by E_i. Thus, the upper bound on y_i is

$$y_i \leq y_0 + i(B-1)/(2B) + E_i \tag{2.13}$$

The upper and lower bounds for x_i are obtained by inverting the lower and upper bounds for y_i. Thus, we have the result

$$\frac{2B}{2By_0 + (B-1)i} \geq x_i \geq \frac{2B}{2By_0 + (B-1)i + 2BE_i} \tag{2.14}$$

In Figures 2.1a-2.1d the throughput of delta networks, obtained from recurrence relation (2.2), is compared with the lower and upper bounds given by equation (2.14). For low arrival rates the bounds are much more accurate than for high arrival rates. For networks constructed from 2×2 switches, the lower bounds are within 5% of the actual throughput and the upper bounds are within 10% of the actual throughput (for $X_{in} = 1$).

For larger switch sizes (sizes > 4×4) the lower bounds are less accurate than the upper bounds. The upper bounds are still within 10% of the actual throughput.

3. Connecting Delta Networks in Parallel

Three techniques for using $K = 2^k$ delta subnetworks or truncated delta subnetworks (constructed from 2×2 switches) of size N×N each in parallel, to obtain a network of size N×N, are discussed in this section. These techniques differ primarily in the distribution policy used to distribute the incoming packets between the subnetworks.

The first technique is to connect the i^{th} input of the network ($0 \leq i \leq N-1$) to the i^{th} input of each delta subnetwork through a 1-to-K demultiplexer. Similarly the i^{th} output of each delta subnetwork is connected to the i^{th} output of the network through a K-to-1 multiplexer (see Figure 1.2a). The demultiplexers forward an incoming packet to any of the K subnetworks equiprobably. If multiple requests arrive at the input of a multiplexer in the last stage, one of them is selected equiprobably and is forwarded to the output of the network. This network is called a Randomly loaded parallel delta network (Rn).

If X_{in} is the arrival rate at the input of the network, then the arrival rate at the input of each delta subnetwork, x_0, is equal to X_{in}/K.

The output rate at the output links of each delta subnetwork, x_n, can be obtained from recurrence relation (2.2). The output rate at the output of the network, X_{out}, is given by the expression

$$X_{out} = 1 - (1 - x_n)^K \tag{3.1}$$

If x_l and x_u are the upper and the lower bounds on x_n, then the upper and lower bounds on X_{out} are

$$1-(1-x_l)^K \leq X_{out} \leq 1-(1-x_u)^K \tag{3.2}$$

In the technique described above, a packet is blocked within the subnetwork with probability

$1 - x_n$. If every packet arriving at an input of the network is forwarded to more than one delta subnetwork simultaneously, then the probability that all copies of the same packet are blocked within the subnetworks is expected to be much less than $1 - x_n$. The reduction in blocking of packets within the subnetworks will in turn increase the throughput of the network. The second technique for combining multiple delta subnetworks in parallel, utilizes this fact. In this technique the packets arriving at the inputs of the network are forwarded to all the subnetworks. This network is called a Multiple loaded parallel delta network (Mn). Since the throughput of this network could not be determined analytically, simulation techniques were used.

The last technique is to demultiplex the incoming packets between K truncated delta subnetworks according to the destination address of the packet. Each subnetwork has n-k stages and the control bit for for stage i of the subnetwork is b_{k+i}. The subnetworks are numbered 0 through K-1. The incoming packet is forwarded to subnetwork j iff $b_1 b_2 \ldots b_k$, the first k bits in the binary representation of the destination address, are also the binary representation of j.

All the K outputs of the j^{th} subnetwork, whose binary representations differ only in the first k bits are connected to one K-to-1 multiplexer (see Figure 3.1). The output of this multiplexer is the network output with binary representation $b_1 b_2 \ldots b_k o_{k+1} o_{k+2} \ldots o_n$, where $o_{k+1} o_{k+2} \ldots o_n$ are the common bits in the binary representation of these subnetwork outputs and $b_1 b_2 \ldots b_k$ is the binary representation of j. This network is called a Selectively loaded parallel delta network (Sn).

If X_{in} is the arrival rate at each input of the network, then the arrival rate at the inputs of the subnetworks, x_0, is equal to X_{in}/K. The output rate at the output of each truncated delta subnetwork will be x_{n-k}, which can be obtained from recurrence relation (2.2). The output rate, X_{out}, at each output of the network is given by the expression

$$X_{out} = 1 - (1 - x_{n-k})^K \qquad (3.3)$$

The upper and lower bounds on X_{out} can be obtained by using the upper and lower bounds of x_{n-k} in the above expression.

In Figures 3.2a and 3.2b the throughputs obtained by using different distribution policies are compared. The performance of Sn is better than that of Rn because the subnetworks in Sn have fewer stages and therefore fewer packets are lost due to collisions in the subnetworks. The performance of Mn was expected to be better than that of Rn, because in Mn all possible paths between an input and an output are tried simultaneously. However, each subnetwork in this situation is more heavily loaded and the number of collisions in the subnetworks are greater. The increased number of collisions almost offsets the advantage of using multiple paths simultaneously.

The throughputs obtained from a simple delta network, from the use of multiple links (to be discussed in the next section), and from an ideal crossbar [12], are also shown in this figure to illustrate the relative advantage of using the two approaches discussed in this paper.

4. Using Multiple Links

In this section we propose the use of delta networks with multiple links. In an N×N delta network constructed from 2×2 crossbar switches, each crossbar switch can receive up to $K = 2^k$ packets at each of its input ports and it can forward at most K packets to any output port. Figure 1.2b shows an 8×8 delta network implemented from 2×2 switches for K=2. To connect the input port of a switch to the output port of another, K independent links are used (since each link carries only one packet in a clock cycle). The input ports of switches in the first stage of the network receive packets only on one of the K links, which is the input link of the network (remaining K-1 links are unused). The packets on the K links of the output are multiplexed on a single link which is an output link of the network. A network with K parallel links is denoted by D_K.

Figures 4.1a and 4.1b show one possible implementation of a switch for D_2 and D_4. The operation of the switch for D_4 is described next. The switch implementation can be easily generalized for arbitrary values of K.

Each switch in D_4 contains two banks of four 1-to-2 demultiplexers, labeled the ´U´ bank and the ´L´ bank. The inputs of the ´U´ bank demultiplexers receive packets from the upper input port and those of the ´L´ bank receive packets from the lower input port. The outputs of the demultiplexers are labeled ´0´ and ´1´. The demultiplexers are followed by four 4_input sorters, labeled uu, ul, lu and ll respectively. The ´0´ and ´1´ outputs of the demultiplexers in the ´U´ bank are connected to the uu and ul sorters, and those of the ´L´ bank are connected to the lu and ll sorters. The sorters are followed by two banks of four 2-to-1 multiplexers, which are again labeled the ´U´ bank and the ´L´ bank. The outputs of uu and lu sorters are connected to the multiplexers in the ´U´ bank, and the outputs of ul and ll sorters are connected to the multiplexers in the ´L´ bank. The outputs of the ´U´ and ´L´ multiplexers form the upper and the lower output of the switch. The demultiplexers in the switch forward the incoming packets to their ´0´ output if they are directed to the upper output of the switch and to their ´1´ output otherwise. The sorters move the x packets arriving at their inputs ($0 \leq x \leq 3$) to their outputs labeled 0 through x. The sorting network is constructed from Bitonic Sorters [2].

In a switch for D_4, each sorter contains six switching elements shown as boxes with arrows. If exactly one input of an switching element has a packet on it, the packet is forwarded to the upper output port if the arrow in the box points upwards and to the lower output port if it points down. If both the input ports of a switching element have packets on them, they are both passed straight through.

If the outputs of the two sorters connected to the same multiplexer bank have a total of K or fewer (more than K) packets, the multiplexers in the bank will forward all the (only K of these) packets to the switch output.

We will derive the expression for the throughput of a 2×2 crossbar switch with K links on each input and output port. This result will be used to derive the throughput of a N×N delta network constructed from such switches.

If j packets arrive at the K links of an input port with probability $X_{in}(j)$, which is independent of the arrival of packets on the second input port, then the probability of finding m packets on an output port is given by the expression

$$X_{out}(m) = \sum_{i=0}^{K} \sum_{\substack{j=m-i \\ j \geq 0}}^{K} X_{in}(i) X_{in}(j) \binom{i+j}{m} 2^{-(i+j)} \quad \text{for } m = 0,1,2,\ldots,K-1$$

$$X_{out}(K) = \sum_{i=0}^{K} \sum_{j=K-i}^{K} X_{in}(i) X_{in}(j)\, 2^{-(i+j)} \sum_{m=K}^{i+j} \binom{i+j}{m} \qquad (4.1)$$

In an N×N delta network made up of such switches, let $x_i(m)$ $(0 \leq m \leq K)$, denote the probability of finding m packets on the output of a switch in stage i (for $1 \leq i \leq n$) and let $x_0(m)$ denote the probability of finding m packets at the input of a switch in stage 1 $(\sum_{m=0}^{K} x_i(m) = 1, \text{ for } 0 \leq i \leq n)$.

If X_{in} is the arrival rate at every input of the network then

$$x_0(0) = 1 - X_{in}$$

$$x_0(1) = X_{in}$$

$$x_0(2) = x_0(3) = \ldots = x_0(K) = 0$$

Since packets arrive at the two inputs of the same switch in any stage independently (the reasons for this are cited in section 2), the probabilities of finding m (m<K) packets at the output of a switch in stage s $(1 \leq s \leq n)$ can be expressed by the following coupled recurrence relations

$$x_s(m) = \sum_{i=0}^{K} \sum_{\substack{j=m-i \\ j \geq 0}}^{K} x_{s-1}(i) x_{s-1}(j) \binom{i+j}{m} 2^{-(i+j)}$$

$$x_s(K) = \sum_{i=0}^{K} \sum_{j=K-i}^{K} x_{s-1}(i) x_{s-1}(j) 2^{-(i+j)} \sum_{m=K}^{i+j} \binom{i+j}{m} \qquad (4.2)$$

The values of $x_n(0), x_n(1), \ldots, x_n(K)$ can be obtained by solving these recurrence relations [10]. The output rate at each output of the network, X_{out} will then be given by

$$X_{out} = 1 - x_n(0)$$

If the modules connected to the output links of the network are capable of accepting $\hat{K}$ $(1 \leq \hat{K} \leq K)$ packets in each clock cycle, then the throughput of the network will be given by the expression

$$X_{out} = \sum_{i=1}^{K} x_n(i) * \text{MIN}\{i, \hat{K}\}$$

In this case the K-to-1 multiplexers, at the outputs of each switch in the last stage, must be replaced by K-to-$\hat{K}$ concentrators.

As mentioned in section 2, the coupled recurrence relations (4.2) do not specify the functional form of X_{out} or its dependence on X_{in} and n. To find such a functional form we assume that for each stage i between the k^{th} and the last stage, $x_i(j)$ specifies a binomial distribution with mean $K*p_i$, i. e.,

$$x_i(j) = \binom{K}{j} p_i^{\,j} (1-p_i)^{K-j}$$

The first k stages can be ignored since there can be no conflict in these stages.

Due to the conflicts in the i^{th} stage p_{i+1} is less than p_i, but $x_{i+1}(j)$ $(0 \leq j \leq n)$ is still assumed to specify a binomial distribution. It can be shown that

$$p_k = X_{in}/K$$

and

$$X_{out} = 1 - \left(1 - \frac{p_{n-1}}{2}\right)^{2K}$$

The following recurrence relation holds for the values of p_is $(k \leq i < n)$

$$p_{i+1} = \left[\sum_{j=0}^{2K} \binom{2K}{j} \left(\frac{p_i}{2}\right)^{j} \left(1 - \frac{p_i}{2}\right)^{2K-j} \text{MAX}\{j,K\} \right] \frac{1}{K} \qquad (4.3)$$

Here $p_i/2$ is the probability that a link at the input of a switch in stage i (one of the K links at each input of a switch) carries a request which must be forwarded to a given output port. This recurrence relation can be simplified to

$$p_{i+1} = p_i - \frac{1}{K} \binom{2K}{K+1} * 2^{-(K+1)} * p_i^{K+1} + \text{high order terms}$$

By deleting the high order terms of p_i and denoting the coefficient of p_i^{K+1} by Q in the above equation, the following recurrence relation is obtained

$$p_{i+1} = p_i - Q * p_i^{K+1}$$

The forward difference function of this difference equation is used as differential operator to obtain the following differential equation

$$\frac{dp}{di} = -Q * p^{K+1}$$

The solution of this differential equation at integer points in the domain will be an approximation for the solution of the difference equation, and thus we have the following approximate solution for equation (4.3)

$$p_i = p_0 \sqrt[K]{\frac{1}{1 + KQ\,i\,p_0^{K}}} \qquad (4.4)$$

The analysis presented above can be generalized to obtain the throughput, X_{out}, of $N \times N$ delta networks constructed from $B \times B$ crossbar switches, where K parallel links are used for each connection between two switches. The actual throughput is obtained from the set of coupled nonlinear recurrence relations listed below. In these equations, I denotes the total number of packets arriving at all the inputs of a crossbar switch and $<i_0, i_1, \ldots, i_{B-1}>$ is a partion of I, which denotes the number of packets arriving at each switch input.

Recurrence relation

$$p_s(m) = \sum_{\substack{i_0, i_1, \ldots, i_{B-1} \\ I = i_0 + i_1 + \ldots + i_{B-1} > m}}^{BK} \left[\left[\prod_{n=0}^{B-1} x_{s-1}(i_n) \right] \binom{I}{m} B^{-I} (B-1)^{I-m} \right]$$

for m < K and s = 1,2,...,n/b

$$p_s(K) = \sum_{\substack{i_0, i_1, \ldots, i_{B-1} \\ I = i_0 + \ldots + i_{B-1} > K}}^{BK} \left[\prod_{n=0}^{B-1} x_{s-1}(i_n) \quad \sum_{m=K}^{I} \binom{I}{m} B^{-I} (B-1)^{I-m} \right]$$

for s = 1,2,...,n/b

Initial Conditions

$$x_0(0) = 1 - X_{in}$$

$$x_1(0) = X_{in}$$

$$x_2(0) = x_3(0) = \ldots = x_K(0) = 0$$

Throughput

$$X_{out} = 1 - x_{n/b}(0)$$

Similarly, by making the binomial approximation, the following equations are obtained for the throughput

$$p_0 = \frac{X_{in}}{K}$$

$$p_i = p_0 \sqrt[K]{\frac{1}{1 + KQ i p_0^K}}$$

where $Q = \frac{1}{K} \begin{matrix} BK \\ K+1 \end{matrix} B^{-(K+1)}$ and

$$X_{out} = 1 - \left(1 - \frac{p_{(n/b-1)}}{B}\right)^{BK}$$

Figures 4.2a and 4.2b show the throughput of network D_2. The actual throughput obtained from recurrence relation (4.2) and the throughput estimates obtained from equations (4.3) and (4.4) are shown together. Figures 4.2c and 4.2d show the same for D_4. The estimates obtained by equations (4.3) and (4.4) are within 10% of the actual throughput for networks of sizes less than $2^{25} \times 2^{25}$. Equation (4.4) turns out to be a very good approximation for the nonlinear recurrence relation (4.3).

From Figure 3.4 it is clear that the improvement in throughput obtained by replacing each link of a simple delta network by K parallel links is greater than that obtained by using K delta subnetworks in parallel regardless of the distribution policy.

The throughput obtained by using four links in parallel is very close to the throughput of a crossbar.

5. Conclusion

Fairly tight lower and upper bounds were derived for the throughput of unbuffered delta networks. These bounds have simple functional forms and they illustrate the dependence of network throughput on the arrival rate, network size and basic switch size.

Two approaches for enhancing the performance of delta networks were discussed. For networks obtained by combining multiple delta subnetworks in parallel, three distribution policies were proposed for distributing the incoming packets between the subnetworks. The effect of the distribution policy on the throughput of the network was investigated.

Then, networks obtained by replacing each link of a simple delta network by multiple links were considered. The throughput of these networks was analyzed. One possible implementation for the basic switches to be used in these networks was described.

References

[1] Barnes, G. H. and Lundstrom, S. F., "Design and Validation of a Connection Network for Many-Processor Multiprocessor systems," Computer 14(12), pp. 31-41 (Dec. 1981).

[2] Batcher, K. E., "Sorting Networks and their Applications," Proc. of the Spring Joint Computer Conference , AFIPS press, Montvale, N. J. (1968).

[3] Dennis, J. B., "Data Flow Supercomputers," Computer 13(11), pp. 48-56 (Nov. 1980).

[4] Dias, D. M. and Jump, J. R., "Packet Switching Interconnection Networks for Modular Systems," Computer 14(12), pp. 43-53 (1981).

[5] Dias, D. M. and Jump, J. R., "Analysis and Simulation of Bufferred Delta Networks," IEEE Trans. on Computers C-30(4), pp. 273-282 (April 1981).

[6] Dias, D. M., "Packet Switching in Delta and Related Networks," Ph. D. Dissertation, Rice University, Houston, Tx. (May 81).

[7] Feng, T., "Data Manipulating Functions in Parallel Processors and Their Implementations," IEEE Trans. on Computers C-23(3), pp. 309-318 (Mar. 1974).

[8] Goke, L. R. and Lipovski, G. J., "Banyan Networks for Partitioning Multiprocessor Systems," Proc. of the 1st Annual Symposium on Computer Architecture , pp. 21-28, ACM, New York, N. Y. (1973).

[9] Gottlieb, A., Grishman, R., Kruskal, C. P., McAullife, K. P., Rudolph, L., and Snir, M., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," IEEE Trans. on Computers C-32(2), pp. 175-189 (Feb. 1983).

[10] Kruskal, C. P. and Snir, M., "The Performance of Multistage Interconnection Networks for Multiprocessors," Private communication .

[11] Lawrie, D. H., "Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers C-24(12), pp. 1145-1155 (Dec. 1975).

[12] Patel, J. H., "Performance of Processor-Memory Interconnections for Multiprocessors," IEEE Trans. on Computers C-30(10), pp. 771-780 (Oct. 1981).

[13] Schwartz, J. T., "The Burroughs FMP Machine, Ultracomputer Note #5," Courant Institute, NYU (1980).

[14] Siegel, H. J. and McMillen, R. J., "Using the Augmented Data Manipulator Network in PASM," Computer 14(2), pp. 25-33 (Feb. 1981).

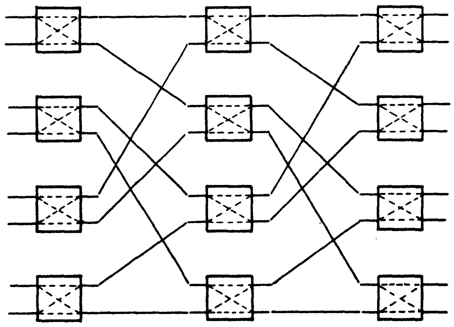


Figure 1.1 : An 8X8 delta network constructed from 2X2 switches.

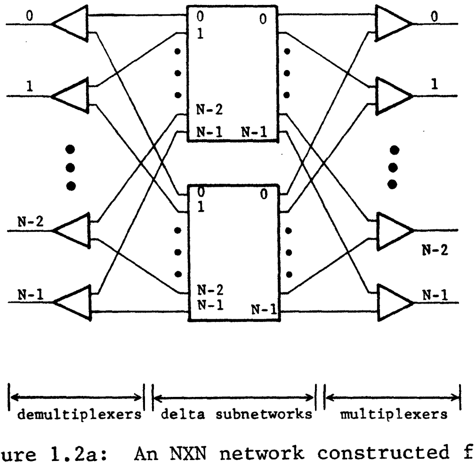


Figure 1.2a: An NXN network constructed from two NXN delta subnetworks

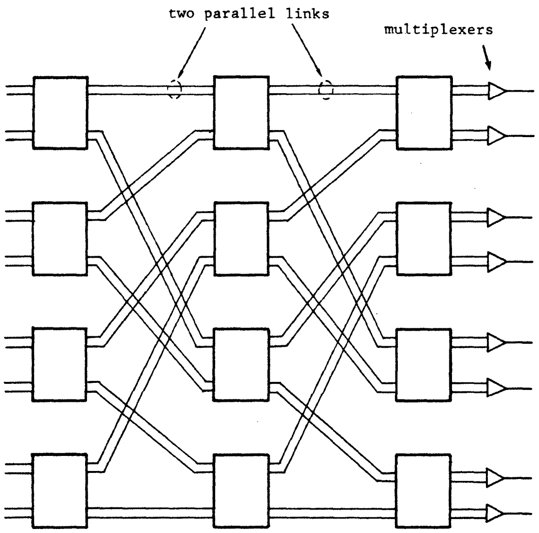


Figure 1.2b : An 8X8 delta network with double links (D_2 network)

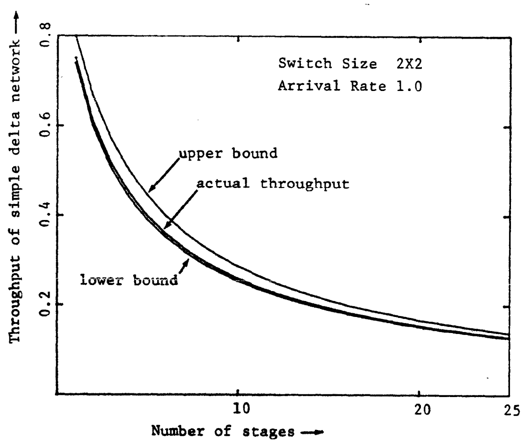


Figure 2.1a

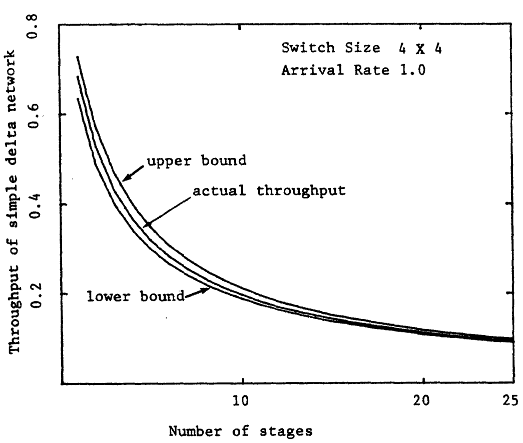


Figure 2.1b

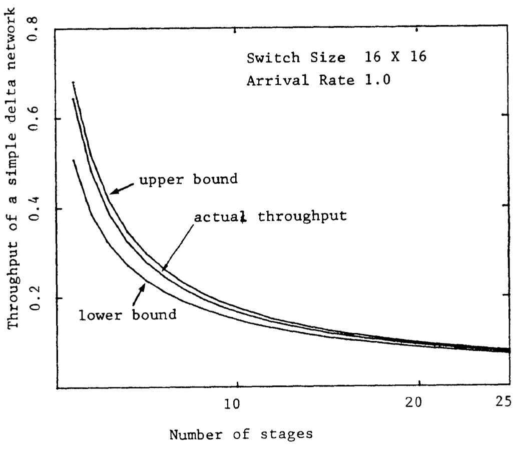


Figure 2.1c

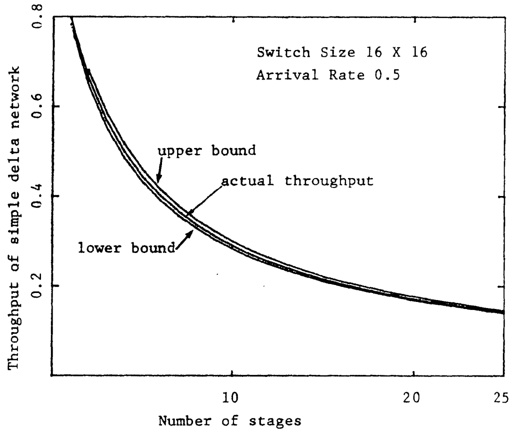


Figure 2.1d

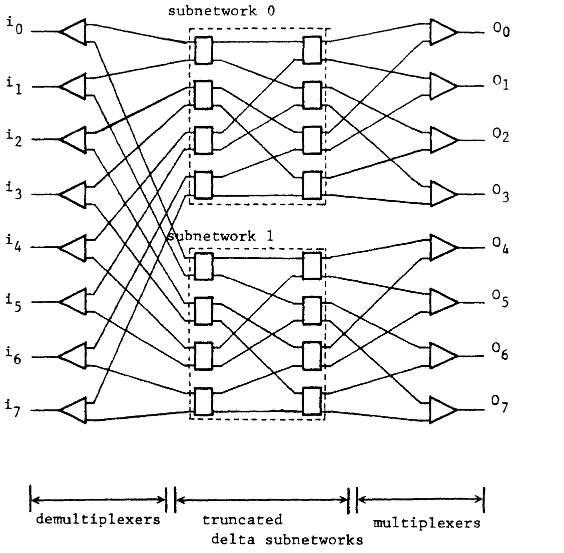


Figure 3.1 : An 8X8 selectively loaded parallel delta network constructed from two 8X8 truncated delta subnetworks.

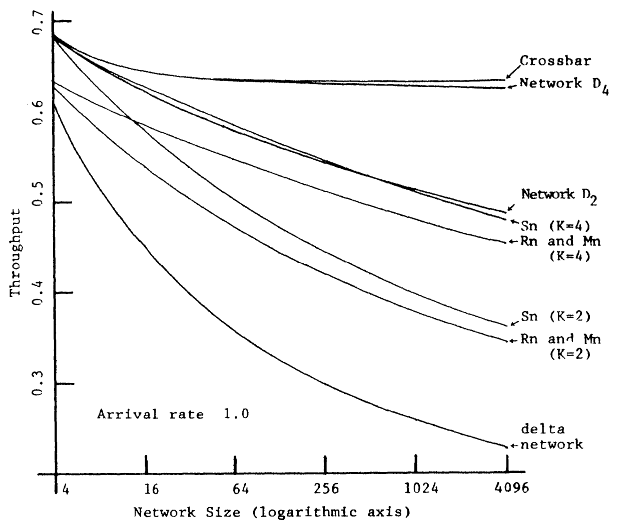


Figure 3.2a

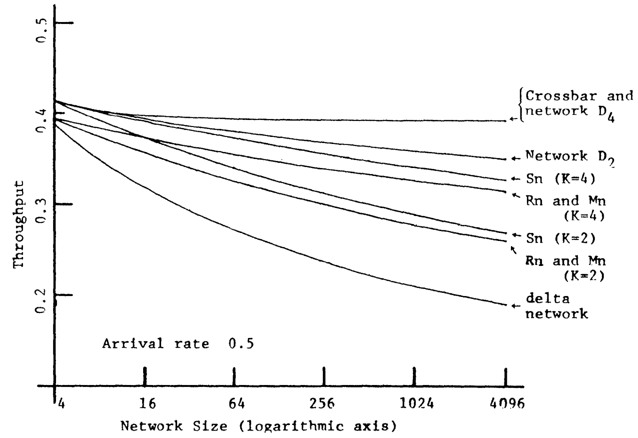


Figure 3.2b

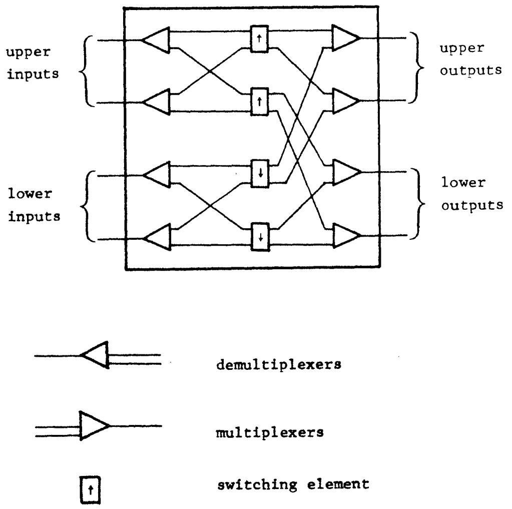


Figure 4.1a : A 2X2 crossbar switch for D_2

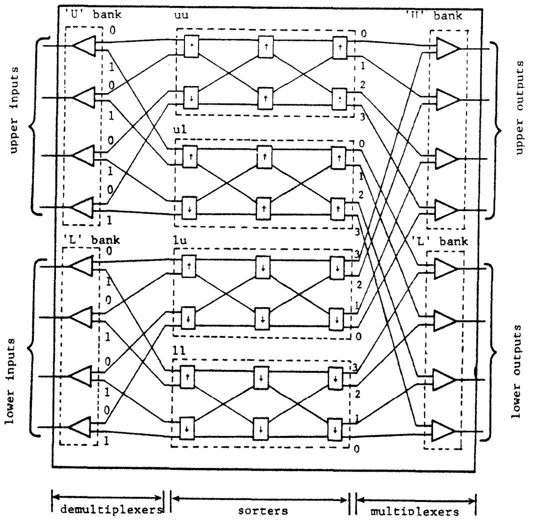


Figure 4.1b : A 2X2 crossbar switch for D_4

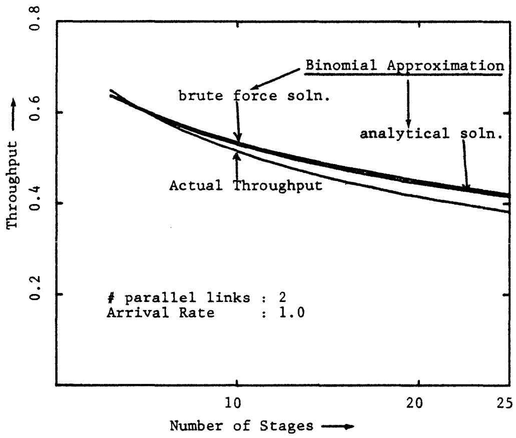


Figure 4.2a

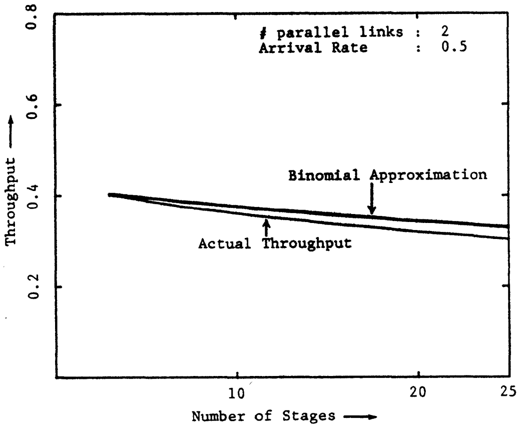


Figure 4.2b

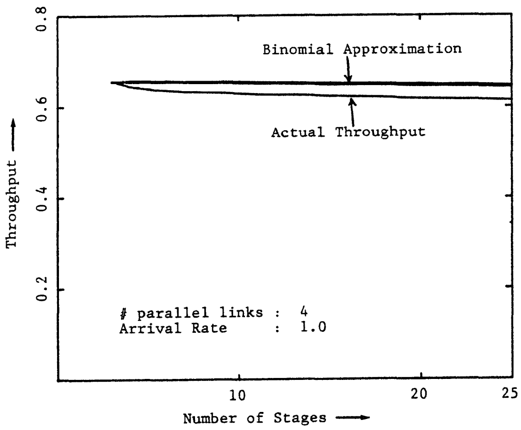


Figure 4.2c

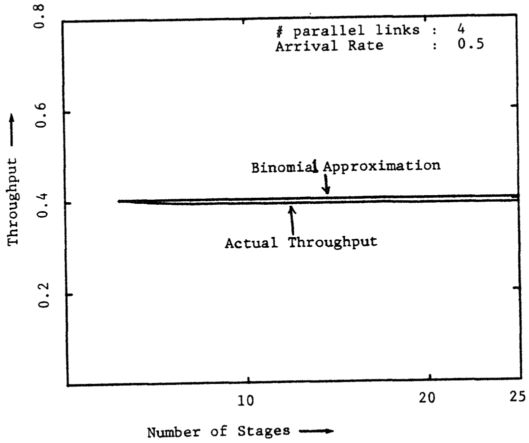


Figure 4.2d

EXPANDING AND CONTRACTING SW-BANYAN NETWORKS

Doug DeGroot
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

Abstract

SW-banyan networks are one of the most promising class of multistage interconnection networks. Their advantages include simplicity of control, partitionability, modularity, and expandability. Most SW-banyan interconnection networks that have been studied have been strongly rectangular nxn networks, constantly growing (expansion) networks, or constantly shrinking (concentration) networks. A class of SW-banyans called expanding and contracting SW-banyans is formally defined. These networks are shown to offer certain significant performance benefits over other multistage SW-Banyans. Because they require more hardware, they are more costly than certain other rectangular SW-banyans. However, they offer significant performance advantages, and thus may be suited for high-performance environments. They retain the advantage of n log n cost functions. More importantly, they retain the "unique-path" property.

INTRODUCTION

It is clear that interconnection networks comprise a major architectural component of large, highly parallel computers. Although many networks have been studied for their suitability to this environment [Siegel, Lawrie], one type receiving much current attention is the "unique-path" multistage network. Given any two resources connected to opposite sides of the network, there is always one and only one path through the network between them (thus the name "unique-path"). The number of stages in these unique-path multistage networks is usually O(log n), with n switches (crosspoints) per stage, yielding a cost function of O(n log n) (as opposed to $O(n^2)$ for crossbars). The Omega network [Lawrie2] is perhaps the best known example of a "unique-path" multistage network. Unique-path networks are in general much easier to control than multiple-path networks.

Unfortunately, most of these unique-path networks also contain only partial interconnectivity capabilities. Thus, given any two resources to be interconnected, if other interconnections presently exist, the two resources may not be able to be interconnected due to blockage by the other already active interconnections. This absence of full interconnectivity can cause serious degradation in system performance if interconnection blockages cannot be held to a minimum by either the operating system or the user application. Clearly, the less blockage inherent in the network the better. One large class of blocking, unique-path networks is the class of SW-banyan networks. It is shown below that expanding and contracting multistage SW-banyans have less blockage inherent in them than other types of multistage SW-banyans. But what is just as important, they are also shown to possess the unique-path property.

BANYAN INTERCONNECTION NETWORKS

In this section a very large, general class of multistage interconnection networks called banyans is described [Lipovski]. SW-banyans, a proper subset of banyans, are a particularly attractive, cost-effective, modular type of banyan that can be recursively synthesized from smaller SW-banyans. A number of single-valued functions are defined which describe certain structural and topological features of SW-banyans.

Banyans

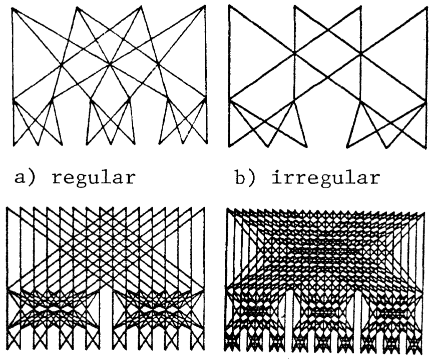
Banyan networks, named for an East Indian and Hawaiian fig tree, are defined in terms of their graph representation. A banyan (or banyan graph) is a Hasse diagram of a partial ordering in which there is one and only one path from every base to every apex. A base is defined as any vertex with no edges incident into it; an apex is any vertex with no edges incident out of it; all other vertices are called intermediate vertices. Some examples of banyans are shown in Figure 1. In these diagrams, bases are at the bottom and apexes are at the top. In the following illustrations, the banyans will be drawn as undirected graphs.

An L-level banyan is a banyan in which every base-apex path is of length L. In an L-level banyan, there are L levels (stages) of arcs but L+1 levels of nodes. By convention, all following banyan illustrations will be numbered baseward, with apexes at level 0 and bases at level L. In an L-level banyan, arcs exist only between vertices in adjacent levels.

In a banyan, the spread of a vertex is the number of edges incident out of the vertex; the fanout of a vertex is the number of edges incident into the vertex. If every node level of a banyan is such that all vertices within the same level have identical spread and fanout values, the banyan is called uniform; otherwise it is called non-uniform. In a uniform banyan, the fanout values of the vertices may be characterized by an L-component vector $\underline{F}$, called the fanout vector. Similarly, the spread values are characterized by an L-component spread vector $\underline{S}$. A rectangular banyan is a banyan for which $\underline{F} = \underline{S}$. It is shown below that a rectangular banyan has the same number of vertices at each level. Non-rectangular banyans have $\underline{F} \neq \underline{S}$.

0190-3918/83/0000/0019$01.00 © 1983 IEEE

If every component of the fanout vector $\underline{F}$ is equal to some constant f and every component of the spread vector is equal to some constant s, the corresponding banyan is called regular; otherwise it is called irregular. When f = s, by necessity $\underline{F}$ = $\underline{S}$, and the corresponding banyan is both regular and rectangular; such banyans are called strongly rectangular. An irregular rectangular banyan is called weakly rectangular. Figure 1 illustrates these concepts. From the above definitions it can be seen that a crossbar is a one-level regular banyan.


a) regular b) irregular

c) weakly rect. d) strongly rect.

Figure 1

SW-banyans

SW-banyans are a particularly interesting proper subset of L-level banyans. They are especially suitable for partitioning and connection networks [Goke, DeGroot]. SW-banyans can be axiomatically defined [Premkumar]. Recall that a banyan is a Hasse diagram of a partial ordering in which there is one and only one path from every base to every apex. The level x reachability set for any base b, $0 \leq x \leq L$, is defined as the set of all nodes in level x that can be reached by directed paths from base b. Similarly, the level x reachability set for any apex a, $0 \leq x \leq L$, is defined as the set of all nodes in level x that can be reached by paths from apex a. A banyan is an SW-banyan if and only if for any two bases b and c, or any two apexes d and e, their level x reachability sets are either disjoint or identical, for $0 \leq x \leq L$.

Constructing SW-Banyans

In addition to the preceding axiomatic definition, a novel constructive definition of SW-banyans will now be given. This definition applies only to uniform L-level SW-banyans, but it can easily be extended to include non-uniform, non-L-level SW-banyans. It differs considerably from previous constructive definitions for SW-banyans and SW-banyan networks [Goke, Goke2, DeGroot]. It can be used to generate regular and irregular SW-banyans, and non-rectangular, strongly rectangular, and weakly rectangular SW-banyans, as well as the expanding and contracting SW-banyans presented here. A number of single-valued functions can be associated with this definition, and they are given below. These functions define and describe certain structural and topological features of SW-banyans.

A uniform one-level SW-banyan is simply a crossbar. A two-level uniform SW-banyan can be constructed as follows. Consider any mxn crossbar, and select t of them. Choose any integer k > 0, and construct another SW-banyan as follows. Take the first (leftmost) apex of each of the t crossbars and connect them to k new nodes. Take the second apex of each crossbar and connect each of these apexes to k other new nodes. Continue until all apexes have been connected to groups of k new nodes. Figure 2 illustrates this procedure. The resulting 2-level network is (km)x(tn). Clearly each of the km new nodes has one and only path to each of the t crossbars (see Figure 2). Further, each crossbar apex has one and only path to each crossbar base. Therefore, each of the km new nodes has one and only one path to each of the tn bases, and thus the constructed network is a banyan. It is easy to see that the constructed two-level banyan satisfies the axiomatic definition given above, and thus the banyan is also an SW-banyan. Now consider any mxn (L-1)-level uniform SW-banyan. Choose some t of these SW-banyans and some k > 0 as before. An L-level SW-banyan can be constructed using the same procedure as above. Because each (L-1)-level SW-banyan has one and only one path between each apex and base, so will the constructed L-level SW-banyan. This procedure can be recursively applied to construct an SW-banyan with any number of levels. There will always be one and only one path between every base and every apex in the constructed SW-banyan, and it can be shown that the axiomatic definition will always hold. Figure 3 illustrates this process.

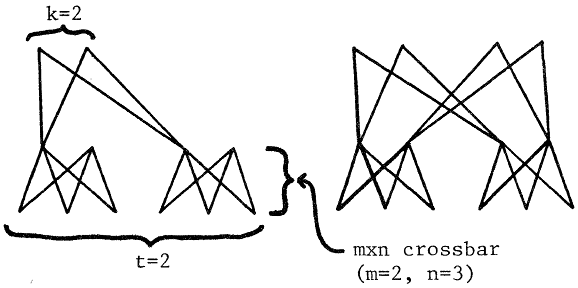


Figure 2

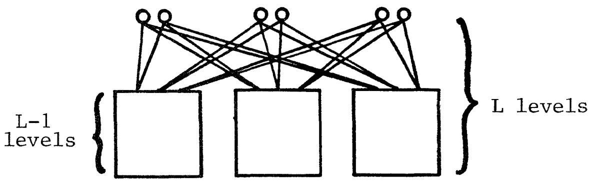


Figure 3

SW-banyan Topological Features

In a uniform SW-banyan, all vertices within a given level have identical fanout values and identical spread values. The fanouts and spreads of a uniform SW-banyan are represented by the L-component vectors $\underline{F}$ and $\underline{S}$. For example, the 3-level SW-banyan in Figure 1.c has $\underline{F} = (2,3)$ and $\underline{S} = (2,2)$. The fanout of every node at level i is denoted $f(i)$ for $0 \le i \le L-1$; the spreads of these nodes are denoted $s(i)$ for $1 \le i \le L$. In other words, $f(i)$ is the i+1'st component of $\underline{F}$, and $s(i)$ is the i'th component of $\underline{S}$. For convenience, we define both $s(0)$ and $f(L)$ to be 1. The number of nodes in any level x of an L-level SW-banyan is defined as $n(x)$. The number of apexes is $n(0)$; the number of bases is $n(L)$.

In this section, a number of topological features of SW-banyans are described. These features are used in the following sections to describe expanding and contracting SW-banyans. Because the proofs of the theorems are so easy and obvious, and because they have appeared elsewhere in the literature ([Goke, DeGroot]), they are omitted here.

Theorem 1:
For any level x in an SW-banyan, $0 \le x \le L-1$, $n(x+1) = n(x)f(x)/s(x+1)$.

Theorem 2:
For any uniform L-level SW-banyan, $n(x) < n(x+1)$ if and only if $f(x) > s(x+1)$. Furthermore, $n(x) > n(x+1)$ if and only if $f(x) < s(x+1)$. And $n(x) = n(x+1)$ if and only if $f(x) = s(x+1)$.

Theorem 3:
In a rectangular SW-banyan, $n(x)$ is constant and equal to B, the number of bases in the banyan, for $0 \le x \le L$.

Corollary 3.1:
If $n(x) = n(x+1)$ for all x, $0 \le x \le L-1$, then $\underline{F} = \underline{S}$.

Theorem 4:
Each base of an SW-banyan S reaches $s(x+1)s(x+2)\ldots s(L)$ nodes at level x, $0 \le x \le L-1$.

Corollary 4.1:
The number of apexes in a uniform SW-banyan is $s(0)s(1)\ldots s(L)$.

Corollary 4.2:
Each node at level x, $0 \le x \le L$, reaches $s(0)s(1)s(2)\ldots s(x)$ apexes.

Theorem 5:
Every apex in a uniform SW-banyan reaches $f(0)f(1)\ldots f(x-1)$ nodes at level x, $1 \le x \le L$.

Corollary 5.1:
The number of bases in a uniform SW-banyan is $f(0)f(1)\ldots f(L)$.

Corollary 5.2:
Each node at level x, $0 \le x \le L$, reaches $f(x)f(x+1)\ldots f(L)$ bases.

Theorem 6:
In a uniform SW-banyan, the number of apexes equals the numbers of bases if and only if $f(0)f(1)\ldots f(L-1) = s(1)s(2)\ldots s(L)$.

EXPANDING AND CONTRACTING SW-BANYANS

Most of the multistage interconnection networks that have been studied have been nxn multistage networks, that is, they have n inputs (apexes) and n outputs (bases). Furthermore, they have almost always been strongly rectangular nxn networks - they have had n switches in every stage of the network. In this section, a certain type of non-rectangular nxn network is presented. These networks are called "expanding and contracting" nxn SW-banyan networks. They are shown to have certain performance advantages over the prevalent rectangular nxn networks.

From Theorem 6 above, any nxn SW-banyan must have $f(0)f(1)\ldots f(L-1) = s(1)s(2)\ldots s(L) = n$. Furthermore, to be rectangular, (that is, to have n switches in every stage), it must be that $f(i) = s(i+1)$ for $0 \le i \le L-1$. It should be clear that we can take the $f(i)$ and permute them and still have their product equal to n. For instance, if $f(0)f(1)f(2) = n$, then clearly $f(1)f(0)f(2)$ also equals n. Therefore, given an nxn L-level SW-banyan with particular fan and spread vectors $\underline{F}$ and $\underline{S}$, another nxn SW-banyan can be derived by simply permuting the components of $\underline{F}$ or $\underline{S}$ (or both). However, unless $f(i)$ still equals $s(i+1)$, for $0 \le i \le L-1$, the nxn SW-banyan will no longer be rectangular (since $\underline{F}$ will no longer equal $\underline{S}$).

For an example, consider the 8x8 weakly rectangular SW-banyan in Figure 4a. It has $\underline{F}$ and $\underline{S}$ both equal to (2,4). Because $\underline{F} = \underline{S}$, $n(x)$ is constant and equal to 8 for $0 \le x \le 2$ (see Theorems 1 and 3). Now consider what happens when $\underline{F}$ is changed to the vector (4,2). Clearly now $\underline{F} \ne \underline{S}$, and so the resulting SW-banyan cannot be rectangular. However, we still have that $f(0)f(1) = s(1)s(2) = 8$. So the banyan is still 8x8 (see Corollaries 4.1 and 5.1). But because $f(0) < s(1)$, $n(0) < n(1)$. In fact, $n(0) = 8$, but $n(1) = 16$. (From Theorem 1, $n(1) = n(0)f(0)/s(1)$. And since $f(0)/s(1) = 4/2 = 2$, $n(1) = 2n(0)$.) In other words, there are twice as many nodes at level 1 than there are at level 0. The corresponding 8x8 SW-banyan is shown in Figure 4b. Since $f(1)/s(2) = 2/4 = 1/2$, there are half as many nodes at level 2 than there at level 1. So level 0 has 8 nodes (apexes), level 1 has 16 nodes, and

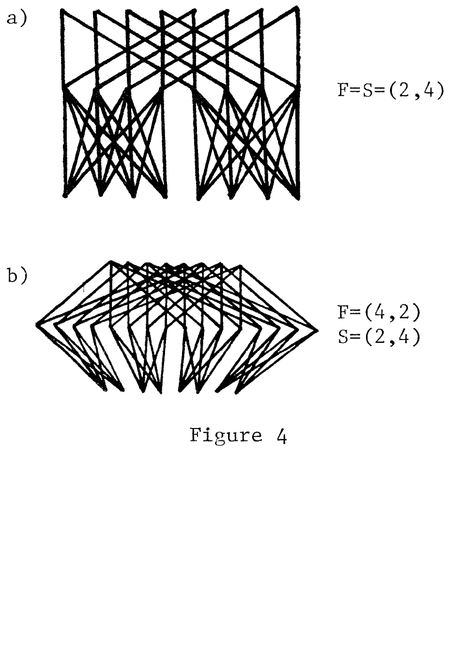


Figure 4

level 2 has 8 nodes (bases). This banyan expands outward from the top (apexes) and then contracts back toward the bottom (bases), somewhat like a diamond.

It is easy to see that it is also possible to construct expanding and contracting mxn SW-banyans in which $m \neq n$.

Preserving the Unique-path Property

It has seemed perplexing to many that a multistage network can have twice as many nodes in an inner level than in the apex or base levels and yet still retain the unique-path property, that is, that there can still be one and only one path between every apex and every base. Figure 4b provides visual proof of one example of this possibility. To see how the unique-path property is maintained, recall the constructive definition of L-level SW-banyans given in Section 2.3. A uniform two-level SW-banyan can be constructed by selecting t mxn crossbars and interconnecting them through km new nodes, for some $k > 0$. It is clear that doing so yields an SW-banyan with $\underline{F} = (t,n)$ and $\underline{S} = (k,m)$. If $t > k$, then there will be more level 1 nodes than there are level 0 nodes. If $n < m$, there will be fewer level 2 nodes than there are level 1 nodes. However, the recursive constructive definition assures us that the constructed network will in fact be an SW-banyan and will therefore possess the unique-path property. In this way, an expanding and contracting network can be constructed, and the unique-path property is maintained, as explained above. The above procedure can be recursively repeated to produce expanding and contracting SW-banyans of even greater numbers of levels, with the unique-path property being easily proven by induction as before.

A mathematical proof of the unique-path property of SW-banyans has been given in [Bhuyan].

Blocking Characteristics

Most unique-path multistage interconnection networks suffer from various types of blockage. In this section, a special type of blockage is considered. It is initially assumed that some sort of dedicated, non-interfering connections are to be used to interconnect apexes to bases, as in circuit switched connections, for example. This assumption is later relaxed.

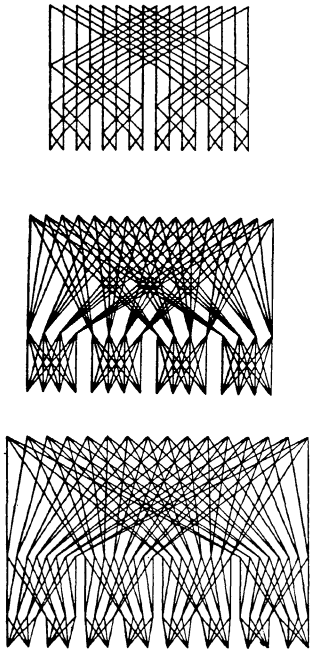
When one apex is connected to a base by means of a dedicated communication path, no other new apex-base connection can be made if that new connection requires a node or link in use by the first connection. The second connection is said to be "blocked." This is a direct consequence of the unique-path property. Only when the first connection is undone can the second be made. Certain networks are inherently more prone to blocking than others.

In this section, a function is defined which gives an indication of the amount of static blockage inherent in an L-level SW-banyan. This function allows different SW-banyans to be compared to each other. The function simply relates how many possible interconnections are rendered impossible (become blocked) by any single connection made on an empty network. Exactly how much run-time blockage this connection causes depends on many factors.

Consider any single interconnection. It uses one and only one node at each level of the SW-banyan. The base and apex being interconnected are obviously rendered unavailable; but because this would be true even in crossbars, their unavailability is not considered as blockage here. Consider the node in use at level x however, $1 \le x \le L-1$. This node can be reached by $s(1)s(2)\ldots s(x)$ apexes. Furthermore, this node can reach $f(x)f(x+1)\ldots f(L-1)$ bases. But one of these apexes and one of these bases are the apex and the base in the given interconnection, so they are not considered as being able to be blocked (they are already in use). For any x, $0 \le x \le L$, define a(x), the number of apexes reachable by a node at level x, to be $s(0)s(1)\ldots s(x)$ (see Corollary 4.2). Define b(x), the number of bases reachable by a node at level x, to be $f(x)f(x+1)\ldots f(L)$ (see Corollary 5.2). Then bp(x), the number of blocked paths that pass through the busy node at level x, is simply $(a(x)-1)(b(x)-1)$, for $1 \le x \le L-1$. We define both bp(0) and bp(L) to be zero. The sum of all bp(x), for $1 \le x \le L-1$, however, does not yield the total number of blocked paths generated by a single active path, since many blocked paths would get counted more than once with this sum. To avoid the multiple counting, we define the function bl(x) as $[a(x)-a(x-1)](b(x)-1)$, for $1 \le x \le L-1$. This equation counts the number of additional blocked paths that are encountered as a communication path is followed from an apex down to a base. The total blockage created by a single connection is then correctly given by the sum of all bl(x), for $1 \le x \le L-1$.

For an example, consider several possibilities of a 16x16 SW-banyan. The most popular such networks are strongly rectangular ones in which either f=s=2 or f=s=4. An expanding and contracting SW-banyan in which $\underline{F} = (4,2,2)$ and $\underline{S} = (2,2,4)$ is also considered. These networks are illustrated in Figure 5. The values for a(x), b(x), and bl(x) are shown for all three. Summing the bl(x)'s, it can be seen that the f=s=2 SW-banyan incurs a total blockage of 17 blocked interconnections for each connection made. The f=s=4 SW-banyan incurs a total blockage of only 9, or almost half as few. But the expanding and contracting SW-banyan incurs a total of only 5. From these figures, it would seem that the expanding and contracting SW-banyan is the best performer of the three, followed by the f=s=4 and then the f=s=2. These results are consistent with recent studies [DeGroot, Malek, McMillen, Bhuyan].

For another example, consider a 64x64 SW-banyan. Using 4x2 and 2x4 nodes, a 64x64 SW-banyan can be built with $\underline{F} = (4,4,2,2)$ and $\underline{S} = (2,2,4,4)$, as shown in Figure 6. Level 0, the apex level, has 64 nodes, level 1 has 128, level 2 has 256, level 3 has 64, and level 4, the base level,

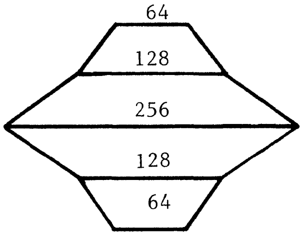


lvl	a(x)	b(x)	bl(x)
0	1	16	0
1	2	8	7
2	4	4	6
3	8	2	4
4	16	1	0

lvl	a(x)	b(x)	bl(x)
0	1	16	0
1	4	4	9
2	16	1	0

lvl	a(x)	b(x)	bl(x)
0	1	16	0
1	2	4	3
2	4	2	2
3	16	1	0

Figure 5



F=(4,4,2,2)
S=(2,2,4,4)

Figure 6

has 64. For each connection made, this 64x64 SW-banyan suffers only 33 blocked interconnections. The standard f=s=4 SW-banyan suffers 81, and the f=s=2 SW-banyan suffers 129. Clearly the expanding and contracting SW-banyan offers significant performance gains over other SW-banyans.

It should be noticed that the increased performance of an expanding and contracting SW-banyan does not come without cost. First, the number of stages in an expanding and contracting SW-banyan may be more than in other networks, leading to increased communication delays. In addition, expanding and contracting SW-banyans may easily require more network switches and wires. However, importantly, the total cost of expanding and contracting SW-banyans still grows at the rate of only n log n.

Bandwidth Characteristics

The above analysis considered all interconnections to be dedicated, non-interfering connections, as in circuit switching. This section considers the analysis of expanding and contracting SW-banyans in a packet switching environment. Bandwidth is defined to be the expected number of memory requests accepted per cycle. To calculate the bandwidth, it is necessary to calculate and sum the probabilities of an output occurring at each base (assuming apexes are the inputs). Bhuyan and Agrawal provide a simple recursive function for doing so [Bhuyan]. The probability of output of a node at network level i is simply

$$p(i) = 1 - (1 - p(i-1)/s(x))^{f(x)}$$

We assume here that p(0)=1 (see [Bhuyan]. for other relevant assumptions. The bandwidth of an SW-banyan is then simply n(L)p(L). For the 16x16 SW-banyan then, the f=s=2 version has a bandwidth of 7.2, the f=s=4 version has a bandwidth of 8.44, but the expanding and contracting SW-banyan with F = (4,2,2) and S = (2,2,4) has a bandwidth of 9.28. Clearly the expanding and contracting SW-banyan is the better performer in packet switching environments. Similar results are obtained for the 64x64 example. It should be easy to prove that this will always be the case for expanding and contracting SW-banyans.

OTHER TOPOLOGICAL POSSIBILITIES

It should be clear from Section 3.1 that with the proper choice of t and k at each recursive step of the construction of an SW-banyan that arbitrary topologies can be achieved. Figure 7 illustrates some of the many possibilities. Each has the unique-path property. What the advantages of any of these topologies are, if any, remains to be investigated.

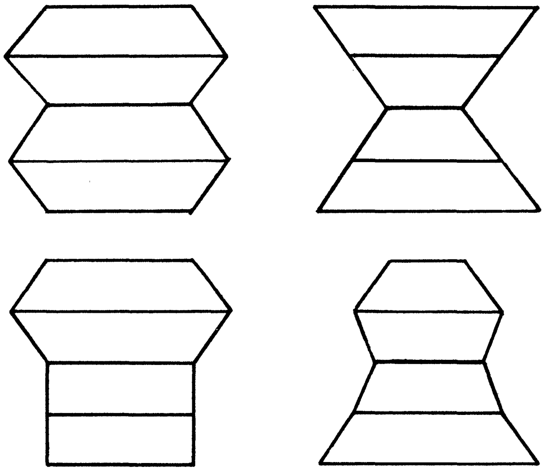


Figure 7

CONCLUSIONS

It has been shown that nxn multistage SW-banyans can be constructed with more than n nodes in an internal level. This was not generally believed to be possible. Because they are SW-banyans, these networks have the unique-path property, that is, there is one and only one path between every apex and every base. Traditionally, nxn SW-banyans are constructed as strongly-rectangular SW-banyans. It has been shown here that the expanding and contracting SW-banyans possess significantly less inherent blockage than the corresponding rectangular SW-banyans. Such networks can be built from only two or three different types of switches. Although they require more switches and may result in more network stages than some strongly rectangular SW-banyans, they retain the advantages of a cost function of only O(n log n). As a consequence, it seems such networks may prove to be more suitable for interconnecting large numbers of system resources than the prevalent rectangular networks, especially when high performance is a major concern.

Acknowledgements

Thanks to Dr. Jack Lipovski for his help with this work.

Bibliography

[Bhuyan] "Design and Performance of a General Class of Interconnection Networks," Bhuyan, Laxmi N. and Agrawal, Dharma P., Proc. 1982 International Conference on Parallel Processing, Aug. 1982, pp. 2-9.

[DeGroot] Mapping Computation Structures Onto SW-banyan Networks, Doctoral Dissertation, Department of Computer Sciences, The University of Texas, Austin, 1981.

[Goke] Banyan Networks for Partitioning Multiprocessor Systems, Goke, Rodney L., Doctoral Dissertation, Univ. of Florida, 1976.

[Goke2] "Banyan Networks for Partitioning Multiprocessor Systems," Goke, Rodney L. and G. Jack Lipovski, First Annual Symp. On Comp. Arch., Dec. 1973, pp. 21-28.

[Lawrie] "Bibliography: SIMD/MIMD Computer Interconnection Networks," Lawrie, Duncan H., Distributed Processing Technical Committee Newsletter, Vol. 3, No. 2, IEEE, June 1981, pp. 6-12.

[Lawrie2] "Access and Alignment of Data in an Array Processor," Lawrie, Duncan H., Trans. on Computers, IEEE, Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.

[Lipovski] "A Theory for Multicomputer Interconnection Networks," Lipovski, G. Jack, and Malek, Miroslaw, accepted for publication in Trans. on Computers, IEEE.

[Malek] A 4x4 Modular Crossbar Design for the Multistage Interconnection Networks, Malek, Miroslaw, DeGroot, Doug, Hung, A.C., and Juang, Ming-Shing, Dept. of Computer Sciences and the Dept. of Elec. Eng., The University of Texas, Austin, Texas, May 31, 1981.

[McMillen] "Performance and Impelmentation of 4x4 Switching Nodes in an Interconnection Network for PASM," McMillen, Robert J., Adams, George B. III, and Siegel, Howard J., Proceedings of 1982 International Conference on Parallel Processing, Aug. 1981, pp. 229-233.

[Premkumar] A Theoretical Basis for the Analysis and Partitioning of Regular SW-banyans, Doctoral Dissertation, Dept. of Elec. Eng., The University of Texas, Austin, Texas, 1981.

[Siegel] "A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems," Siegel, Howard J., Mueller, Phillip T., Jr., AFIPS Conf. Proc. 1979 National Computer Conference, Vol. 48, June 1979, pp. 529-542.

A COMPARISON OF CIRCUIT SWITCHING AND PACKET SWITCHING FOR DATA TRANSFER IN TWO SIMPLE IMAGE PROCESSING ALGORITHMS

by
Mehrad Yasrebi
Communication Products Division
IBM Corporation
Research Triangle Park, NC
and
Sanjay Deshpande and J.C. Browne
Department of Computer Sciences
The University of Texas at Austin

Abstract

The communication costs for parallel versions of two simple algorithms used in image processing are compared in packet switching and circuit switching formulations. The two algorithms are smoothing and histogramming. The histogramming algorithm, the recursive doubling algorithm of Stone, is studied over a range of processor numbers and pixel intensity resolution. The packet and circuit switching properties of the interconnection networks of the multiprocessor systems are based on two network architectured multiprocessors which are well-documented in the literature, PASM and TRAC. Communication based upon circuit switching generally gives a somewhat lower communication cost with the advantage increasing with pixel intensity resolution. The results of the analysis suggest a high utility value for including both circuit switching and packet switching functionality in the networks of network architectured multiprocessor systems.

Introduction and Overview

This paper compares the communication costs for executing two algorithms used in image processing on three parallel computer architectures. The purpose of the comparison is to evaluate packet switching and circuit switching modes of data movement for interconnection network based multiprocessors. The two algorithms used for the comparison are computation of histograms of the intensity values of pixels of an image and smoothing of gray level data across the pixels of an image.

The model for a packet switching architecture is the Partitionable SIMD/MIMD (PASM) System for Image Processing and Pattern Recognition [Siegel81]. The model for a circuit switching architecture is the Texas Reconfigurable Array Computer (TRAC) [Sejnowski80]. The third architecture, all processors sharing a common bus [Bhuyan82], is given as a baseline for the comparison. An analysis of communication costs for the two algorithms executing on PASM has been given in [Siegel81]. The results of an analysis of the execution of the two algorithms in a circuit switching formulation based on TRAC are given here. Space limitations preclude detailing of the analysis.

Communication Analysis for Parallel Algorithms

The major factors determining communication cost for the execution of parallel algorithms on interconnection network (ICN) based multiprocessors include: (i) the topology of the ICN and the configuration of resources on the ICN, (ii) the mapping of the data movement requirements of the algorithm upon the ICN, (iii) choice of switching methodology, (iv) the latency and bandwidth properties of the ICN, and (v) the unit sizes and the total volume of the data to be moved. This paper focuses on the impact of switching methodology and data volume on communication cost.

The choice of packet switching or circuit switching as the mode of network data path establishment can have a substantial effect on each of these architectural parameters. Packet switching tends to give flexibility in topology but fixed unit transfer sizes. Circuit switching tends toward less flexibility in topology, greater flexibility in unit size for transfers, but a longer transfer latency time. Packet switching may also introduce bandwidth degradation due to path contention while circuit switching may introduce path blockages which limit realizable network topologies for all networks short of full cross-bars.

The measure of communication cost is elapsed time. The communication times given herein are reported as number of memory cycles. We assume, in order to normalize computation costs across architectures, that an integer addition takes a single memory cycle and that updating a histogram vector element requires two integer additions. The speed-up of a multiprocessor over a uniprocessor is the ratio of total execution times, T_E, where $T_E = T_{COMM} + T_{COMP}$. All LOG's in this paper are in base 2 unless otherwise noted. The data paths of each ICN are taken to be one integer word in width. For the multistage ICN's of PASM and TRAC it is assumed that a unit of data moves through one stage of the ICN on each memory cycle.

Definition of Architectures

Communication cost for execution of the two algorithms is compared for three ICN-based multiprocessor architectures. The single shared bus architecture (Figure 1) has been characterized by Bhuyan and Agrawal [Bhuyan82]. It is a baseline for ICN-based multiprocessors. There is no distinction between packet and circuit switching in this model of communication. The model for a packet switching data movement architecture is PASM [Siegel81]. The ICN of PASM connects complete processing elements as shown in Figure 2.

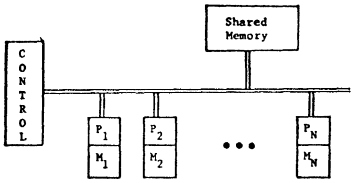


Fig. 1. A Multiprocessor with a Shared Bus

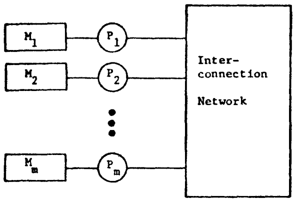


Fig. 2. PE-to-PE Configuration

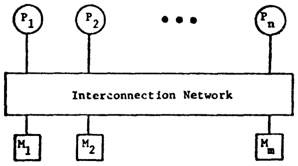


Fig. 3. Processor-to-Memory Configuration

0190-3918/83/0000/0025$01.00 © 1983 IEEE

The interconnection networks proposed for PASM are the generalized cube and the augmented data manipulator (ADM) [Siegel79]. These two networks are optimal for histogramming in the sense that all permutations for the algorithm can be realized by both networks in a single pass. Thus packet transfers can take place without blocking.

The model for a circuit switching data movement architecture is TRAC [Sejnowski80]. TRAC places processors at the apex nodes and memories at the base nodes of its ICN (Figure 3). The ICN of TRAC is an SW-Banyan [Premkumar80] with nodes having spread of two and fanout of three for its ICN. Processor configurations are formed by establishing circuits in the ICN joining processors to memory units. Data flow between processors for different stages of the algorithms can be realized by dynamically switching memories between processor-memory configurations. This network also implements trees of circuits joining one memory to many processors in which any one circuit may be activated and/or deactivated by a single processor instruction. These tree circuits are called shared or switchable memory trees. Data flow between processors may be implemented using this capability by a sequence of circuit activations and deactivations (among the circuits following the tree).

The ICN of TRAC actually implements both circuit switching and packet switching but only the circuit switching mode of use is modeled in the equations given following.

The Algorithms and Their Mapping to the Architecture

Histogramming and smoothing are among the basic operations of image processing, although not usually rate determining steps in the computations. Attention to detailed parallel formulations of major computational steps of image processing such as thresholding and edge detection is needed. It is assumed in the analysis following that the picture is M*M pixels in size ($M=2^m$) and that N ($N=2^n$) processors are available. The resolution of each pixel is λ bits.

Histogramming Algorithm

The parallel algorithm for histogramming is the recursive doubling algorithm of Stone [Stone75]. The structure of the algorithm is shown in Figure 4 for N=8. N partial histograms are computed in parallel at level 0. Each partial histogram is a vector of length 2^λ. The partial histograms are then added in pairs in parallel for LOG(N) stages to complete the algorithm.

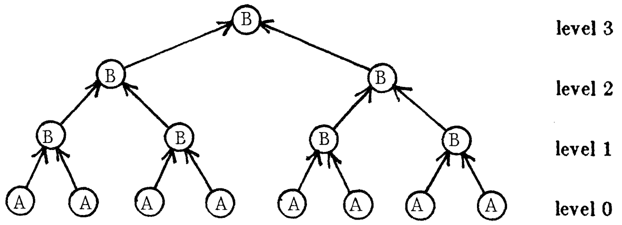


Figure 4: Recursive Doubling Algorithm for Histogramming

Partial histograms are shown at level 0 by A's and vector additions by B's. $N/2^i$ transfers of vectors are done between level (i-1) and level i. The computation time, T_{COMP}, for this algorithm under the assumptions made here is proportional to $T_{COMP} = M^2/N + 2^\lambda LOG(N)$.

A Packet Switching Formulation of Recursive Doubling Histogramming - Siegel *et al* [Siegel81] have given a thorough analysis for the execution of this algorithm on PASM. We adopt the results of this study as our packet switching model of recursive doubling histogramming. It is commonly the case that further steps in the analysis of the image require thresholding so that the final histogram vector must be collected in one processor and the threshold value distributed. The total communication time for this formulation of the algorithm is

$$T_{COMM} = [\underbrace{(LOG(N) + 2^\lambda)}_{\text{travel time through the ICN}} + \underbrace{2}_{\text{switch setting time}}] \text{ x } \underbrace{LOG(N)}_{\text{number of levels in the ICN}}.$$

A Circuit Switch Formulation of Recursive Doubling Histogramming Based on Tree Circuits - Figure 5 illustrates the structure of the circuit switched data movement formulation of recursive doubling for an 8 processor-8 memory configuration.

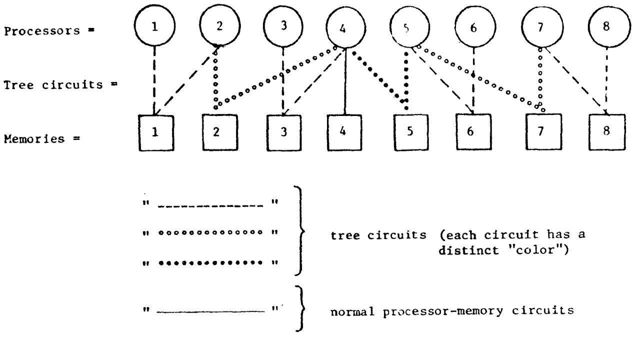


Figure 5: Circuit Switching Using the Tree Circuit Formulation

The M^2 pixels are evenly partitioned among the 8 memories. Each processor computes a partial histogram vector and stores it in the corresponding memory. The computation is then completed in LOG(8)=3 stages of adding partial vectors with the full histogram computed by processor 3 and stored in memory 5. The tree circuits of Figure 5 implement the successive communication paths between levels in Figure 5. The "----" tree circuits implement the data flow between levels 0 and 1 in Figure 4, "ooooo" the data flow between levels 1 and 2 and "....." the data flow between levels 2 and 3. There is a regular pattern of using first the verticals and then the diagonals of each type of tree circuit. Each tree circuit type has a unique number (called COLOR in correspondence with graph theory). LOG(N) colors are required to implement the algorithm in this formulation. Path selection (activation and deactivation) in all tree circuits of identical COLOR can be done in parallel with a latency time proportional to LOG(N)/2. The ICN of TRAC can implement the tree circuit pattern of Figure 5 without blockage. The total communication cost for this formulation of recursive doubling histogramming is

$$\begin{aligned} T_{COMM} &= [LOG(N)]^{-1} \sum_{i=1}^{LOG(N)} N/2^i\,[\underbrace{LOG(N)}_{\text{time to switch all memory with identical COLOR}} + \underbrace{\frac{LOG(N)}{2}}_{\text{latency time}}] \\ &= (3/2)(N-1) \end{aligned}$$

A Circuit Switching Formulation of Recursive Doubling Based on Direct Reconfiguration - Another formulation based on circuit switching can be developed by directly reconfiguring the ICN after each step (level in Figure 4) of the algorithm to conform to the data movement path required at each stage of the algorithm. Each configuration step involving establishment of a circuit between a given processor and a set of memories must be done serially. Thus use of the tree-circuit based algorithm is faster by a factor of LOG(K) where K is the number of COLORs available.

The Smoothing Algorithm

Smoothing is replacement of the intensity of each pixel by the mean of the intensity of the given pixel and its nearest neighbors.

Packet Switching Formulation of Smoothing - Siegel *et al* [Siegel81] have formulated and analyzed a packet data movement formulation of the smoothing algorithm. They show a speed-up of about .8N for a 1024 processor configuration. This estimate is extremely conservatively based. A greater speed-up is probable.

Circuit Switching Formulations of Smoothing - A circuit switching structure for the smoothing operation is suggested by the fact that each computation requires only nearest neighbors. Therefore if the pixels are stored by columns then a processor will need simultaneous access to three columns (say k-1,k,k+1) to execute the computations on column k. A realization of this representation of the smoothing algorithm is given in Figure 6. Extra zero valued rows and columns of pixel values are added to each formulation of boundary conditions. The solid lines of Figure 6 are normal circuits. The dotted lines are tree circuits from which leaf-root paths can be selected. Processor 1 computes in sequence the smoothed values for the pixels in columns 1, 2 and 3. Processor 2 will simultaneously and in sequence compute the smoothed values for the pixels in columns 4, 5 and 6. P1 and P2 must share access to pixel columns 3 and 4. The execution procedure described preceding allows this sharing to be accomplished without conflict if the required circuits can be established in the network. This two processor configuration obviously extends to an N processor 3N-memory configuration so long as the memory unit can hold an entire column of pixel values. A TRAC-like ICN can realize these configurations so long as these restrictions are met. It is also the case that the necessary data movement can be realized by reconfiguration of normal circuits. This is not the method of choice so long as the conditions for a tree circuit representation can be met.

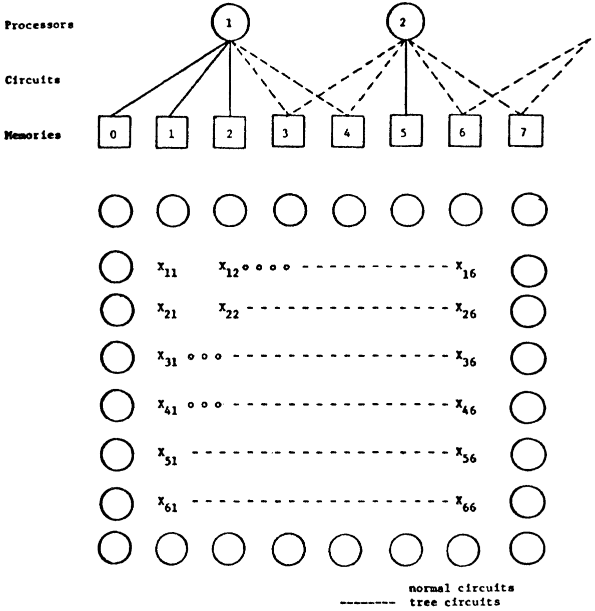

Processors
Circuits
Memories
1
2
0
1
2
3
4
5
6
7
x_{11} x_{12} o o o o ------------ x_{16}
x_{21} x_{22} ---------------- x_{26}
x_{31} o o o ---------------- x_{36}
x_{41} o o o ---------------- x_{46}
x_{51} ------------------ x_{56}
x_{61} ------------------ x_{66}
normal circuits
-------- tree circuits

Figure 6 A Storage Structure and Circuit Configuration for Parallel Smoothing

It may be desired to use a degree of parallelism greater than M (N>M). Then the columns of pixels must be decomposed into vectors of length M/k where N=kM. In this case the establishment of circuits is dependent upon k and may not always be possible. A formulation using both circuit switching and packet routing capabilities for TRAC has been worked out. The pixels appearing at the boundaries created by partitioning of columns have their "nearest neighbors" sent to them by packet movement. This "mixed" communication mechanism is still of lower cost than a pure packet based mechanism. The case N<M (for $N=2^i$, $m=2^j$) is handled by assigning multiple (2^k) columns to processors. This case raises no new problems.

We give here numbers only for the circuit switching representation where N=M and data movement is via tree circuit activation and deactivation. Then the total communication cost is (N/2) LOG(N) (assuming deactivation and activation of all tree circuit paths is done in parallel). If N=M=512, then only 256*9=2304 memory cycles are required for data communication. This is negligible compared to the C*512*512 arithmetic operations on the pixels ($C \geq 10$ and probably $C > 10^2$) since indexing must be accomplished as well as the addition and division of smoothing itself.

We thus conclude that for smoothing data movement costs will be essentially trivial for both packet and circuit switching representations.

Speed-up Analysis and Discussion

Figure 7 shows the net speed-up versus the number of processors for M=1024 and λ=8. There is, in this case, little difference between formulations based on different switching strategies for moderate numbers of processors. There is the suggestion that circuit switching will yield superior performance for large numbers of processors.

Figure 8 shows the speed-up factor as a function of λ for M=1024, and N=256. The amount of data transferred grows exponentially as λ. Thus circuit switching data movement shows a strong advantage as λ increases since the cost of data movement in the circuit switching strategy given here is constant with respect to data volume until the capacity of a memory unit is exceeded.

Smoothing on the other hand shows advantage for packet switching since there are cases where a pure circuit switching formulation becomes rather complex.

The bottom line with respect to parallel histogramming is that circuit switching has an advantage resulting from flexibility in the unit size of transfers and in stability with respect to algorithm parameters but that well-designed architectures should give similar performance for small to moderate numbers of processors.

Circuit switching and packet switching are both extremely efficient for parallel smoothing. Packet switching has an advantage over circuit switching with respect to application of degrees of parallelism with N>M for parallel smoothing. This advantage arises from the greater flexibility in communication topology.

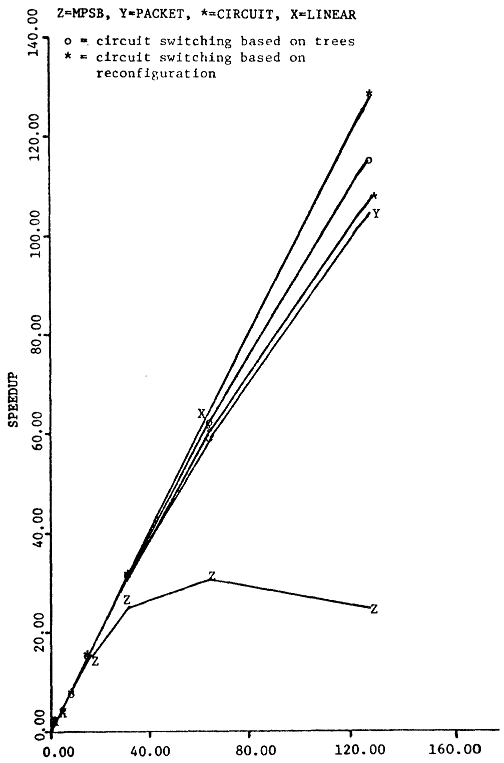


Figure 7 Speedups versus the Number of Processors (M=1024, λ=8)

There is suggestion from these two algorithms that implementation of both packet and circuit switching facilities in the ICN's for multiprocessors will give lower communication cost and greater net speedup than either used separately.

Acknowledgements

This work was sponsored by the Air Force Office of Scientific Research under Grant Number AFOSR-82-0091 and by the National Science Foundation under Grant Number MCS-8116099.

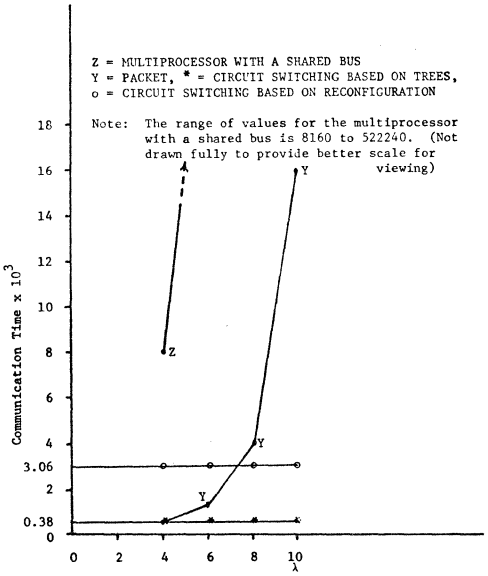


Fig. 8 Communication Time Versus λ
(N=256)

References

[Bhuyan82] Bhuyan, L.N. and Agrawal, D.P., "Applications of SIMD Computers in Signal Processing", AFIPS Conf. Proc. 51, pp. 135-142, 1982.)

[Feng81] Feng, Tse-yun, "A Survey of Interconnection Networks", Computer 14(2), December 1981.

[Premkumar79] Premkumar, U.V., et al, "Interprocessor Communication on the Texas Reconfigurable Array Computer", in 1st Int. Conf. on Distributed Computer Systems, 1979.

[Premkumar80] Premkumar, U.V., et al, "Design and Implementation of the Banyan Interconnection Network in TRAC", AFIPS Conf. Proc., May 1980.

[Sejnowski80] Sejnowski, M.C., et al, "An Overview of the Texas Reconfigurable Array Computer", NCC Conf. Proc., 1980.

[Siegel79] Siegel, H.J., "Interconnection Networks for SIMD Machines", Computer 12, June 1979.

[Siegel81] Siegel, H.J., et al, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition", IEEETC C-30(12), December 1981.

[Stone75] Stone, H., Introduction to Computer Architectures, Science Research Associates, Inc., 1975.

NUMERICAL EXPERIMENTS WITH THE MASSIVELY PARALLEL PROCESSOR[a]

E. J. Gallopoulos
and
S. D. McEwan

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Ill. 61801

Abstract -- The use of the Goodyear Massively Parallel Processor (MPP), an array of 16384 Processing Elements, is described for the solution of the shallow-water equations in a spherical geometry. These partial differential equations arise in Numerical Weather Prediction models and their fast solution is necessary. These are discretized with second order finite-differences on a latitude-longitude grid. Each physical grid point is mapped onto one MPP Processing Element. A set of difference equations results at each grid point, the same set at each grid point. This makes possible the use of a parallel algorithm for their solution at all grid points simultaneously. Only values from neighbourhood points are needed except for a few cases and thus routings between non adjacent Processing Elements are kept at a minimum. The resolution achieved with both available horizontal MPP dimensions is adequate and is comparable with fine resolution models currently in use. The exploitation of the MPP architecture is described and some of the problems facing the algorithm designer when confronting this novel computer architecture together with suggestions for handling them are indicated. Performance comparison estimates indicate that the MPP could achieve equal or better performance than more expensive supercomputers for such a problem. It is concluded that the MPP can competitively solve problems in the area of Numerical Weather Prediction.

Introduction

The Massively Parallel Processor system (MPP) [1] is a bit-serial SIMD computer designed and built as a collaborative effort between the NASA Goddard Space Flight Center and the Goodyear Aerospace Corporation, primarily, to support high-speed Image Processing. We intend to show here by means of a complete example how the MPP can be used very effectively to solve problems in computational physics, in particular the ***shallow-water*** equations occuring in numerical weather prediction. For a system like the MPP which was designed primarily for one particular area, namely Image Processing, we indicate how the available massive parallelism, if used carefully, can give excellent performance levels, comparable to, and for the example better than, current supercomputers. This application study shows that for some problems, the dimensionality constraints imposed by such an architecture are not a problem. The efficient use of the MPP in even a small set of real problems in this area would help the modellers in need of new and faster computational tools. This is the first such study done for the MPP. It is based on a parallel algorithm first set out by Kalnay and Takacs in [2]. It is interesting to note, as mentioned in [2], that the father of Numerical Weather Prediction L. F. Richardson, in his pioneering work [3] had envisaged a "human parallel computer" for performing weather prediction. More recently, references [4, 5, 6, 7] contain studies on the use of unconventional architectures to solve problems occuring in that area.

MPP Description

The Goodyear Massively Parallel Processor (MPP) is a bit-serial Single-Instruction Multiple-Data computer currently being built under NASA contract to support high-speed Image Processing. It consists of four main components [fig. 1]:

Array Unit (ARU)
Array Control Unit (ACU)
Program and Data Management Unit (PDMU)
Staging Memory (SM)

The ARU consists of a 128×128 array of bit-serial Processing Elements (PE) [fig. 2], each having a 1K bit RAM, extensible to 64K, giving an overall 2Mbyte ARU memory capacity. The interconnection network is of nearest-neighbour type with possible open, cylindrical, toroidal and spiral connections for the edges, all under user control. The ACU cycle time is 100ns. The ACU [fig. 3] executes the applications programs, scalar and array operations, and manages the I/O. It consists of the PE Control Unit (PCU), the I/O Control Unit (IOCU) and the Main Control Unit (MCU). These three modules operate in parallel and thus array and scalar arithmetic and I/O can be overlapped. Scalar data and application programs reside in MCU memory. The PCU memory contains the routines that operate on arrays of data in the ARU. Each PCU instruction is 64-bit wide allowing several PE elementary bit operations to be performed simultaneously. PCU routines are called from application programs residing in the MCU memory. A call-queue is provided to queue-up the calls from the MCU to the PCU, enabling the MCU to work concurrently. Since most of the MCU operations consist of subroutine calls or scalar operations that take much less time than the array operations, the PCU rarely waits for a new call to be issued by the MCU.

The S-registers on each PE can shift planes of data without interfering with the computations except when a bit-plane is to be written into or read from ARU memory. Hence only 1 cycle for reading or writing is stolen every 128 PE activity cycles (The time needed to bring a new 128×128 bit-plane in place in the S-registers.) The Staging Memory buffers data between the ARU and the secondary storage devices.

The Program Development Software consists of two assemblers, one each for the PCU and MCU, a System

[a] Research supported in part by NASA under contract NAS5-26405

0190-3918/83/0000/0029$01.00 © 1983 IEEE

Subroutine Library, a set of I/O Macros initiating I/O functions, a Control and Debug module and a Linker. Additionally a parallel version of Pascal, Parallel Pascal [8], will be available.

If assembly language is to be used in order to make the most efficient use of the bit-serial features of the MPP, then to make the task of large-scale programming feasible, a large number of utility routines must be available. These routines can be roughly classified as

1) array unit arithmetic (signed and unsigned integer, floating point),
2) scalar-array arithmetic (eg scalar by array multiplication),
3) scalar arithmetic,
4) array logical operations (all Boolean functions) and comparisons,
5) routing operations,
6) search operations,
7) reduction operations,
8) others.

This is only a rough classification. Nevertheless it gives a flavour of the type of utility routines that should be available to the MPP programmer. At a somewhat higher level, the standard mathematical functions (as in Fortran) operating on array arguments, together with matrix and vector manipulators, must all be available in the utility library. The existence of efficient routines at that level is imperative for maximizing the performance.

At the time of the experiment (Summer '82), the MPP was still under development. Therefore the experiments were done on an MPP simulator system [9]. This system coordinates the execution of programs to emulate the actions of the MPP system. The user can write his programs in the MCU and PCU assembly languages using any of the available library routines. The resulting modules are loaded in the system and then the program trace can be followed through the available Debugger. As such it provides an excellent development tool for routines developed for the MPP. Even when the MPP is available, it would be more convenient to use the simulator first for code debugging and testing. The simulator system records the number of MCU and PCU cycles used during the execution of the application programs. It is written in the C programming language and runs under the Unix [10] or VMS [11] operating systems.

Problem description

The problem under consideration is the solution of a simplified form of the Navier-Stokes equations suitable for weather prediction in meteorology. Our example here could well serve as a guide for modellers in the need of faster computer rates in other Fluid Mechanic areas.

The physical processes occuring in the atmosphere and as a result their mathematical formulations are non-linear. Even with the occasional simplifications, the arising equations cannot be solved analytically and require good numerical techniques.

To decide on the suitable simplifications to make to the full set of equations describing the important physical phenomena in the atmosphere one has to note that although what principally determines the long-term statistical properties of the atmosphere are the cumulative effects of heating and friction, these terms are locally small compared with the fluid-dynamical terms. Hence by concentrating on these latter equations and terms the modeller can draw important conclusions for the behaviour of the complete system. The model to be solved concentrates on these terms as a first step towards a complete model, similar to the one described in [12]. It has been found that the so called *shallow-water* or *barotropic* equations contain the essential numerical aspects of the large scale prediction equations [13] governing an incompressible, homogeneous and hydrostatic fluid. These are strictly two-dimensional and thus refer to phenomena at a single fluid layer. The spherical polar coordinates (λ,ϕ) where λ is the longitude and ϕ the latitude are chosen, as the most natural reference system for motions around the globe. At time t and at position (λ,ϕ) the dependent variables for the model are the height h of the free atmospheric surface under consideration equal to $h_T - h_B$, and the Eastward and Northward velocity components u and v respectively. The equations, broken down to λ and ϕ components, are written as follows:

$$\frac{\partial h}{\partial t} = (\frac{\partial h}{\partial t})_\lambda + (\frac{\partial h}{\partial t})_\phi \tag{1a}$$

$$\frac{\partial hu}{\partial t} = (\frac{\partial hu}{\partial t})_\lambda + (\frac{\partial hu}{\partial t})_\phi \tag{1b}$$

$$\frac{\partial hv}{\partial t} = (\frac{\partial hv}{\partial t})_\lambda + (\frac{\partial hv}{\partial t})_\phi \tag{1c}$$

where

$$(\frac{\partial h}{\partial t})_\lambda = -\frac{1}{a\cos\phi}\left[\frac{\partial hu}{\partial\lambda}\right]$$

$$(\frac{\partial hu}{\partial t})_\lambda = -\frac{1}{a\cos\phi}\left[\frac{\partial(hu)u}{\partial\lambda}\right] - \frac{gh}{a\cos\phi}\frac{\partial h_T}{\partial\lambda} + (f + \frac{u\tan\phi}{a})vh$$

$$(\frac{\partial hv}{\partial t})_\lambda = -\frac{1}{a\cos\phi}\left[\frac{\partial(hu)v}{\partial\lambda}\right]$$

and

$$(\frac{\partial h}{\partial t})_\phi = -\frac{1}{a\cos\phi}\left[\frac{\partial hv\cos\phi}{\partial\phi}\right]$$

$$(\frac{\partial hu}{\partial t})_\phi = -\frac{1}{a\cos\phi}\left[\frac{\partial(hv\cos\phi)u}{\partial\phi}\right]$$

$$(\frac{\partial hv}{\partial t})_\phi = -\frac{1}{a\cos\phi}\left[\frac{\partial(hv\cos\phi)v}{\partial\phi}\right] - \frac{gh}{a}\frac{\partial h_T}{\partial\phi} - (f + \frac{utan\,\phi}{a})uh$$

where f $= 2\Omega\sin\phi$ (with Ω the angular rotation frequency equal to 7.292×10^{-5} sec^{-1}) is the Coriolis parameter. The first equation comes from the law of mass conservation and the last two from the law of momentum conservation. They can be found in this form in [14]. For a model where bottom orography is not included, $h_B = H$, a constant, and thus the h_T variable above can be replaced by h. This is the *quasi-linear hyperbolic* system that must be solved, given suitable boundary and initial conditions. The equations are written in *flux form,* using the time derivatives of the height-velocity products as this gives better results when discretized. The characteristic speed for the above system correspond to i) a pair of fast moving ***gravity waves*** having phase speeds of order of magnitude $\sqrt{gH}$ and to ii) a slow westward moving *Rossby wave* which is of most importance for large scale meteorological processes. The disparity of speeds between these modes creates important constraints for the choice of the integration time step.

Computational Grid and Discretization

A *latitude-longitude grid* with constant angular increments $\Delta\lambda$ and $\Delta\phi$ has been used. The grid is non-staggered (all variables are defined at each node.) Each node lies at the intersection of selected latitude and longitude circles. In contrast to the GLAS model [12] and as in [14] the grid system is shifted by $\frac{\Delta\phi}{2}$ next to the poles. Hence the polar singularity which arises from the $a\cos\phi$ term in the equations above is avoided. For m longitude and n latitude circles

$$\Delta\lambda = \frac{2\pi}{n}, \quad \Delta\phi = \frac{\pi}{m}$$

the first and last latitude circles lying next to the north and south poles respectively, correspond to $\phi = \pm(\frac{\pi}{2} - \frac{\Delta\phi}{2})$. Boundary conditions are doubly periodic as with the GLAS model. In the East-West direction for both scalar and vector elements $s(\lambda + 2\pi, \phi) \equiv s(\lambda, \phi)$. In the Nort-South direction, when variables are needed across the poles, the values are taken from the first grid point encountered by moving along the same latitude circle 180° around the pole. To keep the equations consistent the signs of the vector variables and the trigonometric functions must be changed when combined across the poles [15]. Hence $\vec{v}(\lambda, \pm(\frac{\pi}{2} + \frac{\Delta\phi}{2})) \equiv -\vec{v}(\lambda + \pi, \pm(\frac{\pi}{2} - \frac{\Delta\phi}{2}))$ and $s(\lambda, \pm(\frac{\pi}{2} + \frac{\Delta\phi}{2})) \equiv s(\lambda + \pi, \pm(\frac{\pi}{2} - \frac{\Delta\phi}{2}))$. For the solution of the system, a set of initial conditions for *h, u, v* at all locations must be available. For model testing, the conditions used were derived analytically as in [16], instead of extracting them from a weather map.

The objective is to write finite-difference expressions and discretize the space derivatives $(\frac{\partial}{\partial\phi}, \frac{\partial}{\partial\lambda})$ on the right-hand side of the equations (1). Then a time differencing scheme can be applied to integrate one step ahead in time and update the variables. The standard notation for the average and difference operators will be used: If a function s is defined on a grid having grid increment Δx then the difference operator acting on s would be defined by

$$\delta_{nx} s_i = \frac{s_{i+\frac{n}{2}} - s_{i-\frac{n}{2}}}{\Delta x}$$

and the average operator

$$\overline{s_i}^{nx} = \frac{s_{i+\frac{n}{2}} + s_{i-\frac{n}{2}}}{2}$$

where s_i means the value of the variable s evaluated at point $i\Delta x$. As a result, a second order approximation to $\frac{\partial s}{\partial x}$ at point $i\Delta x$ is given by

$$\delta_x \overline{s_i}^{x} = \frac{s_{i+1} - s_{i-1}}{2\Delta x}$$

Extension to more dimensions are immediate.

By using the aforementioned operators, the right hand sides of (1) could be discretized suitably and consistently as follows:

$$(\frac{\partial h}{\partial t})_\lambda \approx \frac{-1}{a\cos\phi}\left[\delta_\lambda \overline{hu}^{\lambda}\right] \qquad (2a)$$

$$(\frac{\partial hu}{\partial t})_\lambda \approx \frac{-1}{a\cos\phi}\left[\delta_\lambda(\overline{hu}^{\lambda}\ \overline{u}^{\lambda})\right] - \frac{gh}{a\cos\phi}\delta_\lambda \overline{h}^{\lambda} + (f + \frac{u\tan\phi}{a})hv \qquad (2b)$$

$$(\frac{\partial hv}{\partial t})_\lambda \approx \frac{-1}{a\cos\phi}\left[\delta_\lambda(\overline{hu}^{\lambda}\ \overline{v}^{\lambda})\right] \qquad (2c)$$

and

$$(\frac{\partial h}{\partial t})_\phi \approx \frac{-1}{a\cos\phi}\left[\delta_\phi(\overline{hv\cos\phi}^{\phi}\right] \qquad (3a)$$

$$(\frac{\partial hu}{\partial t})_\phi \approx \frac{-1}{a\cos\phi}\left[\delta_\phi(\overline{hv\cos\phi}^{\phi}\ \overline{u}^{\phi})\right] \qquad (3b)$$

$$(\frac{\partial hv}{\partial t})_\phi \approx \frac{-1}{a\cos\phi}\left[\delta_\phi(\overline{hv\cos\phi}^{\phi}\ \overline{v}^{\phi})\right] - \frac{gh}{a}\delta_\phi \overline{h}^{\phi} - (f + \frac{u\tan\phi}{a})hu \qquad (3c)$$

where each variable and constant is evaluated at each grid point. To advance forward in time an *explicit leapfrog* scheme is used, giving that

$$\mathbf{s}^{(n+1)} = \mathbf{s}^{(n-1)} + 2\Delta t(\frac{\partial \mathbf{s}}{\partial t})_{t=n\Delta t} \qquad (4)$$

where for notational simplicity $\mathbf{s}$ is the vector of unknowns $[h, hu, hv]^T$ defined at each grid point.

Such a time-differencing scheme is frequently used with current models. The relevant theory [17] imposes an upper bound on the timestep that one could use to avoid the phenomenon of linear instability. Roughly, the finer the resolution, the smaller that bound. In [14] a stability criterion is shown for the model. A latitude-longitude grid with the converging meridians at the poles, forces a time step much smaller at the polar regions rather than the equators. As a result that minimum time step should be used. The scheme is of second order in space and time . To start the computations, data from two time levels are needed since the chosen time-differencing scheme uses information from the previous two time levels. This is achieved by using a simple forward time scheme for a fractional time step and then proceeding.

MPP Implementation

As mentioned above the problem was implemented on an MPP simulator. The simulator's overhead and the sequential processing done by the host machine, made infeasible the simulation of a 128×128 ARU. Therefore a 32×32 ARU was simulated. It will be noted where that difference would qualitatively alter the statements. Although assembly language was used for the program, the more elegant Parallel Pascal constructs will be used to indicate some features of the code. Explanations are provided for the parallel extensions whenever they are used. The spherical grid is mapped directly on the ARU: each PE and PE memory contained the variables and constants for the corresponding node. The northmost (southmost) PE row corresponded to the longitude circle immediately to the south (north) of the North (South) pole.

The variable elements that are only functions of position like the Coriolis force component f, $\tan\phi$, $\cos\phi$ etc.

can all be precalculated before the start of the integration loop and are constant (with respect to time) arrays. The time dependent variables ***h, u, v*** and their combinations are also defined at each point. Hence the above values are stored in each PE memory at fixed locations and their corresponding type declaration in a high-level language environment supporting parallel constructs (eg. Parallel Pascal) would be

parallel array [1 .. N, 1 .. N] **of** real

where N is the number of rows (columns) of the grid and corresponding ARU systems and where the **parallel** declaration indicates that the array would be used heavily in parallel operations and the compiler is informed of the preference of the user to have the array stored in the ARU memory for the sake of efficiency. The scalar data is either global to the problem and used in the actual equations (2, 3) or is used for counting purposes. To the former category belong g, Δt, $\Delta\phi$, $\Delta\lambda$, a, Ω etc. and to the latter all the variables keeping track of the number of steps and simulated time since the last important event (reinitialization, filtering etc.) The limited available memory per PE (only 1024 bits) forces the programmer to look for ways to economize as much space as possible by using the Main Controller memory when appropriate. The availability of scalar-array arithmetic routines avoids any timing penalty for doing that. In order to avoid unnecessary repetition of floating-point operations some constant arrays and scalars can be combined from the initiation of the computations with the appropriate scale factors (eg. Δt, $\Delta\lambda$ etc.) Thus by storing in the ARU precomputed constant data, the number of required floating operations, which are expensive, could be reduced. This is not a problem for the example. For a more complicated model however, as the first configuration of the ARU memory is limited, the SM must be used. The available East-West ARU topology readily provides for the East-West periodicity, the first and last PE columns corresponding to $\lambda = 0$ and $\lambda = 2\pi - \Delta\lambda$ respectively. The situation is slightly more complicated for the simulation of the periodicity across the poles, as that cannot be mapped to the available ARU topologies. To combine the appropriate elements, the variable arrays would have to be cylindrically rotated (using the same East-West interconnection as above) by $\frac{N}{2}$ columns. For the $N = 32$ case, the cycle count is underestimated for the routing operations viz. the $N = 128$ case. As it will be seen however, the operation is infrequent enough not to significantly change the full timing estimates.

The corresponding angular increments for the simulated and the real ARU are $\Delta\lambda = 2\Delta\phi \approx 11^o$ and $\Delta\lambda = 2\Delta\phi \approx 3^o$ respectively. To compare, a currently used GLAS model utilizes $\Delta\lambda = 5^o$ and $\Delta\phi = 4^o$ whereas an 'ultrafine' version has $\Delta\lambda = 3^o$ and $\Delta\phi = 2.5^o$. We thus see that the MPP dimensions are adequate for the described problem. Hence no 'dimensional sacrifice' need be done to size down the problem for a parallel implementation, an unfortunate practice in some examples demonstrating the usefulness of parallel computers.

The operations applied at each grid point (PE) for the derivation of the spatial differences are local operations. The only place where this is not so is at some of the calculations for the latitudinal differences. For the scheme used, which is of 2^d order, only elements from the immediate neighbours are needed. For a more accurate 4th order scheme, like the one used in [12] the elements involved for a calculation at PE(i,j) would lie in the surrounding PEs at distances of at most 2.

The inner loop of the code would go through the following steps in order to derive the new values at $t = (n+1)\Delta t$ out of known data at the two previous time steps:

1) From the available values for h, u, v at time level n use the equations (2,3) above to approximate the space derivatives and by adding the λ and ϕ contributions at each grid point (PE) as in (1) derive an approximation to $2\Delta t(\frac{\partial \mathbf{s}}{\partial t})$ for the current time level.

2) Apply (4) to derive the values at the new time level. This only needs one addition per component of the **s** vector.

3) Update the variables and counters.

The PE memory restriction of 1K bits was observed: only 680 bitplanes have been used and these could be further reduced down to 600. The numerical stability considerations mentioned above mean that in going from the simulated array to the real ARU a drastic reduction of time step must be performed in order to satisfy the CFL criterion. We have used $\Delta t = 200$sec for the case $N = 32$, but a rough stability analysis gives that for $N = 128$ $\Delta t \approx 20$ seconds! This is a serious constraint in going to the real ARU. Moreover, the allowed time steps near the equator are much larger than the ones allowed near the poles, due to the convergence of the meridians at the poles. Methods that could be used to overcome this are the Fourier filtering of the unstable waves [18] or the use of an implicit time differencing scheme [19] with no stability restrictions. Alternatively, since the time constraint is relaxed by going to a coarser grid, the number of longitude circles can be reduced (eg halved.) Since the North-South interconnections are not used, two independent simulations could be run concurrently (using the same global constants but different initial conditions). The one system would lie in the north half of the ARU (using half of the PE rows) and the other would lie in the south half. At the end of each computation cycle results for both systems would be simultaneously obtained.

In order to avoid the phenomenon of ***non-linear instability*** [20], a filtering procedure [21] was used. Such a filter applied in the East-West direction to an array variable **q** at location (i, j), repeatedly calculates $\frac{q_{ij+1} - 2q_{ij} + q_{ij-1}}{4}$ and then combines the result with the original array values q_{ij}. As explained above, things are slightly more complicated in the North-South direction. This filtering results in the elimination of $2\Delta x$ waves where Δx is the East-West grid distance which may introduce the instability. This procedure was applied every hour on h, u, v, eight times in each direction resulting in a 16^{th} order filter. The MPP implementation of the above operator is very efficient, taking full advantage of the parallelism.

In a Parallel Pascal environment

```
const
   order = 8 ;
type
   arr = parallel array[1 .. N, 1 .. N] of real ;
var
   q, qsv, temp : arr ;
   i : integer ;
 begin
   qsv := q ;
   for i :== 1 to order do
      begin
         temp := q - rotate(q, 0, 1);
         q :=rotate(temp, 0, -1)- temp;
         q := q/2.;
         q := q/2.
      end;
   q := qsv - q;
end;
```

The **rotate**(*a*, *i*, *j*) function returns an array consisting of the elements of *a* circularly rotated by distances *i*, *j* along the two dimensions. The repeated divisions by 2 are done by special purpose routines. The bit-serial nature of the arithmetic makes divisions and multiplications by 2 and its powers considerably faster than if using the floating point division by 2 routines.

Description of results

The resulting simulation corresponded to east to west moving wave patterns for *h, u, v*. At selected time intervals the contents of the corresponding ARU memory locations were examined. A software interface was set between the simulator and a colour graphics display terminal and the real valued array variables were scaled and mapped on a gray scale. The resulting integer valued arrays produced an image of the moving wave. On the real ARU this correspondence can be handled very quickly. The scaling and integer transformations occur at each grid point and full advantage of the parallelism is taken. Thus even for this output oriented consideration the MPP can be used very efficiently. Because of the extraordinary software complexity of the simulator and the slowness of the host machine, the simulation was only for a few hours.

MPP Timings and Comparisons

The number of consumed cycles for each routine and its function are given in the table. The number of PCU cycles is more important as the MCU works ahead and in parallel and takes less time.

The number of cycles given for each routine includes the cycles spent by any called subroutines during the routine's execution. As can be seen from the timing table, a main step, consisting of the space derivative calculations *dtcalc* (by far the most time consuming routine amongst the frequently called ones) and of *update* takes about 43,000 PE cycles, or for 100 ns/cycle there are about 230 iterations/sec. With a 200sec time step and excluding any overhead, we get that about 1 day is simulated in 2 seconds. This of course is a very rough estimate. If filtering is needed, once per hour say, the time estimate above does not significantly increase despite the cost of *filter*.

Name	Description	MCU	PCU
	(standard library floating-point routines)		
fmul	multiplication	63	787
fsub	subtraction	61	381
fadd	addition	51	381
fdiv	division	101	1031
fhalf	division by two	100	266
fmulsa	scalar-array multiplication	64	791
	(user-written special purpose routines)		
barl	longitude averages	162	745
barp	latitude averages	176	1041
ndifr	longitude finite differences	136	1269
ndifc	latitude finite differences	163	1668
filter	Shapiro 16^{th} order filter	18357	96408
	(user-written main step routines)		
dtcalc	space derivatives	4706	38835
update	leapfrog update	390	3403

It is fair to say that there is little agreement on the way the performance of unconventional machines could be evaluated and compared. The MOPS/MFLOPS rates are some of the most popular measures but even for those, their uniform validity across the machine spectrum is disputed. Other standards have also been proposed [22, 23, 24]. As long as we are interested in particular problems and not general evaluations a good strategy is to simply compare the timings for their solution on the examined machines [25]. Even this however may not be a fair criterion since the fastest algorithms for each machine would possess different numerical properties. Dimensions also would probably be chosen to suit the machine (magic numbers like 64 or 128 for the Cray, the DAP or the MPP) rather than the modeller. Since our experiments have been conducted on a simulator it is only approximate comparisons that we could make. The DAP [26] array is 1/4 of the MPP array and also has considerably slower MFLOP rates. As a result, for this problem it would not compete with the MPP. On the other hand the currently used DAPs have 4K bits of memory per PE (to be extended to 16K) and hence it could be used to model a multi-level model, or one containing more equations, without the constraints imposed by frequent I/O exchanges which might be needed for the MPP in its initial memory configuration. Experiments on the CRAY-1 [19] for a similar model with a resolution of $\Delta\lambda = \Delta\phi = 12^o$ required $8.5\times 10^{-4} sec/step$, whereas with $\Delta\lambda = \Delta\phi = 4^o$ the integration required $4.97\times 10^{-3} sec/step$. The $\Delta\lambda = 2\Delta\phi \approx 11^o$ resolution achieved with the MPP simulator took about $4.3\times 10^{-3} sec/step$. As the parallelism is almost fully exploited and redundancy is kept at a minimum by going to the full ARU this timing estimate would only increase slightly but the available resolution would be $\Delta\lambda = 2\Delta\phi \approx 3^o$ which is superior to the finer resolution in the Cray model above. Therefore a better resolution and comparable or better timing would be achieved with the MPP. Moreover, it would be possible to have concurrent solutions, starting from different initial data as mentioned above.

Conclusions

We have discussed the use of the MPP for the integration of a set of non-linear PDEs frequently occuring in Numerical Weather Prediction. We have found that the MPP can be used for the solution of such problems. We have talked about the problems facing the algorithm designer and suggested methods for coping with them. The MPP gives much better performances for integer computations. It is thus worth investigating the possibility of using fixed or block floating point arithmetic in the computations. The use of the MPP makes efficient and possible the simulation of General Circulation models over the entire globe with adequate horizontal resolution. Consequently the modeller doesn't have to worry about imposing artificial lateral boundary conditions. A complete model would have multiple vertical levels. It would also take account of thermal phenomena which here were ignored, by incorporating more equations. A 4th order space differencing scheme would be preferable. For multiple level simulation the PE memory becomes inadequate. The addressing capability of the PE index registers is for 64K bits per PE memory and a future system could contain such memory. Nevertheless the first available MPP will have to use its Staging Memory as an immediate active buffer area. The SM capacity would initially be 4Mbytes and can be expanded to 64Mbytes. In that configuration rates of 160 Mbytes/sec are achieved. The available permutation network has many capabilities for accessing subarrays or other patterns from and to the storage area. A few of the not too frequently used arrays for a large model could be stored in the stager and brought in the ARU under some paging policy designed to minimize interference with the computations. A preliminary theoretical study of the problems related with memory allocation and management for the Staging Memory can be found in [8]. Real or various analytical initial data would be gathered in a data base and from there it would initialize the MPP arrays. Multiple concurrent simulations could then be run as suggested above or a single fine-grid simulation could be instigated and the results at selected time steps would be displayed. For long predictions, periodic updating of the variables could be done by utilizing new observed data. Overall, the MPP would be a powerful computational tool for modellers.

Acknowledgements

This work was done while the authors were working with the Computer Development Section at the Goddard Space Flight Center. We would like to thank Ken Wallgren, Dr. Eugenia Kalnay, Larry Takacs, Jim Fisher and Dr. Jim Strong of the Goddard Space Flight Center, Dennis Lynch of CSC for the friendly environment and the helpful discussions, and Professor Dan Slotnick of the University of Illinois for making this summer project possible.

References

[1] K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers,* (September, 1980), pp. 836-840.

[2] E. Kalnay-Rivas and Larry Takacs, "A Simple Atmospheric model on the Sphere with 100% Parallelism," *Advances in Computer Methods for PDE's,* R. Vichnevetsky ed., Vol. IV, IMACS 1981.

[3] L. F. Richardson, *Weather Prediction by Numerical Process,* Dover, (1965), 236 pp.

[4] A. B. Carrol and R. T. Wetherald, "Application of Parallel Processing to Numerical Weather Prediction," *Journal of the ACM,* (July, 1967), pp. 591-614.

[5] R. I. Wilhelmson, "Solving Partial Differential Equations Using the Illiac IV," *Constructive and Computational Methods for Differential and Integral Equations,* Springer-Verlag, (1974), pp. 453-475.

[6] M. Graham and D. L. Slotnick, *An Array Computer for the Class of Problems Typified by the General Circulation of the Atmosphere,* Department of Computer Science, University of Illinois at Urbana-Champaign, Report UIUCDCS-R-75-761, (December, 1975), 289 pp.

[7] J. B. Dennis and Ken K.-S. Weng, "Application of Data Flow Computation to the Weather Problem," *High-Speed and Algorithm Organization,* Academic Press, (1977), pp. 143-157.

[8] A. P. Reeves and J. D. Bruner, *The Language Parallel Pascal and other Aspects of the Massively Parallel Processor,* School of Electrical Engineering, Cornell University, (December, 1982), 248 pp.

[9] D. J. Kopetzky, *An Array Simulator Generator,* Department of Computer Science, University of Illinois at Urbana-Champaign, Report UIUCDCS-R-80-1031, (September, 1980), 63 pp.

[10] E. Gallopoulos, Scott McEwan and Dianna Visek, *MPP Simulator Manual,* Department of Computer Science, University of Illinois at Urbana-Champaign, Report UIUCDCS-R-82-1075, (April, 1982), 35 pp.

[11] D. Lynch, P. Jones, J.Reese, C. Weger, *The Massively Parallel Processor System,* Computer Sciences Corporation, Report CSC/TM-82/6034, (February, 1982).

[12] E. Kalnay-Rivas and D. Hoitsma, *Documentation of the 4th Order Banded Model,* Laboratory for Atmospheric Sciences, NASA Goddard Space Flight Center, Technical Memorandum 80608, (December, 1979).

[13] Akira Kasahara, "Computational Aspects of Numerical Weather Prediction and Climate Simulation," *Methods in Computational Physics,* Academic Press, (1977), pp. 1-65.

[14] M. Grimmer and D. B. Shaw, "Energy-Preserving Integrations of the Primitive Equations on the Sphere," *Quart. J. Roy. Meteor. Soc.,* (1967), pp. 337-349.

[15] D. L. Williamson, "Difference Approximations for Fluid Flow on a Sphere," *Numerical Methods in Atmospheric Models,* GARP Publication, (September, 1979), pp. 51-120.

[16] Norman Phillips, "Numerical Integration of the Primitive Equations on the Atmosphere," *Monthly Weather Review,* (September, 1959), pp. 333-345.

[17] R. D. Richtmyer and K. W. Morton, *Difference Methods for Initial-Value Problems,* Wiley(Interscience), (1967).

[18] D. L. Williamson, "Linear Stability of Finite-Difference Approximations on a Uniform Latitude-Longitude Grid with Fourier Filtering," *Monthly Weather Review,* (January, 1976), pp. 31-41.

[19] R. L. Gilliland, "Solution of the Shallow Water Equations on the Sphere," *Journal of Computational Physics,* (September, 1981), pp. 79-94.

[20] N. Phillips, "An Example of Nonlinear Computational Instability," *The Atmosphere and Sea in Motion,* Rockefeller Inst. Press, (1959).

[21] R. Shapiro, "Smoothing, Filtering and Boundary Effects," *Rev. Geophys. and Space Phys.,* (May, 1970), pp. 359-387.

[22] L. J. Siegel, H. J. Siegel and P. H. Swain, "Performance Measurements for Evaluating Algorithms for SIMD Machines," *IEEE Transactions in Software Engineering,* (July, 1982), pp.319-331.

[23] D. J. Kuck, *The Structure of Computers and Computations,* Wiley, (1978), 610 pp.

[24] D. Hockney and J. Jesshope, *Parallel Computers,* Adam Hilger, (1982), 423 pp.

[25] D. Parkinson and H. Liddell, "The Measurement of Performance on a Highly Parallel System," *IEEE Transactions in Computers,* (January, 1983), pp. 32-37.

[26] P. M. Flanders, D. J. Hunt, S. F. Reddaway, and D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor," *High Speed Computer and Algorithm Design,* Academic Press, (1977), pp. 113-128.

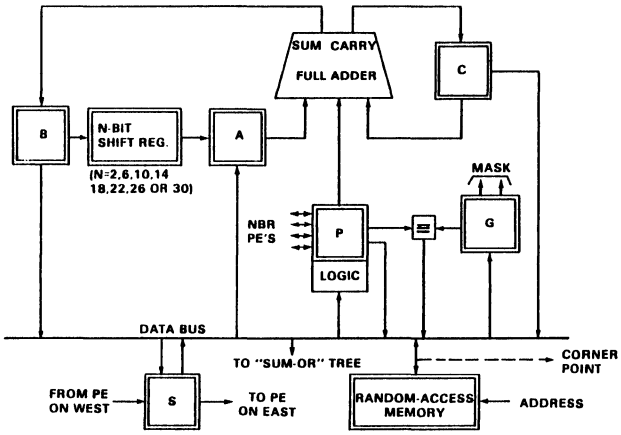


Fig. 2 MPP Processing Element

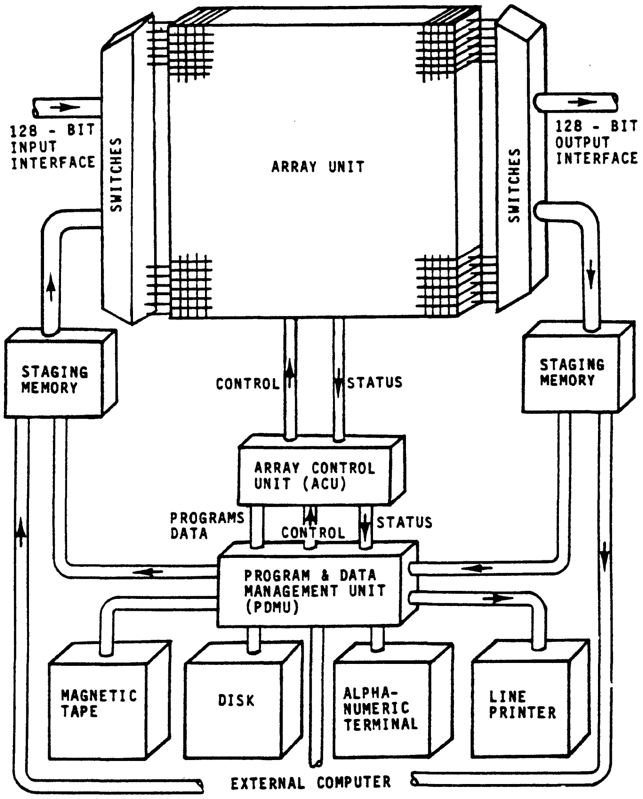


Fig. 1 MPP System

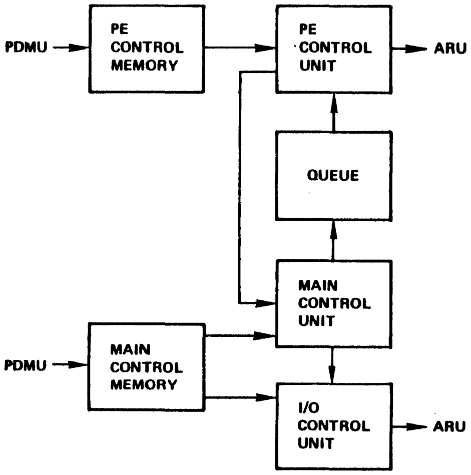


Fig. 3 Array Control Unit (ACU)

AN M-STEP PRECONDITIONED CONJUGATE GRADIENT METHOD FOR PARALLEL COMPUTATION

Loyce Adams
Institute for Computer Applications in Science and Engineering
Hampton, Virginia 23665

Abstract -- This paper describes a preconditioned conjugate gradient method that can be effectively implemented on both vector machines and parallel arrays to solve sparse symmetric and positive definite systems of linear equations. The implementation on the CYBER 203/205 and on the Finite Element Machine is discussed and results obtained using the method on these machines are given.

Introduction

In this paper we are concerned with the solution of a sparse $N \times N$ system of symmetric and positive definite linear equations

$$Ku = f \quad (1.1)$$

by preconditioned conjugate gradient (PCG) methods on both vector computers and parallel arrays. Several descriptions of these methods appear in the literature; see for example, Concus, Golub, O'Leary [1976] and Chandra [1978]. Also, Schrieber [1978] discussed the implementation of conjugate gradient (CG) on vector computers and Podsiadlo and Jordan [1981] discussed its implementation on the Finite Element Machine under construction at NASA Langley Research Center.

The PCG method solves the system $\hat{K}\hat{u} = \hat{f}$ where

$$\hat{K} = Q^T M^{-1} K Q^{-T}, \quad \hat{u} = Q^T u, \quad \hat{f} = Q^{-1} f, \quad (1.2)$$

Q is a nonsingular matrix, and the symmetric and positive definite preconditioning matrix is given by $M = QQ^T$. The algorithm for the solution of u directly is described in Chandra [1978] and is given below where u, r, $\hat{r}$, and p are vectors and (x,y) denotes the inner product $x^T y$.

```
(1) Choose $u^o$
(2) $r^o = f - Ku^o$
(3) $M\hat{r}^o = r^o$
(4) $p^o = \hat{r}^o$
(5) For $k = 0,1,\cdots k_{max}$
        (1) $\alpha = \frac{(\hat{r}^k, r^k)}{(p^k, Kp^k)}$
        (2) $u^{k+1} = u^k + \alpha p^k$
        (3) If $\|u^{k+1} - u^k\|_\infty < \varepsilon$ then
            stop, otherwise continue.
        (4) $r^{k+1} = r^k - \alpha K p^k$
        (5) $M\hat{r}^{k+1} = r^{k+1}$
        (6) $\beta = \frac{(\hat{r}^{k+1}, r^{k+1})}{(\hat{r}^k, r^k)}$
        (7) $p^{k+1} = \hat{r}^{k+1} + \beta p^k$
```

Algorithm 1. PCG Algorithm

We note that the standard conjugate gradient algorithm results by choosing $M = I$.

For vector machines, if $M = I$, all steps of the iteration loop except (1) and (6) can be vectorized. In particular, the multiplication Kp, for K sparse, vectorizes after a suitable ordering of the equations and will be discussed in detail in Section 3. The difficulty arises in the formation of the inner products necessary to calculate α and β. These calculations require a phase in which N partial sums must be added together and therefore do not vectorize well.

For parallel arrays like the Finite Element Machine (Jordan [1978], Adams [1982]), the calculation of u, r, and p can be distributed to the individual processors and the necessary communication between processors can be performed on the dedicated local links. The convergence test in (3) can be performed by using the flag network. However, for a large number of processors, the calculations of α and β can be expensive since the number of values to be summed for each inner product is equal to P, the number of processors. Jordan [1979] realized that this was potentially detrimental to the efficiency of the method on this machine, and as a result, a special hardware circuit (sum/max) was designed to perform the P sums in $O(\log_2 P)$ time.

Since Algorithm 1 has two inner products per iteration that will become costly as N (on vector machines) or P (on arrays) increases, a natural goal is to devise a preconditioner that will reduce the number of CG iterations, and hence the number of inner products, while being inexpensive to implement. In the next section preconditioners that are based on taking m steps of an iterative method are described. In Section 3, the implementations of these methods on the CYBER 203/205 and the Finite Element Machine are

The research reported in this paper was supported in part by the National Aeronautics and Space Administration under NASA Contract No. NAS1-46 while the author was at the University of Virginia, Charlottesville, VA. and in part by the National Aeronautics and Space Administration Contract Nos. NAS1-15810, NAS1-17070 and NAS1-17130 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

0190-3918/83/0000/0036$01.00 © 1983 IEEE

given for a system of equations that results from an example structural engineering problem. Results for this problem on the CYBER 203 and the Finite Element Machine are given in Section 4.

2. M-Step Parallel Preconditioners

2.1 Choosing M

The preconditioned conjugate gradient algorithm of the last section requires a symmetric and positive definite preconditioning matrix M. The question is how to choose M so that the condition number of $\hat{K}$,

$$\kappa(\hat{K}) = \frac{\max_i \lambda_i}{\min_i \lambda_i},$$

is as small as possible.

The best choice for M in the sense of minimizing $\kappa(\hat{K})$ is $M = K$ but this gains nothing since $K\hat{r} = r$ is just as difficult to solve as $Ku = f$. A class of preconditioners that appears to be easily implemented on parallel computers arises by choosing M to be a splitting of K that describes a linear stationary iterative method. As an example, the SSOR splitting of K yields

$$M = \frac{\omega}{2-\omega}\left(\frac{1}{\omega}D-L\right)D^{-1}\left(\frac{1}{\omega}D-U\right) \qquad (2.1)$$

where D,-L, and -U are the diagonal, strictly lower, and strictly upper parts of K respectively. This splitting has been considered extensively in the literature as a preconditioner; for example, refer to Concus, Golub, O'Leary [1976] and the references therein. Now, if the matrix K is ordered by the Multicolor ordering (Adams and Ortega [1982]), the system $M\hat{r} = r$ can be implemented on parallel computers as a forward followed by a backward Multicolor SOR iteration applied to $K\hat{r}=r$ with initial guess $\hat{r}^{(0)}=0$ and will be explained in more detail in Section 3. The question now arises whether it would be beneficial to take more than one step of a linear stationary iterative method to produce a preconditioner M that more closely approximates K. If this is done, the resulting preconditioning matrix is

$$M = P\left(I+G+\ldots+G^{m-1}\right)^{-1}. \qquad (2.2)$$

Now, M must be symmetric and positive definite to be considered as a preconditioner. The necessary and sufficient conditions for M to satisfy these requirements are given in Adams [1982] and we only note here that if P is the SSOR splitting matrix these conditions are met. We also note that Dubois, Greenbaum, and Rodrique [1979] considered a truncated Neumann series for K^{-1} as a preconditioner which corresponded to a Jacobi splitting where P = diag(K).

Even though the preconditioner in (2.2) for the SSOR splitting is symmetric and positive definite, the question of how well the resulting PCG method will reduce the number of CG iterations must be answered. In Adams [1982], for the SSOR splitting, the condition number of the matrix $\hat{K}$ of (1.2) was proven to decrease as the number of steps of the preconditoner in (2.2) increases; however, the maximum ratio of $\frac{\kappa(\hat{K}_1)}{\kappa(\hat{K}_m)}$ was shown to be m. In practice, for larger m, this reduction may not be enough to balance the increase in the work that must be done by the preconditoner (as results in Section 4 verify). However, by parametrizing this preconditoner, the method is very effective. This parametrization is briefly discussed in the next section and the parameters for the SSOR splitting are given.

2.2 Parametrizing M

Johnson, Micchelli, and Paul [1982] have suggested symmetrically scaling the matrix K to have unit diagonal and then taking m terms of a parametrized Newmann series for $K^{-1} = (I-G)^{-1}$ as the value for M^{-1}. This corresponds to a symmetric preconditioning matrix whose inverse is a polynominal of degree m-1 in G,

$$M_m^{-1} = \alpha_0 I + \alpha_1 G + \alpha_2 G^2 + \ldots + \alpha_{m-1} G^{m-1} \qquad (2.3)$$

derived from the Jacobi splitting,

$$K = I - G \qquad (2.4)$$

of K; hence, the solution to $M_m\hat{r} = r$ can be implemented by taking m steps of the Jacobi iterative method applied to $K\hat{r} = r$ with initial guess $\hat{r}^{(o)} = 0$. Johnson, et.al. choose the α_i's so that the eigenvalues of $M_m^{-1}K$, and hence those of M_m, are positive on the interval $[\lambda_1, \lambda_n]$ that contains the eigenvalues of K and are as close to 1 as possible in some sense such as the min-max or the least squares criteria. Clearly, if m = 1, $M_m^{-1}K = \alpha_0 K$ and the condition number of $M_m^{-1}K$ is the same for all $\alpha_0 \neq 0$. Hence, we are only interested in m > 1.

We now generalize this idea for any splitting of the matrix K,

$$K = P - Q. \qquad (2.5)$$

If $G = P^{-1}Q$, then by parametrizing (2.2), the inverse of the m-step preconditioner becomes

$$M_m^{-1} = \left(\alpha_0 I+\alpha_1 G+\alpha_2 G^2+\ldots+\alpha_{m-1} G^{m-1}\right)P^{-1} \qquad (2.6)$$

and will be symmetric if P is symmetric. We choose the values of α_i so that the eigenvalues of $M_m^{-1}K$ are positive on the interval $[\lambda_1 . \lambda_n]$ that contains the eigenvalues of $P^{-1}K$ and are as close to 1 as possible in some sense such as the min-max or least squares criteria. For the least squares criteria, the values of α_i that correspond to the SSOR splitting are given in Table 1 for m = 2,3, and 4.

Table 1.

α Values for the m-step SSOR PCG Method

m	α_0	α_1	α_2	α_3
2	1.00	5.00		
3	1.00	-2.00	7.00	
4	1.00	7.00	-24.50	31.50

In the next section we describe how to implement the m-step parametrized SSOR PCG method on the

CYBER 203/205 and on the Finite Element Machine and in Section 4, results on these machines are given.

3. Implementation of the m-step SSOR PCG Method

We first describe the algorithm for solving $M\hat{r} = r$, where M is the preconditioning matrix given by (2.6). To be concrete, this description will be given for the following test problem.

The domain considered will be a rectangular plate discretized with triangular finite elements over which linear basis functions are defined. The nodes of the triangles are colored Red, Black, and Green so that nodes on a given triangle are different colors as shown in Figure 1. This coloring, as described in Adams and Ortega [1982], decouples the equations so that an implementation on either vector or array computers is possible as will become more apparent later in this discussion.

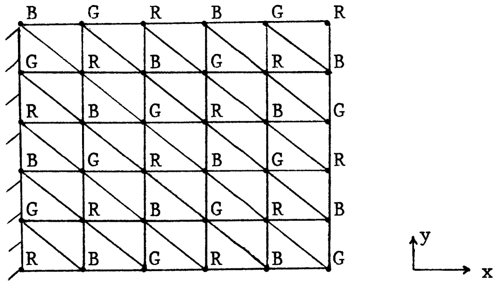


Figure 1. Plate (Triangular Elements)

The problem is to determine the displacements, say u and v, in the x and y directions respectively at each node in the plate whenever the plate is loaded on one edge and constrained on another. The partial differential equations of plane stress that govern these displacements are well known, see Norrie and DeVries [1978], but do not contribute to the discussion here. The important point to make is that the stiffness matrix K of (1.1) will be symmetric and positive definite and will have dimension $2ab \times 2ab$ where a is the number of rows of nodes and b is the number of columns of unconstrained nodes (2 unknowns at each node), and each row of K will contain at most 14 nonzero elements which correspond to the grid point stencil for linear triangular elements shown in Figure 2.

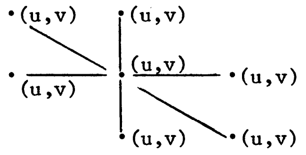


Figure 2. Grid Point Stencil

Observe from Figures 1 and 2 that while there is no coupling between the equations at two nodes of the same color, the equations at a given node do couple. Hence, to completely decouple the system, six colors are necessary; namely, Red(u), Red(v), Black(u), Black(v), Green(u), and Green(v). Now, if the equations at the nodes in Figure 1 are numbered by these six colors from bottom to top, left to right, the system $K\hat{r} = r$ has the form,

$$\begin{bmatrix} D_{11} & B_{12} & B_{13} & B_{14} & B_{15} & B_{16} \\ B_{12}^T & D_{22} & B_{23} & B_{24} & B_{25} & B_{26} \\ B_{13}^T & B_{23}^T & D_{33} & B_{34} & B_{35} & B_{36} \\ B_{14}^T & B_{24}^T & B_{34}^T & D_{44} & B_{45} & B_{46} \\ B_{15}^T & B_{25}^T & B_{35}^T & B_{45}^T & D_{55} & B_{56} \\ B_{16}^T & B_{26}^T & B_{36}^T & B_{46}^T & B_{56}^T & D_{66} \end{bmatrix} \begin{bmatrix} \hat{r}_1 \\ \hat{r}_2 \\ \hat{r}_3 \\ \hat{r}_4 \\ \hat{r}_5 \\ \hat{r}_6 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{bmatrix} \quad (3.1)$$

where B_{12}, B_{34}, B_{56}, and D_{ii}, $i = 1$ to 6 are diagonal matrices.

The SSOR iteration can be realized by a forward followed by a backward Multicolor SOR iteration, (Adams and Ortega [1982]), but is only as expensive as one Multicolor SOR iteration since a technique of Conrad and Wallach [1979] can be used to save results in an auxiliary vector, y, from the forward pass to be used in the backward pass. The procedure is given below for solving $M\hat{r} = r$ of Algorithm 1. The relaxation parameter ω of the SSOR method causes no problems in the implementation and will be set to one here for simplicity.

```
(1) $\hat{r} = 0$;   $y = 0$
(2) For $s = 1$ to $m$
    (1) For $c = 1$ to $6$
        (1) Form $x = -\sum_{j=1}^{c-1} B_{jc}^T \hat{r}_j$
        (2) Solve $D_c \hat{r}_c = x + y_c + \alpha_{m-s} r_c$
        (3) Set $y_c = x$
    (2) For $c = 5$ down to $2$
        (1) Form $x = -\sum_{j=c+1}^{6} B_{cj} \hat{r}_j$
        (2) Solve $D_c \hat{r}_c = x + y_c + \alpha_{m-s} r_c$
        (3) Set $y_c = x$
(3) Solve $D_1 \hat{r}_1 = -\sum_{j=2}^{6} B_{1j} \hat{r}_j + y_1 + \alpha_0 r_1$
```

Algorithm 2. m-step 6-color SSOR

Notice that the values of α_{m-s} above are the parameters that were given in Table 1, and if no parametrization is desired, these are simply set to one. We also point out that Algorithm 2 can easily be modified to solve problems whose domains are discretized by more complicated finite elements or finite differences as long as a multicolor ordering is used. For more details see Adams and Ortega [1982]. We now turn to the implementation of Algorithm 1 in conjunction with Algorithm 2 on the CYBER 203/205.

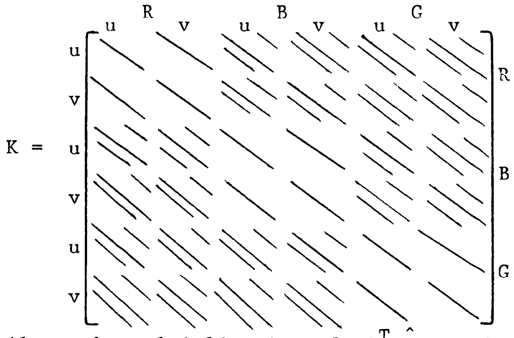
3.1 CYBER 203/205 Implementation

On the CYBER 203/205, vectors consist of contiguous storage locations and maximum efficiency of vector operations is achieved for very long vectors. For vectors of length 1000 around 90% efficiency is obtained, but this drops to approximately 50% or less for vectors of length 100 and 10% for vectors of length 10.

To achieve the maximum vector length for our test problem the u equations at the Red nodes (left to right, bottom to top) including the constrained nodes are numbered first, followed by the corresponding v equations at the Red nodes, then by the Black u, Black v, Green u, and Green v equations. The numbering of the constrained equations is necessary for ease of implementation given the CYBER's contiguous storage requirement but also increases the vector length from 1/3ab to $\frac{1}{3}a(b+1)$. Of course, the actual updating of the storage locations corresponding to these constrained nodes is prohibited by the control vector feature on this machine, see Ortega and Voigt [1977], and for large values of a and b little inefficiency is incurred. For a unit square plate, the maximum vector length for our test problem is $\frac{a^2}{3}$ and is around 1000 when $a \approx 55$, or equivalently when the width of each triangle is equal to 1/54.

The contiguous storage requirement coupled with the manner in which the nodes are colored imposes a restriction on the number of nodes that can be in each row of the plate. In particular, the last node in the first row must be Black so that the coloring R/B/G/R/B/G, etc. wraps around from one row to the next.

Now, the calculations of Ku^o and Kp^k in Algorithm 2 can be done by a straightforward generalization of Madsen, Rodrique, and Karush's [1976] matrix multiplication by diagonals scheme since K of (3.1) has the structure shown in (3.2) (and will be stored by these diagonals as well):



(3.2)

Also, the multiplication of $B_{jc}^T\hat{r}_j$ and $B_{cj}\hat{r}_j$ in Algorithm 2 can be performed by the same technique. The subtraction in the convergence test $\|u^{k+1}-u^k\|_\infty < \varepsilon$ vectorizes and the absolute value is performed by the vector absolute value function that is available on the CYBER. The inner products for the calculation of α and β are done by a call to an inner product routine which utilizes the machine's vector hardware; however, the additions of the partial sums make this operation considerably slower than the other vector operations required in the algorithm.

Next, we turn to the implementation of Algorithm 1 in conjuction with Algorithm 2 on the Finite Element Machine.

3.2 Finite Element Machine Implementation

The first task for the implementation on this machine is to assign the nodes (and hence equations at the nodes) of the plate to the processors. This is done by assigning each processor, as nearly as possible, an equal number of Red/Black/ and Green unconstrained nodes as illustrated in Figures 3a, 3b, and 3c, where in each Figure, the node colorings may repeat beyond the region shown.

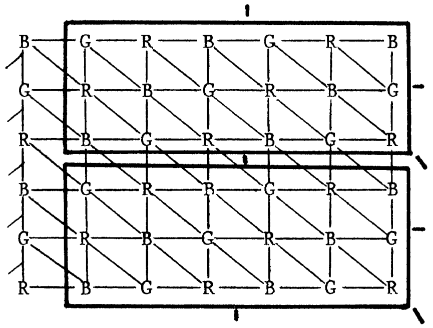


Figure 3a. 18 nodes/procesor

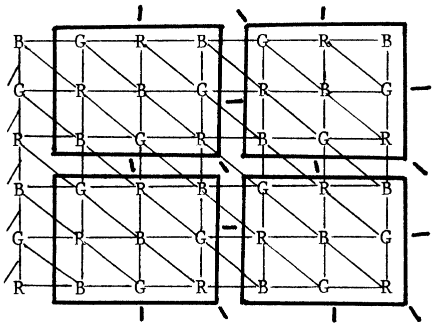


Figure 3b. 9 nodes/processor

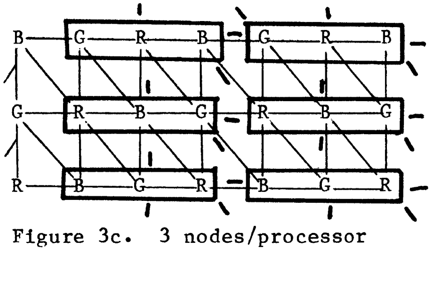


Figure 3c. 3 nodes/processor

In contrast to the CYBER implementation we need not be concerned with numbering the constrained nodes, but instead we should require that each processor receive an equal distribution of each color of the unconstrained nodes.

Since memory is distributed on the Finite Element Machine, each processor stores the portion of u, p, $\hat{r}$, r and K that corresponds to its collection of nodes. For each equation that is assigned to a processor, 14 storage locations are reserved for the nonzero coefficients of K that correspond to the grid point stencil in Figure 2. For more information about these data structures see Adams [1982]. In addition, storage must be reserved in each processor for the portion of p that must be received from neighbor processors during the calculation of Kp each iteration. For example, in Figure 3b, processor 1 must reserve storage for the components of p that correspond to the 3 border nodes in processor 3 and the 3 border nodes in processor 2, but no components are received from processor 4 since no nodes in processors 1 and 4 share a common triangle. This same storage may be used initially for u^o during the calculation of Ku^o. Similarly, storage must be reserved for the $\hat{r}$ components associated with the equations at border nodes in neighbor procesors for the multiplications of $B^T_{jc}\hat{r}_j$ and $B_{cj}\hat{r}_j$ in Algorithm 2.

The sending and receiving of the border p components in each CG iteration in Algorithm 1 and the border $\hat{r}$ components during each step of the preconditioner in Algorithm 2 is only (for rectangular regions) between neighbor processors and in particular for our test problem will require six of the machine's eight nearest neighbor links as shown in Figure 4 for processor P.

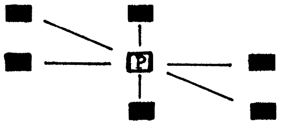


Figure 4. FEM Local Links

Hence, the communication required for the m-step SSOR preconditioner on this machine is completely local and the amount of data that a given processor must communicate can be seen from Figure 3 to be dependent on its number of neighbors as well as the dimension of the rectangle of nodes assigned to it. To reduce the time required for the I/O, the values of each color to be sent to a given neighbor can be packaged and sent as one record and likewise for the values of a particular color to be received from a given neighbor. If this is done, it becomes advantageous to think of the two equations at the same node as being the same color, because, on this machine, it does not matter that they couple since they will always be assigned to the same processor.

The convergence test in Algorithm 1 is implemented by the signal flag network. Each processor raises its convergence flag whenever its portion of u values are within the stopping criterion. The processors are then synchronized and tested to see if all flags are raised; if so, the iteration stops -- if not, all flags are lowered and the iteration continues.

Lastly, we summarize our remarks about the Finite Element Machine implementation of Algorithm 2 by providing a parallel version in Algorithm 3 that will be executed by processor p. The subscript p denotes the portion of a vector that is assigned to processor p, the subscript n denotes the portion of the vector that is received from all of processor p's neighbors and the subscript t denotes the total vector which consists of the components received by, as well as those assigned to, processor p.

```
(1) $\hat{r}_t = 0; y_p = 0$
(2) For s = 1 to m
    (1) For c = 1 to 6
        (1) $x = -\sum_{j=1}^{c-1} B^T_{jc}\hat{r}_{j,t}$
        (2) $D_{c,p}\hat{r}_{c,p} = x + y_p + \alpha_{m-s}r_p$
        (3) $y_p = x$
        (4) If c mod 2 = 0 then
            (1) Send border portion of $\hat{r}_{c-1,p}$ and $\hat{r}_{c,p}$
            (2) Receive $\hat{r}_{c-1,n}$ and $\hat{r}_{c,n}$
    (2) For c = 5 down to 2
        (1) $x = -\sum_{j=c+1}^{6} B_{cj}\hat{r}_{j,t}$
        (2) $D_{c,p}\hat{r}_{c,p} = x + y_p + \alpha_{m-s}r_p$
        (3) $y_p = x$
        (4) If c mod 2 $\neq$ 0 then
            (1) Send border portion of $\hat{r}_{c+1,p}$ and $\hat{r}_{c,p}$
            (2) Receive $\hat{r}_{c+1,n}$ and $\hat{r}_{c,n}$
(3) Solve $D_{1,p}\hat{r}_{1,p} = -\sum_{j=2}^{6} B_{1j}\hat{r}_{1,t} + y_p + \alpha_o r_p$
```

Algorithm 3. FEM m-step 6-color SSOR

4.Results

The example plane stress problem was run on the CYBER 203 at the NASA Langley Research Center for a unit square plate for varying mesh sizes. Table 2 gives the number of iterations, I, and time, T, in seconds to solve this problem using m = 0-10. The parametrized preconditioner results are denoted by P. the number of rows in the plate by a, and the maximum vector length by v.

Table 2. CYBER 203 Iterations and Timings m-step SSOR PCG

	v = 22		v = 41		v = 132		v = 561		v = 1282		v = 2134	
	a = 8		a = 11		a = 20		a = 41		a = 62		a = 80	
m	I	T	I	T	I	T	I	T	I	T	I	T
0	112	.133	157	.213	271	.565	536	3.293	788	11.845	929	22.780
1	52	.129	66	.184	111	.454	214	2.373	311	7.832	395	17.194
2	38	.143	50	.208	79	.478	152	2.428	221	7.773	280	17.380
2P	31	.116	40	.167	61	.369	118	1.885	172	6.052	218	13.534
3	31	.155	39	.216	65	.520	124	2.585	181	8.174	229	18.469
3P	24	.121	30	.167	46	.369	88	1.836	129	5.828	163	13.151
4P	22	.138	24	.166	35	.350	67	1.726	99	5.471	124	12.306
5P	19	.143	20	.167	29	.347	56	1.716	82	5.345	104	12.260
6P	18	.159	18	.175	25	.348	47	1.670	70	5.263	88	12.011
7P					26	.413	43	1.739	64	5.451	80	12.410
8P					21	.375	36	1.634	54	5.139	69	11.985
9P							33	1.660	48	5.056	61	11.731
10P							31	1.709	44	5.070	55	11.594

It should be noted that the inner product routine that was used for these results was developed at Langley and is optimized for the CYBER 203. Several observations can be made from these six test cases.

(1) The parametrized preconditioner is better with respect to both the number of iterations and the execution time than the corresponding unparametrized preconditioner.

(2) The optimal number of steps of the parametrized preconditioner increased as the vector length increased.

In relation to (2), an interesting question is to determine how many steps would be beneficial for a large problem. The answer to this is quite simple if the number of iterations, N_m, could be expressed as a function of m, since the execution time of the m-step method can be expressed as

$$T(m) = N_m(A + mB) \qquad (4.1)$$

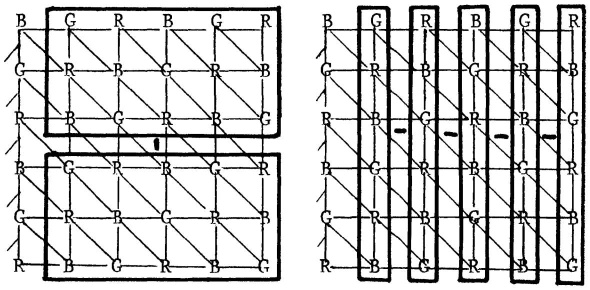
where A is the time for one outer conjugate gradient iteration and B is the time for 1 step of the preconditioner. Now if we assume that $N_{m+1} < N_m$, taking m+1 steps is more beneficial than taking m steps whenever

(1) $(m+1)N_{m+1} - mN_m \leqslant 0$. (This means that the total number of inner loops is less for m+1 steps)

or (2) $$B/A < \frac{N_m - N_{m+1}}{(m+1)N_{m+1} - mN_m} \qquad (4.2)$$

The inequalities in (4.2) explain for larger problems when more steps of the preconditioner should be taken. For instance, the values of the left and right side of inequality (2) when m=9 are (.81,.15), (.68,.5), and (.76,6) for a = 41,62, and 80 respectively. Hence, ten steps are preferable to nine only for a = 80.

We now give the Finite Element Machine results. The example plane stress problem with 6 rows and 6 columns of nodes (60 equations) was solved on a 1, 2 and then on a 5-processor Finite Element Machine using the m-step SSOR PCG method. For this problem the assignment of unconstrained nodes to the processors is shown in Figure 5.



Two Processors Five Processors

Figure 5. FEM Processor Assignments

Observe from Figure 5 that for the two and five processor assignments each processor has an equal number of R, B, and G nodes as well as an equal number of border nodes to be communicated. Therefore, in the absence of communication time and any differences in processor speeds, a speedup of two (five) over the one processor case should be realized.

The number of iterations and the time in seconds for the above assignments are given in Table 3. The speedups for the two and five processor assignments also are included.

Table 3. FEM Iterations, Timings, Speedups m-step SSOR PCG

	p = 1		p = 2			p = 5		
m	I	T	I	T	Speedup	I	T	Speedup
0	48	63.35	48	33.01	1.92	48	17.70	3.58
1	19	47.90	19	25.85	1.85	19	14.85	3.23
2	13	48.75	13	26.65	1.83	13	15.50	3.15
2P	11	41.95	11	22.95	1.83	11	13.30	3.15
3	11	54.95	11	30.15	1.82	11	17.65	3.11
3P	8	41.25	8	22.75	1.81	8	13.25	3.11
4	10	62.40	10	34.30	1.82	10	20.20	3.09
4P	6	39.80	6	22.00	1.81	6	12.90	3.09
5P	5	40.60	5	22.50	1.80	5	13.25	3.06
6P	5	47.05	5	26.20	1.80			

Several observations can be made from Table 3.

(1) The effectiveness of the preconditioner as a function of m was the same for the sequential and two and five processor cases (4p,5p,3p,2p,1,2,3,4).

(2) Taking more than one step of the unparametrized preconditioner was not advantageous.

(3) The overhead for the CG(m=0) algorithm was less than that for the PCG Algorithm because for two and five processors the communications for the preconditioner rather than for the inner products dominate the overhead.

In regard to (3), if we keep the number of nodes per processor fixed and continue to add processors up to a certain number, say n_α, the overhead for the preconditioner will still be more than that for the CG method and hence m = 3P or 2P may become optimal; however, as the number of processors increases beyond n_α, the value of B/A in (4.2) will continue to decrease until $m \geqslant 4p$ steps of the preconditioner will be optimal. The behavior of the m-step PCG Algorithm can be modelled as a function of the number of processors, the problem size, and the relative speed of arithmetic to communication times for the machine. For more details, see Adams [1982].

5. Summary and Conclusions

The m-step multicolor SSOR preconditioned conjugate gradient method described herein has been shown to be effective on vector computers and for a small problem was effective on the Finite Element Machine. As more processors and the sum/max hardware circuit become available on this machine, the method will be tested on larger problems. This method does not face the usual difficulty in choosing the optimal relaxation parameter, ω, for the multicolor SSOR method, since for this ordering and few colors $\omega = 1$ is a good choice, see Adams [1983]. A problem still remains in applying the method to irregular regions since the grid must be colored and for array machines must also be distributed to the processors in light of this coloring.

REFERENCES

Adams, L., Ortega, J. [1982]. "A Multi-Color SOR Method for Parallel Computation," Proceedings 1982 Conference on Parallel Processing, Bellaire, Michigan.

Adams, L. [1982]. "Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers," Ph.D. thesis, University of Virginia (Oct. 1982). Also NASA Contractor Report 166027, NASA Langley Research Center.

Adams, L. [1983]. "M-Step Preconditioned Conjugate Gradient Methods." To appear as an ICASE Report.

Chandra, R. [1978]. "Conjugate Gradient Methods for Partial Differential Equations," Ph.D. thesis, Research Report # 129, Department of Computer Science, Yale University.

Concus, P., Golub, G., O'Leary, D. [1976]. "A Generalized conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations," Sparse Matrix Computations, eds. J. Bunch, D. Rose, Academic Press, pp. 309-332.

Conrad, V., Wallach, Y. [1979]. "Alternating

Methods for Sets of Linear Equations," Numerische Mathematik, Vol. 32, pp. 105-108.

Dubois, P., Greenbaum, A., Rodrique, G. [1979]. "Approximating the Inverse of a Matrix for Use in Iterative Algorithms on Vector Processors," Computing, Vol. 22, pp. 257-268.

Hestenes, M., and Stiefel, E. [1952]. "Methods of Conjugate Gradients for Solving Linear Systems," J. Res. Nat. Bur. Std., pp. 409-436.

Johnson, O., Micchelli, C., Paul, G. [1982]. "Polynominal Preconditioners for Conjugate Gradient Calculations," IBM Research Report 40444, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y.

Jordan, H. [1978]. "A Special Purpose Architecture for Finite Element Analysis," Proc. 1978 Int. Conf. on Par. Proc., pp. 263-266.

Madsen, N., Rodrique, G., Karush, J. [1976]. "Matrix Multiplication by Diagonals on a Vector/Parallel Processor," Information Processing Letters, Vol. 5, No. 2, pp. 41-45.

Norrie, D., DeVries, G. [1978]. An Introduction to Finite Element Analysis, Academic Press, N.Y.

Ortega, J., Voigt, R. [1977]. "Solutions of Partial Differential Equations on Vector Computers," Proc. 1977 Army Num. Anal. Conf., pp. 475-526.

Podsiadlo, D., and Jordan, H. [1981]. "Operating Systems Support for the Finite Element Machine," Computer Science Design Group University of Colorado, Boulder, Colorado.

Reid, J. [1971]. "On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations," Proc. Conf. on Large Sparse Sets of Linear Equations, Academic Press, New York.

Schreiber, R. [1981]. "Implementation of the Conjugate Gradient Method on a Vector Computer," Submitted to SIAM Journal on Scientific and Statistical Computation.

MINIMIZING INNER PRODUCT DATA DEPENDENCIES IN CONJUGATE GRADIENT ITERATION

John Van Rosendale
Institute for Computer Applications in Science and Engineering
Hampton, VA 23665

Abstract

The amount of concurrency available in conjugate gradient interation is limited by the summations required in the inner product computations. The inner product of two vectors of length N requires time c*log(N), if N or more procesors are available.

This paper describes an algebraic restructuring of the conjugate gradient algorithm, which minimizes data dependencies due to inner product calculations. After an initial start up, the new algorithm can perform a conjugate gradient iteration in time c*log(log(N)).

Introduction

Conjugate gradient interation is a method of linear equation solution of great practical importance. It can be used to solve any linear system

$$Au = b$$

where A is symmetric, positive definite, and can be quite efficient when coupled with various preconditioning techniques. However, CG (conjugate gradient) iteration involves the computation of inner products at every iteration. On parallel computers with large numbers of processors, the data dependencies inherent in these inner product calculations will limit the speed of conjugate gradient iteration for large sparse linear systems. See, for example, Schreiber [1981] and Adams [1982]. In fact, given sufficiently many processors, the summation fan-ins in the inner product calculations will dominate the computation time on nearly all large sparse linear systems occurring in practice.

Conjugate Gradient Iteration

This paper presents a solution to this problem through an algebraic restructuring of the CG Algorithm. Consider first the standard CG iteration. One of a number of mathematically equivalent forms of it may be given as follows:

$u^{(0)}$ arbitrary,

$$u^{(n+1)} = u^{(n)} + \lambda_n p^{(n)}, \qquad n=0,1,\cdots,$$

$$p^{(n)} = \begin{array}{ll} r^{(n)}, & n=0, \\ r^{(n)} + \alpha_n p^{(n-1)}, & n=1,2,\cdots, \end{array}$$

$$r^{(n)} = r^{(n-1)} - \lambda_{n-1} Ap^{(n-1)}, \qquad n=1,2,\cdots,$$

$$\alpha_n = \frac{\left(r^{(n)},r^{(n)}\right)}{\left(r^{(n-1)},r^{(n-1)}\right)}, \qquad n=1,2,\cdots,$$

$$\lambda_n = \frac{\left(r^{(n)},r^{(n)}\right)}{\left(p^{(n)},Ap^{(n)}\right)}, \qquad n=0,1,\cdots.$$

The data dependencies here are severe. One cannot generate $\left(r^{(n)},r^{(n)}\right)$ until α_{n-1} and λ_{n-1} are known. But these quantities involve inner products dependent on $r^{(n-1)}$. As pointed out above, an inner product on vectors of length N requires time c*log(N). Thus it would seem that a CG iteration could not be done faster than in time c*log(N).

Idea of New Algorithm

This natural seeming idea, that a CG iteration on vectors of length N cannot be done faster than in time c*log(N), turns out to be incorrect. To see why, consider the computation of a typical inner product required,

$$\left(r^{(n)},r^{(n)}\right).$$

By the formulas above, $r^{(n)}$ is given as

$$r^{(n)} = r^{(n-1)} - \lambda_{n-1} Ap^{(n-1)}.$$

Now suppose we know $r^{(n-1)}$ and $p^{(n-1)}$ but not the value of λ_{n-1}. In this case we would be unable to evaluate $\left(r^{(n)},r^{(n)}\right)$, but we could

Research supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-17070 and NAS1-17130 while the author was in residence at ICASE, NASA langley Research Center, Hampton, VA 23665.

0190-3918/83/0000/0044$01.00 © 1983 IEEE

still perform most of the work involved in evaluating this inner product. Specifically, we can write the recurrence

$$\left(r^{(n)},r^{(n)}\right) = \left(r^{(n-1)},r^{(n-1)}\right)$$

$$- 2\lambda_{n-1}\left(r^{(n-1)},Ap^{(n-1)}\right)$$

$$+ \lambda^2_{n-1}\left(Ap^{(n-1)},Ap^{(n-1)}\right)$$

and can proceed to evaluate all inner products on the right here. If subsequently someone told us the value of λ_{n-1} we could compute the value of $\left(r^{(n)},r^{(n)}\right)$ very rapidly, since only a few more real operations would then be needed to complete evaluation of the recurrence relation.

It is easy to see how this idea can be used to speed the computation of the CG algorithm on parallel computers. We have replaced an inner product computation requiring data not present until iteratin n with inner products of vectors present at interation n-1. Since these vectors are present sooner, we have that much longer to perform their inner products, to achieve the same parallel computation speed. Stated differently, assuming only the inner products limit the speed of the computation, the use of this recurrence relation for $\left(r^{(n)},r^{(n)}\right)$ and the analogous relation for $\left(p^{(n)},Ap^{(n)}\right)$ will approximately double the parallel speed of CG iteration, where it is assumed that sufficiently many processors are available, and communications cost can be neglected.

Recurrence Relations

The recurrence relation just described is one of a large class of such relations which can be exploited to speed up CG iteration. These relations will be given in detail in Van Rosendale [1983], but for now we consider only the general form of such recurrence relations. Consider the typical inner product:

$$\left(r^{(n)},r^{(n)}\right)$$

The value of this inner product may be given in terms of the values of inner products of vectors occurring at any previous iteration together with the values of the real parameters

$$\alpha_{n-1},\alpha_{n-2},\cdots,$$
$$\lambda_{n-1},\lambda_{n-2},\cdots.$$

For example, for any $k > 0$, one can derive recurrence relations of the form

$$\left(r^{(n)},r^{(n)}\right) = \sum_{i=0}^{2k} a_i\left(r^{(n-k)},A^i r^{(n-k)}\right)$$

$$+ \sum_{i=0}^{2k} b_i\left(r^{(n-k)},A^i p^{(n-k)}\right) \qquad (*)$$

$$+ \sum_{i=0}^{2k} c_i\left(p^{(n-k)},A^i p^{(n-k)}\right).$$

The coefficients $\{a_i\}$, $\{b_i\}$, $\{c_i\}$ occurring here are polynomials in the parameters

$$\{\alpha_{n-1},\alpha_{n-2},\cdots,\alpha_{n-k},\lambda_{n-1},\lambda_{n-2},\cdots,\lambda_{n-k}\}.$$

Similar recurrence relations are available for the other type of inner product occurring in CG iteration, $\left(p^{(n)},Ap^{(n)}\right)$.

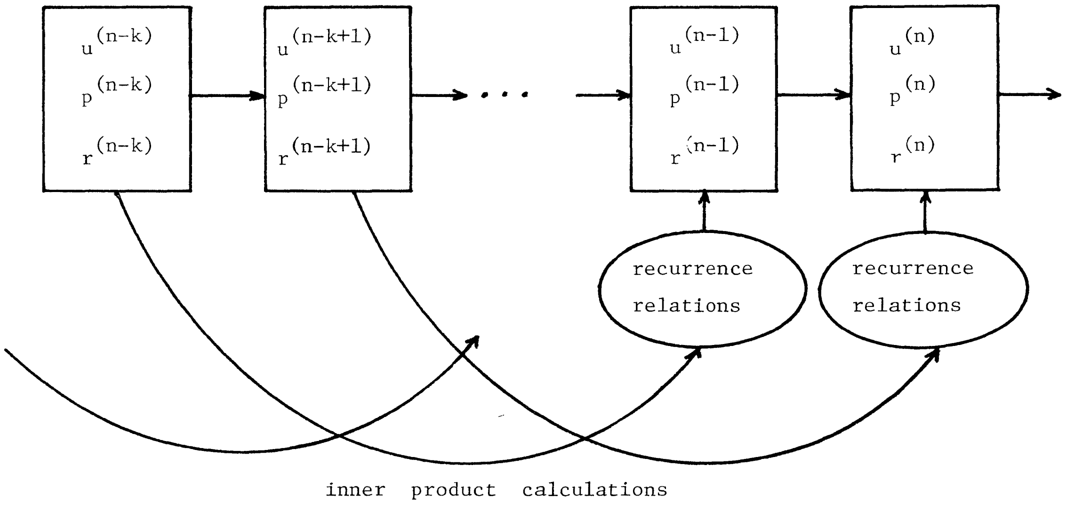


Figure 1. Principal Data Movement in New CG Algorithm.

New Algorithm

To construct a more parallel variant of CG iteration based on these recurrence relations, one begins by selecting a value for the constant k, which may be thought of as a look-ahead parameter. Then at iteration n - k, when vectors $r^{(n-k)}$ and $p^{(n-k)}$ become available one begins forming all of the inner products

$$\left(r^{(n-k)}, A^i r^{(n-k)}\right), \quad i=0,1,\cdots,2k,$$

$$\left(r^{(n-k)}, A^i p^{(n-k)}\right), \quad i=0,1,\cdots,2k,$$

$$\left(p^{(n-k)}, A^i p^{(n-k)}\right), \quad i=0,1,\cdots,2k.$$

The values of these inner products are needed in the recurrence relations for the inner products

$$\left(r^{(n)}, r^{(n)}\right), \ \left(p^{(n)}, Ap^{(n)}\right)$$

at iteration n. Thus we arrive at an algorithm whose data movements are sketched in Figure 1.

Clearly the problems of the delays caused by the summations in the inner products is now solved. If we chose $k = \log(N)$, the inner product summation delays will have no inpact on algorithm speed. However, two new issues now arise. First, we have not dealt with the way in which the parameters

$$\{\alpha_{n-1}, \alpha_{n-2}, \alpha_{n-k}, \cdots \lambda_{n-1}, \lambda_{n-2}, \cdots \lambda_{n-k}\}$$

enter into the recurrence relations. In principle, there could be severe data dependencies here. Second, there seem to be a large number of inner products required now, most involving a relatively high power of the matrix A.

Neither of these problems is as serious as it first appears. For the first, it turns out the coefficients $\{a_i\}\{\alpha_i\}\{c_i\}$ in the recurrence relations above are polynomials in the parameters

$$\{\alpha_{n-1}, \alpha_{n-2}, \cdots \alpha_{n-k}, \lambda_{n-1}, \lambda_{n-2}, \cdots \lambda_{n-k}\}$$

which are at most quadratic in each parameter separately. This fact, coupled with the observation that the parameters

$$\alpha_{n-k}, \alpha_{n-k+1}, \cdots, \lambda_{n-k}, \lambda_{n-k+1}, \cdots$$

gradually become available, enables us to effectively perform the coefficient evaluations in a pipelined fashion. Thus at iteration n, when we need the inner product $\left(r^{(n)}, r^{(n)}\right)$, we can have the recurrence relation (*) completely evaluated, except for performing these summations, or the analogous summations in the recurrence for $\left(p^{(n)}, Ap^{(n)}\right)$. This requires parallel time

$$\log(k) = \log(\log(N)).$$

The second problem mentioned above, the occurence of high powers of the matrix A in the recurrence realtion (*), can be resolved by the use of additonal recurrence relations. First, observe that there is no need to compute powers of the matrix A, since we have the recurrences:

$$A^i r^{(n)} = A^i r^{(n-1)} - \lambda_{n-1} A^{i+1} p^{(n-1)},$$

$$A^i p^{(n)} = A^i r^{(n)} + \alpha_n A^i p^{(n-1)}.$$

Thus the set of vectors $\{A^i p^{(n)}\}_{i=0}^k$ and $\{A^i r^{(n)}\}_{i=0}^k$ can be updated with only one matrix vector product.

Next observe that nearly all of the inner products needed can also be obtained by recurrences. We have

$$\left(r^{(n)}, A^i r^{(n)}\right) = \left(r^{(n-1)}, A^i r^{(n-1)}\right)$$
$$- 2\lambda_{n-1}\left(r^{(n-1)}, A^{i+1} p^{(n-1)}\right)$$
$$+ \lambda_{n-1}^2\left(p^{(n-1)}, A^{i+2} p^{(n-1)}\right),$$

and similar recurrences for the other types of inner products occurring in relation (*). Given the values of the inner products

$$\{r^{(n)}, A^i r^{(n)}\}_{i=0}^{2k},$$

$$\{r^{(n)}, A^i p^{(n)}\}_{i=0}^{2k},$$

$$\{p^{(n)}, A^i p^{(n)}\}_{i=0}^{2k},$$

at iteration n, we can obtain nearly all of the inner products needed at iteration n+1. Only two inner products need to be computed directly.

Computational Complexity

As pointed out above, the summations in the recurrence relations (*) require time

$$\log(k) = \log(\log(N)).$$

Thus if matrix A has at most d nonzeroes per row or column, this algorithm requires parallel time

$$\max\left(\log(d), \log(\log(N))\right).$$

The sequential complexity of this algorithm is essentially the same as that of the usual CG algorithm; we still need two inner products and a matrix vector product at every iteration.

REFERENCES

Adams, L. [1982]. "Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers," NASA Contractor Report 166027, NASA Langley Research Center.

Schreiber, R. [1981]. "Implementation of the Conjugate Gradient Method on a Vector Computer," submitted SIAM J. Sci. Statist. Comput..

NEW MATRIX EQUATION SOLVERS IN GF(2) EMPLOYING CRAMER WITH CHIO METHOD

Yoshiyasu TAKEFUJI, Takakazu KUROKAWA,
Masato ISHIZAKI, and Hideo AISO

Department of E.E. KEIO University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223 JAPAN

ABSTRACT

In our former paper [6] a parallel and pipelined fast matrix equation solver employing the conventional Gauss Jordan Elimination Method in GF(2) has been proposed where the elements are 0s or 1s. In this paper two new solvers employing Cramer with Chio method [2] are proposed which are more suitable for VLSI implementation. The new solvers have a much more flexible expandability toward the increase of the matrix size. $O(n^4)$ gates are required for realizing an n-pyramid solver through solving

$$A\mathbf{X} = \mathbf{b}$$

where A is a regular matrix of n*n, $\mathbf{X}$ and $\mathbf{b}$ are vectors respectively. The solvers can be applied to the real-time decryption [1][3][4][5], hashing, error correction/detection [1], and so on.

1. INTRODUCTION

In our former paper [6] an ultra high speed solver for the regular matrix equations in GF(2)* has been proposed where the elements are 0s or 1s. The parallel and pipelined solver composed of the iterative logic circuits which are suitable for VLSI implementation can be realized by employing the conventional Gauss Jordan Elimination Method [6]. However, the solver has a drawback on flexibility in expanding hardware logic circuits according to the increase of the matrix size [6].

In this paper two new solvers employing Cramer with Chio Method are proposed which can overcome the drawback. The design of the new solvers is discussed from the viewpoints of the total number of gates and that of gate stages for computation. The proposed solvers can be applied to the decryption of encrypted codes [3], and hashing, and so on.

In order to decrypt a polynomial from multiresidue polynomials in GF(2), a regular matrix equation

$$A\mathbf{X} = \mathbf{b}$$

has to be solved where A is a regular matrix of n*n, $\mathbf{X}$ and $\mathbf{b}$ are vectors respectively. In the next section an example of Cramer with Chio method is described.

* Note that GF(2) means Galois Fields 2.

2. CRAMER WITH CHIO METHOD

Cramer with Chio method [2] can be used for solving the regular matrix equation

$$A\mathbf{X} = \mathbf{b}$$

where A is a regular matrix of n*n, $\mathbf{X}$ is a vector of

$$X = (x_1 x_2 x_3 \cdots x_n)^t$$

and $\mathbf{b}$ is also a vector of

$$\mathbf{b} = (b_1 b_2 b_3 \cdots b_n)^t.$$

It is believed that the Cramer scheme is unsuitable for the large size of matrices because the number of computations becomes too large. When Cramer with Chio method in GF(2) is employed for solving

$$A\mathbf{X} = \mathbf{b},$$

the Cramer scheme is attractive because the number of computations can be drastically reduced by Chio method ; $O(n!) \rightarrow O(n^4)$, and the basic operations become simple in GF(2); multiplication and addition correspond to AND and Exclusive OR functions in GF(2) respectively.

A simple example of Cramer with Chio method is shown as follows:

< example 1 >

Find $X = (x_1 x_2 x_3)$ where

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Solution:

$$x_1 = \begin{vmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} = 1$$

$$x_2 = \begin{vmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} \overset{*}{=} \begin{vmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} = 1$$

$$x_3 = \begin{vmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{vmatrix} \overset{*}{=} \begin{vmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 1 \\ 0 & 0 \end{vmatrix} = 0$$

* column exchange

0190-3918/83/0000/0047$01.00 © 1983 IEEE

3. 3-DIMENSIONAL MATRIX EQUATION SOLVER

A 3-dimensional matrix equation solver employing Cramer with Chio method is proposed in this section. The 3-dimensional parallel and pipelined solver for solving

$$A\mathbb{X} = \mathbb{b}$$

where A is a regular matrix of n∗n is composed of n pyramids as shown in Fig.1. A pyramid which can produce one element of vector $\mathbf{x}$ consists of n-stage pipes. A single pipe of the ith stage shown in Fig.2 is composed of a panel of k∗k D Flip Flops where k is n-i+1, a 1-detector, a column exchanger, and a logic unit shown in Fig.3-6 respectively. The 1-detector in the ith stage detects the leftmost element to be 1 and produces the row number of the element for exchanging the leftmost column with the column of the row number. The row number corresponds to the output of case i in Fig.4. The column exchanger exchanges the required two columns. The logic unit described in Fig.6 calculates the arithmetic operations in GF(2). In GF(2) the addition and subtraction corresponds to the function of Exclusive OR logic and the multiplication to that of AND logic. The division corresponds to no-operation when the divisor is 1. In order to find x_l for

$$x_l = \begin{pmatrix} a_{11} & a_{12} \cdots & b_1 \cdots & a_{1n} \\ a_{21} & a_{22} \cdots & b_2 \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} \cdots & \underset{\substack{| \\ l\text{ th. column}}}{b_n} \cdots & a_{nn} \end{pmatrix}$$

the required data flows from bottom to top through a pyramid and the x_l can be obtained as shown in Fig.1. The proposed solver is suitable for VLSI implementation because of a high regularity of iterative logic circuits. The solver has a flexible expandability to the increase of a matrix size. The solver for solving a regular matrix equation of

$$A\mathbb{X} = \mathbb{b}$$

where A is a regular matrix of (n+1)∗(n+1), can be easily realized by appending the (n+1)th pipe to the bottom of a pyramid composed of n-stage pipes.

The total number of required gates for realizing a matrix solver to solve

$$A\mathbb{X} = \mathbb{b}$$

where A is a regular matrix of n∗n, is shown in Table 1 and Fig.7. The total number of required gates and that of required gate stage for solving a regular matrix equation are described in Fig.7. $O(n^4)$ gates are required for realizing a n-pyramid matrix solver where n is the size of vector $\mathbb{X}$.

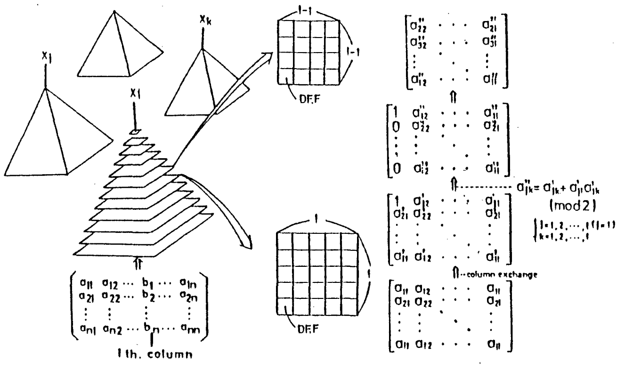


Fig.1 Cramer with Chio method

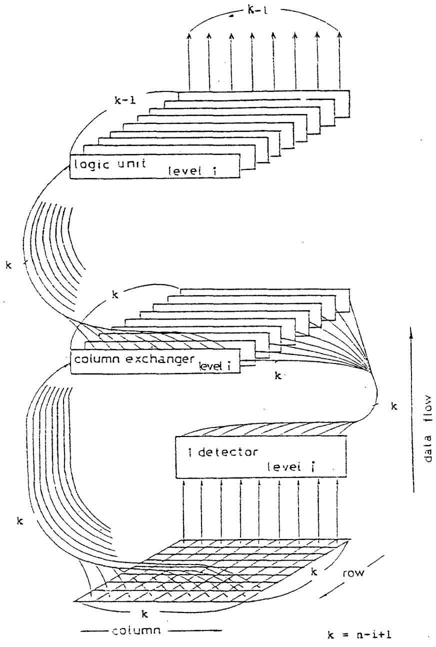


Fig.2 A single pipe of the ith stage

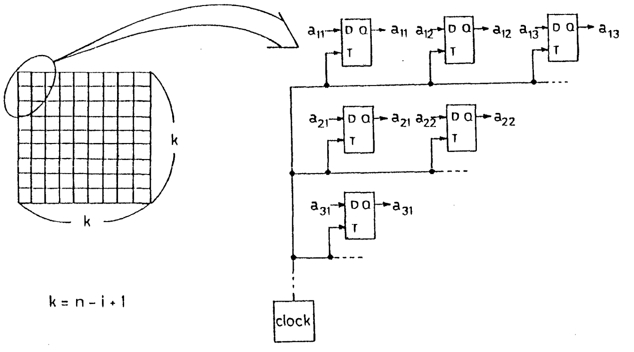


Fig.3 A panel of D Flip Flops of the ith stage

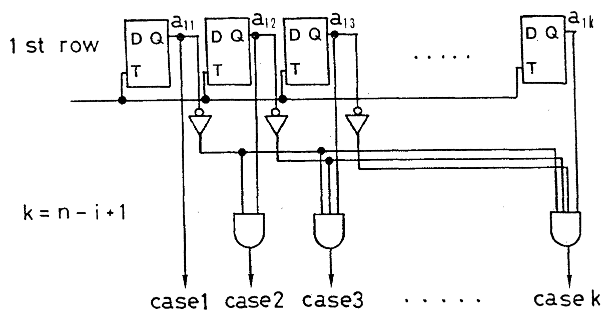
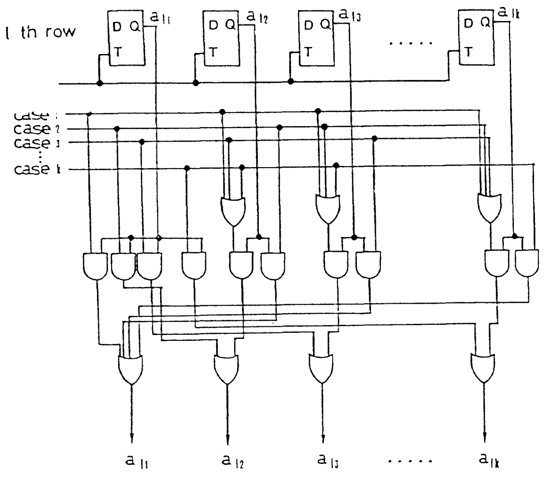


Fig.4 A 1-detector of the ith stage



$k = n - i + 1$
$l = 1, 2, \cdots k$

Fig.5 A column exchanger of the ith stage

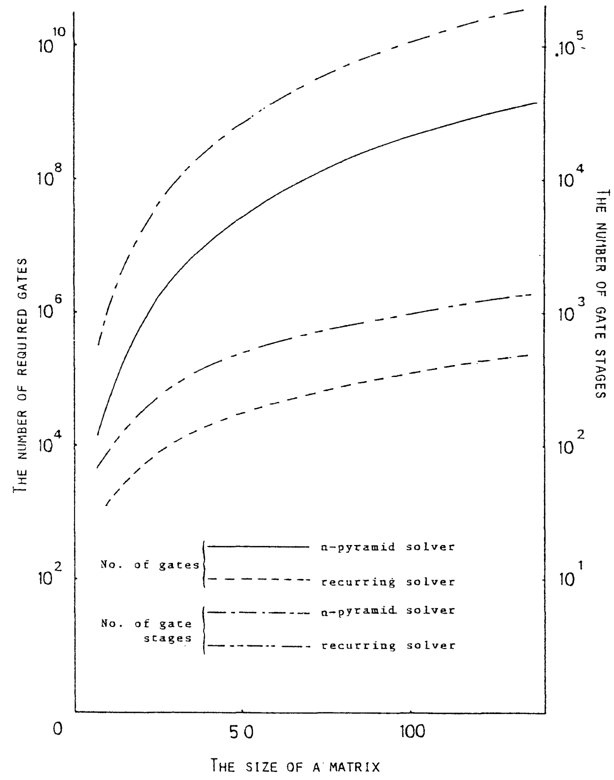
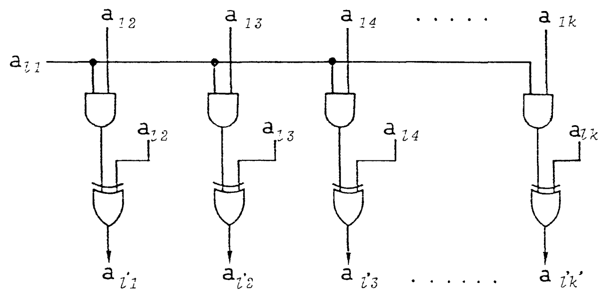


Fig.7 Evaluation of the matrix solvers



a_{l2} a_{l3} a_{l4} ..... a_{lk}
a_{l1}
a_{l2}
a_{l3}
a_{l4}
a_{lk}
$a'_{l'1}$ $a'_{l'2}$ $a'_{l'3}$ ...... $a'_{l'k'}$

Fig.6 A logic unit of the ith stage

$k = n - i + 1$
$k' = k - 1$
$l = 2, 3, \cdots k$
$l' = l - 1$

Table 1 The number of required gates of the matrix solver

	AND	OR	NOT	XOR	DFF
PANEL OF DFF	0	0	0	0	$(n - i + 1)^2$
1 - DETECTOR	$n - i$	0	$n - i$	0	0
COLUMN EXCHANGER	$(3n-3i+1)(n-i+1)$	$(2n-2i+1)(n-i+1)$	0	0	0
LOGIC UNIT	$(n - i)^2$	0	0	$(n - i)^2$	0
TOTAL	$4i^2 - (8n+5)i$ $+ (4n^2+5n+1)$	$2i^2 - (4n+3)i$ $+ (2n^2+3n+1)$	$n - i$	$i^2 - 2ni + n^2$	$i^2 - 2(n+1)i$ $+ (n^2+2n+1)$

AMOUNT OF WHOLE STAGES

The number of required gates $= \frac{13}{3} n^4 + 2n^3 + \frac{1}{3} n^2 - 6n$

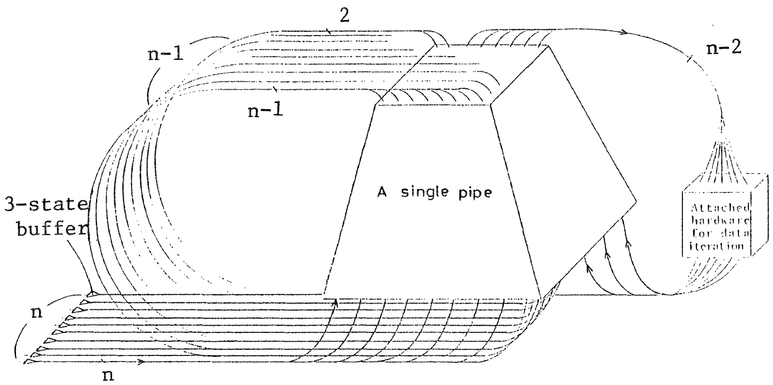


Fig.8 A recurring matrix solver employing a pipe

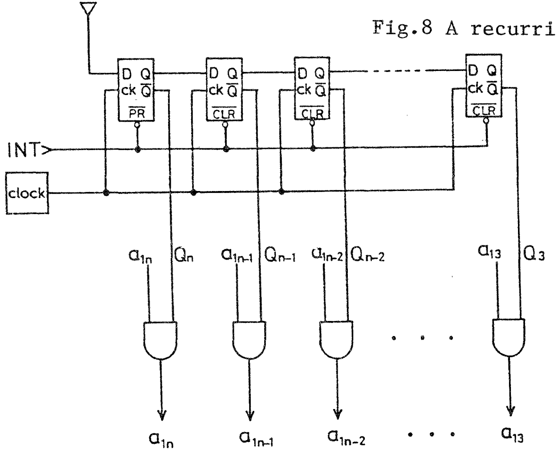


Fig.9 Attached hardware for data iteration

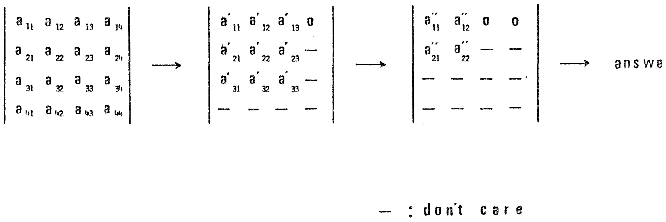


Fig.10 An example of masking

4. RECURRING MATRIX SOLVER

A simplified recurring matrix solver is proposed in this section. The recurring solver for solving

$$A\,\mathbf{X} = \mathbf{b}$$

where A is a regular matrix of n×n, is composed of a single pipe together with an attached hardware shown in Fig.8. Once the required data is set, the elements circulate from bottom to top where the attached hardware synchronously masks the needless elements for calculation. An example of masking is shown in Fig.10.

The attached hardware is composed of an (n-1)-stage D Flip Flop circuit and AND-logic circuits shown in Fig.9. Initial seed of the D Flip Flop circuit is

$$(\overline{Q}_n\,\overline{Q}_{n-1}\overline{Q}_{n-2} \cdots \overline{Q}_1\ ) = (0\ 1\ 1 \cdots 1),$$

and in the next clock

$$(\overline{Q}_n\,\overline{Q}_{n-1}\overline{Q}_{n-2} \cdots \overline{Q}_1\ ) = (0\ 0\ 1 \cdots 1).$$

The recurring matrix solver can drastically reduce the amount of required hardware for solving a regular matrix equation as shown in Fig.7. $O(n^2)$ gates are required for realizing a recurring matrix solver where n is the size of vector $\mathbf{X}$. The number of required gate-stage is also shown in Fig.7.

5. CONCLUSION

The matrix equation in GF(2) should be solved in various application fields [6]. Two new matrix equation solvers in GF(2) employing Cramer with Chio method were proposed. The one is an n-pyramid solver where n is the size of vector $\mathbf{X}$. A pyramid is composed of n-stage pipes. The other is a recurring matrix solver employing only a single pipe for saving the hardware amount. The total number of required gates for realizing the solvers and that of gate-stage are discussed. The proposed solvers can be implemented by 3-dimensional VLSI circuits because of a high regularity of the iterative logic. The solvers have a flexible expandability to the increase of the matrix size.

6. REFERENCES

[1] Nicholas S. Szabo, Richard Tanaka, "Residue Arithmetic and its Applications to computer technology," McGraw Hill (1967).

[2] Louis A. Pipes, Shahen A. Hovanessian, "Matrix Computer Methods in Engineering," John Wiley & Sonns, Inc. (1969).

[3] Yoshiyasu Takefuji, Koichiro Tsujino, Mari Ibuki, and Hideo Aiso, "A NOVEL APPROACH TO PARALLEL PROCESSING CRYPTOSYSTEM," Proc. of ICPP (1982).

[4] Yoshiyasu Takefuji, PhD Dissertation, "A STUDY OF FAULT- TOLERANT HARDWARE," (1983).

[5] Takakazu Kurokawa, BS paper, "A proposal of multiresidue codes in polynomial ring applied to cipher codes," (1983).

[6] Yoshiyasu Takefuji, Takakazu Kurokawa, and Hideo Aiso, "FAST MATRIX SOLVER in GF(2)," Proc. of Computer Arithmetic (1983).

SPECIFICATION AND IMPLEMENTATION OF AN INTEGRATED PACKET COMMUNICATION FACILITY FOR AN ARRAY COMPUTER

Bharat Deep Rathi, Sanjay Deshpande, Matthew Sejnowski, Don Walker*
Roy Jenevein**, G. J. Lipovski and J. C. Browne*

Departments of Electrical Engineering and *Computer Science
University of Texas at Austin, Texas
**Department of Computer Science
University of New Orleans

ABSTRACT

Four distinct packet communication requirements for network architectured computer systems are: system control, dataflow data type movement, SIMD, data realignment and movement of high volume data between MIMD configurations when memory sharing is unavailable or too costly. This paper defines and describes a packet switching mechanism which meets each of these requirements. Mechanisms are also defined and described for breaking and restoring SIMD execution structures which are required to complete the implementation of packet switching for SIMD execution. The mechanisms were defined and are described in the context of the Texas Reconfigurable Array Computer (TRAC), but should be in large measure adaptable to other network architectured systems.

1.0 PROBLEM STATEMENT AND OVERVIEW

A computer system incorporating a multistage netowrk can utilize either packet switching or circuit switching or both. The choice of modes may depend upon other architectural factors, such as whether the network couples processing elements to memories or processors to processors, or whether the system model of computation is SIMD or MIMD, and also may depend upon the selection of problems the system is intended to execute.

The Texas Reconfigurable Array Computer (TRAC) [SEJ80] uses its network to create configurations by coupling processing elements to memories to form processors and then coupling processors to form SIMD tasks, if desired [BRO82]. Tasks, whether SISD or SIMD, run independently from each other in MIMD fashion. TRAC is thus capable of SIMD and MIMD models of computation. The varistructure capabilities [LIP77] of the architecture are implemented through carry look-ahead techniques which are similar to SIMD synchronization techniques. The breadth of representation capability of TRAC has forced us into a thorough analysis of the requirements of network architectured systems for the packet switching mode of network communication. This paper reports the results of this study.

There are four distinct communication requirements for network architectured systems which can be met by appropriate packet switching facilities. These are:

1. system control (both inter and intra-task communication)
2. data movement for dataflow models of computation (inter-task communication)
3. data realignment for SIMD processing of arrays (intra-task communication) and
4. movement of data between MIMD configurations when either circuit switching is not provided, or when the switch configuration required for establishment of circuits, needed to link specific process/data pairs, cannot be realized or realized only with excessive reorganization cost (inter-task communication)

All of these communication requirements have been previously recognized and discussed in the literature. This paper defines a coherent integrated implementation of packet switching which serves all of these requirements. Some of the mechanisms and implementation schemes have been previously reported [TRI79,PRE79]. The overlap with this integrated discussion will be noted in the text at the appropriate places.

Commonly, packets will be sent to some subset of the processors in an SIMD task. The selection of packet switching as a mode of communication thus requires an efficient mechanism for desynchronizing and resynchronizing SIMD tasks. Two such mechanisms are implemented as a part of the communication functionality for TRAC. This capability is particularly significant for TRAC since tasks of data width greater than a single byte may be constructed through SIMD synchronization of single byte wide processors. However, both should be adaptable to other architectures.

The next section states the requirements in the context of the TRAC architecture. Section 3 gives an overview of the packet switching capabilities and their implementation. Section 4 defines the desynchronizing and resynchronizing functions and their implementation. Section 5 describes the implementation scheme including software requirements. Section 6 analyzes the design space for implementation and justifies the selections.

2.0 REQUIREMENTS FOR PACKET SWITCHING FUNCTIONALITY IN TRAC

Although TRAC has been previously described [SEJ80,PRE79] we review the background for packet

0190-3918/83/0000/0051$01.00 © 1983 IEEE

communication. We emphasize that software design and application studies have been conducted in parallel and sometimes in advance of definition and implementation of architectural features. The integrated packet communication facilities are an example of the interplay of requirements and architecture.

TRAC uses circuit switching to establish task spaces (partitions) of resources conforming to some desired model of computation. The tasks may be SIMD or SISD. Partitions run independently of each other in MIMD fashion. A circuit switching mode of communication and data movement is available within a task space. Circuits are used as broadcast buses for instructions and carry linkage. They are also used to implement explicit sharing of memory between processors, since any one of the processors can explicitly detach a memory, and another can explicitly select this memory in a single instruction. Reconfiguration of the network can also be used to move memory units and thus data between configurations. There would, however, be many restrictions on possible usage of the system if these circuit switch functions were all that were available for interprocess communication. System control functions could be executed only by extensive reconfiguration. It would be awkward to implement the dataflow model of computation, for more than a small number of processors (the number which can effectively communicate through a single switchable memory unit). Data realignment between phases of SIMD computations (such as transposing a matrix) would have to be serialized. Network blocking would restrict the set of configurations which could realize sharing among configurations and be useful for MIMD processing. On the other hand, packets provide complete processor-processor communication capability. However, the disadvantage of packet mechanism is that the packets arrive at the receiving process at a non-deterministic time, thereby forcing a set of receiving processors out of synchronisms, thus requiriing resynchronism overhead.

Each of these functions requires somewhat different capabilities from the packet communication mechanism. System control requires small interrupting packets. Dataflow and realignment requires carrying of addresses together with data. General data movement between MIMD configurations requires efficient transmission of large volumes of data.

The mechanisms described in succeeding sections integrate implementation of all of these requirements. The succeeding discussion clarifies the mechanisms as inter- or intra-task depending on their functions.

3.0 OVERVIEW OF TRAC'S PACKET SUPPORT

On TRAC each task is assigned its own task space of processors, memories and I/O devices that are interconnected as required. One processor of each task space is called its task-head. This processor is chosen during task set up time and implements the required synchronization, security, authorization mechanism and other operating system functions needed to support inter/intra task packet communication. When any task executes, the processors assigned to it execute in SIMD mode (in lock step). Therefore the processors assigned to this task are simultaneously informed about any intra-task communication. But for any inter-task packet communication, a sending task needs to inform the receiving task of its desire to communicate. This is done by the use of an inter-task protocol implemented strictly through interrupting packets (described below).

On TRAC, packets can be transmitted within a task space and between task spaces. The packet network provides full interconnection between the 'w' processors in the system. The SW-banyan network provides this communication using less than w**2 links; this implies that links must be shared. Transmission delays dependent on traffic patterns may be introduced by blocking in the network.

The addition of a packet switching (store and forward) network increases the switch cost only marginally. This is due to the fact that the switch can be time multiplexed. The time slice devoted to packet forwarding on the switch corresponds to the period where primary memory is executing a read or a write cycle (i.e. when it cannot receive data nor send data on the circuit switch "bus"). The extra hardware needed consists of some control circuitry, and buffers to provide the store and forward function for the packet network. Thus this hardware requires some additional logic (chip area) which is relatively inexpensive. Further it does not require any additional pins, since the pins already used for circuit switching are data multiplexed.

A packet consists of an address header and a number of data words. The addressing scheme for directing the packet movement in the banyan network is the one suggested by Tripathi and Lipovski [TRI79]. An improved method has been suggested by Siegel and McMillen [SIE81]. A processor Pi transmits a packet to another processor Pj by first loading relevant data into a packet generating unit, in a designated primary memory module in its private memory ensemble. As links and nodes become available the packet proceeds towards the target processor along a unique path. Some form of arbitration is provided at the switch nodes to resolve any packet conflicts. The details of this store and forward hardware on TRAC are given in [SEJ81].

The inter- and intra-task packet communication use this packet network. The primary difference between intra- and inter-task packets is the amount of context switching required for their reception at a target processor. Packets which cause interrupts upon reception are called interrupting packets, while packets which are explicitly sent and explicitly received are called mapping packets.

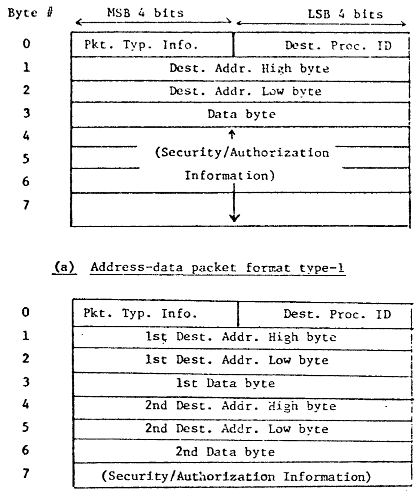
Interrupting packets are used to transmit small amounts of control information. Mapping packets are used when movement of substantial amounts of data is required, but a small amount of controlling data movement is needed to initiate the activity. These two types of packets use the packet network at different times (i.e., the network is time-multiplexed). Therefore, in effect we have an interrupting packet channel and a mapping packet channel.

Inter-task mapping packet communication is initiated by following a "request-acknowledge" protocol. This protocol requires that the sending task's task head send an interrupting packet to the receiving task's task head. This packet will indicate the desire for multi-byte wide data communication between these two tasks (since there are multiple processors in either task). On receiving such an interrupting packet, the receiving task head is interrupted along with all other processors of the receiving task. This task head then reviews and validates this interrupting packet and sends an "acknowledge" packet to the requesting task head. It then informs other processors in the receiving task to execute a "receive MAP" instruction. The sending task's task head receives the acknowledge and instructs its task's processors to execute a "send MAP" instruction. These "MAP" instructions are executed by microcode in the processors. Both instructions have a parameter, which is the number of bytes to be sent or received. After the multi-byte wide data communication is initiated, the processors of the sending and receiving tasks operate at their own pace. A processor sends a packet whenever its packet input port is free, and/or receives one when it arrives. This communication is terminated when the send/receive counts in the respective tasks' processors reach zero. Resynchronization to SIMD mode is needed because each processor can finish its MAP instruction at different times (due to blocking in the network). This synchronization of the processors is achieved by mechanisms defined later.

Intra-task mapping packet communication does not require this "request-acknowledge" protocol. This is because the processors of the task are operating in lock step and therefore initiate this communication together. This is done by initiating the execution of a "send MAP/receive MAP" instruction. This instruction handles the required sending and receiving of mapping packets. It executes in a similar manner as the "send/receive MAP" instructions described above.

To support dataflow implementation on TRAC, we might need multiple tasks to send to a single receiving task simultaneously. Since we cannot ensure that these sending tasks will enter the MAP function at the same time, we need to implement the MAP instructions so that they can be interrupted by another sending task wishing to join the MAP function. This allows the sending tasks to join the MAP function "asynchronously" (they still have to follow the "request-acknowledge" protocol described preceding). The advantage gained by this scheme is that the sending tasks are made to wait for the least amount of time (time required for a transmit acknowledge). The two other schemes are : 1) to completely serialize the senders in the request order, and 2) to enable transmission (from all the senders) only after the last sender has requested transmit permission. The disadvantage with the first scheme is that it does not allow parallel communication and good dataflow implementation. The disadvantage in the second alternative scheme is that new senders have to wait upon the last sender. Another possible problem with this scheme is that the senders may have to wait indefinitely, if any one of them does not join the MAP request.

TRAC packets are in fact trains of 1 byte wide words and 8 bytes long. There are two basic formats for mapping packets. These formats (Fig. 1 and 2) allow us to communicate information for all the applications cited above. In the first format (called an address/data packet format) the memory address in the destination processor, where the data will be stored, always accompanies the data. Here we can either send one byte of data (Fig. 1(a)), or two bytes of data per packet (Fig. 1(b)). If we send



(a) Address-data packet format type-1

(b) Address-data packet format type-2

Figure 1: Address-Data Packet Formats

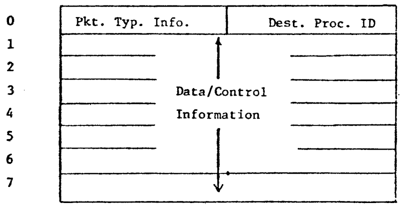


Figure 2: Data/Control Information Packet Format

only one byte of data per packet, then the remaining three bytes in the packet can contain any security/authorization information if needed. In the second format (Fig. 2) the packet contains 6 bytes of data/control information. The microcode in the receiving task implicitly knows the destination address for placing the packet data, or it is informed of this address prior to receiving this information. The microcode need not be aware of the contents of the packet data. This information is analyzed by the local higher level software construct. The format of Fig. 2 is also used for interrupting packets.

There is the need for the simultaneous use of both packet formats for mapping packets on TRAC. The address-data format of Fig. 1 looks attractive for supporting dataflow algorithms and data realignment in SIMD tasks. While the data/control packet format of Fig. 2 may be more suitable for transferring "large" amounts of data between tasks. Looking at the two formats we see that the formats in Fig. 1 are special cases of the format in Fig. 2. The two formats must be distinguished because these two formats require different microcode (or hardware) for handling them.

Some information will be needed in each packet to indicate its format. The first byte of a packet contains 4 bits of destination processor ID, which is required to route the packet through the banyan to one of the sixteen processors (in the current configuration of TRAC). The other 4 bits of this byte are used to indicate the packet type information required to indicate the packet's format to the microcode. This also facilitates the reception of multi-typed packets concurrently. For example tasks A and B send data simultaneously to task C using mapping packets. Then it is possible that task A sends packets of the type shown in Fig. 1(a) and task B sends packets of the type shown in Fig. 2. The microcode is thus required to recognize the packet type for each packet received.

4.0 RESYNCHRONIZATION MECHANISMS

The protocols for executing packet transfers require the use of the desynchronizing and resynchronizing mechanisms. The processors of a SIMD task are desynchronized when a task is broken into subtasks, or during exception handling, or during the execution of an instruction that requires the processors of a task to operate independently (e.g., "MAP" instruction execution). There are two resynchronization mechanisms. One is intended to reassemble the sub-tasks into the original task structure when all the sub-tasks have terminated. The second is designed for use when a single processor requires to interrupt the synchronous execution of a single task. This mechanism can also be used to reassemble the original task structure. Both of these resynchronization mechanisms are supported in hardware.

4.1 Resynchronization Mechanism - A

This mechanism is used to synchronize the sub-tasks into their original task, when the sub-tasks have terminated. It is used only when the circuit switched path linking the processors (called the instruction tree) of the task has been broken to create the sub-tasks. The instruction tree is a tree-shaped broadcast path, rooted at a memory module, and can be recreated by a single instruction. The logic elements of the instruction tree of interest here are illustrated in Figure 3.

After breaking a task into sub-tasks, the task head processor places the count of the number of processors to be resynchronized in the synchronization register in the memory module at the root of the instruction tree. After finishing asynchronous processing each processor acquires this memory module and decrements the count by 1. Mutual exclusion for such an access is provided by hardware in the interconnection network. If the count is non-zero the processor simply activates the part of the instruction tree that links it to the memory module at the root. It then executes an arithmetic operation so that it asserts a true propagate, but does not produce a generate in the carry-lookahead logic that is part of the instruction tree (Fig. 3). The processor then waits for the incoming carry to become true. The last processor to acquire the shared memory module sees a zero count in the synchronization register. It creates its branch of the instruction tree and then executes an arithmetic operation to assert the generate signal. All the processors of the original task see a carry due to this operation. This indicates that the original instruction tree has been recreated and that the processors are resynchronized.

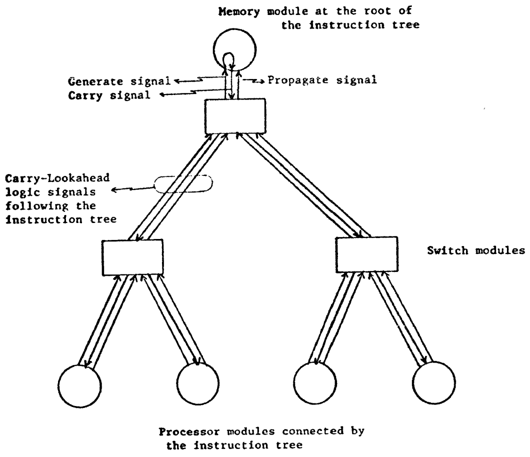

Memory module at the root of the instruction tree
Generate signal
Carry signal
Propagate signal
Carry-Lookahead logic signals following the instruction tree
Switch modules
Processor modules connected by the instruction tree

Figure 3 Resynchronization Mechanism-1

(Only the carry-lookahead signals following the instruction tree are shown here.)

4.2 Resynchronization Mechanism - B

The following hardware mechanism is used when a single processor wants to communicate an asynchronous event to the other processors of the desynchronized tasks. It can also be used to bring the processors back in synchrony. This mechanism can be used when the instruction tree is active or has been deactivated.

When an asynchronous event which has to be communicated to the rest of the processors occurs in any one of the processors of a task, the processor recognizing the event, asserts a signal-A via a tree shaped single line (see Figures 4 and 5). This signal goes down to the root of the tree over line-A, gets turned around and comes back up along the broadcast tree to all the task's processors over line-B. This signal is recorded in all the processors, including the asserting processor, by setting of the SYNC flip-flop.

As each processor completes its current "atomic" operation it tests this flip-flop. If it is set, it recognizes that an asynchronous event has occurred and that it needs to get back into lock step with its companion task processors. If the SYNC flip-flop is not set, it continues to execute the next "atomic" operation independently. On recognizing the occurrence of an asynchronous event it clears the SYNC flip-flop and waits for a wire-AND line (Fig. 5) to become TRUE. This line connects all the processors in a task.

While in an asynchronous mode of operation the inverted output of the SYNC flip-flop is fed onto this wire-AND line. Before entering the asynchronous mode this flip-flop is cleared and the wire-AND line is set TRUE. Whenever any of the SYNC flip-flops attached to this wire-AND line is set, this line is set FALSE. Thus after the asynchronous event has occurred and this wire-AND line is TRUE, it is known that the processors once again are in lock step. They then proceed ahead in lock step to service the event.

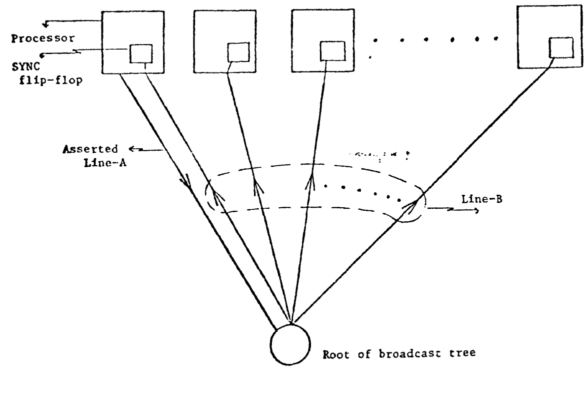


Figure 4: Resynchronization Hardware Mechanism

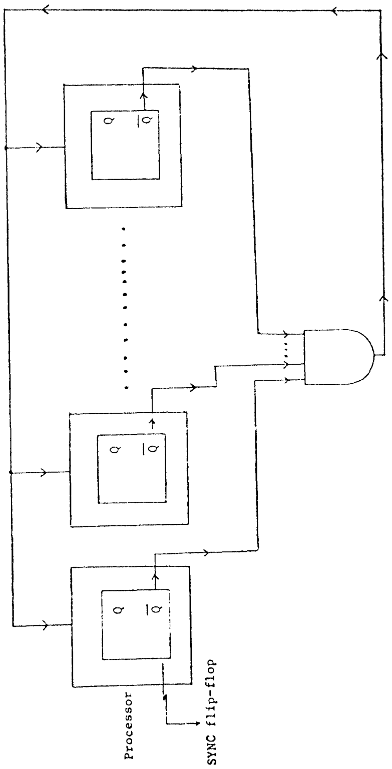


Figure 5 - Wire and Logic

The "MAP" instruction's microcode uses part of the second resynchronization mechanism to resynchronize the original task's processors. An overview of the "MAP" function's microcode operations and the use of the resynchronization mechanism by a processor is given in Fig. 6.

Further analysis of the resynchronization hardware is very much dependent on the processor implementation. The implementation in TRAC is described in a report [RAT83].

5.0 IMPLEMENTATION OF THE MAP FUNCTION

In this section we discuss the operands, data structures, operations and hardware mechanisms required for the sender in a MAP function. Here we state only the basic requirements and indicate the sequence required of the microcode to support such data communication. Then we propose a communication scheme to transmit the information required by the receiver.

The two cases of multi-byte communication are the intra- and inter- task data communication/realignment. Their initiation and execution sequence has been discussed in Section 3.

In the two cases of task communication described above we found that we need a resynchronization mechanism at the end of the MAP function to bring the processors of the original task back in lock step. The resynchronization mechanism described in Section 4.2 is used for this purpose.

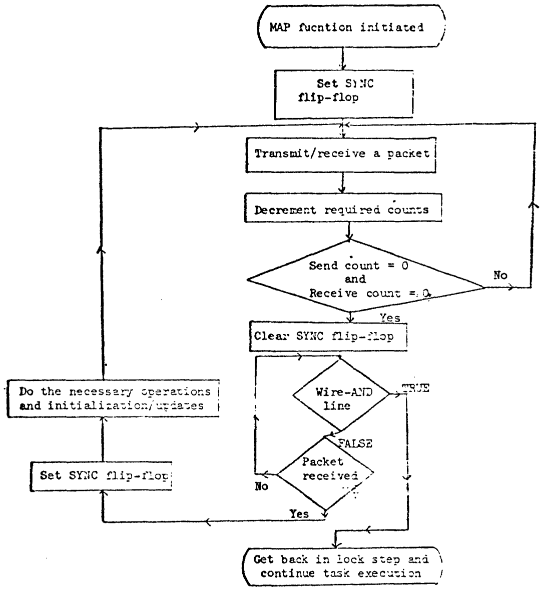

MAP fucntion initiated
Set SYNC flip-flop
Transmit/receive a packet
Decrement required counts
Send count = 0 and Receive count = 0
No
Yes
Clear SYNC flip-flop
Wire-AND line
TRUE
FALSE
Packet received
No
Yes
Do the necessary operations and initialization/updates
Set SYNC flip-flop
Get back in lock step and continue task execution

Figure 6: Overview of the use of the hardware resynrchronization mechanism to terminate the MAP function

5.1 THE MAP FUNCTION SENDER

Regardless of the mapping packet format used by any two communicating tasks the microcode requires the following minimum information for implementing the MAP function :

1. Send Count : Number of packets a processor will transmit.
2. Receive Count : Number of packets a processor will receive.
3. Destination Processor IDs : This is required by the packet network to route the packet to the required processor.
4. A pointer to the memory of the sender task where the data to be transmitted is located.
5. A pointer to the memory of the destination processor where the data is to be stored. This may be a specified index register or an absolute address in the receiving task's space.

Some additional information may also be required to allow the operating system to exercise its data security and authorization mechanisms.

This information can be provided by one of the following three schemes:

1. Load time binding : Here a compiler creates the template for the transformation or the data communication, and the loader binds the addresses, destination processor numbers IDs, and the send and receive counts. This is done for all communicating tasks.
2. Dynamic binding : The interrupting packet and/or the mapping packets are used to communicate this information.
3. A combination of the above two schemes.

The first scheme is suitable for the address-data packet format (Fig. 1), while the second/third scheme seems more suitable for the data/control information packet format (Fig. 2).

5.2 COMMUNICATION TO THE RECEIVER

We assume here that the required send counts, destination processor IDs and pointer to the data to be sent, have been specified to the sending task's processors in some fashion. We are concerned here with the specification of the packet receive counts for the receiving task.

The receiving task receives an interrupting "request" packet when a sender wants to communicate with it. The task head recognizes this packet, and after checking authorization and security of this communication, it acknowledges the sender appropriately. If mapping packets are to be received it must inform its task's processors of this. After the processors of the receiving task are made aware of a sender, they initialize their packet receive counts and then start reading the MAP packets. It is possible that while servicing one sender other senders also may desire communication. As each new sender is allowed to communicate, all the processors of the receiving task must update their packet receive counts appropriately. Therefore a scheme is needed to allow us to do this initializing/updating of the packet receive counts properly.

Four schemes were considered. They differed in whether the receiver would be given information at load time, by means of interrupting packets or by means of the mapping packet mechanism. They all required the sender have descriptors and templates set up at load time, requiring the loader to execute two passes to create and bind the operands for a MAP function. A detailed analysis of these schemes is available in a report [RAT83].

The scheme we have selected for implementation assumes that the MAP functions' "receive operands" are bound at load time in the sending task's space. When the sending task desires to execute a MAP function it must communicate the necessary "receive operands" to the receiving task dynamically, that is the "receive operands" are sent during execution time. These operands are sent over the mapping packet channel.

The actual sequence of operations is as follows :-

1. The sending task's task head sends a "request" interrupt packet to the receiving task's task head.

2. The receiving task head validates this request and sends an appropriate acknowledgement.

3. On receiving a "transmit-enable" acknowledge the sending task head directs its task's processors to start MAP packet communication.

4. Each sending task processor first sends a special "receive operand" mapping packet. This packet's type is set to indicate its contents. After sending this packet it continues to send the required data MAP packets.

5. Each receiving task's processor on receiving a packet, checks the packet type. If it contains "receive operand" information it updates its receive count (maintained in its working registers) and destination address (if any) using the packet data. Otherwise on receiving a data packet it stores the data at the required location.

6.0 ANALYSIS OF TRAC IMPLEMENTATION

This section completes the definition of the integrated packet switching mechanism and discusses the selections among design alternatives.

The packet formats (Figs. 1 and 2) must both be supported. Each is suitable for different applications (as cited in Section 3). When handling MAP packets the microcode must know which format it is transmitting or receiving. The sending task's processors have to know the format so that they can write the packet data appropriately, the receiving task's processors need to know the format so that they can read the contents properly, and also because they need to know the destination address to place this data. For the address-data packet format (Fig. 1) this address is specified in the packet. But for the data/control packet format (Fig. 2), the receiving task already knows this address implicitly, or has been informed of this prior to the multi-byte data communication. This packet type information is indicated by 4 bits in the first byte of each packet (Fig. 1 and 2).

It is necessary because of the way the destination addresses for the data/control packet format (Fig. 2) are specified to restrict the number of simultaneous senders of data/control format MAP packets. While receiving this format the receiving task's processors holds the data destination address in an internal working register. An internal register needs to be used if the microcode has to receive the packet in reasonable time. On TRAC the number of internal registers available is limited. Therefore when multi-byte data communication is done, using MAP packets of data/control format, we restrict the receiver to receive them from only one sender at a time. This serialization of communication leads to loss of parallelism. It does now, however, affect the basic application. Such MAP packets were assumed to be used by the operating system and/or by tasks for transmitting "large" amounts of data between tasks. They will not be used to support dataflow or data realignment. Removal of the internal register restriction or providing the processor with a greater number of internal working registers would allow multiple senders to a single receiver increasing potential parallelism.

It is possible to receive address-data packets (Fig. 1) from multiple senders, because the destination address for the data is specified in the packet. The microcode in the receiving task's processors does not have to use its internal working registers to store this address. Therefore there is no restriction imposed by the microcode on the number of simultaneous senders of address/data packets to a single receiver. Dataflow and data realignment can be effectively implemented with packets of this format.

It is possible that while handling address/data senders, another sender requesting transmission of a data/control format packet requests to join the transmission. It is possible for the microcode to receive packets from one such sender along with the other address-data packet senders. The receiving task will not allow a second data/control packet sender to join the transmission, until the previous such sender terminates. On terminating, the sending task's task-head indicates completion by sending an interrupting packet of a special format. On receiving this interrupting packet the receiving task enables another data/control packet sender if any are queued. In order to allow such a mix of packet formats to be received, the microcode must to identify the packet format for each MAP packet it receives.

The restrictions of simultaneous packet reception can be implemented by the receiving task head. Whenever a sending task requests mapping packet communication, it will specify the packet format to be used. This is specified in the "request" interrupting packet sent to the receiving task head. This receiving task's task head checks its mapping packet status data to see if it can allow packet communication in the request format. The task head will not allow two data/control packet senders to transmit at the same time. If the task head finds that the sender can join the MAP function, it acknowledges that sender. If not, the task head either sends a "transmit-deny" acknowledge to the sender immediately; or it queues this request and sends the acknowledge at the end of the current data/control format MAP function execution. In this case the task head will also have to inform its task's processors about this new sender at the end of the current MAP function. In the first case the sender will try to gain permission at a later time. If we choose this option, there is a possibility of congesting the interrupting channel with request-deny/acknowledge packets. This overhead is reduced in the second case since the request is queued and the sender waits to receive the transmit acknowledgement.

The next implementation option to be reviewed is the specification of the MAP function's operands. These are the send count (number of packets to be sent), the destination processor ID and the source address of the data to be sent, for each processor in the sending task. Each processor of the task must know the receive counts (number of packets to be received) and the destination address of the data. These MAP function operands must be specified and bound at compile and load time. Special data descriptors and data structures (templates) are used to store them. The sending task's operands are bound in its task's space.

This scheme somewhat increases the complexity of the systems loader. The MAP information is bound at load time to the sending tasks address space and therefore requires a two pass loader. It does not require any space in the receiving task to store the "receive operands". It thus gives a small storage space overhead. Further it does not congest the interrupting packet channel and uses only the mapping packet channel to communicate its control and data interrupted explicitly to transfer "receive operands" it does not require resynchronization within the MAP instruction. It does, however, require resynchronization to terminate the MAP instruction.

7.0 CONCLUSION

This paper has defined a multi-purpose packet data movement capability for a network architectured multiprocessor computer system. It has been shown that such capabilities can be effectively implemented in an integrated manner, and that the packet switching functions are compatible with and complimentary to a circuit switching functionality for the network. This integrated packet communication system is operational in the current four processor, nine memory configuration of TRAC. Exploration of the design space for implementation gave a clear resolution of desirable choices. The functionality and the implementation techniques are largely independent of the choice of network structures. The implementation concepts should be broadly applicable to network architectured multiprocessors.

8.0 ACKNOWLEDGEMENTS

This work was sponsored by the National Science Foundation under Grant Number MCS-8116099 and by the Department of Energy under Grant Number DE-AS05-81ER10987.

REFERENCES

1. [BRO82] J.C. Browne and G.J. Lipovski; "Reconfigurable Network Architectured Computer Systems: An Environment for Parallel Computing", Int. Workshop on High-Level Language Computer Architecture; Fort Lauderdale, Florida; pp. 40-49, 1982.

2. [LIP77] G.J. Lipovski and A. Tripathi; 'A Reconfigurable Varistructured Array Processor'; 1977 Int. Conf. on Parallel Processing; pp 165-174; August 1977.

3. [PRE79] U.V. Premkumar, R.N. Kapur and G.J. Lipovski; 'Interprocessor Communication on the Texas Reconfigurable Array Computer'; Proc. of the 1st Int. Conf. on Dist. Comp. Systs.; Huntsville, Alabama; pp 51-62; October 1-5, 1979.

4. [RAT83] B.D. Rathi, S. Deshpande, R. Jenevein, M. Sejnowski, D. Walker, G.J. Lipovşki and J.C. Browne; "Inter/Intra Task Packet Communication on the Texas Reconfigurable Array Computer"; TRAC Technical Report; Dept. of Elect. Eng. and Comp. Sci.; U.T. at Austin; 1983.

5. [SEJ80] M.C. Sejnowski, E.T. Upchurch, R.N. Kapur, D.P.S. Charlu, and G.J. Lipovski; 'An Overview of the Texas Reconfigurable Array Computer'; Proc. of AFIPS NCC Conf.; pp 631-641; 1980.

6. [SEJ81] M.C. Sejnowski; 'Packet Support in the Texas Reconfigurable Array Computer'; M.A. Report; Dept. of Comp. Sci.; U.T at Austin; Texas-78712; 1981.

7. [SIE81] H.J. Siegel and R. McMillen; "The Multistage Cube : A Versatile Interconnection Network"; COMPUTER Vol.14, No.12; pp 65-76; December, 1981.

8. [TRI79] A. Tripathi and G.J. Lipovski; 'Packet Switching in Banyan Networks'; 6th Annual Symp. on Comp. Arch.; 1979.

TIMING CONTROL OF VLSI BASED NLOGN AND CROSSBAR NETWORKS*

Sanjay Dhar, Mark A. Franklin and Donald F. Wann
Department of Electrical Engineering,
Washington University,
St.Louis, Missouri 63130.

ABSTRACT

Two basic data flow control methods for circuit switched, pipelined networks of the general NLogN and Crossbar (CB) topologies are modelled and their effects on overall data rates achievable are determined. The synchronous method uses a global clock and as network modules grow, clock skew and and clock tree charge/discharge times grow resulting in lower data rates. The asynchronous method relies on local request/acknowledge signals to control data movement and hence it´s performance is less affected by system growth.

1.0 Introduction

Advances in VLSI technology have made available a number of low cost yet powerful microprocessor chips. This has led to a host of proposals ([8], [9], [10]) for the design of closely coupled multiple processor systems in which a number of processors are connected together by a communications network. The network handles interprocessor communication and enables resource sharing. Its design is a key factor in determining overall system performance.

The principal issue of interest in this paper is the type of control scheme to be used for control of data movement in a large circuit switched interconnection network environment where the network is partitioned into a number of subnetwork chips [2], and data transfer through the network is pipelined. Two principal methods that can be used in controlling data movement along the network are referred to as the synchronous (or clocked) and the asynchronous (or self-timed) schemes [7]. The synchronous control scheme has been traditionally favoured, especially in small systems, because of its logic design simplicity. The presence of global clock signals, however, makes such systems difficult to expand and as the system grows, system performance may deteriorate due to the increases in clock skew. The absence of any global signals in an asynchronous system makes it inherently modular and expandable and hence it becomes attractive in systems where the size of the system cannot be predicted in advance, where a number of subsystems operate independently, or where system size (and clock skew) require inordinately large clock periods for proper operation.

The analysis in this paper follows that of [11], focusing here, however, on the data rates achievable in both CB and NlogN networks when asynchronous and clocked control schemes are utilized. The analysis provides a quantitative approach to making the CB/NlogN, asynchronous/synchronous design decisions. Decision curves are provided for a particular example to illustrate the procedure.

2.0 Protocol Issues

A complete interconnection network requires control provisions for path establishment, transfer of data from source to destination, detection of a blocked path and indication of end of transmission with path clearing. We will assume that these requirements are satisfied as shown in [2] and that the network has been partitioned following a bit slice architecture approach with one bit per plane. The present analysis focuses on the data rates achievable after a path has been established from a source to a destination. Hence this analysis will hold for systems where the average message length is much larger than the average number of modules in a path in the network, that is, data tranfer time is much larger than path establishment time.

For the asynchronous network, a delay insensitive control structure is adopted. That is, insertion of arbitrary delay between modules will not cause the network to malfunction. Also transition sensitive logic is employed. Figure 1 shows the interconnection between two asynchronous modules. A transition on R1 indicates the presence of a "1" data bit, while a transition on R0 indicates the presence of a "0" data bit. The "A" line supplies the acknowledge response signal. Interconnection of two synchronous modules is shown in Figure 2. For the synchronous network, the standard two-phase level sensitive clock is used for the data transfer. Data at the input of a module is captured at phase one of the clock and is transferred to the output of the module at phase two of the clock.

3.0 Asynchronous Banyan Delay Model

The Huffman finite state machine representation of a logical implementation of the asynchronous module is shown in Figure 3. If we assume that the environment of the module can cause a change at the module input as soon as the environment receives a change at the module output, then it can be shown [1] that the sufficient conditions on the various delays to achieve race-free operation are given by the relations

$$dF >= dL \qquad (3.1)$$

$$d0 >= dF + dL \qquad (3.2)$$

where dL is the maximum delay of the combinational logic.

A pair of communicating modules i and j in a path k is modelled as shown in Figure 4. Considering module i, the maximum propagation delay from any input to any output of the combinational logic is dLi, the propagation delay of the feedback path is dFi and the propagation delay from module i to module j is dPij. Similarly for module j, we have dLj and dFj and dPji, the propagation delay in the acknowledge path from module j to module i. The delay from the output of the combinational logic of module i

* This work was supported in part by NSF Grant MCS-78-20731 and ONR Contract N00014-80-C-0761.

0190-3918/83/0000/0059$01.00 © 1983 IEEE

through the combinational logic of module j to the input of the combinational logic of module i, corresponding to the term d0 of Figure 3, is given by

$$d0 = \max(dFi,dPij) + dLj + \max(dFi,dPji) \tag{3.3}$$

If we assume that condition (3.1) is satisfied ($dFi >= dLi$ and $dFj >= dLj$), then d0 given by (3.3) satisfies condition (3.2).

The minimum delay in transferring two successive pieces of data (e.g. successive words which are part of the same message) between the pair of Asynchronous BA modules i and j is equal to the maximum loop delay as given below.

$$dABAij = dLi + \max(dFi,dPij) + dLj + \max(dFi,dPji) \tag{3.4}$$

Consider next all of the pairs of communicating modules along a particular path k in the network. Since data transfer is pipelined, we next determine the maximum delay between module pairs on that path.

The path k is modelled as shown in Figure 5 where each pair of communicating modules and the maximum loop delay associated with that pair is shown. Since the network is pipelined, the minimum time between transfer of two successive data items along the path k, dABAk, is given by the maximum of the delays dABA12, dABA23,...., dABA(n-1)n.

$$dABAk = 2dL + 2(\max(dPk,dF)) \tag{3.5}$$

where dL and dF are the maximum values associated with the combinational logic and feedback delays; dPk is the maximum delay between modules for path k, that is

$$dPk = \max(dPij,dPji) \quad \text{for all communicating modules i and j in path k} \tag{3.6}$$

Notice that dPk will be dependent on the particular path under consideration since this delay reflects the layout of module chips on a printed circuit board and that layout is in turn dependent on the topology of the network being considered. The average of dABAk over all paths (a total of M are present) in the network gives the average delay between successive data transfers. Assuming all paths in the network are equally used this average can be expressed as

$$dABA = \Big(\sum_{k=1}^{M} dABAk\Big)/M \tag{3.7}$$

We will assume here that the maximum delays associated with the combinational logic and feedback are equal for all paths in the network. Then if $dPk >= dF$ for all k, equation (3.7) becomes:

$$dABA = 2dL + 2\Big(\sum_{k=1}^{M} dPk\Big)/M \tag{3.8}$$

Letting dPBA be the average of dPk over all paths we obtain

$$dABA = 2dL + 2dPBA \tag{3.9}$$

where $dPBA = \sum_{k=1}^{M} dPk/M$

3.1 Synchronous Banyan Delay Model

A pair of synchronous modules in a path from a source to a destination is modelled as a finite state machine in Figure 6. A two phase clocking scheme is used to clock the memory elements 1 and 2 of each module. Considering module i, the maximum combinational logic delay is dLi, the memory delays are dMi1 and dMi2, the interconnection path delay from module i to module j is dPij, and the clock delays are dCi1 and dCi2. Similarly for module j.

The following three constraints on the clock period (obtained as in [11]) must hold to ensure proper operation:

$$T >= dMi1+dMi2+dPij+dLj+(dCi1-dCj1) \tag{3.10}$$

$$T >= dMi2+dPij+dLj+dMj1+(dCi2-dCj2) \tag{3.11}$$

$$T >= dMi1+dMi2+dLi \tag{3.12}$$

In most designs the third constraint on T is smaller than either of the first two and will not be considered further. The quantities (dCi1-dCj1) and (dCi2 - dCj2) are the differences between the times the phases 1 and 2 of the clock arrive at the corresponding memory elements of the two modules and are referred to as the clock skew, defined as

$$deltaC1 = dCi1 - dCj1 \tag{3.13}$$

$$deltaC2 = dCi2 - dCj2 \tag{3.14}$$

If dM, dPBAmax, dL and delta represent maximum values which can occur over any data path, then the constraints of (3.10) and (3.11) can be written as

$$T >= dL + 2dM + dPBAmax + delta \tag{3.15}$$

where dPBAmax is the maximum path delay between any pair of communicating modules over the entire network, that is,

$$dPBAmax = \max(dPij,dPji) \quad \text{for all communicating modules} \tag{3.16}$$

Another constraint on the clock period relates to the clock tree charge/discharge time. For reliable operation of the system the clock period must be greater than the time required to charge and discharge the clock tree to voltage levels which can be reliably sensed by the gates in the network. Let this time be represented by tau and thus $T >= tau$. The worst case condition clock period for the Synchronous BA network, dSBA, is now given by

$$dSBA = \max(dL+2dM+dPBAmax+delta,\ tau) \tag{3.17}$$

3.2 Asynchronous Crossbar Delay Model

The delay model for the CB network is obtained in a similar manner as for the BA network. The maximum Asynchronous CB loop delay for modules i and j is then obtained as

$$dACBij = dLi + \max(dPij,dFi) + dLj + \max(dPji,dFi) \tag{3.18}$$

Since data is pipelined as in the BA network, we obtain the average delay as dACB given by

$$dACB = 2dL + 2*\max(dPCB,dF) \tag{3.19}$$

where dPCB is the path delay between two communicating modules. It should be noted that because of the planar construction of the CB network, the distance between two interchip communicating modules is constant independent of network size. The maximum path delay between two communicating modules is a constant not dependent on any particular path being considered. This is a key difference in the analysis of the two networks. If $dPCB >= dF$ then equation (3.19) can be written as

$$dACB = 2dL + 2dPCB \tag{3.20}$$

3.3 Synchronous Crossbar Delay Model

The synchronous delay model for the CB network is similar to the model developed for the BA network (Figure 6). The delay in the network is given by dSCB where

$$dSCB = max(dL + 2dM + dPCB + delta, tau) \qquad (3.21)$$

Note that in the CB network, dPCB is also the maximum delay between two communicating modules.

4.0 Delay Parameters

Consider next the various delay parameters needed for evaluation of dABA, dSBA, dACB and dSCB in Equations (3.9), (3.17), (3.20) and (3.21) respectively. To estimate these values the synchronous and asynchronous modules were designed using NMOS technology with a minimum feature size of 2.5 microns. From these designs values for dL and dF were obtained (see section 5.0). Considering the path delays dPBA, dPBAmax and dPCB, the delay of on-chip paths is negligible compared to the delay of off-chip paths. The off-chip delay can be minimized by using exponential drivers and is given by [5]:

$$dP = d*e*ln(CL/Cg) \qquad (4.1)$$

where Cg is the capacitance of an elemental gate and CL is the load capacitance. The capaciatance CL consists of two pin capacitances (Cpin) and the external path capacitance. For the CB case, the maximum path length between communicating modules in any path is L2 (Figure 7), and (4.1) becomes:

$$dPCB = d*e*ln((2Cpin+Cb*L2)/Cg) \qquad (4.2)$$

where Cb is the capacitance per unit length of the printed circuit board. For the BA case dPBA and dPBAmax are derived in [12] as:

$$dPBA = d*e*ln(2Cpin + Cb*L1 + ((N**2+1)*(N-1)/2N**4)*N'*Cb*L2/Cg) \qquad (4.3)$$

$$dPBAmax = d*e*ln((2Cpin + Cb*L1 + (N-1)*N'*Cb*L2/N**4)/Cg) \qquad (4.4)$$

where L1 and L2 are the spacing between chips used in the BA and CB networks as shown in Figure 7, N is the network size in a chip module and N′ the overall network size. This more complex expression reflects the changing path lengths between banyan network stages.

We next consider the clock skew. For this analysis it is assumed that the clock as presented to the individual chip modules has no skew and that all skew occurs within the chip. The clock skew can be attributed to :

(a) Differences in line lengths.
(b) Differences in the passive line parameters like resistance, dielectric constant that determine the line time constant.
(c) Differences in the threshold voltages of the two modules.

One possible clock distribution scheme that guarantees equal length paths, thus eliminating (a) from consideration is shown in Figures 8 and 9. As shown in Figures 8 and 9, the section AB of the clock tree is common to all the modules in the chip and hence does not contribute towards the clock skew. Let the maximum and minimum time constants of the clock tree from B to all the leaf nodes be RCmax and RCmin. Then, given equal length paths the clock skew can be found as [11]:

$$delta = RCmin*ln(1-(VTmin/Vdd)) - RCmax*ln(1-(VTmax/Vdd)) \qquad (4.5)$$

where VTmax and VTmin are the maximum and minimum values associated with the threshold voltages of the gates of the network and Vdd is the power supply voltage. RCmax and RCmin can be obtained as functions of the clock tree time constant RC (e.g. RCmax=k1*RC, RCmin=k2*RC).

Determination of the clock tree time constant is a problem that has been solved in [4] and [6]. Using the development of [4] for the tree starting at B (Figures 8,9) we get RC as:

$$RC = 9*(1-1/N)*(N^{3} - 1)*R0C0/7 \qquad (4.6)$$

where R0 and C0 are the resistance and capacitance of the last (and smallest) section of the clock tree. The time constant (RCf) for the entire clock tree (starting at A) is:

$$RCf = (3 - 2/N)*(10N^{3} - 3)*R0C0/7 \qquad (4.7)$$

The total time to charge and discharge the clock tree, tau, has been derived in [12] and is given by

$$tau = RCf*\left\{ln\left(\frac{Vdd-VTmin+Vn}{Vdd-VTmax-Vn}\right) + ln\left(\frac{VTmax+Vn}{VTmin-Vn}\right)\right\} \qquad (4.8)$$

where Vn is the noise margin rquired for reliable circuit operation. Notice that for this simplified analysis RCf is a function of the module size N and not the overall network size N′. That is, only the clock tree charge/discharge time within the chip module is considered.

5.0 Example

As an example let us consider a N′*N′ network built from N*N size module chips which are laid on copper printed circuit boards. The pin capacitance for this type of construction is about 4pF and the capacitance of an elemental gate is 0.01pF. The various delays are:

d = 2 nsec; dM = 4 nsec
dL = dF = 45 nsec. for synchronous module
dL = dF = 34 nsec. for asynchronous module

We will assume that the size of the largest chip available is 1cm*1cm on which a single bit slice 32*32 network can be implemented. Assume that only one layer of metal is available. Let a fraction q of the clock line be distributed in diffusion and the rest in metal. If Rd is the resistance per square of diffusion, Cd and Cm the capacitance per unit area in pF/sq.micron of diffusion and metal respectively, then the time constant of the last section of the clock tree R0C0 is given by [12]:

$$R0C0 = ((10000/32)**2)*Rd*q*(2*q*Cd + 3*(1-q)*Cm)/8000 \text{ nsec} \qquad (5.1)$$

The fabrication constants of the current NMOS technology have the following values:

$$Rd = 20 \text{ ohms/sq.}, Cm = 10^{-4} \text{ pf/sq. micron},$$
$$Cd = 0.3*10^{-4} \text{ pF/sq. micron}$$

Also, the variation of time constant and threshold voltage during fabrication is about 20% (i.e. RCmax=1.2*RC, RCmin=0.8*RC). We take the supply voltage Vdd=5 V, the typical threshold voltage as 2.5 V and the noise margin as 0.5 V. Then VTmax=3.0 V, VTmin=2.0 V and Vn=0.5 V. For illustration purposes, take q=5%, L1=1 inch, L2=2 inches and Cb=1 pF/inch (note values of L1 and L2 are dependent on board technology factors such as whether multilayer, wirewrap or other technology is used). The delays dABA and dACB for the asynchronous modules and the delays dSBA and dSCB for the synchronous modules of the BA and CB networks are plotted in Figures 10 and 11 against

N′, the network size. In Figure 12 the same delays are plotted against N, the module size.

From Figure 12 we can make a comparison of the delays associated with the asynchronous and synchronous control schemes. It is clear that in the case of the Banyan network, for small N the synchronous control scheme results in a smaller delay. Consider a particular network size N′. From the intersection of the curves for the asynchronous and synchronous control schemes for this value of N′, we can obtain the range of values of N for which a particular control scheme is better. For example, for N′=512 we get the following result:

$$\text{BANYAN}\begin{cases}\text{Synchronous control better for } N < 20\\ \text{Asynchronous control better for } N > 20\end{cases}$$

Similar conclusions can be reached for the CB network. network curves. Notice that the CB network delays are independent of the network size because the intermodule distances are constant independent of the module or network size. Thus we get the following result:

$$\text{CROSSBAR}\begin{cases}\text{Synchronous control better for } N < 24\\ \text{Asynchronous control better for } N > 24\end{cases}$$

6.0 Conclusions

The Banyan and the Crossbar networks have been modelled according to the type of control scheme used. These models were used to obtain the delay associated with each type of network and control scheme. The delay equations were then used to obtain the delay curves of Figures 10 and 11 and Figure 12.

Comparison of the delays in the Banyan and the Crossbar network were made for both types of control schemes, the synchronous and the asynchronous. It can be observed from Figure 10 that the BA network delays increase with N′, the network size, because of the increase in the physical inter-chip path lengths. The asynchronous network delay decreases with the module size because the implementation of a network of a given size requires fewer modules chips and hence the physical path lengths between communicating modules decrease. Physical path lengths decrease with N for the synchronous case also, however here large modules result in larger clock skew which dominates the inter-chip delay. The synchronous network delay is the maximum of two terms. For small module sizes the first term (includes inter-chip delay and clock skew) of (3.17) dominates, while for large module sizes (e.g. N=32) the clock tree charge/discharge time, tau,dominates. Tau increases rather rapidly (O(N**3)) with the module size and this results in the steep rise in Figure 12 for large N. The behaviour of the delays of the Crossbar network were similar except that the planar construction of the network resulted in smaller delays. Notice also that only a single curve is obtained for the asynchronous case because of the modular and planar structure of the Crossbar network. This was also the reason for the delays in the Crossbar network for both types of control schemes being less than those in the Banyan network. Delay curves were obtained for a particular example clearly showing the delay tradeoffs for the Banyan and Crossbar networks, and the synchronous and the asynchronous control schemes.

REFERENCES

[1] Fang,T.P. "On the Design of Hazard Free Circuits", Comp. Sys. Lab., Tech. Mem. 285, Washington University, St.Louis, MO (Nov. 81).

[2] Franklin,M.A., Wann,D.F. and Thomas,W.J. "Pin Limitations and Partitioning of VLSI Interconnection Networks", IEEE Trans. on Comp., Vol. C-31, No. 11, Nov. 1982.

[3] Goke,L.R. and Lipovski,G.J., "Banyan Networks for Partitioning Multiprocessor Systems", Proc. 1st Annu. Symp. Comput. Arch., 1973.

[4] Kung, S.Y. and Gal-Ezer,R.J. "Synchronous vs Asynchronous Computation in VLSI Array Processor", Proc. SPIE, Vol. 341, May 1982.

[5] Mead,C. and Conway,L., INTRO. TO VLSI SYSTEMS, Addison-Wesley Pub.Co. Reading ,MA (1980).

[6] Penfield, P. and Rubinstein,J. "Signal Delay in RC Tree Networks", Proc. 18th Design Auto. Conf., June 1981.

[7] Seitz,C.L., "Self-timed VLSI Systems", Proc. Caltech Conf. VLSI, Jan.1979.

[8] Sejnowski,M.C.,et. al. ′An overview of the Texas Reconfigurable Computer′, AFIPS Proc., Nat. Comp. Conf. (1980).

[9] Sullivan,H. and Bashkow,T.R. ′A Large Scale Homogeneous, Fully Distributed Parallel Machine I′, Proc. 4th Ann. Symp. on Comp. Arch. (March 1977).

[10] Swan, R.J. et. al. ′Cm* A Modular Multi-Microprocessor′, AFIPS Proc. Nat. Comp. Conf. (1977).

[11] Wann, D.F. and Franklin, M.A. "Asynchronous and Clocked Control Structures for VLSI Based Interconnection Networks", IEEE Trans. on Comput., March 1983.

[12] Dhar,S., Franklin,M.A. and Wann,D.F. "Timing Control of VLSI based NlogN and Crossbar Networks", Center for Computer Systems Design, Washington Univ., St. Louis, MO, Tech. Rpt. CCSD83-101 (May 1983).

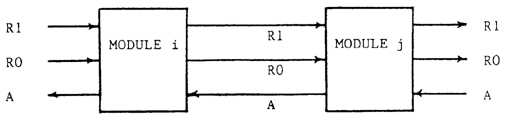


Figure 1: Interconnection of two asynchronous module.

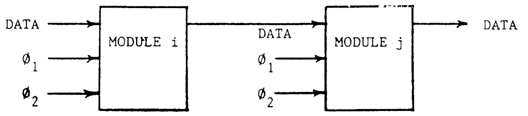


Figure 2: Interconnection of two synchronous modules.

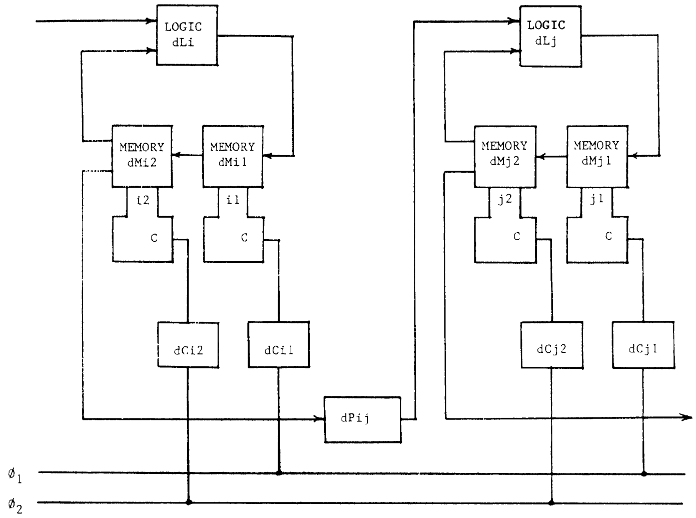


Figure 6: Model for two adjacent synchronous modules.

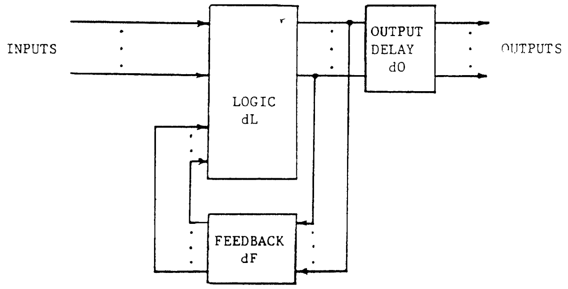
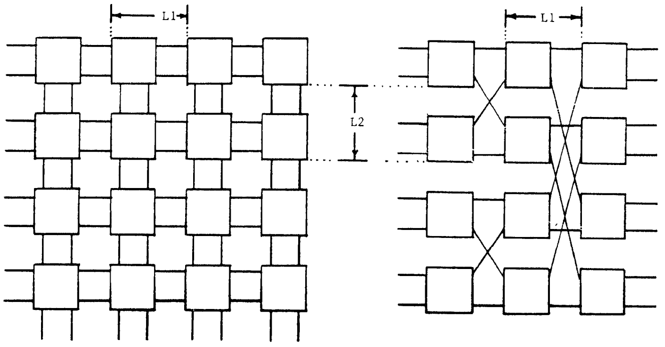


Figure 3: Huffman asynchronous circuit model



(a) Crossbar (b) Banyan

Figure 7: 8*8 Crossbar and Banyan networks built from 2*2 module chips.

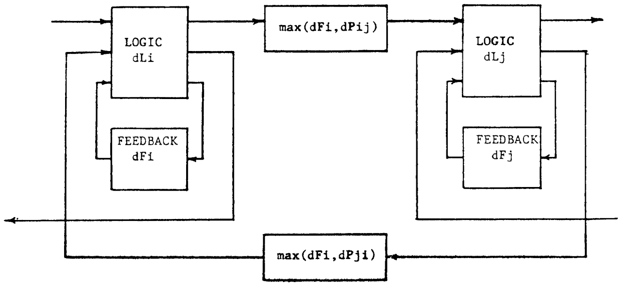
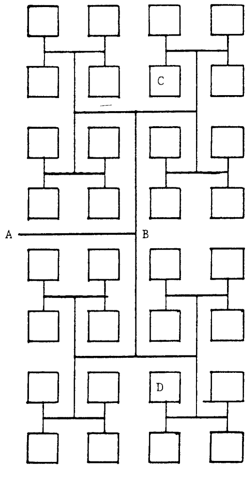


Figure 4: Delay model for two adjacent asynchronous modules.



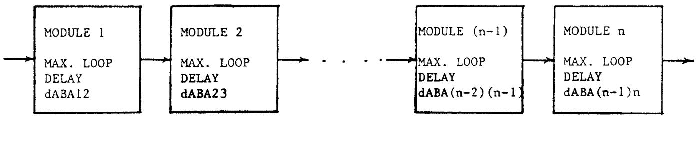


Figure 5: Delay model for a path in the asynchronous Banyan network.

Figure 8: Clock distribution for a 16*16 Banyan network.

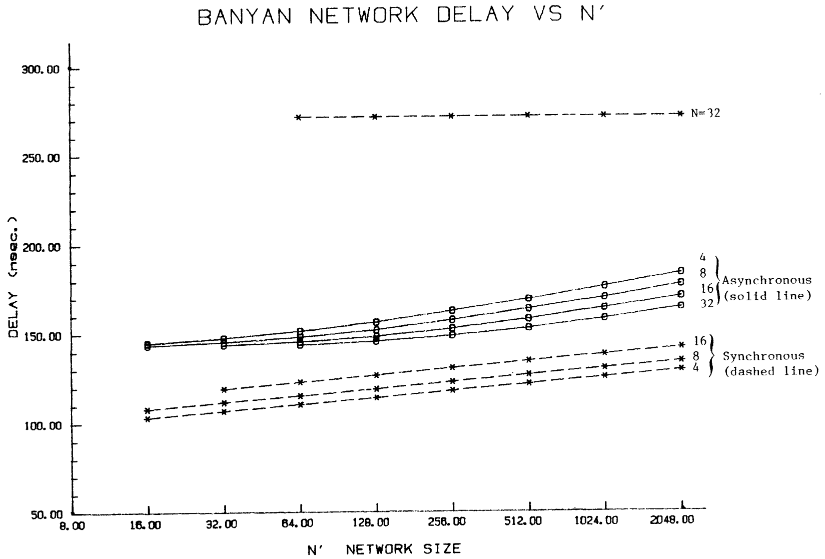


Figure 10: Delay variation of Banyan network as a function of network size N' with module size N as a parameter.

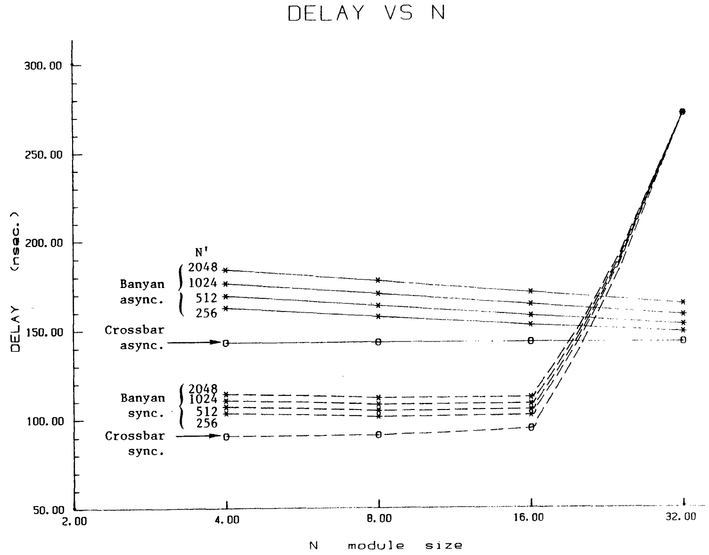


Figure 12: Delay variation of Crossbar and Banyan networks as a function of module size N with network size N' as a parameter.

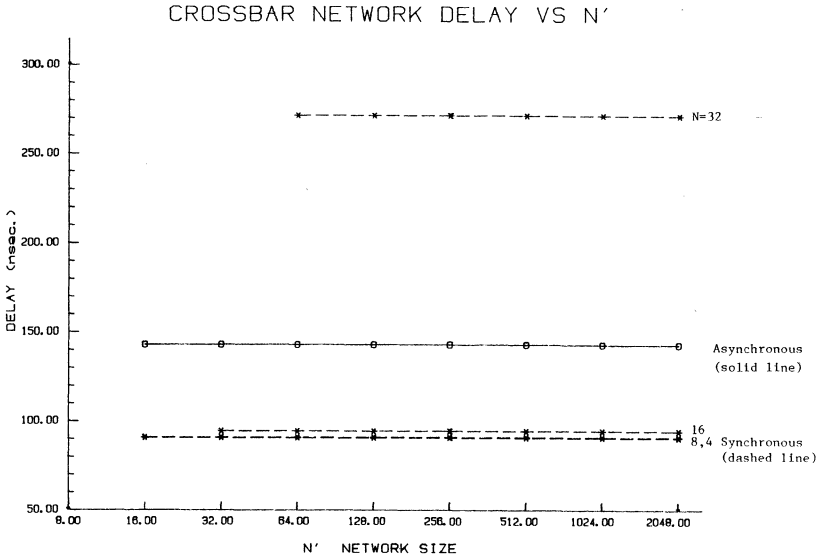


Figure 11: Delay variation ofCrossbar network as a function of network size N' with module size N as a parameter.

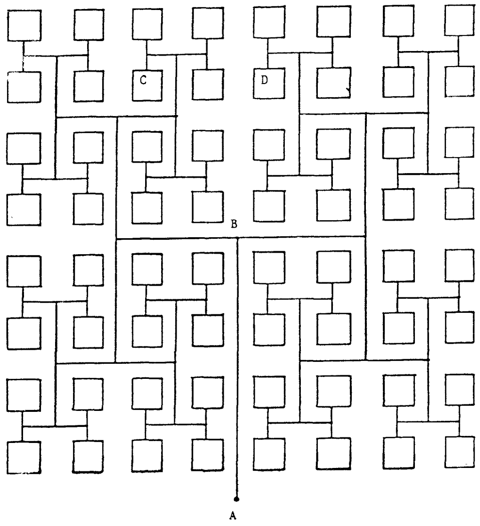


Figure 9: Clock distribution for an 8*8 Crossbar network.

Easily-Testable (N,K) Shuffle/Exchange Networks

David C.H. Lee* and **John Paul Shen**

Department of Electrical Engineering
Carnegie-Mellon University
Schenley Park, Pittsburgh PA 15213 U.S.A.

Abstract -- *This paper focuses on the testing of an important class of interconnection networks called (N,K) shuffle/exchange networks. A sequential circuit model is used for the basic switching element. A general fault model for the switching element is introduced. A testing strategy is presented which involves the exhaustive testing of each switching element without exhaustively testing the entire network. Each switching element is exhaustively tested via the application of a checking sequence. It is shown that the class of (N,K) shuffle/exchange networks is C-testable. A network is C-testable if it can be fully tested using a constant number of test patterns. A test sequence of constant length is constructed which when applied to a (N,K) shuffle/exchange network will fully test the entire network.*

1. Introduction

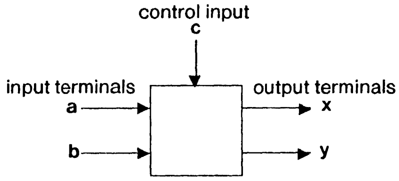
Previous works

In recent years, many multistage interconnection networks have been proposed and extensively studied. Most research efforts focus on the network topologies, routing algorithms, and potential applications [7]. More recently, issuses involving reliability, fault tolerance and fault diagnosis are being addressed [1]. There has been limited investigation of the testing and testability of such networks, which constitute the focus of this paper.

Most of the previous works assumed each basic switching element to be a combinational circuit and each requires a separate control line. Most of the fault models assumed are quite restrictive. Frequently only faults involving single line stuck at a logic value or single switching element stuck at a switch state are considered. A more general fault model is needed.

(N,K) shuffle/exchange networks

A β-element is a 2 x 2 switching element that can be set to one of two states, namely the "Through" (0) state or the "Cross" (1) state, corresponding to the two possible permutations of its two input terminals; see Figure 1a . A β-element can be implemented as a two-state sequential circuit; see Figure 1b. Each β-element in a network can be independently set to either the 0 or the 1 state. To facilitate self-routing and to reduce the number of I/O pins needed, Levitt et al [6] propose a β-element which uses the two data inputs a and b for transmitting data as well as destination address tag bits used for routing. A third input, c, determines whether the input terminal lines contain data or routing tag bits. Similar routing scheme is assumed in this paper. Details of it can be found in [5].



(a) The beta element

		input vector <abc>							
next state/ outputs <xy>		000	001	010	011	100	101	110	111
current state	0	0/00	0/00	0/01	0/01	0/10	1/10	0/11	1/11
	1	1/00	0/00	1/10	0/10	1/01	1/01	1/11	1/11

(b) The state table of the beta element

Fig. 1. The β element and its state table.

A N x N shuffle/exchange stage has N input terminals and N output terminals and consists of a perfect shuffle connection [9] followed by (N/2) β-elements. For convenience, N is assumed to be a power of 2. Let m = $\log_2 N$, then each terminal can be identified by a binary number of m bits. Starting from the top, each β-element can be identified by a binary number of m-1 bits. All the β-elements in the same stage examines synchronously the routing tag bits. A single control signal c can be used for all the β-elements in the same stage. This scheme reduces the number of control signals needed from (N/2) down to one per stage.

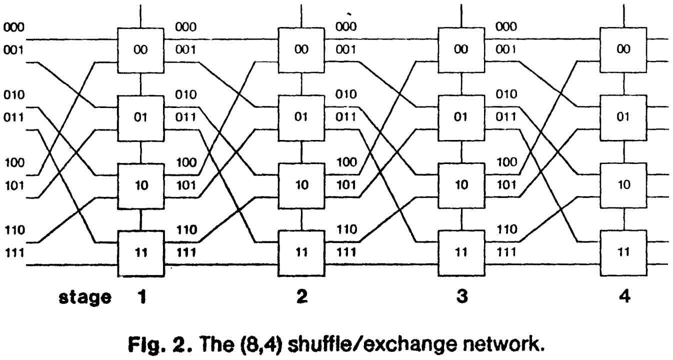


Fig. 2. The (8,4) shuffle/exchange network.

A (N,K) shuffle/exchange network has N input terminals, N output terminals, and consists of a cascade of K identical N x N shuffle/exchange stages. The stages can be numbered from left to right as 1,2,...,K. The outputs from stage i are connected to the inputs of stage i+1 as shown in Figure 2. $(\beta_{m-2}\beta_{m-3}...\beta_0)_j$ denotes the β-element $(\beta_{m-2}\beta_{m-3}...\beta_0)$ in the jth stage.

It is assumed that the (N,K) shuffle/exchange network uses a routing scheme involving destination address tags [6]. Before inputing data at an input terminal a K-bit routing address tag (d_1

0190-3918/83/0000/0065$01.00 © 1983 IEEE

$d_2 ... d_K$), one for each stage, is used to set the switches so as to provide the desired connection path. The β-element in the ith stage examines d_i and sets its state according to the value of d_i.

Many well known networks are (N,K) shuffle/exchange networks [7]. When K = log$_2$N, the (N,K) shuffle/exchange network is the omega network, and is topologically equivalent to a class of well-known networks. These networks include the modified data manipulator, the flip network used in STARAN, the indirect binary n-cube network and the regular SW banyan network with spread and fanout of 2. When K = 1, it is the well known shuffle/exchange network proposed by Stone [9].

In Section 2, the fault model and the testing strategy is formally introduced. In Section 3, the concept of C-testability is introduced and applied to the testing of (N,K) shuffle/exchange networks. In Section 4, it is shown that the class of (N,K) shuffle/exchange networks is C-testable. A test sequence is constructed whose length is independent of the network size.

2. Testing Methodology

Beta-element fault model

The state table of a sequential machine completely characterizes the machine's behavior. Since a β-element is modelled as a 2 state sequential machine, any β-element failure which causes an arbitrary change to the original state table is considered a fault. Our fault model assumes:

1. A fault in a β-element is any arbitrary deviation from the fault-free state table of the β-element without increasing the number of states.

2. There is at most one faulty β-element.

3. The fault is permanent.

* : faulty signals

next state/ outputs <xy>		input vector <abc>							
		000	001	010	011	100	101	110	111
current state	0	$0/1\overset{*}{0}$	$\overset{*}{1}/\overset{*}{1}0$	$0/1\overset{*}{1}$	$1/1\overset{*}{1}$	0/10	1/10	0/11	1/11
	1	$1/01^*$	$\overset{*}{1}/01^*$	$1/11^*$	$\overset{*}{1}/11^*$	1/01	1/01	1/11	1/11

Fig. 3. Input line a stuck at 1.

next state/ outputs <xy>		input vector <abc>							
		000	001	010	011	100	101	110	111
current state	0	$0/01^*$	$0/01^*$	0/01	0/01	$0/11^*$	$1/11^*$	0/11	1/11
	1	$1/01^*$	$0/01^*$	$1/11^*$	$0/11^*$	1/01	1/01	1/11	1/11

Fig. 4. Output line y stuck at 0.

next state/ outputs <xy>		input vector <abc>							
		000	001	010	011	100	101	110	111
current state	0	$\overset{*}{1}/00$	$\overset{*}{1}/00$	$\overset{*}{1}/\overset{*}{1}\overset{*}{0}$	$\overset{*}{1}/\overset{*}{1}\overset{*}{0}$	$\overset{*}{1}/\overset{*}{0}\overset{*}{1}$	$1/\overset{*}{0}\overset{*}{1}$	$\overset{*}{1}/11$	1/11
	1	1/00	$\overset{*}{1}/00$	1/10	$\overset{*}{1}/10$	1/01	1/01	1/11	1/11

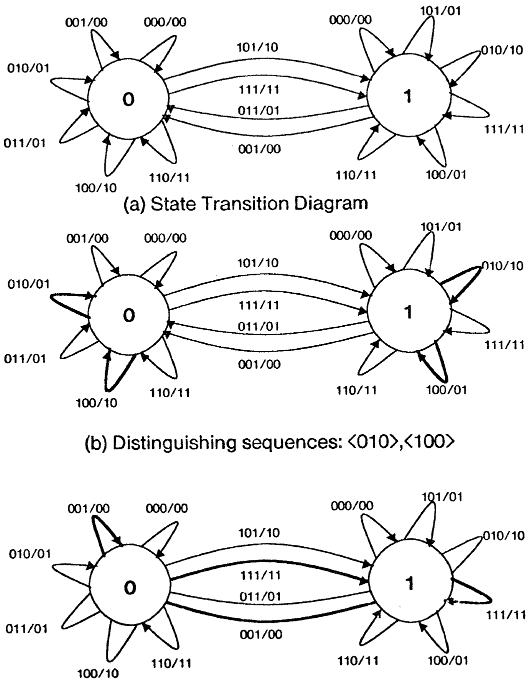
Fig. 5. β-element stuck at the 1 state.

As shown in Figures 3, 4 and 5, a conventional fault involving either a line stuck at some logical value or a β-element stuck at the 0/1 state can be represented using this fault model. Many other fault types can be modelled. This fault model is quite compatible with VLSI implementations. On a VLSI chip, faults tend to be arbitrary but confined to certain area of the chip.

Exhaustive testing of beta elements

Since the fault model allows a β-element to fail in an arbitrary way, each β-element must be exhaustively tested. A β-element, modelled as a 2 state sequential machine, can be exhautively tested using the checking experiment approach. Hennie described a method for sequential machine identification using a checking experiment [3]. A checking experiment for a machine involves the construction of a checking sequence of the machine. A checking sequence consists of an input sequence and the corresponding output sequence which can uniquely characterize a sequential machine. The sequential machine must have a strongly connected state diagram and a distinguishing sequence in order for a checking sequence to exist. A distinguishing sequence is an input sequence the application of which allows the current state of the machine to be determined from the output sequence. A checking sequence must perform the following three functions: (1) Initialize the machine into a known state S. (2) Verify the number of states in the machine. (3) Starting from state S, for every entry in the state table, an input vector is applied to stimulate that entry and then a distinguishing sequence is applied to verify the state transition.

If at any time during the checking experiment, the actual machine responds in a manner other than that dictated by the expected output sequence, the sequential machine must be faulty. If a β-element behaves correctly throughout the checking experiment and assuming the number of states has not been increased by a



(a) State Transition Diagram

(b) Distinguishing sequences: <010>,<100>

(c) Synchronizing sequences: <001>,<111>

Fig. 6. State transition diagram and some useful input sequences of the β-element.

fault, then the state table of this machine must be the same as the fault free one. By carefully designing the input checking sequence, the length of the sequence can be reduced [3]. Figure 6 shows the state transition diagram and some useful input sequences of the β-element. The diagram is strongly connected. It has two distinguishing sequences of length one, namely the two input vectors <abc> = <010> and <100>. Futhermore it has two synchronizing sequences, <001> and <111>, also of length one. (A synchronizing sequence is an input sequence which when applied to a machine results in a unique final state independent of the initial state.)

The checking experiment for the β-element involves the application of a synchronizing sequence (<001> or <111>) followed by the activation of a state transition and then followed by a distinguishing sequence (<010> or <100>) for each of the 16 entries or state transitions in the state table. A (N,K) shuffle/exchange network has N/2*K β-elements. If the entire network is considered as a single sequential machine, it will have $2^{N/2 * K}$ states. It is infeasible to design a checking sequence for such a state machine even for relatively small N and K. Consequently, the appropriate testing strategy is to exhaustively test each β-element without exhaustively testing the entire network. The main task now is to construct the smallest possible sequence of network input test patterns which will result in the efficient simultaneous application and observation of the checking sequences to all the β-elements.

3. C-testability and Test Vectors

C-testability

The problem of testing iterative arrays was first studied by Kautz [4], who assumed that an individual cell can be tested for all its possible faults only by applying all possible input vectors to that cell. The necessary and sufficient conditions were given by Kautz for testing an iterative array with a single faulty cell. Friedman [2] studied a class of one-dimensional unilateral combinational iterative arrays which requires a constant number of tests to detect all faults, independent of the size of the array. He called them C-testable iterative arrays. He also assumed that there is only one faulty cell in the array. In this paper, the concept of C-testability is generalized and applied to two-dimensional (N,K) shuffle/exchange networks. Unlike the combinational arrays studied by Kautz and Friedman, a (N,K) shuffle/exchange network consists of cells which are sequential circuits. Applying Kautz's necessary and sufficient conditions to (N,K) shuffle/exchange networks we have the following:

Definition 1: A (N,K) shuffle/exchange network is testable if the following conditions are met:

1. For each β-element, all entries in the state table can be stimulated, i.e. all the state transitions can be activated, and then verified.
2. For each β-element, any faulty signal produced by a faulty β-element can be propagated to an observable network output.

Condition 1 is necessary and sufficient for the exhaustive testing of every β-element. Condition 2 ensures the detection of any faulty signal.

Definition 2: A (N,K) shuffle/exchange network is C-testable if it is testable and the number of network test vectors, or test patterns, required is a constant and independent of the size of the network.

Four useful test vectors

A test vector for the (N,K) shuffle/exchange network consists of two sub-vectors. The first sub-vector consists of the data inputs to the β-elements in the first stage, and the second sub-vector are the K control signals, $c_1, \ldots, c_K$. Since all K control signals are normally inactive except when new communication paths are being established by routing tag bits, during which time the same active signal is applied to all K control lines sequentially, hence all K lines can be considered as one logical control line. $T = \langle t_0\, t_1 \ldots t_{N-1}\rangle^c$ is used to denote the test vector, where t_i is the input to the ith input terminal of the (N,K) shuffle/exchange network and c is the control signal input. A shift register can be used to shift a c pulse to successive stages in synchronism with the arrival of destination address tag bits at the data inputs of successive stages. $\overline{T}$ is used to denote $\langle \bar{t}_0\, \bar{t}_1 \ldots \bar{t}_{N-1}\rangle^c$, where $\bar{t}_i$ = 0 (or 1) if t_i = 1 (or 0).

Definition 3: The ith terminal of any stage with i = $(p_{m-1}\, p_{m-2} \ldots p_0)$ is even-weighted if $\Sigma_{j=0}^{m-1} p_j$ = an even number, and is odd-weighted if $\Sigma_{j=0}^{m-1} p_j$ = an odd number.

Four useful network test vectors are now introduced. Let $(p_{m-1}\, p_{m-2} \ldots p_0)$ be the binary representation for i, denoting the ith input terminal of the network. We define the four test vectors as follows:

1. $T_0^c = \langle 00\ldots0\rangle^c$ i.e. $t_i = 0$ for $0 \leq i \leq N-1$
2. $T_1^c = \langle 11\ldots1\rangle^c$ i.e. $t_i = 1$ for $0 \leq i \leq N-1$
3. $T_2^c = \langle 01\ldots1\rangle^c$ where
   a. $t_i = 0$ for all even-weighted t_i.
   b. $t_i = 1$ for all odd-weighted t_i.
4. $T_3^c = \langle 10\ldots0\rangle^c$ where
   a. $t_i = 1$ for all even-weighted t_i.
   b. $t_i = 0$ for all odd-weighted t_i.

Note that $T_0^c = \overline{T_1}^c$ and $T_2^c = \overline{T_3}^c$. The four test vectors for a shuffle/exchange network with N = 8 are illustrated below:

Input terminal	T_0^c	T_1^c	T_2^c	T_3^c
0 0 0	0	1	0	1
0 0 1	0	1	1	0
0 1 0	0	1	1	0
0 1 1	0	1	0	1
1 0 0	0	1	1	0
1 0 1	0	1	0	1
1 1 0	0	1	0	1
1 1 1	0	1	1	0

Under the fault free condition, the test vector T_0^c will apply <00c> to each β-element in the (N,K) shuffle/exchange network and the test vector T_1^c will apply <11c> to each β-element in the (N,K) shuffle/exchange network. When the β-elements in a shuffle/exchange stage are all in state 0, then the input terminal $(p_{m-1}\, p_{m-2} \ldots p_0)$ to this stage is connected to the output terminal $(p_{m-2}\, p_{m-3} \ldots p_0\, p_{m-1})$ of the same stage [8]. When the β-elements in a shuffle/exchange stage are all in state 1, then the input terminal $(p_{m-1}\ p_{m-2}\cdots p_0)$ to this stage is connected to the output terminal $(p_{m-2}\ p_{m-3}\cdots p_0\ \bar{p}_{m-1})$ of the same stage [8].

Lemma 1 Under the fault free condition, the test vector $T_0^{\ 1}$ will apply <001> to each β-element and set all β-elements in a (N,K) shuffle/exchange network to the 0 state; the test vector $T_0^{\ 0}$ will apply <000> to each β-element regardless of the current states of the β-elements.

Proof: By definition, the test vector $T_0^{\ 1}$ will apply <001> to every β-element in the first stage. From the state table of a β-element, the outputs should be <00> and the next state will be state 0. The output from the first stage are all 0's, then the input to the following stage are all 0's too. Hence every β-element in the subsequent stages will receive the input vector <001>. $T_0^{\ 0}$ will apply <000> to each β-element in the first stage. The same foregoing argument applies except the next state is still the same as the current state. Δ

Lemma 2 Under the fault free condition, the test vector $T_1^{\ 1}$ will apply <111> to each β-element and set all β-elements in a (N,K) shuffle/exchange network to the 1 state; the test vector $T_1^{\ 0}$ will apply <110> to each β-element regardless of current states of the β-elements.

Proof: Similar to the proof of Lemma 1.

Theorem 1 Under the fault free condition, if every β-element in a (N,K) shuffle/exchange network is in state 0, then the test vector $T_2^{\ 0}$ (or $T_3^{\ 0}$) will apply the same input vector $\langle t_0 t_1 \ldots t_{N-1}\rangle^0$ to all K stages. Every β-element remains in state 0 after the test vector is applied to the network.

Proof: We know that the ith stage of β-elements simply connect the input terminal $(p_{m-1}p_{m-2}\cdots p_0)$ of stage i to the output terminal $(p_{m-2}p_{m-3}\cdots p_0 p_{m-1})$ of stage i, which is also the input terminal $(p_{m-2}p_{m-3}\cdots p_0 p_{m-1})$ of stage i+1, for all $1 \leq i < K$. Since the ith stage only permutes the terminal $(p_{m-1}p_{m-2}\cdots p_0)$, the input values t_j, $0 \leq j \leq N-1$, of the (i+1)th stage is the same as that of the ith stage. So if the test vector for the first stage is $\langle t_0 t_1 \ldots t_{N-1}\rangle^0$, then every subsequent stage will receive the same vector $\langle t_0 t_1 \ldots t_{N-1}\rangle^0$ under fault free condition. Since the control signal c is 0, the β-elements do not change states. Δ

Figure 7 illustrates Theorem 1. The current state, x, and the next state, y, of each β-element are denoted as x/y.

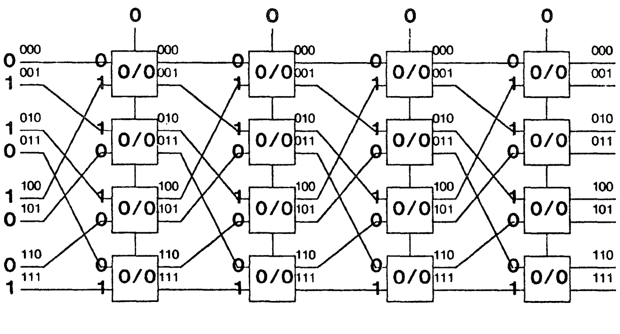


Fig. 7. Illustration of Theorem 1 using $T_2^{\ 0}$.

Theorem 2 Under the fault free condition, if every β-element in a (N,K) shuffle/exchange network is in state 1, then the test vector $T_2^{\ 0}$ (or $T_3^{\ 0}$) will apply the input vector $\langle t_0 t_1 \ldots t_{N-1}\rangle^0$ to the input terminals of the first stage of β-elements. For all the subsequent stages, $T_2^{\ 0}$ (or $T_3^{\ 0}$) will apply the same vector $\langle t_0 t_1 \ldots t_{N-1}\rangle^0$ to the input terminals of the odd stages, and the complemented vector $\langle \bar{t}_0 \bar{t}_1 \ldots \bar{t}_{N-1}\rangle^0$ to the input terminals of the even stages. All the β-elements remain in state 1.

Proof: The proof is similar to the proof of Theorem 1. Δ

4. C-Testable (N,K) Shuffle-Exchange Networks

Test Sequence I : Stimulation

The stimulation of each entry in the state table of every β-element is considered first. If the test sequence $\mathcal{T}_1 = \{T_0^{\ 0}\ T_1^{\ 0}\ T_2^{\ 0}\ T_3^{\ 0}\}$ is applied to a (N,K) shuffle/exchange network, with all its β-elements in state 0, the entries with current state = 0 and input vectors <000>, <010>, <100> and <110> will be stimulated. For convenience, the symbol [s,<xxx>] is used to denote the entry in the state table with current state s and input vector <xxx>. Thus the above stated entries are denoted as [0,<000>], [0,<010>], [0,<100>] and [0,<110>].

If all the β-elements are in state 1, by Theorem 2, the test sequence $\mathcal{T}_2 = \{T_0^{\ 0}\ T_1^{\ 0}\ T_2^{\ 0}\ T_3^{\ 0}\}$ will stimulate the following entries [1,<000>], [1,<010>], [1,<100>] and [1,<110>]. The stimulation of those entries in the state table involving state changes are now considered. Using similar arguments as in the proofs for Theorems 1 and 2, we can derive the following results.

Theorem 3 Under the fault free condition with all the β-elements in state 0, the test vector $T_2^{\ 1}$ will apply the same input vector to the input terminals of every stage of the network. All the β-elements $(\beta_{m-2}\ \beta_{m-3}\cdots\beta_0)$ with $\Sigma_{j=0}^{m-2}\beta_j$ = even will receive the input vector <011> and remain in state 0 and the β-elements with $\Sigma_{j=0}^{m-2}\beta_j$ = odd will receive the input vector <101> and then change from state 0 to state 1. (See Figure 8.)

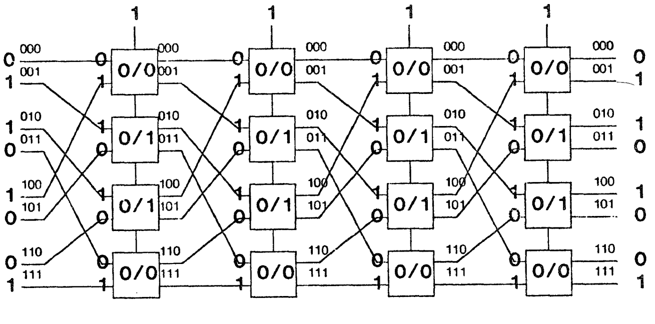


Fig. 8. Illustration of Theorem 3 using $T_2^{\ 1}$.

Corollary 1 Under the same circumstance as in Theorem 3, the test vector $T_3^{\ 1}$ will apply the same input vector to the input terminals of every stage of a (N,K) shuffle/exchange network. Every β-element with $\Sigma_{j=0}^{m-2}\beta_j$ = odd will receive the input vector <011> and remain in state 0, while every β-element with $\Sigma_{j=0}^{m-2}\beta_j$ = even will receive the input vector <101> and then change from state 0 to state 1.

Theorem 4 Under the fault free condition with all the β-elements in state 1, the test vector $T_2^{\ 1}$ will apply the input vector $\langle t_0\ t_1 \ldots t_{N-1}\rangle^1$ to all odd stages and the input vector $\langle \bar{t}_0\ \bar{t}_1 \ldots \bar{t}_{N-1}\rangle^1$ to all even stages. For a β-element in the jth stage, if j + $\Sigma_{j=0}^{m-2}\beta_j$ = odd, the β-element will receive the input vector <011> and change from state 1 to state 0, and if j + $\Sigma_{j=0}^{m-2}\beta_j$ = even, the β-element will receive the input vector <101> and remain in state 1. (See Figure 9.)

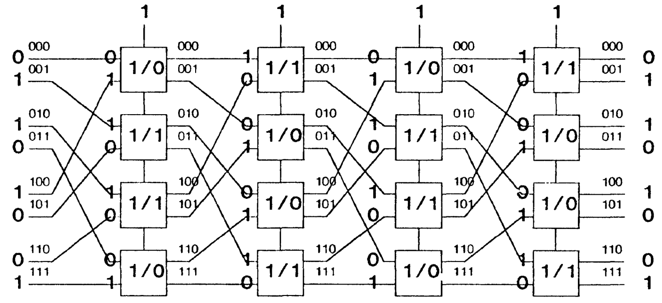


Fig. 9. Illustration of Theorem 4 using $T_2{}^1$.

Corollary 2 Under the same circumstance as in Theorem 4, the test vector $T_3{}^1$ will apply the input vector $\langle \bar{t}_0\ \bar{t}_1 \ldots \bar{t}_{N-1}\rangle^1$ to all odd stages and the input vector $\langle t_0\ t_1 \ldots t_{N-1}\rangle^1$ to all even stages. For a β-element in the jth stage, if j + $\Sigma_{j=0}^{m-2}\beta_j$ = odd, the β-element will receive the input vector <101> and remain in state 1, and if j + $\Sigma_{j=0}^{m-2}\beta_j$ = even, the β-element will receive the input vector <011> and change from state 1 to state 0.

From Lemma 1, Theorem 2 and Corollary 1, if all the β-elements in the network are in state 0 and the entire network is fault free, applying the test sequence $\mathcal{T}_3 = \{T_2{}^1\ T_0{}^1\ T_3{}^1\ T_0{}^1\}$ will stimulate the four entries, [0,<011>], [0,<001>], [0,<101>], and [1,<001>], in all the β-elements. After the test sequence, all the β-elements of the network will be back in state 0 as they were in before the application of this test sequence.

Another iest sequence $\mathcal{T}_4 = \{T_2{}^1\ T_1{}^1\ T_3{}^1\ T_1{}^1\}$ is useful when all the β-elements are in state 1. Again assuming the entire network is fault free, then from Lemma 2, Theorem 4 and Corollary 2, this test sequence will stimulate the four entries, [1,<101>], [1,<111>], [1,<011>], and [0,<111>], in all the β-elements. In the process of applying $\mathcal{T}_4$, all the β-elements start in state 1 and return to state 1. As can be seen, using the foregoing four test sequences, all sixteen entries in the state table of each β-element can be stimulated. Figure 10 illustrates the entries stimulated by the four test sequences.

next state/ outputs <xy>		input vector <abc>							
		000	001	010	011	100	101	110	111
current state	0	$\mathcal{T}_1$	$\mathcal{T}_4$	$\mathcal{T}_1$	$\mathcal{T}_4$	$\mathcal{T}_1$	$\mathcal{T}_4$	$\mathcal{T}_1$	$\mathcal{T}_3$
	1	$\mathcal{T}_2$	$\mathcal{T}_4$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_2$	$\mathcal{T}_3$

Fig. 10. Entries stimulated by $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4$.

Test Sequence II: Verification

Under the fault free condition, as the above test sequences are applied, the entire network at any point in time can be in only one of six possible states. Denote the jth β-element of ith stage with $(\beta_{m-1}\ \beta_{m-2} \ldots \beta_0)_i$ where $\beta_{m-1}\ \beta_{m-2} \ldots \beta_0$ = j. These six states of the network are:

- S_0 : all the β-elements are in state 0
- S_1 : all the β-elements are in state 1
- S_2 : 1. β-elements with $\Sigma_{j=0}^{m-2}\beta_j$ = even are in state 0

  2. β-elements with $\Sigma_{j=0}^{m-2}\beta_j$ = odd are in state 1
- S_3 : 1. β-elements with $\Sigma_{j=0}^{m-2}\beta_j$ = even are in state 1

  2. β-elements with $\Sigma_{j=0}^{m-2}\beta_j$ = .odd are in state 0
- S_4 : 1. every $(j)_i$ is in state 0 if i + $\Sigma_{j=0}^{m-2}\beta_j$ = odd

  2. every $(j)_i$ is in state 1 if i + $\Sigma_{j=0}^{m-2}\beta_j$ = even
- S_5 : 1. every $(j)_i$ is in state 1 if i + $\Sigma_{j=0}^{m-2}\beta_j$ = odd

  2. every $(j)_i$ is in state 0 if i + $\Sigma_{j=0}^{m-2}\beta_j$ = even

For each of the six possible network states, a set of network test vector(s) is needed to verify the current states of all the β-elements.

Definition 4: A set of input vector(s) to a (N,K) shuffle/exchange network is a distinguishing set of test vectors or simply distinguishing set if when they are applied to the network, a distinguishing sequence, i.e. the input vector <010> or <100>, is applied to each β-element in the network.

Theorem 5 $\{T_2{}^0\}$ is a distinguishing set, if the network is in state S_0 or state S_1. $\{T_3{}^0\}$ is also a distinguishing set for state S_0 or state S_1.

Proof: The β-element $(\beta_{m-2}\beta_{m-3} \ldots \beta_1\beta_0)$, receives its input a from the input terminal $(0\beta_{m-2} \ldots \beta_1\beta_0)$ and input b from the input terminal $(1\beta_{m-2} \ldots \beta_1\beta_0)$. When $T_2{}^0$ is applied to a network, a β-element will receive a <010> if $\Sigma_{j=0}^{m-2}\beta_j$ = even and a <100> if $\Sigma_{j=0}^{m-2}\beta_j$ = odd. Similarly, when $T_3{}^0$ is applied, a β-element will receive a <100> if $\Sigma_{j=0}^{m-2}\beta_j$ = even and a <010> if $\Sigma_{j=0}^{m-2}\beta_j$ = odd. From the discussion above, it is easy to see that either $T_2{}^0$ or $T_3{}^0$ will apply a distinguishing sequence to each β-element in the network if all the β-elements are in the same state. Δ

To verify that all the β-elements are in state 0, i.e. the network is in S_0, we need to apply the distinguishing set, $T_2{}^0$ (or $T_3{}^0$), to the network after the application of each of the four test vectors in $\mathcal{T}_1$. Similarily, $T_2{}^0$ (or $T_3{}^0$) is needed after each test vector in $\mathcal{T}_2$ to verify that all the β-elements remain in state 1. Note that each of these distinguishing sets consists of only one input vector.

Two particular test vectors, to be used for constructing distinguishing sets, are now defined. A test vector called $V_{odd} = \langle v_{N-1}\ v_{N-2} \ldots v_1\ v_0\rangle^0$ is defined as follows: v_i = 0 for all terminal i = $\langle p_{m-1}\ p_{m-2} \ldots p_0\rangle$, such that $\Sigma_{j=odd}\ p_j$ = even, and v_i = 1 for all terminal i such that $\Sigma_{j=odd}\ p_j$ = odd. Similarily, another test vector called V_{even} is defined as follows: v_i = 0 for all i such that $\Sigma_{j=even}\ p_j$ = even, and v_i = 1 for all i such that $\Sigma_{j=even}\ p_j$ = odd.

Theorem 6 $\{V_{odd}\}$ constitutes a distinguishing set for a (N,K) shuffle/exchange network, if the network is in one of the states, S_2, S_3, S_4, and S_5, and $\log_2 N$ is odd.

The proof of this Theorem is rather lengthy and is omitted here but is fully documented in [5].

Complete Test Sequence

Before the construction of a complete test sequence, we must show that any erroneous signal produced by a faulty β-element is always propagated to and visible at a network output. Any faulty β-element in a (N,K) shuffle/exchange network, will generate a faulty signal D, denoting a logic 0 becoming a faulty logic 1, or $\bar{D}$ denoting a logic 1 becoming a faulty logic 0, in response to the network test sequence. If a D or $\bar{D}$ singal from a previous stage arrives at an input terminal of a fault free β-element, the fault free β-element will always propagate the D or $\bar{D}$ to one of its output terminals. This is illustrated in Figure 11 for the case of the propagation of D. Since our fault model assumes that there is only one faulty β-element in a network, the faulty signal D or $\bar{D}$ once generated, will always be propagated to an output terminal of the Kth stage.

next state/ outputs <xy>		input vector <abc>							
		D00	D01	D10	D11	0 D0	0 D1	1 D0	1 D1
current state	0	0/ D0	D/ D0	0/ D1	D/ D1	0/0 D	0/0 D	0/1 D	1/1 D
	1	1/0 D	D/0 D	1/1 D	D/1 D	1/ D0	0/ D0	1/ D1	1/ D1

Fig. 11. Faulty signal propagation by a β-element.

We can construct a complete test sequence for a (N,K) shuffle/exchange network with no limitations on N and K by using the test sequences discussed above. This complete test sequence, under the fault free condition, will stimulate and verify every entry in the state table of every β-element. First, use T_0^1 to set all the β-elements into state 0. The test sequence $\mathcal{T}_1 \cup \mathcal{T}_4 = \{T_0^0\, T_1^0\, T_2^0\, T_3^0\, T_2^1\, T_0^1\, T_3^1\, T_0^1\}$ is then applied. After the application of this sequence all the β-elements should be in state 0. Now T_1^1 is applied to set all the β-elements into state 1. The test sequence $\mathcal{T}_2 \cup \mathcal{T}_3 = \{T_0^0\, T_1^0\, T_2^0\, T_3^0\, T_2^1\, T_1^1\, T_3^1\, T_1^1\}$ is then applied. The composition of the above test sequences produces a test sequence of length 18 consisting of the following 18 test patterns: $T_0^0 \cup \mathcal{T}_1 \cup \mathcal{T}_4 \cup T_1^1 \cup \mathcal{T}_2 \cup \mathcal{T}_3 = \{T_0^1\, T_0^0\, T_1^0\, T_2^0\, T_3^0\, T_2^1\, T_0^1\, T_3^1\, T_0^1\, T_1^1\, T_0^0\, T_1^0\, T_2^0\, T_3^0\, T_2^1\, T_1^1\, T_3^1\, T_1^1\}$. The applicaion of this set of 18 test patterns will fully exercise all 16 transitions in the state table of every β-element.

The application of each of the above 18 test patterns must be followed by a distinguishing set to verify that all the β-elements are indeed in the correct states. We assume that $m = \log_2 N$ is odd. Based on Theorem 5, if the network is in either state S_0 or S_1, the distinguishing set $\{T_2^0\}$ or $\{T_3^0\}$ can be used. If the network is in one of the remaining four states: S_2, S_3, S_4 or S_5, based on Theorem 6, $\{V_{odd}\}$ can be used as the distinguishing set. Hence, the complete test sequence must include the original 18 test patterns for stimulating all the state table entries of each β-element and another 18 test patterns, i.e. distinguishing sets, for the verification of network states. Hence the length of the complete test sequence, assuming $m = \log_2 N$ is odd, is 36 and is independent of the network size. The foregoing discussions lead to the following result.

__Theorem 7__ A (N,K) shuffle/exchange network, with $\log_2 N$ = an odd integer, is C-testable and requires a test sequence of 36 test patterns.

If $\log_2 N$ is an even integer, it is believed that the following conjectures are true:

__Conjecture 1__ $\{V_{odd}, V_{even}\}$ constitute a distinguishing set for a (N,K) shuffle/exchange network, if the network is in one of the following states, S_2, S_3, S_4, and S_5, and $\log_2 N$ is even.

Each of the four states: S_2, S_3, S_4, and S_5, occurs only once during the application of the original 18 test patterns for stimulation. Hence the distinguishing set $\{V_{odd}, V_{even}\}$, consisting of two test vectors, need to be applied only four times. Consequently, if $\log_2 N$ is even, the total number of test patterns needed for verification is 22 instead of 18. Hence,

__Conjecture 2__ A (N,K) shuffle/exchange network, with $\log_2 N$ = an even integer, is C-testable and requires a test sequence of 40 test vectors.

The state table of a β-element has 16 entries. In order to perform the exhaustive testing of a β-element, it is necessary and sufficient to stimulate all 16 entries and verify all the state transitions. A minimum of two test vectors are needed for every entry in the state table, thus a minimum of 32 test vectors are needed to exhaustively test a β-element. Furthermore, the initialization of the β-element requires another two test vectors. Hence the minimum number of test vectors required for testing an entire (N,K) shuffle/exchange network is at least 34. The test sets obtained in this section consisting of 36 or 40 test vectors, depending on whether $\log_2 N$ is odd or even, is believed to be the actual minimum.

__Conjecture 3__ The class of (N,K) shuffle/exchange networks is C-testable and the minimum number of test patterns required is 36 if $\log_2 N$ is odd, and is 40 if $\log_2 N$ is even.

A C-testable (16,5) shuffle/exchange network has been designed, using three micron NMOS technology, and a layout has been generated. Details of this design are documented in [5].

__5. References__

[1] Agrawal, D.P., "Testing and Fault Tolerance of Multistage Interconnection Networks," __Computer__, pp. 41-53, April, 1982.

[2] Friedman, A.D., "Easily Testable Iterative Systems," __IEEETC__, pp. 1061-1064, Dec., 1973.

[3] Hennie, F.C., __Finite State Models for Logical Machines__, Wiley, 1968.

[4] Kautz, W.H., "Testing for Faults in Celluar Logic Arrays," __Proc. Symp. Switch. and Autom.__, pp. 161-174, 1967.

[5] Lee, D.C.H., "Fault Diagnosis of (N,K) Shuffle/Exchange Networks," __M.S. Thesis__, EE Dept., CMU, Feb., 1983.

[6] Levitt, K.N., M.W. Green and J. Goldberg, "A Study of the Data Commutation Problems in a Self-Repairable Multiprocessor," __Proc. SJCC__, pp. 515-527, 1968.

[7] Masson, G.M., G.C. Gingher, and S. Nakamura, "A Sampler of Circuit Switching Networks," __Computer__, pp. 32-48, June, 1979.

[8] Shen, J.P., "Fault Tolerance Analysis of Several Interconnection Networks," __Proc. ICPP__, pp.113-122, 1982.

[9] Stone, H.S., "Parallel Processing with Perfect Shuffle," __IEEETC__, pp. 153-161, Feb., 1971.

~~~~~~~~~~~~~~~~~~~~~~~~~~

This research was supported in part by the Semiconductor Research Corporation (SRC) under contract No. 82-11-007.

* Mr. David C.H. Lee is currently with the Digital Equipment Corporation, 200 Forest Street, Marlboro MA 01752.

~~~~~~~~~~~~~~~~~~~~~~~~~~

FAULT TOLERANCE SCHEMES IN
SHUFFLE-EXCHANGE TYPE INTERCONNECTION NETWORKS

Krishnan Padmanabhan and Duncan H. Lawrie
Laboratory for Advanced Supercomputers
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ABSTRACT

As a solution to the fault tolerance problem of the shuffle-exchange type networks, a class of networks is proposed which provide non-unique paths between inputs and outputs. The topology of the multiple paths is specified by means of a redundancy graph and the procedure to construct a multipath network with a specified redundancy graph is presented. We provide practical schemes for utilization of the alternate paths' and evaluate how well they perform in the presence of faults in the network.

1. INTRODUCTION

Several topologically equivalent multistage interconnection networks have been proposed in the literature for applications in closely coupled multiple processor systems [3]. Such networks possess the property that between any input and any output there is a unique path made up of switching nodes. Breakdown of any such node or an edge thus makes some outputs inaccessible to certain inputs.

As a solution to this fault tolerance problem of the shuffle-exchange type networks, we introduce in this paper several classes of networks called Multipath Omega Networks. Such networks provide multiple ways of getting from an input to an output and their close relation to the Omega topology [5] helps maintain all the connection and control properties of the latter in a no-fault situation. Multipath networks behave as gracefully degrading systems, operating at a reduced level of performance in the presence of faults, but nevertheless providing full connectivity.

Several papers in the recent past have considered the idea of using more than one path to get from a source to a destination in multistage networks. Some networks inherently possess this property ([2], [9]), while others are obtained by augmenting an existing network to provide the multiple paths [1]. The extra stage approach in [1] will be seen to be a special case of the class of networks we discuss in this paper.

We present, in the next section, the theoretical development of the multiple path networks. We first introduce a convenient means of specifying the topology of the redundant paths, and show how a Multipath Omega Network of any size with a specified redundancy graph can be constructed. We then provide some schemes for implementing the networks and utilizing the alternate paths in practice, along with evaluations of their performance in the presence of faults.

2. MULTIPLE PATH NETWORKS

A $B^m \times B^m$ Omega Network [5] consists of m stages of $B \times B$ crosspoint switches with $B*B^{m-1}$ shuffles interconnecting the stages. A $P*Q$ shuffle is a permutation of PQ elements, $0 \leq i \leq PQ-1$, whose effect when i is represented as a $p+q$ bit binary number is to rotate left the bits by p positions. A 16×16 Omega Network constructed out of 4×4 switching elements is shown in Fig. 1. In its general form, an Omega Network of N inputs and N outputs, where N is an arbitrary integer, is constructed out of a set of switches the sizes of which correspond to a complete set of factors of N.

In an Omega Network, there is exactly one path between any source S and any destination D. Such a path can be characterized by concatenating the $n(=log_2N)$ destination address bits to the n source address bits [5]:

$$s_0 s_1 \ldots s_{n-1} d_0 d_1 \ldots d_{n-1} \qquad \ldots(1)$$

In addition, the terminal (of a switch) that a path occupies at the output of stage i $(0 \leq i \leq m)$ is given by the n-bit window in (1) starting at bit position bi where $b = \log_2 B$:

$$s_0 s_1 \cdots \boxed{s_{bi} s_{bi+1} \ldots s_{n-1} d_0 d_1 \ldots d_{bi-1}} d_{bi} \cdots d_{n-1} \qquad \ldots(2)$$

The existence of such a unique path between each source and destination is what leads to many of the useful properties of the Omega Network [5] and its distributed control algorithm. However, the uniqueness of paths implies that a fault anywhere in the network will destroy its connectivity. While reinforcing the links and the logic within the switches will mask some failures as when they occur [6], a more fundamental form of tolerance is provided by eliminating this uniqueness property and providing more than one way to get from a source to a destination. In this case, if one path is faulty, it *may* be possible to find an alternate route to the destination. We now characterize the construction and properties of Modified Omega Networks with multiple paths between sources and destinations.

An ordered factorization of N corresponds to an f-tuple $<B_1,B_2,\ldots,B_f>$ of factors satisfying $B_1B_2\ldots B_f=N$. The (1-path) Omega Network corresponding to such a factorization [5] consists of f stages of crosspoint switches, with stage i made up of $B_i \times B_i$ switches. In addition, each stage is preceded by the $B_i * \frac{N}{B_i}$ shuffle interconnection.

Define a *pseudofactorization* of N to be an f-tuple $<B_1,B_2,\ldots,B_f>$ of integers, with $B_1B_2\ldots B_f=B$, that satisfy

This work was supported in part by the National Science Foundation under Grants No. US NSF MCS81-00512 and US NSF MCS80-01561, the US Department of Energy under Contract No. US DOE DE-AC02-81ER10822, and by the Department of Computer Science at the University of Illinois at Urbana-Champaign.

0190-3918/83/0000/0071$01.00 © 1983 IEEE

the following conditions:

$$B > N \quad \text{and} \quad B/B_j \leq N \;\; 1 \leq j \leq f.$$

Let B/N be equal to R. Then *an* R-path Omega Network corresponding to the above pseudofactorization consists of f stages of crosspoint switches with stage i consisting of $B_i \times B_i$ switches; each stage i is preceded by k_i $B_i \times \frac{N}{k_i B_i}$ shuffles ($k_i \geq 1$) such that there are exactly R ways to get from each source to each destination.

We will refer to R as the *redundancy* of the multipath Omega Network. Figs. 2 and 4 show two 4-path 16×16 Omega Networks (corresponding to the pseudofactorizations <2,4,2,4> and <2,2,4,4> of 16). The two networks are different in the manner in which the four different paths from a source to a destination interact. The topology of the redundant paths is specified by means of a *redundancy graph* indicated below each network.

2.1 Redundancy Graphs

A *redundancy graph* is a flow graph with the following restrictions:

(1) The set of nodes in the graph is divided into S classes corresponding to the S stages of switches in the network.

(2) Each edge connects a node in class i to a node in class $i+1$, $1 \leq i \leq S\text{-}1$.

(3) The in-degrees of all nodes in a class are the same and so are the out-degrees of all nodes in a class.

Three examples of redundancy graphs are shown in Fig. 3. Fig. 3a corresponds to a *disjoint path network* (all redundant paths are disjoint) [7], while Fig. 3c is the redundancy graph of a standard (1-path) Omega Network. The nodes in the graph correspond to switches in the network and the edges to links. In that sense the redundancy graph is a subgraph of the network and the subgraph connecting any input to any output in the network will be isomorphic to the redundancy graph. The number of faults a network can tolerate is given by λ-1, where λ is the *line connectivity* of its redundancy graph.

The control scheme for setting up the paths in such a network is the distributed tag control scheme, much like the one for the standard Omega Network. Each stage i is controlled by $b_i = \log_2 B_i$ bits so that the entire destination tag consists of $\sum b_i$ bits. The difference, of course, is that the destination tag in an R-path network consists of $n+r$ bits where $r = \log_2 R$. Only n of these bits are the destination address bits $d_0 d_1 \ldots d_{n-1}$ and a particular path out of the R alternates is chosen by a specific setting of the r *redundant* bits. We go into this in more detail in section 2.2. The broad scheme for using the multiple paths is to *backtrack*, in the event of a fault, up to the point of the last fork and then take an alternate route. This backtracking can also be done in case of blocking along one of the paths. Referring to Fig. 3, it can be seen that the graphs for a given S and R differ basically in the following three aspects:

(1) The stage(s) at which fork/join is done

(2) The magnitude of the fork/join done at each stage

(3) The number of disjoint paths between source and destination

The effect of these variables on the performance and cost of the system is considered in [8].

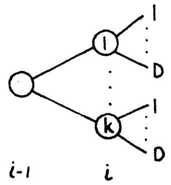
2.2 Derivation of NW from Redundancy Graph

A redundancy graph specifies both the number of stages S and the redundancy R. The total number of bits needed to control the network, irrespective of how it is constructed, is $n+r$. The only variables are the sizes of the switches used in different stages and the distribution of the redundant bits among the S stages. Note that at any stage the smallest switch that can realize an out-degree (or in-degree) D is a $D \times D$ switch.

The terminal an input-output path occupies at the output of stage i is given by the n-bit window defined earlier in (2). Consider such a window W_i at stage i:

$$\underset{\text{switch}}{w_0 w_1 \ldots w_{n-b-1}}\;\underset{\text{terminal}}{w_{n-b} \ldots w_{n-1}}$$

For a fork size of D at this stage, exactly $d = \log D$ of the redundant bits $r_0 r_1 \ldots r_r$ should be a part of the subwindow $w_{n-b} \ldots w_{n-1}$; this will ensure that from the same switch D paths will fork out. Similarly, if at this stage there are k disjoint paths in the graph, i.e., there are k nodes at this stage in the redundancy graph, then $\log k$ of the redundant bits should be a part of the subwindow $w_0 w_1 \ldots w_{n-b-1}$. The above two conditions will yield the subgraph shown below at stage i:



Consider joins now. A join of size D at stage i reduces the number of disjoint paths by a factor of D, i.e., d of the redundant bits in window W_{i-1} are replaced by destination bits in W_i. The b_i tag bits at stage i always replace the least significant b_i bits in W_{i-1}; hence the removal of the d redundant bits is achieved by choosing an appropriate shuffle to precede stage i. This may necessitate choosing shuffles other than the $B_i * \frac{N}{B_i}$ shuffle.

Parallel edges correspond to a join immediately following a fork; in this case, the redundant bits introduced in the terminal subwindow $w_{n-b} \ldots w_{n-1}$ in stage i-1 should be replaced by d-bits at stage i. For a join in the absence of parallel edges, r-bits in the switch subwindow $w_0 w_1 \ldots w_{n-b-1}$ should be replaced (resulting in a net reduction in the number of redundant paths). This can, in general, be achieved by choosing the shuffle connection preceding the stage appropriately.

Other than the above conditions and the inclusion of the right number of r-bits and a minimum number of d-bits at each stage, we have a lot of freedom in choosing the sizes of switches at each stage.

Consider, as an example of the above procedure, the construction of a 4-path 16×16 Omega Network with the redundancy graph shown in Fig. 4. We have $r=2$ (redundant bits r_0 and r_1) and $n=4$ (destination address bits d_0, d_1, d_2, and d_3). Since stages 1 and 2 involve a binary fork each, r_0 has to be a part of stage 1 and r_1, of stage 2. Stages 3 and 4 involve a join each and hence one r bit each should be replaced by d-bits in these stages; at stage 3, the r-bit introduced in stage 2 (in the terminal subwindow) should be replaced and at stage 4 the r-bit introduced in stage 1 should be replaced. There are many ways to distribute the remaining two d-bits among the four stages. Consider the following distribution as an illustration:

$$r_0d_0 \quad r_1 \quad d_1 \quad d_2d_3$$

This would correspond to using 4×4 switches in the first and last stages and 2×2 switches in the intermediate stages. Let us determine the connections to precede each stage to realize the above redundancy graph. A $4*4$ shuffle preceding stage 1 would give $W_1=s_3s_4r_0d_0$. Thus at the output of this stage, we have two alternate terminals ($s_3s_40d_0$ and $s_3s_41d_0$) that a path could occupy to get to the same destination. A $2*8$ shuffle preceding stage 2 will make $W_2=s_4r_0d_0r_1$. Now the redundancy is increased to four, with the four paths occupying two different switches (s_40d_0 and s_41d_0). In stage 3, r_1 must be replaced by d_1 (to ensure the join and parallel edges); hence stage 3 is preceded by the identity connection making $W_3=s_4r_0d_0d_1$. Stage 4 is just preceded by the $4*4$ shuffle leading to $W_4=d_0d_1d_2d_3$, the correct destination. This results in the network shown in Fig. 4.

3. IMPLEMENTATION SCHEMES FOR DISJOINT PATH NETWORKS

Disjoint path networks have been dealt with in detail in [7]. They provide the highest tolerance to faults (among all multiple path networks) - an R-path network of this type can tolerate (R-1) arbitrary internal stage faults and potentially many more. The set of internal switches and links in such a network can be divided into R disjoint classes; a path from a source to a destination consists of (internal) switches and links from only one such class. In addition, the path has counterparts in each of the other R-1 classes. Classes containing no faults in essence support the paths that would encounter faults in the other classes.

A modular organization of a $B\times B$ crosspoint switch is shown in Fig. 5a. A connection is established in the switch when an input module is connected to an output module. When a request is made to an input module i along with a base B address j, the input module i requests module j for connection; if the output module is not currently in use (and if it is not faulty - a situation we will consider shortly), such a connection can be set up. Following this setup, direct connections are available between input i and output j in the switch for all control and data signals. The technique to set up a sequence of such switches is the *set-and-forward* scheme. In cycle $2i$-1, stage i is set up and in cycle $2i$, the address for stage $i+1$ is passed on to the next stage by stage i. In $2S$ cycles, the entire path is set up (if no *exceptions* arise in between).

Two exceptions could arise in the process of setting up such a path - a block, and a fault. A block is signalled first by the BL line in the switch where the block occurred, and then propagated back to the source. Once a fault is detected, generation and propagation of the F signal takes place in a manner similar to that of the block signal.

3.1 Fault Recognition

Most published research on fault analysis in networks ([4], [10]) have used as the model of a fault an *entire switch* being stuck at one of its states. We outline here a more realistic model used in [8]. We shall consider each module along with the lines leaving it as our ***units***, in the sense that we shall not be concerned with the logic within the blocks (Fig. 5c). Such a unit will be considered to be *faulty* if any value of the input vector does not produce an appropriate output vector.

Such a general class of faults can be detected by a *replication check*, where we duplicate or triplicate every module in the switch. (See [6] for such a design.) We propose a *self-checking* scheme inherent in our protocol, which is less versatile than replication, but requires little increase in hardware. The purpose of every control signal is to *exercise* a portion of the logic in the receiving module. If a plausible response is elicited for an output signal (resulting in a proper handshake as specified in Fig. 5b), it is reasonable to expect that the portion of the logic exercised by that signal works properly. Consider Fig. 5c. When an output module B initiates a handshake by asserting the REQ line, it monitors the three input lines ACK, BL, and F. If none, or more than one of these is asserted, the unit C is assumed to be faulty. If B did not assert the REQ line properly as it should have, it will not find any input signal asserted and will know of a fault. (Note that the return-to-zero protocol detects all line stuck-at faults.) Thus each module checks a portion of the logic in the next module as a path gets set up. When a fault is detected, two actions are taken by the detecting module. It makes a note of this by setting a *fault flag* within the module. And it signals the fault back by asserting the F line which propagates back to the source. By setting the fault flag in the module, subsequent requests to the module are notified immediately of the fault. (Note that the state-updating portion of the logic in the second module is not exercised by handshaking.)

We have considered here only the control portion of each module. To ensure that data (which includes address and data bits) is transferred properly from module to module, an encoding scheme can be used.

3.2 Fault Notification

For the purposes of this section it is convenient to consider each sequence of the form output module→link→input module as a single entity called an *element*. The reasons for this are twofold. First, faults anywhere within such an element affect the operation of the network in the same manner. Second, the terminal an input-output path occupies in a stage is given by such an element. The network, in this view, consists of $S+1$ stages of elements. There are several options for notification of faults back to the sources and the subsequent action to be taken.

Non-adaptive Routing: In this approach, each path learns of a fault only when it reaches the faulty element. When the fault signal reaches the source, it sends the same request out on the next alternate path, without retaining knowledge of the presence of the fault along the path just tried. Advantages of this scheme are that nonfaulty paths will be utilized to the maximum extent and that there is almost no additional hardware required. The drawback of the approach is that a long time could be spent trying alternate paths (especially if the faults are located in the later stages of the network).

Adaptive Routing - Notification on Demand: In this scheme, knowledge of the location of the faults encountered in various tries is maintained at the sources and is used in subsequent routing decisions. A source learns of a faulty element upon requesting it the first time. It can determine, by keeping count of the number of clock cycles after which the fault signal is received, where along the path (the stage and element) the fault is located. The ID's of the faults thus determined can be stored in a set of B tables associated with the B terminals that a source could branch into at the first stage. Trying a path now entails first checking the appropriate table to see if the path would encounter any of the faults currently in the table.

Adaptive Routing - Broadcast Notification: This is a conceptual scheme that would be difficult to implement in practice. However the network performance under this scheme will provide us a standard to evaluate the previous proposals. Under the broadcast scheme, notification of a fault is done immediately upon recognition (or occurrence) to all the sources that could potentially use that element. Once the fault ID's reach the sources, they are handled in exactly the same manner as in the demand notification scheme.

In the two adaptive schemes above, the size of the fault tables maintained at the sources is an important parameter. When a table of size T is maintained, a terminal will have to be shutdown when more than T faults are reported at that terminal. A larger T increases not only the logic complexity and the storage required, but also the initial overhead associated with searching it. Reducing the table size would close down terminals faster leading to increased traffic along fewer paths and earlier shutdown of the system. The effect of T on the performance is considered in the next section.

3.3 Some Evaluation Studies

Fig. 6 presents the average normalized delay in a 64×64 network as a function of the percentage of faulty elements, for a variety of cases. Normalized delay is defined as the ratio of the actual delay to the minimum delay. (Faults are permitted only in the internal stages of the network.) The request rate m is the probability of a source making a new request in a cycle when it has no outstanding request; requests are not allowed to be queued at the sources. The 2-path network is constructed using seven stages of 2×2 switches; the 4-path network uses four stages of 4×4 switches.

Let us briefly account for the shape of the curves first. Recall that the set of intermediate elements can be divided into R classes and that the non-faulty classes support the paths that will encounter the faulty elements. As the number of faults increases, so does the number of paths that a non-faulty class supports. Effectively then, the load on the non-faulty paths is much higher with a higher redundancy. This accounts for the higher delay of the 4-path networks as F gets higher. *Breakdown* of the system is said to occur when all R paths from some source to some destination contain faults. To obtain the numbers in Fig. 6, we have kept the system running after such breakdowns.

The non-adaptive scheme performs very close to the adaptive scheme and does better than the latter with very small table sizes because it uses non-faulty elements to the maximum possible extent. The $T=0$ option in the adaptive routing scheme closes down terminals (or classes) much earlier than necessary leading to increased load on the rest of the alternates. This overkill results in it performing worse than the non-adaptive scheme.

4. CONCLUSION

We have, in this paper, introduced schemes for fault-tolerance in shuffle-exchange type networks based on redundant paths between inputs and outputs. The interaction of the multiple paths between each source and destination is specified by means of a redundancy graph, and the derivation of a network with a given redundancy graph is discussed. We have also considered some practical schemes for using the multiple paths in practice and shown that some inexpensive schemes provide good performance in the presence of faults.

REFERENCES

[1] G.B. Adams and H.J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Trans. Computers*, Vol. C-31, No. 5, May 1982, pp 443-454.

[2] V.E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic,* Academic Press, New York, 1965.

[3] *Computer,* IEEE Press, Vol. 14, No. 12, December 1981.

[4] T.-Y. Feng and C.-L. Wu, "Fault-Diagnosis for a Class of Multistage Interconnection Networks," *IEEE Trans. Computers*, Vol. C-30, No. 10, Oct. 1981, pp 743-758.

[5] D.H. Lawrie, *Memory-Processor Connection Networks*, Department of Computer Science Report No. UIUCDCS-R-73-557, University of Illinois at Urbana-Champaign, February 1973.

[6] C. Leung and J. Dennis, *Design of a Fault-Tolerant Packet Communication Architecture*, MIT Laboratory for Computer Science, Computation Structures Group Memo 196, July 1980.

[7] K. Padmanabhan and D.H. Lawrie, "A Class of Redundant Path Multistage Interconnection Networks," To appear in *IEEE Trans. Computers.*

[8] K. Padmanabhan and D.H. Lawrie, *Techniques for Fault Tolerance in Omega Type Interconnection Networks*, Cedar Document 11, Laboratory for Advanced Supercomputers, University of Illinois at Urbana-Champaign, February 1983.

[9] D.S. Parker and C.S. Raghavendra, "The Gamma Network: A Multiprocessor Interconnection Network with Redundant Paths," *Proceedings of the 9th Annual Symposium on Computer Architecture*, 1982, pp 73-80.

[10] J.P. Shen and J.P. Hayes, "Fault Tolerance of a Class of Connecting Networks," *Proceedings of the 7th Annual Symposium on Computer Architecture*, 1980, pp 61-71.

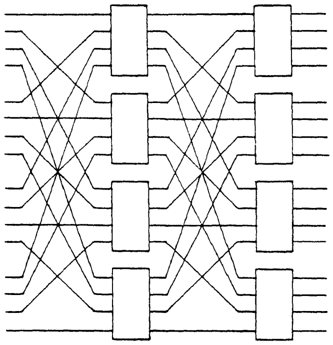


Fig. 1. A 16×16 Omega Network

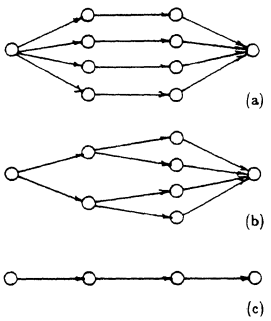


Fig. 3. Examples of redundancy graphs

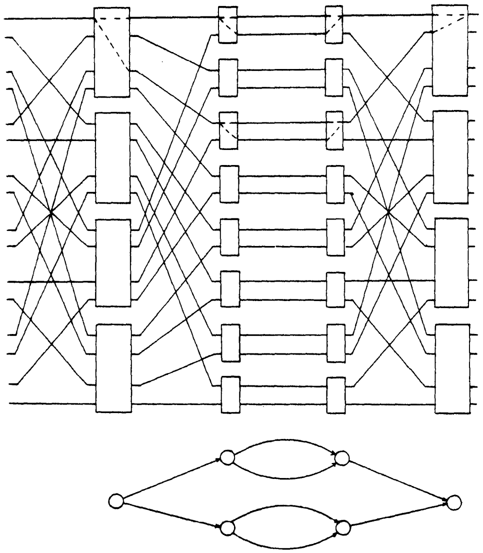


Fig. 4. The 4-path Omega Network considered in sec. 2.2

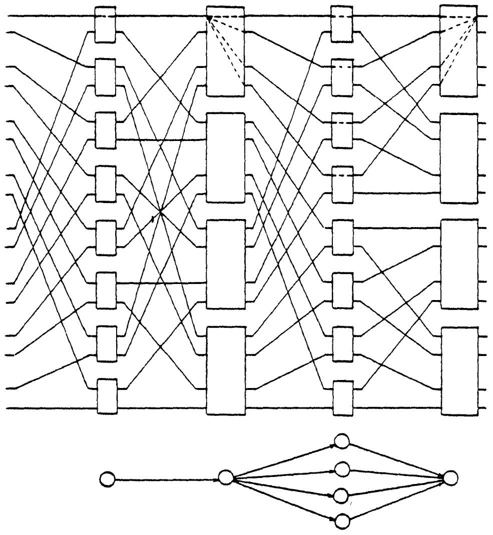


Fig. 2. A 4-path Omega Network and its redundancy graph

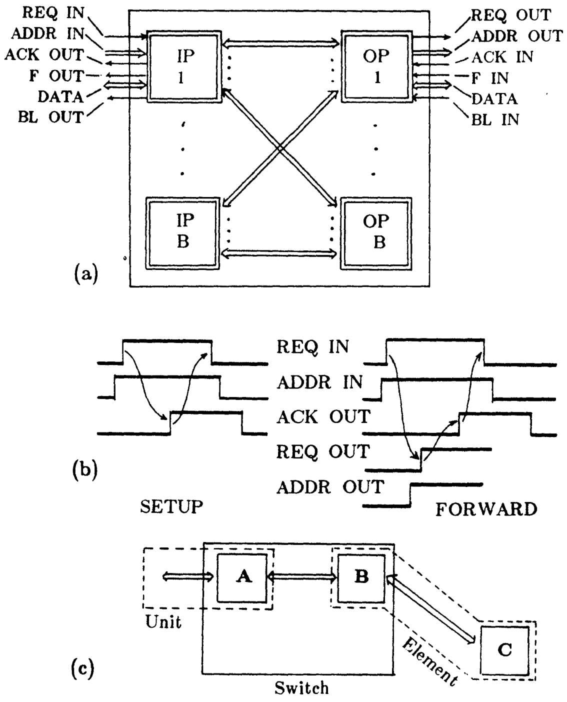


Fig. 5 a. Modular organization of a $B \times B$ switch
b. Handshake protocol between switches
c. Definition of unit and element

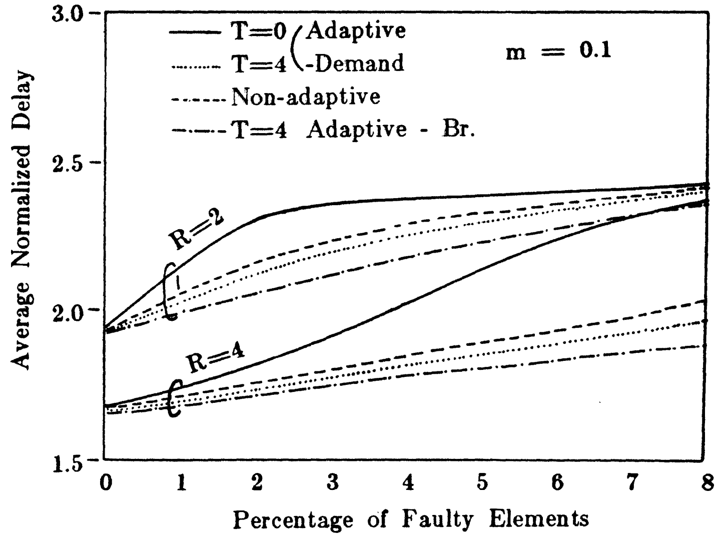


Fig. 6 Average normalized delay in the network

A CONDITION KNOWN TO BE SUFFICIENT FOR REARRANGEABILITY OF THE BENES CLASS OF INTERCONNECTION NETWORKS WITH 2X2 SWITCHES IS ALSO NECESSARY

S.C. Kothari and S. Lakshmivarahan
School of Electrical Engineering and Computer Science
University of Oklahoma
Norman, Oklahoma 73019

Abstract

An NxN (N inputs and outputs) multistage interconnection network is said to be rearrangeable if it can realise all the possible connections of the N input terminals to the N output terminals in a one-to-one fashion. Starting with the pioneering works of Clos and Benes to this date a variety of sufficient conditions for rearrangeability are known in the literature. In this paper it is shown that the well known sufficient condition due to Benes on the link permutation is also necessary for rearrangeability if the network is made up of 2x2 switches.

Introduction

An NxN switching network (with N inputs and N outputs) is an arrangement of switches which is capable of performing certain permutations of inputs. The combinatorical power (CP) of a given switching network is often measured [2] as the ratio of the number permutations the network can realise to the total number of possible permutations of N inputs. Clearly $0 \leqslant CP \leqslant 1$. A switching network is said to be rearrangeable if CP=1, that is, there exist settings of its component switches such that, the switching network as a whole, can realise all the N! permutations. The well known NxN cross-bar switch has CP=1. Most of the early studies on (rearrangeable) switching networks have been exclusively in the context of telephone networks [1][2][6]. The interest in multistage (rearrangeable) switching networks revived recently in connection with the development of parallel computers and parallel algorithms [4][13]. At the present time there is considerable interest in the interconnection network as evidenced by the special issue on this topic [8] as well as the extensive bibliography in [6][12].

Clos in 1953[1] in a fundamental paper exhibited for the first time a three stage switching network which is rearrangeable. Benes in a series of papers analysed a rich class of switching networks from the point of view of rearrangeability. Among many other interesting results, Benes gave a method for the design of a multistage switching network made up of odd number of stages and with square (cross-bar) switches. We call this class of networks as the general Benes class (GBC). The contributions by Benes are succinctly summarised in his now classic book [2]. Most of our notations follow those of Benes [2]. Consider the following subclass of the GBC defined recursively using 2x2 switches. For easy reference, we call this subclass as the class of block structured network (BSN). Let $N=2^n$.

Definition 1: An NxN block structured network is made of three (3) stages. The first and the third stages are each made of 2x2 switches and there are N/2 of them in each of these stages. The middle stage however, has two copies of N/2 x N/2 block structured networks called blocks A and B. The N outputs of stage i are connected to N inputs of stage i+1 by an interconnection scheme $\emptyset_i$ called the link permutation i=1,2. Referring to figure 1, let X_1, $X_2 \ldots X_{N/2}$ and $Y_1, Y_2, \ldots Y_{N/2}$ denote the 2x2 switches at the first and third stages. Notice 2i-1 and 2i are the inputs of X_i and 2j-1 and 2j are the outputs of Y_j. In the following we define a special class of link permutations.

Definition 2: The link permutation $\emptyset_1(\emptyset_2)$ in the block structured network is said to be distributive if the two outputs (inputs) of each of the switch $X_i(Y_j)$ are connected one to each of the two blocks A and B at the input (output). Similarly, when the blocks are furthur reduced to three stage networks of proper sizes, the link permutations in each of these reductions could be required to satisfy the above condition of distributivity. From theorem 3.9 of Benes [2], it readily follows that a block structured network is rearrangeable if all the link permutations are distributive. In this paper, we prove that distributivity of all the link permutations is indeed necessary for the class of block structured networks to be rearrangeable.

Main Result

Our main result is the contents of the following:

Theorem: A block structured network is rearrangeable if and only if all the link permutations are distributive.

Proof: The if part follows from the theorem 3.9 of Benes [2]. To prove the only if part, first assume that $\emptyset_1$ is not distributive but $\emptyset_2$ is. Referring to figure 2, let both the outputs of X_i be connected to the block A. Now any permutation which takes both the inputs to the switch X_i to the output of a single switch Y_k (say) for any k=1,2...N/2 is not realisable by the overall network. A similar argument follows if $\emptyset_2$ or $\emptyset_1$ and $\emptyset_2$ are not distributive [15].

The above analysis clearly establish the fact that for the rearrangeability of the overall network it is necessary that $\emptyset_1$ and $\emptyset_2$ be distributive. Now to show that all the link permutations that are embeded in the blocks A and B must also be distributive, assume without loss of generality, there is a link permutation which is part of the block A that is not distributive. Then, by inductive hypothesis it follows that there exists a permutation η(on N/2 objects) which is not realisable by block A. Since $\emptyset_1$ is distributive, there exists subassignment Ψ_1 of the switches X_i i=1 to N/2 to the inputs of block A which are numbered from 1 to N/2. That is, referring to figure 3 $\Psi_1(X_i)=j$ if an output of the switch X_i is connected to input j of block A under $\emptyset_1$. Similarly, $\Psi_2(r)=Y_s$ if the output terminal r of block A is

0190-3918/83/0000/0076$01.00 © 1983 IEEE

connected to an input of the switch Y_s under $\emptyset_2$. Define

P_A: $\{X_s | s=1 \text{ to } N/2\} \to \{Y_r | r=1 \text{ to } N/2\}$ where $P_A=\Psi_2.\eta.\Psi_1$. Consider a permutation p (on N objects) which takes the two inputs to switch X_i to the two outputs of Y_j where $p_A(X_i)=Y_j$. This permutation p is clearly not realisable by the overall network since η is not realisable by block A. The proof that distributivity is necessary when n=2 (the basis for the induction) is very similar to the one presented above. Hence the theorem.

COMMENTS

1. In all the examples of the Benes network given in the literature [3][9][10], it is assumed that $\emptyset_2=\emptyset_1^{-1}$ and $\emptyset_1$ is distributive. Our result shows that one can choose $\emptyset_1$ and $\emptyset_2$ independently so long as they are distributive.

2. One of the interesting open questions in the context of the shuffle-exchange-type networks is whether or not two passes of the Omega network is rearrangeable [14]. Looking at this problem from the point of view of necessary conditions, we found out, for N=8, by simply rewriting the network and fixing the states of certain switches as in figure 4, that two passes of Omega network is rearrangeable. In particular, the first and third switches in the 4th stage of figure 4 are set to the "straight" (S) state but the second and fourth switches are set to the "cross" (C) state. If the state of a switch is fixed, it can as well be replaced by permanent connections. The link permutation between stages 3 and 5 resulting from this elimination of switches in the 4th stage is shown in figure 5. The rest of the stages 1,2,3, 5, and 6 as well as the inputs to the stage 1 in figure 5 are all obtained by permuting the switches in figure 4. A closer examination of figure 5 reveals that it is a block structured network where all the link permutations are distributive. Hence, by our theorem it is rearrangeable. Since rearrangeable networks remain rearrangeable even if the inputs are permuted, the input terminals in figure 5 without loss of generality can be renumbered in the natural order 1 through 8 instead of as shown in figure 5. It is interesting to note that while Parker [14] has also arrived at the same conclusion for N=8, he uses brute force enumeration method instead.

REFERENCES

[1] C. Clos. "A Study of Non-Blocking Switching Networks." The Bell System Technical Journal, Vol. 32, 1953, pp. 406-424.

[2] V.E. Benes. Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, 1968.

[3] A. Waksman. "A Permutation Network." Journal of ACM, Vol. 15, 1968, pp. 159-163.

[4] T. Feng. "A Survey of Interconnection Networks." Computer, Vol. 14, 1981, pp. 12-27.

[5] H. J. Siegel. "Interconnection Networks for SIMD Machines." Computer, Vol. 12, 1979, pp. 57-66.

[6] K.J. Thurber. "Interconnection Networks-A Survey and Assessment." Proceedings of NCC, 1974, pp. 909-919.

[7] N. Pippenger. "On Rearrangeable and Non-Blocking Switching Networks." Journal of Computer and System Science. Vol. 17, 1978, pp. 145-162.

[8] C. Wu. (Editor) "Interconnection Networks." Special Issue. Computer, Vol. 14, Issue number 12, December 1981.

[9] D.C. Opferman and N.T. Tsao-Wu. "On a Class of Rearrangeable Switching Network, Part I Control Algorithm." The Bell System Technical Journal. Vol. 50, 1971, pp. 1579-1600.

[10] R. W. Hockney and C.R. Jesshope. Parallel Computers. Adam Hilger Ltd., Bristol 1981.

[11] D.H. Lowrie. "Access and Alignment of Data In An Array Processor." IEEE Transactions on Computers. Vol. 24, 1975, pp. 1145-1155.

[12] J. T. Schwartz. "Ultra Computers." ACM Transactions on Programming Languages, Vol. 2, 1980, pp. 484-521.

[13] H.S. Stone. "Parallel Processing with Perfect Shuffle." IEEE Transactions on Computers, Vol. 20, pp. 153-161.

[14] D.S. Parker Jr. "Notes on Shuffle/Exchange Type Switching Networks." IEEE Transactions on Computers, Vol. 29, 1980, pp. 213-222.

[15] S.C. Kothari and S. Lakshmivarahan. "A condition know to be sufficient for rearrangeability of the Benes class of interconnection networks with 2x2 switches is also necessary." Technical Report, School of EECS, University of Oklahoma, January 1983.

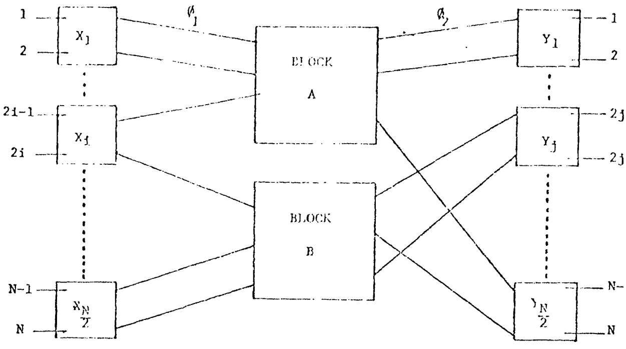


Figure 1: BLOCK STRUCTURED NETWORK

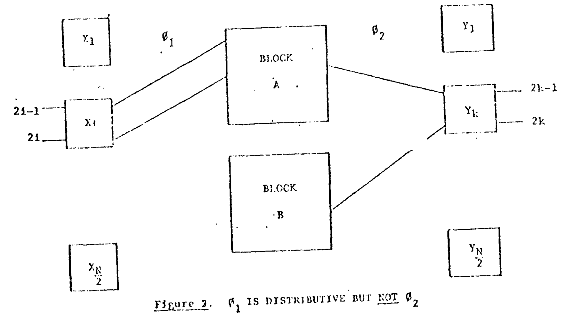


Figure 2. ϕ_1 IS DISTRIBUTIVE BUT NOT ϕ_2

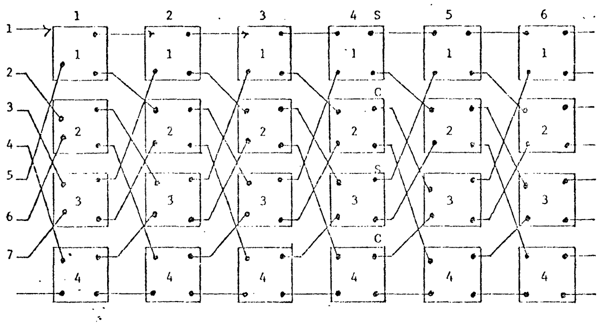


Figure 4. TWO PASSES OF OMEGA NETWORK

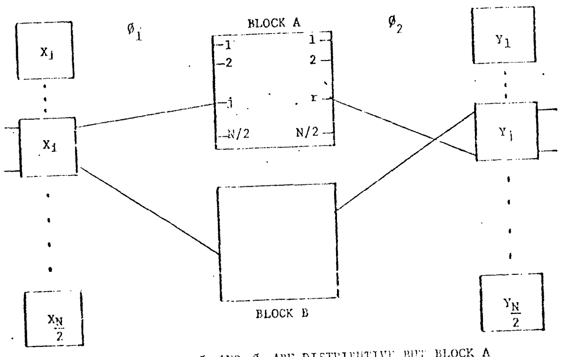


Figure 3. ϕ_1 AND ϕ_2 ARE DISTRIBUTIVE BUT BLOCK A HAS A LINK PERMUTATION THAT IS NOT DISTRIBUTIVE

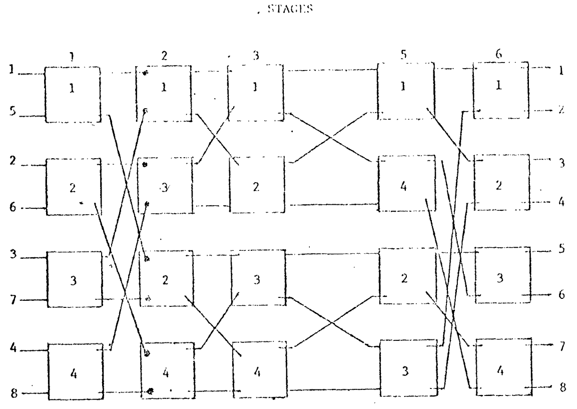


Figure 5. A BLOCK STRUCTURED NETWORK OBTAINED FROM 8A

A FAST ALGORITHM FOR CONCURRENT LU DECOMPOSITION AND MATRIX INVERSION*

Ming-Yang Chern and Tadao Murata
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
P.O. Box 4348, Chicago, Illinois 60680

Abstract -- This paper presents an efficient algorithm for LU decomposition and matrix inversion based on the concurrent data-loading array architecture. The algorithm performs the LU decomposition of a strongly nonsingular matrix A initially loaded in the array, in parallel with computing the inverse matrices L^{-1}, U^{-1}, and A^{-1} : the L^{-1} and U^{-1} can be taken out together with L and U; and A^{-1} will appear in the array at the end of the computation. For an n x n matrix executed on the array of the same size, the total time required for the above computation is $n(t_a+t_m+t_d)$; where t_a, t_m, and t_d represent the time for addition, multiplication, and division, respectively. A simple augmentation of this algorithm can lead to the solution of a linear system of equations without additional time. The performance of this algorithm is analyzed and compared favorably with an improved version of systolic arrays.

I. Introduction

Many scientific and engineering problems can be reduced to the problem of solving linear systems of equations (LSEs). The recent availability of low cost, high density, fast VLSI devices has opened a new avenue for using processor arrays to perform special-purpose parallel computations. Efficient algorithms and cost-effective hardware structures have been intensively searched. The VLSI computing structures related to solving LSEs have been suggested by several researchers. Kung [3,4] proposed systolic arrays which can be used for LU decomposition, matrix multiplication, and linear convolution. Preparata and Vuillemin [7] analyzed an implementation of triangular matrix inversion from small construction modules. Hwang and Cheng [5] presented a complete set of computational structures for solving LSEs. This set includes some special modules for LU decomposition, solving a triangularized LSE, triangularized matrix, and matrix multiplication.

In Hwang and Cheng's design, the modulized construction units can also be integrated to LU-decompose large-scale matrices as reported in their partitioned matrix algorithms [6]. These modules and linear equation solvers were recently applied to a system for image processing and database management [9]. The usefulness of these modules is evident. However, the utilization of processing elements (PEs) in these modules is still low. Moreover, these modules, like other systolic arrays, suffer from the data reshaping problem. The data of a dense matrix must be reshaped before being fed into the processor array. The scheme of using on-chip delay latches can solve the problem partially, but evoke a more severe problem in the array chip interconnection.

The process of solving LSEs in the above designs is divided into several steps, where a different hardware module is used in each step. Some of these steps can not be executed concurrently. For example, the computation of U^{-1} can not start unless the LU decompostion is completed. In addition, the output data of a step may not be arranged in the same way as required by the input of the next computational module. Some special provision must be arranged. This may need extra hardware and cause extra time delay. According to [6], four types of VLSI arithmetic module chips are required. Four is not a large number. However, it is still desirable to use fewer types of module chips.

This paper presents a highly efficient algorithm based on a Concurrent Data-Loading Array Processor (CDLAP) [2]. Only one type of module chip is required for the construction of LSE solvers. According to our algorithm, the CDLAP can perform LU decomposition a little faster than Hwang and Cheng's design [5], and the inverse matrices of U, L, and A can all be obtained in the same process. Further, the solution vector or matrix in a LSE can be concurrently computed on an associated CDLAP array.

II. LU Decomposition and Matrix Inversion

For LU decomposition of an n x n matrix $A = [a_{ij}]$, we consider only the case in which all the principal minor submatrices of A are nonsingular. This provides a necessary and sufficient condition to produce a unique lower triangular matrix $L=[l_{ij}]$ with all $l_{ii} = 1$, and a unique upper triangular matrix $U = [u_{ij}]$ such that $L * U = A$. Both L and U are n x n nonsingular matrices. In Crout's reduction method [10], the matrix $A = L * U$ is decomposed according to the following computations:

For i = 1,2, ..., n,

$$\left.\begin{aligned} u_{ik} &= a_{ik} - \sum_{j=1}^{i-1} l_{ij} u_{jk} && \text{for } i \le k \le n; \\ l_{ki} &= (a_{ki} - \sum_{j=1}^{i-1} l_{kj} u_{ji})/u_{ii} && \text{for } i < k \le n; \\ l_{ii} &= 1 . && \end{aligned}\right\} \quad (1)$$

* This work was supported by the National Science Foundation under Grant ECS 81-05649.

0190-3918/83/0000/0079$01.00 © 1983 IEEE

Equation (1) can be transformed into another form more suited for parallel computations. We use $a_{ij}^{(k)}$ to denote the a_{ij} value in the kth recursion of computation. Set $a_{ij}^{(1)} = a_{ij}$ at the beginning. Using "t" as a variable to denote the recursion sequence from 1 to n, we have the following algorithm equivalent to (1) :

For t = 1,2, ..., n,

$$u_{tj} = a_{tj}^{(t)} \qquad \text{for } j \geq t \qquad (2.a)$$

$$l_{it} = a_{it}^{(t)} / u_{tt} \qquad \text{for } i > t \qquad (2.b)$$

$$a_{ij}^{(t+1)} = a_{ij}^{(t)} - l_{it} u_{tj} \qquad \text{for } i,j > t \qquad (2.c)$$

In each recursion, the above three equations are executed in the order (2.a), (2.b), and (2.c). (The same convention will apply to the equations (4), (6), (8), (10), and (12) in the following.) When t equals n, the above computation ends after computing (2.b). Note that the computation of l_{it} in (2.b) is not necessary for i = t, since l_{tt} is always equal to 1.

To illustrate the computation of the inverse matrices of L and U, let $L^{-1} = M = [m_{ij}]$ and $U^{-1} = V = [v_{ij}]$. Since L and U are nonsingular, both M and V do exist and are unique. It is easily verified that M is an n x n lower triangular matrix with all its diagonal elements equal to 1, just like L. Similarly, V is an n x n upper triangular matrix like U. To compute M, we make use of the relation L * M = I, where I is the identity matrix of order n. Thus we may write

$$\left.\begin{array}{ll} m_{kk} = 1 & \text{for } k = 1, 2, \ldots, n; \\ m_{ij} = -\sum_{k=j}^{i-1} l_{ik} m_{kj} & \text{for } 1 \leq j < i \leq n \end{array}\right\} \quad (3)$$

As in the previous case, (3) can be transformed into the following recursive algorithm:

For t = 1, 2, ... , n,

$$\left.\begin{array}{ll} m_{tt} = m_{tt}^{(t)} = 1 \; ; \quad m_{it}^{(t)} = 0 & \text{for } i > t; \\ m_{tj} = m_{tj}^{(t)} & \text{for } j < t. \end{array}\right. \quad (4.a)$$

$$m_{ij}^{(t+1)} = m_{ij}^{(t)} - l_{it} m_{tj} \qquad \text{for } i > t \text{ , } j \leq t \qquad (4.c)$$

Note that (4.b) is missing. The labeling (4.c) is used instead of (4.b) for the convenience of later illustration. The computation of V based on the relation V * U = I can be written as:

$$\left.\begin{array}{ll} v_{kk} = 1/u_{kk} & \text{for } k = 1, 2, \ldots, n; \\ v_{ij} = -\left(\sum_{k=i}^{j-1} v_{ik} u_{kj}\right)/u_{jj} & \text{for } 1 \leq i < j \leq n \end{array}\right\} \quad (5)$$

which in turn can be transformed into the following recursive algorithm:

For t = 1, 2, ..., n,

$$v_{tt}^{(t)} = 1 \; ; \quad v_{tj}^{(t)} = 0 \qquad \text{for } j > t \qquad (6.a)$$

$$v_{it} = v_{it}^{(t)} / u_{tt} \qquad \text{for } i \leq t \qquad (6.b)$$

$$v_{ij}^{(t+1)} = v_{ij}^{(t)} - v_{it} u_{tj} \qquad \text{for } i \leq t \text{ , } j > t \qquad (6.c)$$

When t equals n, the inversions of L and U will be completed after computing the (a), (b) portions of the above equations.

To describe the inversion of matrix A, let $S = [s_{ij}] = A^{-1}$. Since A = L * U, we have $S = U^{-1} * L^{-1} = V * M$. Thus, for any i and j,

$$s_{ij} = \sum_{k=1}^{n} v_{ik} m_{kj} = \sum_{k=\max\{i,j\}}^{n} v_{ik} m_{kj} \qquad (7)$$

Transforming (7) into the recursive algorithm, we have:

For t = 1, 2, ..., n,

$$s_{it}^{(t)} = s_{tj}^{(t)} = 0 \qquad \text{for } i, j \leq t \qquad (8.a)$$

$$s_{ij}^{(t+1)} = s_{ij}^{(t)} - v_{it} m_{tj} \qquad \text{for } i, j \leq t \qquad (8.c)$$

After the above n recursions, set

$$s_{ij} = - s_{ij}^{(n+1)} \qquad (8.d)$$

Step (8.d) at t = n+1 is added, since $s_{ij}^{(t+1)}$ at t = n is equal to $- s_{ij}$.

III. Parallel Computation Architecture

The Concurrent Data-Loading Array Processor (CDLAP) is introduced in [2]. The architecture is most suited for the computations in which processing elements (PEs) in the same row or same column share the same operand data as in a column-row vector multiplication. By accumulating resultant values in the PEs, the array processor can be used for computations involving recursive algorithms. The CDLAP has been shown to have an excellent efficiency in performing matrix multiplications [2]. The one-dimensional case of the concurrent data-loading design is reported in [8].

The careful examination of the recursive algorithms in Section II reveals some important characteristics of the algorithms. Consider the following three groups of equations:

{(2.a), (4.a), (6.a), (8.a)} --- group (a)
{(2.b), (6.b)} --- group (b)
{(2.c), (4.c), (6.c), (8.c)} --- group (c)

The equations in each of the above groups can be executed concurrently. For group (a) in each recursion, there are always 2n+1 new superscripted variables assigned. All these variables are set to zero except $m_{tt}^{(t)}$ and $v_{tt}^{(t)}$, which are both set to 1. In addition, the total number of the variables u_{tj} and m_{tj} (without superscript) assigned is n+1 for any t. For group (b), there are exactly n divisions sharing the same divisor (operand) u_{tt}. Group (c) has more complex sharing of operands. In the recursion of t = k, (n-k) x n and k x n PEs are required for concurrently executing equation sets {(2.c), (4.c)} and {(6.c), (8.c)}, respectively. Here each operand l_{it} or v_{it} will be shared by exactly n PEs. The same characteristics can be observed for the other combination of the equation sets; that is, n x (n-k) and n x k PEs are needed for the equation sets {(2.c), (6.c)} and {(4.c), (8.c)}, respectively. Again, each operand u_{tj} or m_{tj} will be shared by exactly n PEs.

The CDLAP configuration suited for the above complex computation requirements is shown in Fig.1 for the case n = 4. For the LU decomposition and matrix inversion of an n x n matrix A, the CDLAP must have n dividers and n x n adder/multipliers. The PEs in the same row can share the same operand and the PEs in the same column can share the other common operand. The shifts are in the diagonal direction to the upper-left PEs. At the upper boundary of this array, n+1 bus registers are provided. These registers are used to hold the data for output and for the concurrent data-loading in the corresponding columns. The dividers represented by circles in Fig. 1 have similar bus registers for output and for the concurrent data loading in the corresponding rows. Hence, there are totally 2n+1 output channels. (One of them is not necessary, since it is always 1 for this computation.) At the bottom and right boundaries, there are a total of 2n+1 input channels. The channels I_0 and I_5 will always be set to 1 since they correspond to $m_{tt}^{(t)}$ and $v_{tt}^{(t)}$, and the remaining input channels will be set to zeros.

In order to illustrate the computation process on the CDLAP, we may consider the case n = 4 without loss of generality. The recursive algorithms in Section II can be extended, grouped, and illustrated in the following detailed computation steps.

Step 0: Initial data loading of matrix A:

i.e., $a_{ij}^{(1)} = a_{ij}$ for all i, j.

Step 1: Initial shift:

$u_{1j} = a_{1j}^{(1)}$ for j = 1, 2, 3, 4;

$m_{11} = m_{11}^{(1)} = 1;$ $m_{i1}^{(1)} = 0$ for i > 1;

$v_{11}^{(1)} = 1;$ $v_{1j}^{(1)} = 0$ for j > 1;

$s_{11}^{(1)} = 0$

Step 2: Division:

$l_{i1} = a_{i1}^{(1)} / u_{11}$ for i > 1;

$v_{i1} = v_{i1}^{(1)} / u_{11}$ for i = 1.

Step 3: Multiplication and accumulation:

$a_{ij}^{(2)} = a_{ij}^{(1)} - l_{i1}\ u_{1j}$ for i, j > 1;

$m_{ij}^{(2)} = m_{ij}^{(1)} - l_{i1}\ m_{1j}$ for i > 1, j = 1;

$v_{ij}^{(2)} = v_{ij}^{(1)} - v_{i1}\ u_{1j}$ for i = 1, j > 1;

$s_{ij}^{(2)} = s_{ij}^{(1)} - v_{i1}\ m_{1j}$ for i, j = 1.

Step 4: Shift:

$u_{2j} = a_{2j}^{(2)}$ for $j \geq 2$;

$m_{22}^{(2)} = 1;$ $m_{i2}^{(2)} = 0$ for i > 2;

$m_{2j} = m_{2j}^{(2)}$ for $j \leq 2$;

$v_{22}^{(2)} = 1;$ $v_{2j}^{(2)} = 0$ for j > 2;

$s_{i2}^{(2)} = {}_{2j}^{(2)} = 0$ for i, $j \leq 2$.

Step 5: Division:

$l_{i2} = a_{i2}^{(2)} / u_{22}$ for i > 2;

$v_{i2} = v_{i2}^{(2)} / u_{22}$ for $i \leq 2$.

Step 6: Multiplication and accumulation:

$a_{ij}^{(3)} = a_{ij}^{(2)} - i_{i2}\ u_{2j}$ for i, j > 2;

$m_{ij}^{(3)} = m_{ij}^{(2)} - i_{i2}\ m_{2j}$ for i > 2, $j \leq 2$;

$v_{ij}^{(3)} = v_{ij}^{(2)} - v_{i2}\ u_{2j}$ for $i \leq 2$, 2 > 2;

$s_{ij}^{(3)} = s_{ij}^{(2)} - v_{i2}\ m_{2j}$ for i, $j \leq 2$.

Step 7: Shift:

$u_{3j} = a_{3j}^{(3)}$ for $j \geq 3$;

$m_{33}^{(3)} = 1;$ $m_{i3}^{(3)} = 0$ for i > 3;

$m_{3j} = m_{3j}^{(3)}$ for $j \leq 3$;

$v_{33}^{(3)} = 1;$ $v_{3j}^{(3)} = 0$ for j > 3;

$s_{i3}^{(3)} = s_{3j}^{(3)} = 0$ for i, $j \leq 3$.

Step 8: Division:

$l_{i3} = a_{i3}^{(3)} / u_{33}$ for i > 3;

$v_{i3} = v_{i3}^{(3)} / u_{33}$ for $i \leq 3$.

Step 9: Multiplication and accumulation:

$a_{ij}^{(4)} = a_{ij}^{(3)} - l_{i3}\ u_{3j}$ for i, j > 3;

$m_{ij}^{(4)} = m_{ij}^{(3)} - l_{i3}\ m_{3j}$ for i > 3, $j \leq 3$;

$v_{ij}^{(4)} = v_{ij}^{(3)} - v_{i3}\ u_{3j}$ for $i \leq 3$, j > 3;

$s_{ij}^{(4)} = s_{ij}^{(3)} - v_{i3}\ m_{3j}$ for i, $j \leq 3$.

Step 10: Shift:

$u_{4j} = a_{4j}^{(4)}$ for $j \geq 4$;

$m_{44}^{(4)} = 1;$ $m_{4j} = m_{4j}^{(4)}$ for $j \leq 4$;

$v_{44}^{(4)} = 1;$

$s_{i4}^{(4)} = s_{4j}^{(4)} = 0$ for i, $j \leq 4$.

Step 11: Division:

$v_{i4} = v_{i4}^{(4)} / u_{44}$ for $i \leq 4$.

Step 12: Multiplication and accumulation:

$s_{ij}^{(5)} = s_{ij}^{(4)} - v_{i4}\ m_{4j}$ for i, $j \leq 4$.

Step 13: Sign change:

$$s_{ij} = - s_{ij}^{(5)} \qquad \text{for } i,\ j \leq 4.$$

According to the above computation steps, the operation of the CDLAP is repetitive and regular except the initial data loading and the sign change in the last step. In each recursion, the three steps corresponding to equation groups (a), (b), and (c) are executed sequentially; shift first, then division, and then multiplication and accumulation. For each shift step, all data will be moved one position in the upper-left direction. The labeling of each datum shifting within the (n+1) x n PEs is not changed. The initial values of all variables $v_{tj}^{(t)}$, $m_{it}^{(t)}$, and $s_{ij}^{(t)}$ are set to either 1 or 0 at the time these enter the array. On the other hand, the data $a_{tj}^{(t)}$ and $m_{tj}^{(t)}$ become u_{tj} and m_{tj} at the time they enter the output registers at the upper boundary. Similarly, after the division step, the l_{it} and v_{it} values are sent to the output registers. The data in the output registers are also used as the operands in next step of multiplication and accumulation.

Fig. 2 shows the data flow of the above computation process. The four snapshots display the position of variables at the beginning of Steps 3, 6, 9, and 12, respectively. The L, U, L^{-1}, and U^{-1} values can be obtained from the output ports, and the inverse matrix A^{-1} will appear in the array at the end of the last step. Detailed input and output sequences are shown in Tables 1 and 2, where the last two columns under I_{Aj} and O_{Aj} will be refered to later in Section IV.

IV. Solving Linear Systems of Equations

Consider a family of linear equations A * X = B, where B = $[b_{ij}]$ is a given n x m matrix and X = $[x_{ij}]$ is an n x m unknown matrix. When m = 1, then B becomes a vector $\underline{b} = [b_i]$ and X becomes an unknown vector $\underline{x} = [x_i]$.

Since A = L * U, we have L * U * X = B. Let U * X = D = $[d_{ij}]$. Then we have the relation L * D = B. Using this relation, we can write

$$b_{ij} = \sum_{k=1}^{i} l_{ik} d_{kj}$$

Since l_{ii} = 1, we have

$$d_{ij} = b_{ij} - \sum_{k=1}^{i-1} l_{ik} d_{kj} \qquad \text{for any } i \text{ and } j. \qquad (9)$$

Equation (9) can then be transformed into the recursive algorithm shown below.
For t = 1, 2, ... , n,

$$d_{tj} = b_{tj}^{(t)} \qquad (10.a)$$

$$b_{ij}^{(t+1)} = b_{ij}^{(t)} - l_{it} d_{tj} \qquad \text{for } i > t \qquad (10.c)$$

This computation, similar to that for LU decomposition, will end at t = n after completing (10.a).

For U * X = D, the algorithm is similar. Using the relation X = U^{-1}*D = V * D, we can write

$$x_{ij} = \sum_{k=i}^{n} v_{ik} d_{kj} \qquad \text{for any } i \text{ and } j. \qquad (11)$$

Transforming (11) into a recursive form, we have:
For t = 1, 2, ..., n,

$$x_{tj}^{(t)} = 0 \qquad (12.a)$$

$$x_{ij}^{(t+1)} = x_{ij}^{(t)} - v_{it} d_{tj} \qquad \text{for } i \leq t \qquad (12.c)$$

After completing (12.c) at t = n, the following sign change must be done to obtain the correct x value.

$$x_{ij} = - x_{ij}^{(n+1)} \qquad \text{for all } i \text{ and } j. \qquad (12.d)$$

To perform the above computations, an extra n x m CDLAP will be needed. This extra array must be associated with the original array from which the l_{it} and v_{it} data are transferred to the horizontal data buses of the associated array. The data of matrix B are initially loaded into this associated array and the shift must be in the "up" or "north" direction. The configuration can be implemented as shown in Fig. 3. The detailed steps for n = 4 are listed in the following (where j = 1, 2, ..., m).

Step 0: Data loading:

$$b_{ij}^{(1)} = b_{ij} \qquad \text{for } i = 1, 2, 3, 4.$$

Step 1: Initial shift:

$$d_{1j} = b_{1j}^{(1)}, \quad x_{1j}^{(1)} = 0$$

Step 3: Multiplication and accumulation:

$$b_{ij}^{(2)} = b_{ij}^{(1)} - l_{i1} d_{1j} \qquad \text{for } i > 1.$$

$$x_{ij}^{(2)} = x_{ij}^{(1)} - v_{i1} d_{1j}. \qquad \text{for } i \leq 1.$$

Step 4: Shift:

$$d_{2j} = b_{2j}^{(2)}, \quad x_{2j}^{(2)} = 0$$

Step 6: Multiplication and accumulation:

$$b_{ij}^{(3)} = b_{ij}^{(2)} - l_{i2} d_{2j} \qquad \text{for } i > 2.$$

$$x_{ij}^{(3)} = x_{ij}^{(2)} - v_{i2} d_{2j} \qquad \text{for } i \leq 2.$$

Step 7: Shift:

$$d_{3j} = b_{3j}^{(3)}, \quad x_{3j}^{(3)} = 0$$

Step 9: Multiplication and accumulation:

$$b_{ij}^{(4)} = b_{ij}^{(3)} - l_{i3} d_{3j} \qquad \text{for } i > 3.$$

$$x_{ij}^{(4)} = x_{ij}^{(3)} - v_{i3} d_{3j} \qquad \text{for } i \leq 3.$$

Step 10: Shift:

$$d_{4j} = b_{4j}^{(4)}, \quad x_{4j}^{(4)} = 0$$

Step 12: Multiplication and accumulation:

$$x_{ij}^{(5)} = x_{ij}^{(4)} - v_{i4} d_{4j} \qquad \text{for } i \leq 4.$$

Step 13: Sign change:

$$x_{ij} = - x_{ij}^{(5)} \qquad \text{for } i = 1, 2, 3, 4.$$

The numbering of the above steps follows the system in Section III. Fig. 4 shows the data flow of this process in the asociated array. Referring to Tables 1 and 2, the input and output sequences are listed under I_{Aj} and O_{Aj}, respectively. The corresponding time units can be easily verified.

To solve $A * \underline{x} = \underline{b}$ directly, the associated array is actually not necessary. The computation process is the same as LU decomposition except that the $\underline{b}$ data must be input through the channels I_0 to I_3 at step 1. The snapshots of data flow for $n = 4$ are shown in Fig. 5. At the end of Step 13, the solution vector $\underline{x}$ is obtained in the array. This scheme offers an efficient way to solve LSEs (for $m = 1$) without extra hardware.

V. Estimation of Data Broadcast Delay

In analyzing the time required for one recursion of the above computations, it is necessary to consider the time required for setting up data signals on the data-loading lines before the data can be loaded. This data broadcast delay may be a controversial point about the CDLAP. However, the following estimates of this delay on some practical design indicate encouraging results. The physical parameters used in this estimation are as listed in Mead and Conway [1]:

	Resistance	Capacitance
Metal	0.03 ohms/□	$0.3 * 10^{-4}$pf/µm^2
diffusion	10 ohms/□	$1 * 10^{-4}$pf/µm^2
poly	15-100 ohms/□	$0.4 * 10^{-4}$pf/µm^2
gate-channel		$4 * 10^{-4}$pf/µm^2

Assume that we have a VLSI array chip of 1.2 cm square, PE of 1 mm square, metal line of 6 µm (3 λ) wide, metal line space of 6 µm, and 16 bits per word. The data-loading buses are mostly made of metal lines and partly connected by polysilicon or diffusion layer as shown in Fig.6. Then the width of the one data-loading channel would be about 200 µm or 0.2 mm. For the horizontal loading lines, there is a 0.2 mm diffusion link for every 1 mm of metal line. The diffusion link is assumed to have the same width as the metal line. Under this situation, the R_1*C_1 value of a vertical data line would be 0.13 nsec (R_1= 60 ohms, C_1= 2.2 pf); while for a horizontal line, it is about 10 nsec (R_1= 3400 ohms, C_1= 3.0 pf). Here, R_1 is the resistance along the data line, and C_1 is the capacitance due to the data line itself. The R_1*C_1 value gives us a rough estimate of the intrinsic time delay due to the data bus itself; it is about two times that of the real time constant.

Two-layer metal lines were used in a recent work [11] in which a 32-bit processor was fabricated. This technique allows smaller chip area and easier layout cross-over. Using this technique in our implementation, both the horizontal and vertical loading buses will have the same high speed. Thus, the time delay is mainly caused by the output impedance of the driver circuit and the capacitance of the gates and non-metal pathways. For the layout shown in Fig.6, the longest diffusion link needed to connect the data bus to the input buffer of a PE is about the layout width of the data bus, about 200 µm in our calculation above. Let Rg be the resistance along this diffusion link, and Cg be the capacitance due to the diffusion link and its associated gate area. Assume this diffusion link has the width of 4 µm (2 λ). Then the maximum Rg is about 340 ohms. The maximum capacitance due to this diffusion link is 0.08 pf. If the area of the gate and other diffusion regions connecting to this link is within 200 µm^2 (which should be large enough), the Cg value for one PE would be less than 0.16 pf.

Taking a practical estimate, the output resistance of a driver circuit in the chip is assumed to be 1K ohms. The maximum resistance R to any gate area is thus 1000 + 60 + 340 = 1.4 K ohms. The total capacitance C associated with the single data line would be within (2.2 pf + 0.16 pf * 10) = 3.8 pf. Thus the R * C estimate of the time delay of the data bus is only 5.3 nsec. For the above matrix operations, the time required for one step of computation (one addition plus one multiplication, or one division) would be at least one order of magnitude longer than this delay. Therefore, the time required for the data broadcast is negligible.

VI. Performance Analysis and Comparisons

The advantages of the algorithm in this paper can be easily seen by comparing the required structural complexity and time efficiency with those of other array processors performing the same functions. The systolic algorithm for LU decomposition presented in [1] is not as efficient as that in [5]. In Kung's paper [12], only the net processing time is counted. The LU decomposition needs $n(t_a+t_m+t_d)$ processing time. While for solving $A * \underline{x} = \underline{b}$ (by back substitution), it takes an additional $n(t_a+t_m+t_d)$ time, disregarding the data arrangement problems. Here, t_a, t_m, and t_d represent the time for one addition, multiplication, and division, respectively.

A close examination of the CDLAP shown in Fig.1 shows that the first two columns can be implemented as one column, since the dividers are not busy at the same time as the remaining PEs. On the other hand, the diagonal shift can be split into the vertical and horizontal shifts. And these vertical or horizontal shift lines can be used for loading and unloading n^2 data in n shift steps without matrix data reshaping. Since the data shift is fast, the memory data rate is the most probable speed bottleneck, which must be at least fast enough to accomodate the I/O data in one unit of computation time. Therefore, the initial data loading of n^2 data would take less than n units of computation time.

With an appropriate provision of cache memory or buffer registers, the initial data loading time would be very short, just like the shift time in the case of systolic arrays. Thus we neglect the initial data loading time in our later analysis. The turnaround time required for LU decomposition in our design is $(n-1)(t_a+t_m+t_d)$. This is less than that in [12]. For solving $A * \underline{x} = \underline{b}$, our algorithm needs only $n(t_a+t_m+t_d)$, while that in [12] requires $2n(t_a+t_m+t_d)$ counting only the net processing time. Matrix inversion takes $n(2t_a+3t_m+t_d)$ in [12], compared with only $n(t_a+t_m+t_d)$ turnaround time in our case.

For the convenience of analysis, we assume that $t_d = t_a + t_m$. It is also reasonable to assume that the time delay due to a shift, data broadcast, or sign change is negligible. A time unit can thus be defined as the time needed for a division or a multiplication plus an addition. The total time units spent up to each step is also shown in the 2nd column of Tables 1 and 2. From Table 2, it can be found that the computations of L, U, L^{-1}, U^{-1}, and A^{-1} will take 2n-3, 2n-2, 2n-2, 2n-1 and 2n time units, respectively.

Hwang and Cheng's paper [5] offers a complete set of modules for constructing the LSE solvers. Their results are summarized in Table 3. Their turnaround times for obtaining U^{-1}, A^{-1}, and solving LSEs from A are also computed and listed in the entries from (8) to (11) in Table 3. Their computation of U^{-1} has to wait until the completion of U, while for A^{-1} the completion of U^{-1} is required.

The performance of our algorithm and the required CDLAP structural complexity are also summarized in Table 3. The result of comparison is self-explanatory. Our design uses less hardware and less time to obtain L^{-1}, U^{-1}, or A^{-1}, and to solve the LSEs. Further, only one type of module is required in our design, since the vertical /horizontal shift implementation of the CDLAP has the capacity of the associated array in Fig.3.

VII. Conclusions

The algorithm presented in this paper offers an efficient way to perform LU decomposition and matrix inversions concurrently in the same array. For an n x n matrix, it requires only n^2 PEs. This algorithm uses fewer PEs and takes less time than existing algorithms on systolic arrays to the best of our knowledge. In addition, an n x m unknown matrix of an LSE can be computed by attaching an n x m associated array. The computation of the unknown matrix can be completed in parallel with the above process. All of the above can be achieved using only one type of module, which is suitable for mass production.

The processor array on which the present algorithm is executed is a version of the CDLAP. A similar structure can be used for matrix multiplications [2]. In all the cases mentioned so far, no data reshaping problem is encountered for dense matrices. This architecture has shown very encouraging results for the applications considered so far. More applications are being investigated and will be reported later.

References

[1] C. Mead and L. Conway, Introduction to VLSI Systems, Reading, MA: Addison-Wesley, 1980.

[2] M.Y. Chern and T. Murata, "Efficient matrix multiplications on a concurrent data-loading array processor", Proc. 1983 Int. Conf. on Parallel Processing.

[3] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)", Sparse Matrix Proc., 1978, Society for Industrial and Appl. Math., 1979, pp.256-282.

[4] H.T. Kung, "Let's design algorithms for VLSI systems", in Proc. Caltech Conf. VlSI, Jan. 1979, pp.65-90.

[5] K. Hwang and Y.H. Cheng, "VLSI computing structures for solving large-scale linear system of equations", Proc. of 1980 Int. Conf. on Parallel Processing, pp.217-227.

[6] K. Hwang and Y.H. Cheng, "Partitioned matrix algorithms for VLSI arithmetic systems", IEEE Trans. Computers, Vol.31, Dec. 1982, pp.1215-1224.

[7] F.P. Preparata and J. Vuillemin, "Optimal integrated-circuit implementation of triangular matrix inversion", Proc. of 1980 Int. Conf. on Parallel Processing, pp.271-279.

[8] K.H. Huang and J.A. Abraham, "Efficient parallel algorithms for processor arrays", Proc. 1982 Int. Conf. on Parallel Processing, pp.271-279.

[9] K. Hwang and K.S. Fu, "Integrated computer architectures for image processing and database management", Computer, Jan. 1983, pp.51-60.

[10] P.D. Crout, "A short method for evaluating determinants and solving systems of linear equations with real or complex coefficients", in Proc. American Inst. Elec. Eng., Vol.40, 1941, pp.1235-1240.

[11] J.M. Mikkelson, L.A. Hall, A.K. Malhotra, S.D. Seccombe, and M.S. Wilson, "An NMOS VLSI process for fabrication of a 32-bit CPU chip", IEEE J. Solid-State Circuits, Vol.16, Oct. 1981, pp.542-547.

[12] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V. B. Rao, "Wavefront array processor: language, architecture, and applications", IEEE Trans. Computers, Nov. 1982, pp.1054-1066.

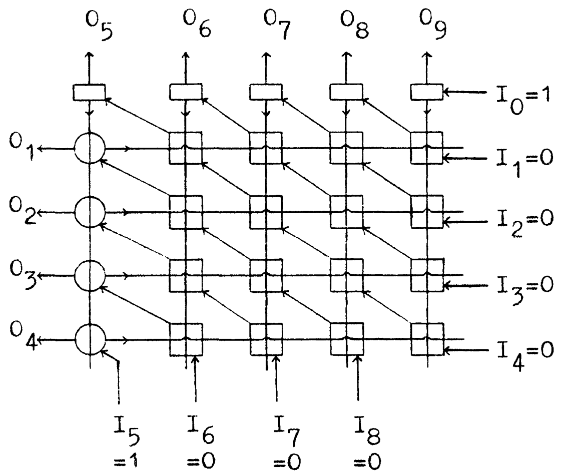
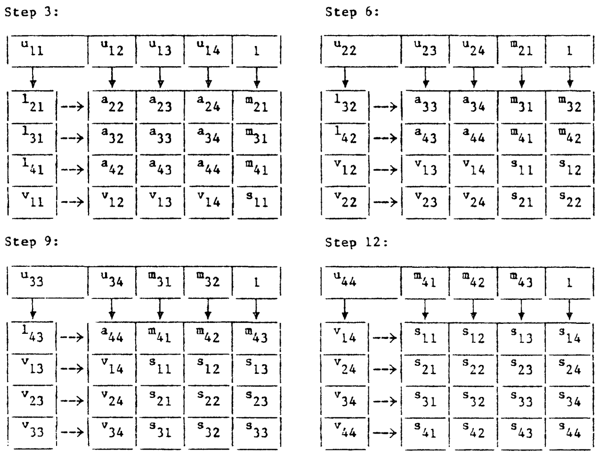


Fig. 1 CDLAP configuration for LU decomposition and matrix inversion; where squares, rectangles and circles represent PEs, output registers, and dividers, respectively.



Step 3:

u_{11}		u_{12}	u_{13}	u_{14}	1
l_{21}	-->	a_{22}	a_{23}	a_{24}	m_{21}
l_{31}	-->	a_{32}	a_{33}	a_{34}	m_{31}
l_{41}	-->	a_{42}	a_{43}	a_{44}	m_{41}
v_{11}	-->	v_{12}	v_{13}	v_{14}	s_{11}

Step 6:

u_{22}		u_{23}	u_{24}	m_{21}	1
l_{32}	-->	a_{33}	a_{34}	m_{31}	m_{32}
l_{42}	-->	a_{43}	a_{44}	m_{41}	m_{42}
v_{12}	-->	v_{13}	v_{14}	s_{11}	s_{12}
v_{22}	-->	v_{23}	v_{24}	s_{21}	s_{22}

Step 9:

u_{33}		u_{34}	m_{31}	m_{32}	1
l_{43}	-->	a_{44}	m_{41}	m_{42}	m_{43}
v_{13}	-->	v_{14}	s_{11}	s_{12}	s_{13}
v_{23}	-->	v_{24}	s_{21}	s_{22}	s_{23}
v_{33}	-->	v_{34}	s_{31}	s_{32}	s_{33}

Step 12:

u_{44}		m_{41}	m_{42}	m_{43}	1
v_{14}	-->	s_{11}	s_{12}	s_{13}	s_{14}
v_{24}	-->	s_{21}	s_{22}	s_{23}	s_{24}
v_{34}	-->	s_{31}	s_{32}	s_{33}	s_{34}
v_{44}	-->	s_{41}	s_{42}	s_{43}	s_{44}

Fig. 2 The data flow on the CDLAP (of Fig. 1) for LU decomposition and matrix inversion.

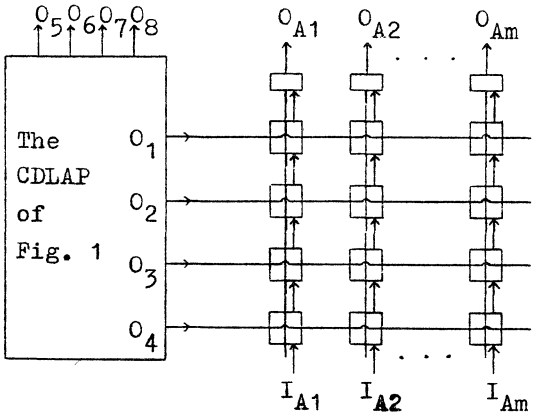
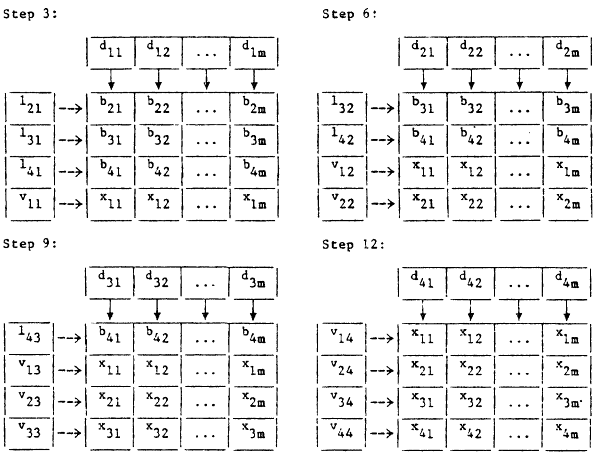


Fig. 3 The CDLAP configuration for solving a family of linear equations.



Step 3:

		d_{11}	d_{12}	...	d_{1m}
l_{21}	-->	b_{21}	b_{22}	...	b_{2m}
l_{31}	-->	b_{31}	b_{32}	...	b_{3m}
l_{41}	-->	b_{41}	b_{42}	...	b_{4m}
v_{11}	-->	x_{11}	x_{12}	...	x_{1m}

Step 6:

		d_{21}	d_{22}	...	d_{2m}
l_{32}	-->	b_{31}	b_{32}	...	b_{3m}
l_{42}	-->	b_{41}	b_{42}	...	b_{4m}
v_{12}	-->	x_{11}	x_{12}	...	x_{1m}
v_{22}	-->	x_{21}	x_{22}	...	x_{2m}

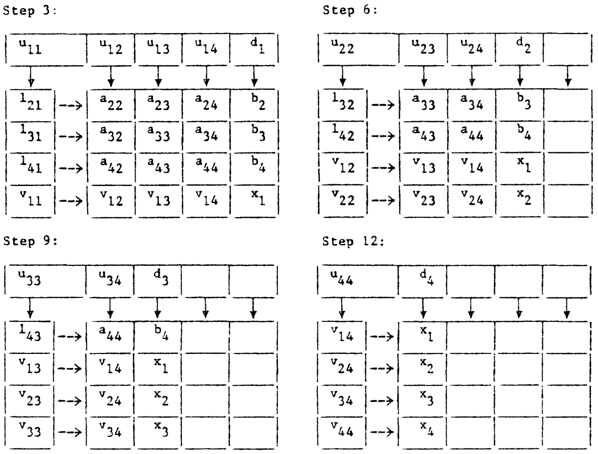
Step 9:

		d_{31}	d_{32}	...	d_{3m}
l_{43}	-->	b_{41}	b_{42}	...	b_{4m}
v_{13}	-->	x_{11}	x_{12}	...	x_{1m}
v_{23}	-->	x_{21}	x_{22}	...	x_{2m}
v_{33}	-->	x_{31}	x_{32}	...	x_{3m}

Step 12:

		d_{41}	d_{42}	...	d_{4m}
v_{14}	-->	x_{11}	x_{12}	...	x_{1m}
v_{24}	-->	x_{21}	x_{22}	...	x_{2m}
v_{34}	-->	x_{31}	x_{32}	...	$x_{3m'}$
v_{44}	-->	x_{41}	x_{42}	...	x_{4m}

Fig. 4 The data flow on the associated array of Fig. 3 for solving A * X = B.



Step 3:

u_{11}		u_{12}	u_{13}	u_{14}	d_1
l_{21}	-->	a_{22}	a_{23}	a_{24}	b_2
l_{31}	-->	a_{32}	a_{33}	a_{34}	b_3
l_{41}	-->	a_{42}	a_{43}	a_{44}	b_4
v_{11}	-->	v_{12}	v_{13}	v_{14}	x_1

Step 6:

u_{22}		u_{23}	u_{24}	d_2	
l_{32}	-->	a_{33}	a_{34}	b_3	
l_{42}	-->	a_{43}	a_{44}	b_4	
v_{12}	-->	v_{13}	v_{14}	x_1	
v_{22}	-->	v_{23}	v_{24}	x_2	

Step 9:

u_{33}		u_{34}	d_3		
l_{43}	-->	a_{44}	b_4		
v_{13}	-->	v_{14}	x_1		
v_{23}	-->	v_{24}	x_2		
v_{33}	-->	v_{34}	x_3		

Step 12:

u_{44}		d_4			
v_{14}	-->	x_1			
v_{24}	-->	x_2			
v_{34}	-->	x_3			
v_{44}	-->	x_4			

Fig. 5 The data flow on the array of Fig. 1 for LU decomposition and solution of A * $\underline{x}$ = $\underline{b}$.

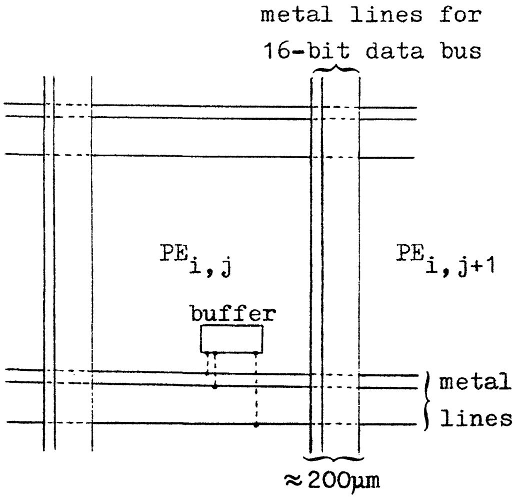


Fig. 6 A layout design for the data-loading buses (where dotted lines represent the diffusion links).

Table 1 :
Input Sequence for LU Decomposition on the CDLAP

Step	Time unit	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_{Aj}
1	0	m_{11}	$m_{21}^{(1)}$	$m_{31}^{(1)}$	$m_{41}^{(1)}$	$s_{11}^{(1)}$	$v_{11}^{(1)}$	$v_{12}^{(1)}$	$v_{13}^{(1)}$	$v_{14}^{(1)}$	$x_{1j}^{(1)}$
2	1										
3	2										
4	2	m_{22}	$m_{32}^{(2)}$	$m_{42}^{(2)}$	$s_{12}^{(2)}$	$s_{22}^{(2)}$	$v_{22}^{(2)}$	$v_{23}^{(2)}$	$v_{24}^{(2)}$	$s_{21}^{(2)}$	$x_{2j}^{(2)}$
5	3										
6	4										
7	4	m_{33}	$m_{43}^{(3)}$	$s_{13}^{(3)}$	$s_{23}^{(3)}$	$s_{33}^{(3)}$	$v_{33}^{(3)}$	$v_{34}^{(3)}$	$s_{31}^{(3)}$	$s_{32}^{(3)}$	$x_{3j}^{(3)}$
8	5										
9	6										
10	6	m_{44}	$s_{14}^{(4)}$	$s_{24}^{(4)}$	$s_{34}^{(4)}$	$s_{44}^{(4)}$	$v_{44}^{(4)}$	$s_{41}^{(4)}$	$s_{42}^{(4)}$	$s_{43}^{(4)}$	$x_{4j}^{(4)}$
11	7										
12	8										

Table 2 :
Output Sequence for LU Decomposition on the CDLAP

Step	Time unit	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_{Aj}
1	0					u_{11}	u_{12}	u_{13}	u_{14}	d_{1j}
2	1	l_{21}	l_{31}	l_{41}	v_{11}					
3	2									
4	2					u_{22}	u_{23}	u_{24}	m_{21}	d_{2j}
5	3	l_{32}	l_{42}	v_{12}	v_{22}					
6	4									
7	4					u_{33}	u_{34}	m_{31}	m_{32}	d_{3j}
8	5	l_{43}	v_{13}	v_{23}	v_{33}					
9	6									
10	6					u_{44}	m_{41}	m_{42}	m_{43}	d_{4j}
11	7	v_{14}	v_{24}	v_{34}	v_{44}					
12	8									
13	8	appearing in the array : A^{-1}								X

Table 3 : Summary of Structural Complexity and Computation Time

* For the design in [5]	no. of required PEs	no. of I/O channels	start-up time	net computation time	turnaround time
(1) LU decomposition	n^2-n	4n-2	n-1	2n-1	3n-2
(2) Triangularization of LSEs	n^2-1	4n	n	2n	3n
(3) Triangular linear system solver	n	n+2	n	2n-1	3n-1
(4) Inversion of U	$(n^2+n)/2$	2n	1	2n-1	2n
(5) Inversion of L	$(n^2-n)/2$	2n-2	1	2n-3	2n-2
(6) Matrix multiplication (to obtain A^{-1})	$2n^2-n$	4n-1	2n-1	2n-1	4n-2
(7) Solving a family of LSEs using A^{-1}	n^2	n^2+2n	n	n	2n
* The combined results from the above design					turnaround time
(8) Solving A * $\underline{x}$ = $\underline{b}$	[(1)+(3)]				6n-3
(9) Obtaining U^{-1} from A	[(1)+(4)]				5n-2
(10) Obtaining A^{-1} from A	[(9)+(6)]				9n-4
(11) Solving a family of LSEs using A^{-1}, with A^{-1} computed from A.	[(10)+(7)]				11n-4
** The algorithm based on CDLAP in this paper:	no. of PEs	I/O channels	initial loading		(after loading) turnaround time
(12) LU Decomposition only	n^2-n	3n-1	n^2 data in n steps		2n-2
(13) LU Decomposition plus L^{-1}, U^{-1}	n^2	4n+1	n^2 data in n steps		2n-1
(14) The above plus A^{-1}	n^2	4n+1	n^2 data in n steps		2n
(15) The above plus solving A * X = B, with B of n x m.	n^2+nm	4n+1+2m	n^2+nm data in n steps		2n
(16) Solving A * $\underline{x}$ = $\underline{b}$	n^2	4n+1	n^2 data in n steps		2n

VECTOR COMPUTER FOR SPARSE MATRIX OPERATIONS

Gao Qing-Shi, Wang Rong-Quan
The Institute of Computing Technology, Academia Sinica
Beijing, China

Abstract

A vector computer architecture for sparse matrix operations is introduced in this paper. Apart from having all the functions of an ordinary vector computer, it can also efficiently make operations on the non-zero elements of the sparse vectors and sparse matrices in pipeline manner. In comparison with the execution of sparse matrix operations in an ordinary vector computer, the computation speed can be increased several times or even over ten times.

On the basis of extending the standard high-level language to the vector high-level language, the further extension to the sparse vector high-level language, the basic operations of sparse vector and sparse matrix and its implementation in machine are discussed.

Raising the problem

A large number of applications, such as linear programming, the numerical solution of differential equations, structural analysis, network analysis, genetic theory, behavior and social science and so on, require solving the problems of higher order sparse linear equation. With the rapid development of technology, large-scale sparse matrices will be constantly encountered in the application related to large-scale system problems.

The Sparse Matrix (SM) described in this paper refers to not only the matrix of which the non-zero elements take a small percentage, such as less than 5%, but also the matrix of which the non-zero elements take a considerable ratio. SM technology has been studied carefully in traditional computers and ordinary vector computers [2]—[4]. Utilizing the SM technology, only non-zero elements are stored and calculated, and this results in great reducing of the needed storage space and raising of the computation efficiency. But these discussions are from the angle of algorithm. Although SM technology can speed up calculation considerably, when non-zero elements of SM are calculated it is necessary to spend more overhead such as comparing, discriminating and controlling of subscripts of non-zero elements, and this results in treating an ordinary pipeline-vector computer as a traditional scalar computer only. The Vector Computer for Sparse Matrix Operation (VCSMO) uses hardware to implement the above overhead operations from the angle of architecture. Thus the non-zero elements of SM can be effectively calculated in pipeline manner. Therefore, the VCSMO introduced in this paper can be several times or even over ten times faster than an ordinary vector computer in the implemantation of SM operations under the same technological conditions.

The Notation of Sparse Vector and Sparse Matrix

The Sparse Vector (SV) can be represented as triad $[\vec{\beta},\vec{y},l]$, Where $\vec{\beta}$ is a monotone-increasing-integer-vector called sparse-integer-control-vector which is composed of the subscripts of non-zero elements of SV, $\vec{y}$ is a data vector which is composed of the non-zero elements of SV, l is the length of $\vec{\beta}$ or $\vec{y}$.

In the assembly language a vector can be represented with $[D_y,l]$ and a SV with a triad $[D_\beta,D_y,l]$, where D_β and D_y are the initial addresses of $\vec{\beta}$ and $\vec{y}$ respectively, and l is the length of $\vec{\beta}$ or $\vec{y}$. It should be noted that l varies constantly in the operation. $\vec{\beta}$ and $\vec{y}$ are usually strored in sequence.

The SM can be represented with a triad $[\beta,Y,\vec{l}]$, which is composed of a group of ordered sparse row vectors $[\beta[i,*],Y[i,*],\vec{l}[i]]$, $i=1,2,\cdots,n$. Generally speaking, if $i\neq j$ then $\vec{l}[i]\neq\vec{l}[j]$.

In the assembly language, the sparse row vector is represented with $[D_\beta[i], D_Y[i], \vec{l}[i]]$, where $i=1,2,\cdots,n$, $D_\beta[i]$ and $D_Y[i]$ are the initial addresses of the row vectors $\beta[i,*]$ and $Y[i,*]$ respectively. Therefore in the assembly language the SM is respresented as $[\vec{D}_\beta,\vec{D}_Y,\vec{l}]$, where $\vec{D}_\beta=[D_\beta[1], D_\beta[2],\cdots,D_\beta[n]]$ and $\vec{D}_Y=[D_Y[1],D_Y[2],\cdots,D_Y[n]]$.

Generally speaking, β and Y are continuously stored row by row and in sequence in the main memory. Therefore, SM can also be represented as $[D_\beta,D_Y,\vec{l}]$ in assembly language, where D_β and D_Y are the initial addresses of β and Y respectively.

Non-sparse vector can use a flag bit instead of $\vec{\beta}$.

SV and SM can also be represented with quadruple $[\vec{\beta},\vec{y},l,L]$ and $[\beta,Y,\vec{l},\vec{L}]$. Where L is upper bound of the length l, $\vec{L}=[\vec{L}[1],\vec{L}[2],\cdots,\vec{L}[n]]$ and $\vec{L}[i]$ is upper bound of $\vec{l}[i]$, the meaning of other parameters is the same as the triad notation. In some cases the calculation can be speeded up when the quadruple notation is used.

The standardization representation of zero-vector or zero-Matrix is blanks.

The Extension of Standard Vector High-Level Language to SV and SM

The High-Levle Language (HLL) system for SM operations must be extended on the basis of the standard HLL. First it must be extended to the standard Vector HLL (VHLL) [1], and then fur-

0190-3918/83/0000/0087$01.00 © 1983 IEEE

ther to the language which includes sparse vector operations and sparse matrix operations.

The Extension of the Declaration Part

On the basis of extending HLL to standard VHLL, we introduce SPARSE VECTOR and SPARSE ARRAY, the type of which can be REAL or INTEGER or BOOLEAN. The mode is as follows:

$$\left\{\begin{matrix}\text{REAL}\\ \text{INTEGER}\\ \text{BOOLEAN}\end{matrix}\right\}\ \text{SPARSE}\ \left\{\begin{matrix}\text{ARRAY [<identifier>,}\\ \\ \text{VECTOR[<identifier>,}\end{matrix}\right.$$

$$\left.\begin{matrix}\text{<identifier>][}a_1\text{:}b_1\text{,}a_2\text{:}b_2\text{;N]}\\ \text{<identifier>][a:b;N]}\end{matrix}\right\}$$

where "{ }" denotes "or". The first identifier is the name of the sparse control matrix or sparse control vector of which the corresponding type is the integer. The second identifier is the name of the compressed matrix or compressed vector.N is the upper bound of needed storage space, (b_1-a_1+1) and (b_2-a_2+1) is number of rows and number of columns of original matrix respectively, b-a+1 is the length of original vector. Several sparse arrays or sparse vectors are permitted to be declared all at once.

What requires our attention is that the column SV of SM is stored very irregularly and can not be noted directly with$[\beta[*,j],Y[*,j],\vec{I}[j]]$.

The Sequence of Operations

The priority of +,-,*,/,↑,¬,∧,V, comparison and other operations are the same as that in the standard HLL.

Sign, Abs, Max, Min, |Max| (absolute maximum), |Min| (absolute minimum), #B(standardization), ⌊⌋(lower integer),⌈⌉(upper integer), ⦃ ⦄(decimal part),#Z(extract main-diagonal vector of matrix), #MI(matrix inversion), #IV(Inverted Sequence Vector),#E(extract a element), ∑(iterative addition of vector), #RT(retun operation) and etc. have the highest priority.

#IV and #MT(matrix transpose) and ↑ belong to the same priority. $\dot{*}$(inner product of vector),SM multiply SV and *,/ belong to the same priority. #HF(Sign replacing), #H+(Sign-bit addition) and +,- belong to the same priority. ⊽(NOR) and V(OR) belong to the same priority, ⊼(NAND) and ∧ (AND) belong to the same priority. For the detail of the above operations, see next section.

The Sparse Vector Expression and Sparse Matrix Expression

If operation result of an expression is the expresentation form of SV or SM, the expression is respectively called the SV expression or SM expression. The SV expression or SM expression can make operation connections with the corresponding non-sparse expression. The expression formed after the connection is called sparse expression or non-sparse expression depending on whether its operation results are sparse or non-sparse vectors (matrixes).

Assignment

SV and SM can be assigned with SV expression and SM expression individually. Besides, non-SV and non-SM can also be assigned with SV expression and SM expression individually and this is return operation. SV and SM can also be assigned with non-SV expression or non-SM expression individually and this is standardization.Non-SV and non-SM can be assigned with a constant, but in general, SV and SM can not be assigned with constant.

Standard Functions

In addition to normal standard functions of general HLL,further extended standard functions would contain length of original vector, number of column and row of original matrix, length of compressed vector, vector consisting of lengths of all compressed row-vector of matrix and needed maximal storage space of SM or SV.

Basic Operations of HLL

The SV computation system should include non-SV operations, scalar operations, SV operations and the mixed operations of non-SV, SV and scalar. All the operations can be made under the control of the operation control vector [1]. The discussion here is concentrated on basic operations of SV and SM.

Standardization

$$[\vec{\beta}_3,\vec{y}_3,l_3]:=\#B([\vec{\beta}_1,\vec{y}_1,l_1]).$$

Where l_3 is the number of non-zero elements of $\vec{y}_1$, $\vec{y}_3$ is a compressed vector eleminated zero-elements of $\vec{y}_1$, $\vec{\beta}_3$ is a corresponding subscript vector.

The Operations of Addition, Subtraction, Multiplication, Division, AND, OR, Exclusive-OR,NAND and NOR

$$[\vec{\beta}_3,\vec{y}_3,l_3]:=[\vec{\beta}_1,\vec{y}_1,l_1]\,\theta\left\{\begin{matrix}y_2\\ \vec{y}_2\\ [\vec{\beta}_2,\vec{y}_2,l_2]\end{matrix}\right\}$$

Where operator θ is +,-,*,/,∧, V,⊕, ⊼ and⊽. When θ is "/" the second operand has to be non-zero scalar or vector without any zero-element. The result of NAND or NOR is usually non-SV.

Comparison

The result of SV comparison operation (>,⩾, =,<,⩽) is a sparse bit-vector (zero-elements must be considered). The result of all satisfying-comparison operation of SV is a bit-scalar.

Inverted Sequence Vector (#IV)

Assume $\vec{X}=(x_1,x_2,\cdots,x_n)$, $\#IV(\vec{X})=(x_n,x_{n-1},\cdots,x_1)$ is a inverted sequence vector of $\vec{X}$.

$$[\vec{\beta}_3,\vec{y}_3,l_3]:=\#IV(\vec{\beta}_1,\vec{y}_1,l_1).$$

Where $\vec{\beta}_3=L+1-\#IV(\vec{\beta}_1)$, $\vec{y}_3=\#IV(\vec{y}_1)$, L is the length of original vector $[\vec{\beta}_1,\vec{y}_1,l_1]$.

Other Operations

The result of vector inner-product is a scalar. The results of Iterative-addition of vector (∑), NOT(¬) and Return Operation (#RT) are usually non-SV. In Max and Min operations zero-elements must be considered. In addition, there are #HF, #H+, Sign function (Sign), |Max|, |Min|,Abs, ⌊⌋, ⌈⌉,⦃ ⦄, Extract subvector, Extract an element of SV and Transmission.

Most of the above operations can be extended to SM. Besides, the other basic operations of SM also include #MI, #MT, #Z, Extract the submatrix of SM, Multiplication of SM and SV or of SM and vector, k-th power of SM and so on.

Implementation of Basic Operation of SV and SM

System Hardware Structure

System hardware consists of a very large memory, a memory controller, instruction control unit and sparse vector ALU. The block-diagram is shown in Fig.1.

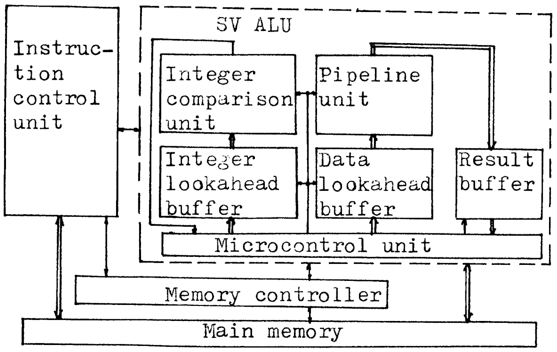


Fig 1. Block diagram of system structure

Accoding to the character of SV and SM, the vectical processing pipeline should be adopted. Therefore, one of the keys of hardware implementation is to provide high-speed access to the memory to ensure data access rate needed for integer comparison unit and pipeline unit. To solve this problem we can use cache or multi-bus multi-bank interleave access technology.

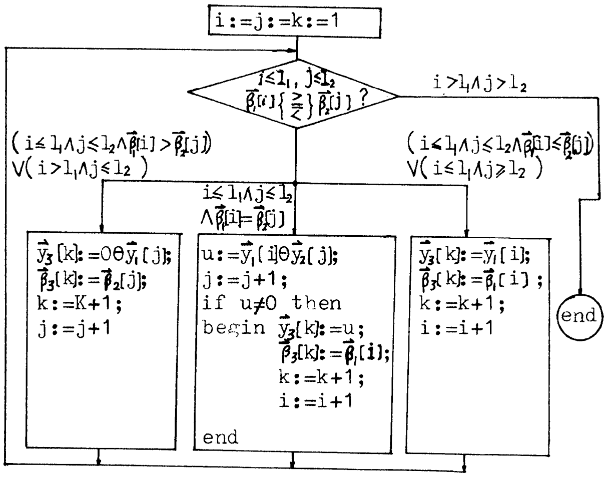
Vector Processing of the VCSMO

The descriptor of a SV is used to provide the parameters of SV. The descriptor is stored in several successive locations of memory. It is placed into index registers before processing.

There are two Vector Parameter Files, VPF_0 and VPF_1, which are especially used for processing vector operations. When one VPF is being used for processing the current vector instruction, another VPF can be used for preparing the parameters of next vector instruction. This structure can minimize the effects of set-up time on computation speed in the vertical processing pipeline.

An Example of Implementation of the Basic Operation in Machine

The basic operations mentioned in the previous section can be implemented directly with hardware. For example, Fig.2. presents block diagram of machine implementation for operation of $[\vec{\beta}_3,\vec{y}_3,l_3]:=[\vec{\beta}_1,\vec{y}_1,l_1]\theta[\vec{\beta}_2,\vec{y}_2,l_2]$, where θ may be +,-, $\vee$ and $\oplus$ operation.



$i:=j:=k:=1$
$i\leq l_1,\ j\leq l_2$
$\vec{\beta}_1[i]\left\{\begin{matrix}>\\=\\<\end{matrix}\right\}\vec{\beta}_2[j]\ ?$
$i>l_1\wedge j>l_2$
$(i\leq l_1\wedge j\leq l_2\wedge\vec{\beta}_1[i]>\vec{\beta}_2[j])$
$\vee(i>l_1\wedge j\leq l_2)$
$(i\leq l_1\wedge j\leq l_2\wedge\vec{\beta}_1[i]<\vec{\beta}_2[j])$
$\vee(i\leq l_1\wedge j\geqslant l_2)$
$i\leq l_1\wedge j\leq l_2$
$\wedge\vec{\beta}_1[i]=\vec{\beta}_2[j]$
$\vec{y}_3[k]:=0\theta\vec{y}_1[j];$
$\vec{\beta}_3[k]:=\vec{\beta}_2[j];$
$k:=K+1;$
$j:=j+1$
$u:=\vec{y}_1[i]\theta\vec{y}_2[j];$
$j:=j+1;$
if $u\neq0$ then
begin $\vec{y}_3[k]:=u;$
$\vec{\beta}_3[k]:=\vec{\beta}_1[i];$
$k:=k+1;$
$i:=i+1$
end
$\vec{y}_3[k]:=\vec{y}_1[i];$
$\vec{\beta}_3[k]:=\vec{\beta}_1[i];$
$k:=k+1;$
$i:=i+1$
end

Fig. 2.

Conclusion

The hardware cost of the integer comparison unit of a VCSMO is very small in comparison with the pipeline unit. Nevertheless, this makes it possible for the non-zero elements in the SV and SM to be operated in high-speed pipeline manner. Therefore, in comparison with the ordinary vector computer, the VCSMO increases the speed of SV operations and SM operations by several to over ten times. The times of speed-up is closely related to the distribution and ratio of non-zero elements of the SM of a specific problem.

Acknowledgement

The authors would like to thank associate Prof. Zhang Xiang for stimulating discussion.

Reference

[1] Gao Qing-Shi, Zhang Xing, A General-purpose Cellular Supercomputer—Cellular Vector Computer of Vertical and Horizontal Processing with Virtual Common Memory, Chinese Journal of Computers, Vol.2, No.1, January (1979) PP 1-13.

[2] Tewarson, R.P., Sparse matrices, Academic Press(1973).

[3] Ogbuobiri, E.C., Dynamic Storage and Retrieval in Sparsity Programming. IEEE. Trans. Power Apparatus System, PAS89,(1970), PP 150-155.

[4] Jennings, A., Solution of Variable Bandwidth Positive Difinite Simultaneous Equations, Comp. J.15. (1971). PP 446.

[5] Bell, G.Fuller, S.H. and Siewiorek, D., The CRAY-1 Computer System, Communications of the ACM, Vol. 21, No.1, Jan. 1978, PP 63-72.

[6] Hintz, R.G. and Tate, D.P., CDC STAR-100 Processor Design, Compcon Proc., Sept. 1972. PP. 1-4.

EFFICIENT MATRIX MULTIPLICATIONS ON A CONCURRENT DATA-LOADING ARRAY PROCESSOR*

Ming-Yang Chern and Tadao Murata
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
P.O. Box 4348, Chicago, Illinois 60680

Abstract -- This paper introduces a VLSI-compatible architecture called concurrent data-loading array processor (CDLAP). Many matrix operations on systolic arrays have the matrix data reshaping problem. The CDLAP can execute the multiplication for dense matrices without data reshaping. A partitioned multiplication algorithm is also presented for matrices larger than the array size. Based on the design in this paper, the utilization of processing elements is virtually the best achievable for large matrices. The CDLAP, with small variations, can be used for band matrix multiplications. The performance, taking into account the total computation time and data transfer bandwidth, is found better than systolic arrays.

I. Introduction

In recent years, VLSI architectures for highly parallel computations have been extensively investigated [1, 2]. The systolic array structure for matrix operations was first proposed by Kung [3-5]. Various versions of systolic arrays designed for different applications have been proposed. The computational structure for solving a linear system of equations (LSE) presented by Hwang and Cheng [6] involves some special modules for LU decomposition, solving a triangularized LSE, triangularized matrix inversion, and matrix multiplication. More recently, the wavefront concept and the wavefront array processor (WAP) were proposed [7]. Although the asynchrony and local memory of the WAP offer more flexibility in computation, the data flow of the WAP is basically the same as systolic arrays.

All the above systolic-type array processors for matrix operations suffer from the common problem of data arrangement. For dense matrix operations, the conventionally arranged matrix must be reshaped before being fed into the processor array. By providing on-chip delay latches, the data arrangement problem can be solved partially, but this evokes a more severe problem in the array chip interconnection. The on-chip delay scheme is suitable only for the case of a one-chip array, in which case the array size is too restricted. Secondly, the chip I/O circuit for a systolic array is complex in order to achieve an acceptably easy interconnection of array chips [8]. In general, the utilization of PEs in the above mentioned arrays is half or less for matrix operations. This leaves some space for further improvement.

The use of global data communication, together with the systolic scheme, in a linear convolution array has been presented by Kung [11]. Huang and Abraham [9] proposed some algorithms for matrix multiplications. For the case of dense matrices, one of their algorithms makes use of a one-dimensional data broadcast scheme and has an improved performance. However, it still has the matrix reshaping requirement and the utilization of PEs is only 50%.

This paper presents an array architecture which removes the above mentioned deficiencies. In this architecture, a data broadcast scheme in two directions across the array is introduced. From the functional point of view, we categorize such a computational array as a concurrent data-loading array processor, or CDLAP for short.

II. Dense Matrix Multiplications

The CDLAP for a dense matrix multiplication is configured as shown in Fig. 1. It is an array of n x n processing elements (PEs). The data, upon arrival at the data bus of each column or each row, can be concurrently loaded in or broadcast to all PEs in that column or row, respectively. The functional block diagram of the PE is shown in Fig.2. Let the accumulator value of the PE at the ith row and jth column be denoted by d_{ij}; $D = [d_{ij}]$ will represent the accumulator matrix of the processor array. For the initialization of a computation, all accumulators can be set to zero by a common reset line. When the multiplexer M selects u_{in} as the input of d_{ij}, the accumulator can acquire data from its right-neighbor PE. In another instance when the multiplexer M selects the output of the adder, the product of x_i and y_j will be accumulated for each step of computation, i.e., $d_{ij} \leftarrow d_{ij} + x_i y_j$. There are two schemes to output the accumulator data. The first is to connect the d_{ij} output directly to the neighboring PEs on the left side; this output channel is denoted by u_{out}. The second is to transfer d_{ij} to a buffer register t_{ij} which will later send out the data via the horizontal bus at an appropriate time. Under this scheme, the horizontal bus is used alternatively for input and output. The tri-state output of t_{ij} is usually open-circuited unless the "output enable" is on. For the t-registers in the same row, the array can enable the output of only one at a time. To keep control simple, the array is designed such that all PEs of

* This work was supported by the National Science Foundation under Grant ECS 81-05649.

0190-3918/83/0000/0090$01.00 © 1983 IEEE

the same column will output their data at the same time.

In order to illustrate a dense matrix multiplication, consider $A = [a_{ij}]$ and $B = [b_{ij}]$, both N x N matrices. The resultant N x N matrix is denoted by $C = [c_{ij}] = A * B$.

Case 1 : Assume that N = n. Let A_i be the ith column vector of A and B_j be the jth row vector of B. Then we may write

$$C = \sum_{k=1}^{n} A_k * B_k$$

This matrix multiplication can be carried out in n recursions, executing

$$D^{(k)} = D^{(k-1)} + A_k * B_k \tag{1}$$

recursively for k = 1,2,....,n; where $D^{(o)} = [0]_{nxn}$ and $D^{(n)} = C$. The input data arrangement is shown in Fig.1. Note that the input operands A_k and B_k are loaded concurrently so that the (i,j)th PE has the input operands a_{ik} and b_{kj} at the beginning of the kth recursion in (1). During the kth recursion, n^2 multiplications and then n^2 additions are performed at the n^2 PEs, respectively, in parallel with loading A_{k+1} and B_{k+1} on the bus lines for the (k+1)th recursion. At the end of this computation, when k = n, the value of matrix C will appear in the accumulator array D.

The data arrangement in this algorithm is concordant with the conventional way of storing matrices and thus eleminates the matrix reshaping problem. The time required to complete this matrix multiplication, with the result staying in the array, is $n(t_a+t_m)$, where t_a and t_m represent the time to perform one addition and one multiplication, respectively. The resultant matrix C can be transfered out of the array processor later, which will take some extra time. Or, it may stay in the array for further processing, which saves both the time of unloading C and the time of data loading for the next stage of a computation. The detailed performance evaluation is presented in Section VI.

III. Partitioned Matrix Multiplications

In practice, many matrices to be computed are larger than the available array size, and thus the computation extendability is important.

Case 2 : Assume that the matrix size N is larger than the array size n, and that m = N/n is an integer. The matrix C = A * B then can be expressed in the following form:

$$c_{ij} = \sum_{k=1}^{mn} a_{ik} * b_{kj} \tag{2}$$

Let $i = (I-1)*n + i'$ and $j = (J-1)*n + j'$ for $1 \leq i',j' \leq n$. The matrix element c_{ij} can be expressed as ${}_{IJ}c_{i'j'}$. Then equation (2) can be rewritten as

$${}_{IJ}c_{i'j'} = \sum_{K=1}^{m} \sum_{p=1}^{n} {}_{IK}a_{i'p} * {}_{KJ}b_{pj'} \tag{3}$$

When expressed in the submatrix form, we have $C_{IJ} = [\ {}_{IJ}c_{i'j'}]$

and $$C_{IJ} = \sum_{K=1}^{m} A_{IK} * B_{KJ} \tag{4}$$

where A_{IK} , B_{KJ} , and C_{IJ} are n x n submatrices of A, B, C at the positions (I, K), (K, J), and (I, J), respectively.

The submatrix multiplication $A_{IK} * B_{KJ}$ in (4) can be carried out on the n x n CDLAP as was shown in Section II. Since the resultant matrix can be accumulated in the array, it is convenient to compute C_{IJ} on the CDLAP. The recursions $D^{(K)} = D^{(K-1)} + A_{IK} * B_{KJ}$ are carried out and accumulated until K= m. The resultant matrix $D^{(m)}$ is then transfered to C_{IJ}. The processor will then reset the accumulator matrix D and continue its computation for the next I, J.

The algorithm for our partitioned matrix multiplication can be easily expressed in a high-level language as shown below:

```
       DO  40  I = 1,m
       DO  40  J = 1,m
       reset D
       DO  20  K = 1,m
20     $D = D + A_{IK} * B_{KJ}$
       $C_{IJ} = D$
40     continue
```

With the provision of output buffer registers, the array processor only initiates the action $C_{IJ} = D$; it need not wait for its completion. The actual transfering of output data will take place in parallel with the next stage of a computation in the array.

To avoid extra time delay caused by the data output, the output data transfer must be finished before the arrival of the next output data; that is, before the completion of the computation stage concurrently running on the array. In our case, an n x n submatrix multiplication is performed in n computational steps and there are n^2 data to be output. Considering the design simplicity and the acceptable data transfer bandwidth, n words of data will be output for each computational step in the CDLAP until all output data in the buffers are delivered. Neglecting the short time for array reseting and output initialization, the total time required for the case N = mn would be $(m^3+1)n$ units of (t_a+t_m). The maximum data I/O rate is 3n words per unit of computation time.

IV. Band Matrix Multiplications

The CDLAP architecture can also be applied to the case of band matrix multiplications.

Case 3 (Band matrix/dense matrix multiplication): Assume that A is an N x N band matrix of width W = n, where N is larger than n. The matrix C = A * B is to be computed, where B and C are both N x W matrices. In Fig.3(a), an example of this multiplication with W = n = 4 is shown. The CDLAP configuration and its associated data arrangement for this multiplication are shown in Fig.3(b). This computation algorithm using CDLAP is similar to that described in [9] except that a two-dimensional array is used in our case. The function of a basic cell is shown in Fig.3(c), where $u_{out} = u_{in} + x_i y_j$. This function can be implemented using the cell shown in Fig.2.

Since the accumulators of the CDLAP can be cleared by giving a "reset" signal before a computation, the actual time required to complete the operation (including the data output) lies between N and (N + W) units of (t_a+t_m). The array data I/O rate for this case is 2W + W = 3n words per time unit.

Case 4 : The band matrix/band matrix multiplication on the CDLAP is illustrated by the example shown in Fig.4(a), where the widths of the matrices are W1 = 3 and W2 = 4. Note that in this case the data shift must be in the diagonal direction as shown in Fig.4(b), while the function of a basic cell is still the same as in case 3. The time required to complete this operation is between N and (N + min{W1, W2}) time units. And the data I/O rate is W1 + W2 + (W1 + W2 - 1) = 2(W1 + W2) - 1 words per time unit.

V. Implementation Considerations

The CDLAP can be implemented by cross-connecting microprocessors using some bus lines. In order to achieve mass production and low cost, however, the CDLAP should be implemented in VLSI chips. Since the systolic array architecture has been considered suitable for VLSI, the CDLAP is compared with systolic arrays in terms of their VLSI implementation characteristics.

Like the systolic array, the CDLAP is regular and homogeneous. The basic cell (or PE) of a two-dimensional systolic array for matrix multiplications has six I/O channels while that of the CDLAP needs at most four channels, as shown in Fig.2. For an n x n array implemented in one VLSI chip, the systolic array has a total of 8n-2 I/O channels from its border PEs [8]. For a dense matrix multiplication on the CDLAP, only 2n channels are needed for data input and n channels for data output. In Case 3, 3n input and n output channels will be required. Only in Case 4, the CDLAP needs 6n-2 I/O channels, which are still 2n less than that required in systolic arrays. Therefore, we expect that the I/O circuits for the CDLAP chip will be less complex than those for systolic arrays. Taking similar I/O schemes proposed for systolic arrays [8], fewer numbers of I/O pins (bit-serial I/O scheme) or less time delay (byte-serial grouped I/O scheme) is expected for the CDLAP.

For the mask layout, it appears that systolic arrays have very short interconnections between neighboring PEs. However, a detailed study shows that data lines are still needed to link the I/O ports on the opposite sides of each PE in systolic arrays. These data lines may be laid across the PE or along the border of the PE, depending on design. Whatever the route is, we can at least implement the data-loading lines for the CDLAP along the data shifting route used in systolic arrays without occupying extra layout area. It is probable that a simple, straight route along the edge of a column or a row of PEs is a better choice.

The most controversial point about the CDLAP may be the time delay due to the data-loading lines. However, our estimation [10] shows that this time delay is much less than the unit computation time t_a+t_m. Therefore, the data broadcast scheme of the CDLAP should not be a speed bottleneck.

VI. Performance Comparisons

The advantage of the above algorithms can be seen by comparing their performance with existing algorithms on other array processors. We assume no pipelining in all PEs for the convenience of comparisons. We also assume that the access of system memory modules is fast enough and appropriately arranged. For the case of dense matrix multiplications, we let P (number of PEs) be fixed as n^2. The time efficiency can then be seen from T (the turnaround time of the entire computation). Since a smaller T can possibly be achieved at the expense of heavy data communication and the communication cost is not low, the transfer bandwidth should be considered. Here the data transfer bandwidth, B, is defined as the maximum number of words which have to be transferred through the I/O ports of the border PEs in a unit of computation time. On the other hand, the larger the processor array is, the larger the B value would be. Therefore, the value of PBT^2 will be used to evaluate the performance as in [9]. For an ideal case, we expect that an n x n array can complete the n x n dense matrix multiplication in n time units. With a uniform rate of data input and output, the data transfer bandwidth would be 3n. The PBT^2 value, $3n^5$, in this ideal case will be used as a reference, and we define the ratio $R = PBT^2/3n^5$.

For the first three algorithms in Table 1, in which the resultant matrix stays in the array, the CDLAP has the best R (= 2/3) as compared with the 6 and 8/3 of the other two. Assuming that the CDLAP uses the output data rate of n words, its R will be 8/3. This ratio is better than those for the systolic array and the algorithm in [9] as shown in Table 1. In the case that the CDLAP outputs in the same data transfer bandwidth as the input, a further improvement (T = 3n/2, R = 3/2) can be obtained.

For the mn x mn matrices, the reference PBT^2 should be $3(mn)^5$. The turnaround time T of the CDLAP in this case is $(m^3+1)n$, and the R value is

Table 1 Performance of Parallel Algorithms for Dense Matrix Multiplications

** for n x n matrices	P	T	B	PBT^2	R	matrix reshaping
WAP[7] (with results in array)	n^2	3n	2n	$18n^5$	6	yes
Broadcast algorithm in [9] (with results in array)	n^2	2n	2n	$8n^5$	8/3	yes
CDLAP (with results in array)	n^2	n	2n	$2n^5$	2/3	no
Systolic Array in [1]	n^2	4n	6n	$96n^5$	32	yes
Algorithm in Section 4.4 of [9]	n^2	2n	6n	$24n^5$	8	yes
CDLAP with the data output rate = n	n^2	2n	2n	$8n^5$	8/3	no
CDLAP with the data output rate =2n	n^2	3n/2	2n	$9n^5/2$	3/2	no
** for mn x mn matrices						
CDLAP (data output in parallel with computation)	n^2	$(m^3+1)n$	3n	$3(m^3+1)^2n^5$	$\sim$ m	no

$m+(2/m^2)+(1/m^5)$. Comparing this T value with the actual computation time m^3n of each PE, the PE utilization is virtually perfect for large m. (For example, if m equals 5, the PE utilization would be 99.2%.) The R value ($\sim$ m) reflects the repeated use of the same data (m times) in the input matrices.

The band matrix multiplication on CDLAP is very similar to the broadcast algorithm in [9]. Both have exactly the same input/output data arrangement. Hence their PE utilization and the data transfer bandwidth would be the same. Since the CDLAP can be set to zero at the biginning of a computation, the total time duration may be equal to or a few time units (smaller than the minimum width) shorter than the algorithm in [9]. Since this algorithm has been proven to be better than the systolic algorithm in [5], the CDLAP should be better yet.

VII. Conclusions

Several cases of matrix multiplications on the CDLAP have been presented in this paper. Their performance has been shown to be excellent from the PE utilization point of view. The time performance is either better or no worse than those for presently known algorithms on systolic arrays. For dense matrix multiplications, no matrix data reshaping is required; for band matrices, the data are arranged in a way similar to that of systolic arrays. The simple data arrangements on CDLAPs have made it easier to design the extended algorithms for large scale matrices.

The preliminary feasibility study in Section V indicates some encouraging results on the VLSI implementation of the CDLAPs. This paper, together with other matrix operations presented in [10], has shown the high potential of the CDLAP as a cost-effective VLSI architecture.

References

[1] C. Mead and L. Conway, Introduction to VLSI Systems, Reading, MA: Addison-Wesley, 1980.

[2] L.S. Haynes, R.L. Lau, D.P. Siewiorek, and D.W. Mizell, "A survey of highly parallel computing", Computer, Jan. 1982, pp.9-24.

[3] H.T. Kung, "Let´s design algorithms for VLSI systems", in Proc. Caltech Conf. VlSI, Jan. 1979, pp.65-90.

[4] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)", Sparse Matrix Proc., 1978, Society for Industrial and Appl. Math., 1979, pp.256-282.

[5] H.T. Kung, "The structure of parallel algorithms", in Advances in Computers, Vol.19, New York: Academic, 1980, pp.65-111.

[6] K. Hwang and Y.H. Cheng, "VLSI computing structures for solving large-scale linear system of equations", Proc. 1980 Int. Conf. on Parallel Processing, pp.217-227.

[7] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V. B. Rao, "Wavefront array processor: language, architecture, and applications", IEEE Trans. Computers, Nov. 1982, pp.1054-1066.

[8] M.Y. Chern and T. Murata, "Comparison of Various Chip-I/O Schemes for Interconnecting VLSI Systolic Array Processor Chips", to appear in the Proc. of 1983 Int. Conf. on Computer Design: VLSI in Computers.

[9] K.H. Huang and J.A. Abraham, "Efficient parallel algorithms for processor arrays", Proc. 1982 Int. Conf. on Parallel Processing, pp.271-279.

[10] M.Y. Chern and T. Murata, "A fast algorithm for concurrent LU decomposition and matrix inversion", Proc. 1983 Int. Conf. on Parallel Processing.

[11] H.T. Kung, "Why systolic architectures?", Computer, Jan. 1982, pp.37-46.

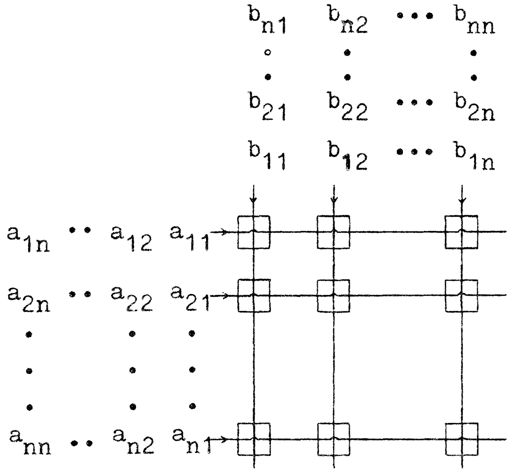

b_{n1} b_{n2} ••• b_{nn}
b_{21} b_{22} ••• b_{2n}
b_{11} b_{12} ••• b_{1n}
a_{1n} •• a_{12} a_{11}
a_{2n} •• a_{22} a_{21}
a_{nn} •• a_{n2} a_{n1}

Fig. 1 CDLAP configuration for dense matrix multiplication.

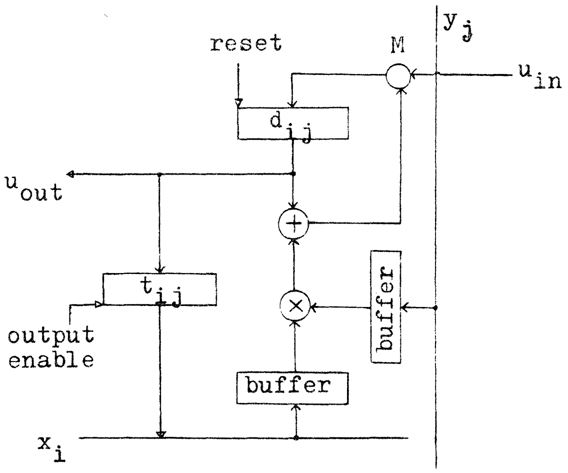


Fig. 2 The functional block diagram of a processing element in CDLAP.

$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & \cdot & \vdots \\ & a_{42} & a_{43} & \cdot & \vdots \\ & & a_{53} & \cdot & \vdots \\ 0 & & & \cdot & \cdot \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & \cdot & \vdots \\ b_{31} & & \vdots & \vdots \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ b_{N1} & \cdot & \vdots & b_{N4} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & \cdot & \vdots \\ c_{31} & \cdot & \vdots & \vdots \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ c_{N1} & \cdot & \vdots & c_{N4} \end{bmatrix}$$

Fig.3(a) Band matrix/dense matrix multiplication with W = 4.

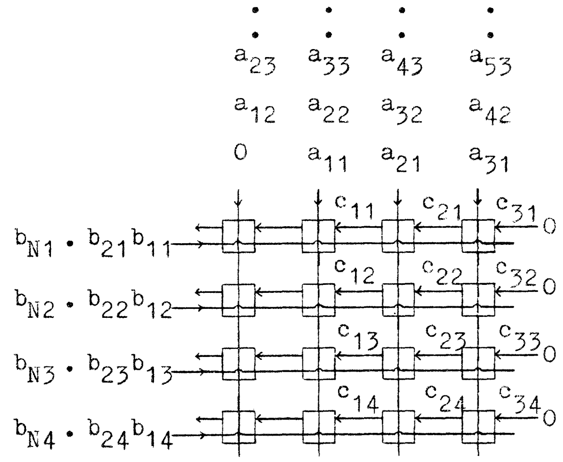
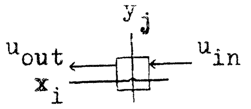

a_{23} a_{33} a_{43} a_{53}
a_{12} a_{22} a_{32} a_{42}
0 a_{11} a_{21} a_{31}
c_{11} c_{21} c_{31} 0
b_{N1} • $b_{21}b_{11}$
c_{12} c_{22} c_{32} 0
b_{N2} • $b_{22}b_{12}$
c_{13} c_{23} c_{33} 0
b_{N3} • $b_{23}b_{13}$
c_{14} c_{24} c_{34} 0
b_{N4} • $b_{24}b_{14}$

Fig.3(b) CDLAP configuration for Fig. 3(a).



$$u_{out} = u_{in} + x_i y_j$$

Fig.3(c) The function of a basic cell in Fig.3(b)

$$\begin{bmatrix} a_{11} & a_{12} & & 0 \\ a_{21} & a_{22} & a_{23} & \\ & a_{32} & a_{33} & a_{34} \\ 0 & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} & \\ & b_{32} & b_{33} & b_{34} & \\ 0 & & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & & 0 \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & \\ c_{31} & c_{32} & c_{33} & c_{34} & & \\ 0 & & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Fig.4(a) Band matrix/band matrix multiplication with W1 = 3 and W2 = 4.

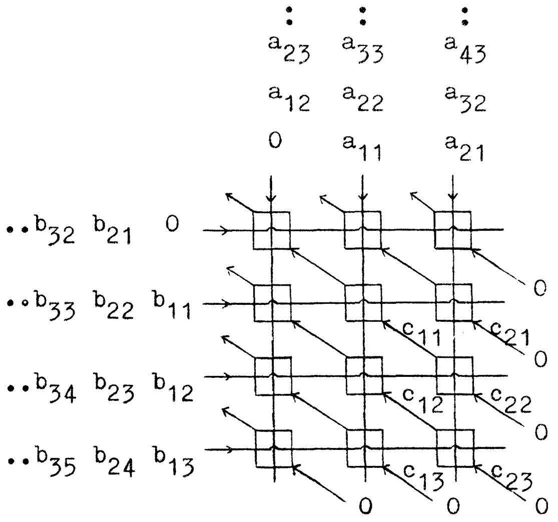

a_{23} a_{33} a_{43}
a_{12} a_{22} a_{32}
0 a_{11} a_{21}
••b_{32} b_{21} 0
••b_{33} b_{22} b_{11}
••b_{34} b_{23} b_{12}
••b_{35} b_{24} b_{13}
0
c_{11}
c_{21}
0
c_{12}
c_{22}
0
c_{13}
c_{23}
0 0 0

Fig 4(b) CDLAP configuration for Fig. 4(a).

HIGHLY PARALLEL PROCESSOR ARRAY "PAX" FOR WIDE SCIENTIFIC APPLICATIONS

Tsutomu Hoshino, Tomonori Shirakawa, Takeshi Kamimura,
Takahisa Kageyama, Kiyo Takenouchi, Hidehiko Abe,
Institute of Engineering Mechanics, University of Tsukuba, Sakura, Niihari, Ibaraki, JAPAN.

Satoshi Sekiguchi, Yoshio Oyanagi,
Institute of Information Sciences, University of Tsukuba, Sakura, Niihari, Ibaraki, JAPAN.

Toshio Kawai,
Department of Physics, Keio University, Hiyoshi, Kouhokuku, Yokohama, JAPAN.

Abstract -- Architecture of microcomputer array PAX is characterized by end-around nearest-neighbor interprocessor connection, data broadcasting capability, asynchronous MIMD operation, and global synchronization control. The system with 128 processors was built, and has been dedicated to the scientific calculations, not only limited to the partial differential equation models, but also extended to the non-nearest-neighbor type, i.e. Monte Carlo simulation with interacting particles, linear equation solving of Gauss-Jordan and conjugate gradient methods, and fast Fourier transform algorithm. High efficiency upto 98 % was demonstrated in these applications on PAX with 32 processors. The parallel execution times were measured and summarized as the scaling law expressing the execution time as a function of problem size and the number of processors in the array. The proven scaling law permits us to extrapolate the present high performance to the super parallel machine with processors more than 1000.

1. INTRODUCTION

Scientific calculations are characterized by the universal principle "action through medium," stating that the physical actions come from the immediately neighboring medium or field. The principle assures that the interprocessor communication is limited in the nearest neighbor processors, if the original space is projected directly onto the array of processors distributed analogously to the physical space. The multiprocessor architecture that utilizes this inherent proximity property is the well-known nearest neighbor mesh (NNM) connection of the processors, which has been studied in machines such as ILLIAC-IV [1] and ICL-DAP [2].

This connectivity, however, has been supposed to be inefficient when the data exchange is required between any pair of processors, such as in the matrix calculus or in the general particle transport problems. The pessimistic conclusion for the matrix product or inversion [3] is based on the assignment of each matrix element to a processor, where the operational load for each matrix element (in O(N), N = size of the matrix) is by far less than that for the data move (in $O(N^2)$). However, if we partition the matrix row-wise, i.e. a single processor takes care of an unknown variable(s), the operational load has the same order, $O(N^2)$ as that for the data broadcast. In the case of particle transport that requires global data circulation in the array, the operational load shows even higher order dependence on the problem size than that of the interprocessor communication. In both cases, the users always demand to expand the problem size, far larger than the parallelism available at present and in near future, so that such a row-wise partitioning would be justified.

We have started the project of developing an array of processors with NNM connection as well as data broadcasting capability, in order to clarify this question by implementing the practical models and by demonstrating the capability of the NNM architecture to a number of typical scientific problems. It is our standpoint that the applicability must be proven based on the concrete applications. It is because the relative value of the communication overhead to the net local calculation is a crucial factor, no matter how the absolute value of the overhead becomes large. Also from the practical viewpoint, not only the order dependence but also the coefficients of the dependent terms are important.

The processor array developed was previously called "PACS", Processor Array for Continuum Simulations, [4]-[6], and two systems with 9 and 32 procesors were actually built. The present system under testing is named "PAX-128", Processor Array eXperiment with 128 processors. A number of applications were implemented so far with successful performance; among them were partial differential equation models in aerodynamics [4], potential problem associated with Poisson equation [4], realistic nuclear power reactor calculation [4],[7], and Monte Carlo simulations of non-interacting plasma particles [4] and spin systems in fundamental physics [6]. These models are characterized by the proximity property inherent to the physical processes simulated. The applications were then extended to the models of non-proximity type, such as Monte Carlo problems with general particle interactions, and matrix calculations in linear equation solving.

These application problems were implemented in PAX-32 system consisting of 32 microcomputers. The time for parallel computation was measured and analyzed in terms of the problem size and number of processors. The derived "scaling law" is the proven base on which we proceed the plan for future super parallel system with processors of more than 1000 and speed of more than GFLOPS.

2. SYSTEM ARCHITECTURE

2.1 System Configuration

0190-3918/83/0000/0095$01.00 © 1983 IEEE

Our series of parallel computer has, at present, two models, the prototype machine PAX-32 [4]-[7] and the proof machine PAX-128. Both models employ 2-dimensional end-around NNM-connection, as well as data broadcating bus line. They comprised of three parts: Host computer (HOST), control unit (CU), and processing unit (PU) arrays. HOST and CU are commonly used by both machines, as shown in Fig. 1.

The PU array executes the parallel tasks. Each PU is essentially a single-board microcomputer. In the PAX-32 system, 32 PUs are implemented, constituting an 8 x 4 rectangular array, and in the PAX-128 system, 128 PUs constitute an 8 x 16 rectangular array. The CU is a microcomputer that controls the PU array and communicates with the HOST computer, via a parallel interface. The CU can also broadcast the data to all PUs. The HOST is a general purpose minicomputer, Texas Instruments 990 model 20. It is used to compile/assemble the source program, load the object program into the CU and PU, initiate the parallel tasks, tranfer and receive the data to and from the CU, and output the computed results.

2.2 Processing Unit

Each processing unit consists of the components shown in Fig. 2, and described below.

Micro processing unit (MPU). An 8-bit microprocessor. The Motorola MC6800 (1 MHz) and MC68B00 (2 MHz) are used in PAX-32 and PAX-128, respectively. It is in charge of the program execution such as 8-bit fixed point arithmetic and logical operations, 8-bit data transfer between memories, and the control of the arithmetic processing unit. Address space is 32 Kbytes.

Arithmetic processing unit (APU). A microprogrammed attached processor that executes fixed-point arithmetic (both in 16-bit and 32-bit lengths), floating-point arithmetic (in 32-bit length), and the elementary function calculations, such as logarithms, square roots, etc. The Advanced Micro Device's Am9511 (2 MHz) and Am9511A-4 (4 MHz) are used in PAX-32 and PAX-128 respectively. The data transfer to and from the APU and the initiation of every operation in the APU are perfectly controlled by the MPU. By taking the average times over the floating-point addition and multiplication in Am9511, we estimate the average floating-point execution time is equal to 65 micro-seconds, which means that 0.0154 MFLOPS (Million FLoating-point Operations Per Second) can be performed by each PU. The PAX-32 system's peak performance is thus estimated to be 0.0154 MFLOPS x 32 = 0.5 MFLOPS. In PAX-128, it is expected to be 0.031 MFLOPS x 128 = 4.0 MFLOPS.

Local memory (LM). The local memory for local data and program store. The MPU accesses to LM of 22 Kbytes in PAX-32, and 28 Kbytes in PAX-128.

Communication memory (CM). The memory shared between neighboring PUs. This memory is used for the data transfer between the PUs.

Control register (CR). This is used to transfer the control word from CU to PU. CR is write-only for the CU and read-only for the PU.

Status register (SR). This is used to report the PU state to CU. It is write-only for the PU and read-only for the CU. The PU informs its status by the code defined in the control software.

2.3 PU Array Control by the CU

Before starting the parallel tasks in PU array, the CU has to load the programs and data into the proper memories in the PU array, and also it has to start the MPUs. In order to control and terminate the parallel tasks, it is necessary for the CU to control and monitor the PU array without disturbing the task execution. Together with the microprocessor MC6800, and local memory (LM), the CU has the following registers to be used for control purposes.

Status Register of all PUs (SR-ALL). This indicates the logical AND and OR of the contents of the SR registers in the all PUs. This information is used in detecting the consistency of all SRs, in order to take the task level synchronization quickly in all the PUs.

Unit Reset register (URES). This is used to reset the MPU, and APU.

Unit Halt register (UHLT). This is installed only in PAX-128, and used to halt the MPU.

Unit Nonmaskable Interrupt register (UNMI). This is used to interrupt the MPU in the PU.

Unit Select register(USEL). This is used to control the bus-switch between the CU-bus and the PU-bus. When a PU is selected by the USEL register, the CU-bus is connected to the PU-bus.

Each function of URES, UHLT, UNMI, and USEL is achieved by writing to the particular register of the CU the code to select the row and column of the particular PU in the processor array. For example, the following selections can be made: PU in row 0 and column 1, all PUs in row 2, all PUs in columns of odd numbers, all PUs, no PUs, etc.

By connecting the CU-bus with all PU-buses, CU can directly broadcast the data and the program to all PU memories.

2.4 Hardware Implementation

Special care was taken in the hardware implementation in order to minimize the connection wiring. The structure of PAX is topologically a torus. The conventional method, parallel placement of the boards and connection on the backplane would result in connections too long.

Our implementation of PAX-32 is a "folded array" ; that is, the original torus geometry is folded, as shown in Reference [4], while the implementation of PAX-128 is exactly a "torus" as shown in Fig. 3. The CU bus and the connections between PUs use flat cables. These methods of implementation reduce wiring length to nearly 1/10 of that in the conventional method.

2.5 Software

Software support for the PAX includes language processors and Fortran subroutines executed on the HOST, a parallel array monitor executed on the CU, and a library of control procedures executed on the PU. Source programs for applications are usually written by the use of Fortran for the serial part of the job executed in the HOST, and by the use of high-level language SPLM for the parallel part of the job that is executed in the PUs. The cross-assembler MASP is available, if necessary, to code the parallel part of the job and the control procedures.

Parallel execution of tasks can be initiated in PUs and data can be transferred to or from the HOST by calling the PCMS subroutine in the user's FORTRAN source program. This subroutine sends several commands to the CU to control the execution in PUs and the data transfer between PU and HOST.

The high-level language SPLM, a structured programming-type language, was developed for parallel processing in the PAX system. Domain declaration is necessary to reserve specified memories for the storage of variables, constants, and program. The user must declare the domain CONST in order to reserve memory space for the program and constants. The domains for the variables are declared by the followings.

VAR : For the local variables in LM memory.

FRONT, BACK, LEFT, RIGHT : For the variables in CM memories used for inter-PU communications.

Variables should be declared with their domains, types and size of array variables. The variable type can be either real, integer, or byte.

In contrast to those languages used for SIMD architecture, the SPLM describes the function limited in a single PU; how the PU takes data from the CM area, how it processes the data, and how the data are returned to a CM. Usually the same program is loaded in all PUs utilizing the broadcast function of the CU. However, different programs can be sequentially loaded to PUs. The overall parallel computation proceeds as each PU executes its own program. The processing, therefore, inherently asynchronous.

Task synchronization, however, is supported by software control. By the procedure SYNC, the task can be synchronized at any user-specified point. The synchronization is taken to ensure that the transferred data is correctly updated. It is also taken in such algorithms as SOR that uniform iteration is necessary for all PUs. Execution of the task resumes by exiting the SYNC procedure. It takes 107 and 57 micro-sec to synchronize all PUs of PAX-32 and PAX-128, respectively.

Nonexpert users require more system software support than that already described. One of the inconveniences occurs in the determination of the PU boundary when a physical space is projected onto the PU array space. There are several subroutines that assist users in these situations. One example is a subroutine which distributes the multi-dimensioned array variable in the HOST to the PU array mesh. Another is a procedure providing the values of the dimensioned variable at the nearest neighbor mesh points without regard to the PU boundary.

Data can be routed in two directions in the PU array by the ROUT procedure. A general routing procedure GR exchanges data with arbitrary PU, using the tags indicating the destination of the data. Data are routed in two directions, and each PU catches the data with tags circulating in the array.

Another possibility for the data move among PUs is the data broadcasting through the CU. By calling procedure BRDCAST, the user can move the data in any PU to all other PUs.

3. APPLICATIONS AND SCALING LAW

Three applications are introduced here; these are typical non-proximity type problems. The first example is a Monte Carlo simulation of moving particles that interact each other through the potential field that they create. Example was taken from the molecular structure simulation for the amorphous material. The second example is the well-known linear equation solving by typical schemes: Gauss-Jordan and conjugate gradient methods. The third example is the well-known fast Fourier transform algorithm. It is the final goal of the study to express the measured time of execution as a function of the problem size and the number of processors.

3.1 Execution time and efficiency

Let us define the efficiency of parallel processing by the array of P PUs as follows.

$$\alpha = Ts/(PTp)$$

where Ts = time for serial execution of the job by a single PU, Tp = time for parallel execution of the same job by P PUs, and P = the number of PUs constituting the array.

Let us limit the discussion to the iterative processing, where one iteration is a sequence of (1) synchronization of all PUs, (2) data move, and (3) net calculation, as shown in Fig. 4. The process is iterated until convergence. The processor may fall into the idling state before the synchronization point, since the net calculation may be different from PU to PU.

The overhead is defined here as those tasks that do not appear in the serial processing. Here it consists of the synchronization, the data move, and the processor idling before the synchronization. Then the following expressions are derived.

$$Tp = \max_j Tj + \max_j Tcj, \qquad Ts = \sum_j Tj,$$

$$PTp = \sum_j Tj + \sum_j Twj + \sum_j Tcj,$$

$$\alpha = (Tp - Tw - Tc)/Tp = \text{(net calculation time)/(total processing time)},$$

where Tj = time for the net calculation in the j'th PU, Tcj = time for the data move and synchronization with the j'th PU, Twj = idling time of the j'th PU, Tw = idling time of PU averaged over all PUs, Tc = data move time averaged over all PUs.

If the algorithm for parallel processing differs from that for serial processing, then the algorithmic degradation factor [8] must be applied to discuss the speed-up ratio over the conventional computer.

3.2 Monte Carlo Simulation of Interacting Particles

3.2.1 Model

Monte Carlo simulation models can be classified into three categories:

1. Non-interaction type model, in which the particles move independently from each other. Radiation transport model represents this category.
2. Nearest-neighbor type model, in which the particles or spins interact with those in the nearby space, but do not move. Spin systems or field models in the fundamental physics fall into this category.
3. Interaction type, in which the particles interact with each other through the field that they create. This is the model of the most general type. The plasma particle simulation, molecular dynamics, and galaxy model belong to this category.

The model of the third 3. category is taken as an example, i.e. the simulation of the structure of amorphous material [9]. The physical space is the 3-dimensional rectangular space with the cyclic boundary condition, i.e. one end of the space is connected with the opposite end. Simulation goes as follows:

1. The particles are randomly distributed in the space.
2. The potential and its derivative (i.e. force) are calculated between all pairs of particles.
3. The particles are pushed by the force which they receive. The moving distance of each particle is determined so as to minimize the potential energy that the particle creates with the nearby particles.
4. The physical parameters of interest, such as total energy are calculated.
5. The procedures 2. and 4. are repeated until the overall potential energy reaches the minimum.

3.2.2 Parallel Scheme

The parallel partitioning of the simulation follows either one of the following two methods or the mixture of them.

1. "Lagrangian scheme", where the particles are divided into subgroups of equal number of particles, each to be assigned to a PU, and the PU is in charge of everything that happens to the particles, no matter how the particles are moving in the physical space.
2. "Eulerian scheme", where the physical space is divided into subspaces of equal size, each to be assigned to a PU, that takes everything happening in the subspace.

Analyzing the implemented program and measuring the execution time, we found Eulerian scheme unattractive, since the non-uniform particle distribution causes the degradation of the efficiency [10]. It was found, in the Eulerian scheme, that, as the number of particles N becomes large, the calculations of potential, force and particle move in the busiest PU dominate in the total execution time. The efficiency becomes asymptotic to 1/f as P tends to infinity, where f is the peaking factor of the particle number distribution, i.e., ratio of the maximum over average numbers of particles per PU.

Discussion will be centered on the following Lagrangian scheme.

Parallel Lagrangian scheme.

1. The particles are uniformly distributed in the PU array.
2. The all particle data are circulated in the PU array. This task is carried out by synchronized routing of data, as follows:
   2-1. The particles ("data" is omitted hereafter) are routed in the RIGHT directions in all PUs, until each of the four PUs in the row of the array shares all particles located in this row.
   2-2. Then the particles are transferred in the FRONT directions in all PUs until all particles encounter each other in the array.
3. The potential energy and force are calculated in each PU during the circulation of particles, that goes in parallel in all PUs.
4. The new particle positions are calculated in each PU in parallel.
5. The physical parameters such as the total potential energy, and average particle move during the iteration are calculated. These global sum quantities over all PUs can be calculated in parallel by the well-known cascade sum method [11], where $\log_2 32 = 5$ additions and the nearest-neighbor data transfer of the distance of 10 PUs (i.e. data are routed twice in RIGHT direction by 1, and 2-PU distances, and then routed three-times in FRONT direction by 1, 2, and 4-PUs).
6. The procedures from 2. to 5. are repeated until the potential energy reaches minimum.

3.2.3 Parallel Execution Time and Scaling Law

Suppose we have N particles on the 2-dimensional rectangular array of P = P1 · P2 PUs. A few parameters are defined; n=N/P, i.e. average number of particles per PU, d = the 1-dimensional size of the physical space, d* = cut-off length of potential, i.e. the distance where the potential reaches zero, and c = d*/d.

Parallel processing program and measured time are analyzed as follows. If not specified, execution time is measured in unit of msec throughout this paper.

1. Particles are uniformly distributed among PUs; n particles per PU.
2. The n particle data are transferred to RIGHT neighbor in (P1-1) times, and nP1 data are moved to FRONT neighbor in (P2-1) times, so that this global circulation of three coordinate data

takes

$$T1 = 1.2\ n\ (P-1) + 3\ ts.$$

3. Potential and force calculations take

$$T2 = 3.8\ n\ N.$$

4. Particle move takes time proportional to the number of particles within the reach of potential function d*, i.e.

$$T3 = n\ (5.7 + 30.7\ Nc).$$

5. Three physical parameters which require the cascade sum and global data routing are calculated. It takes

$$T4 = 3\ (\log_2 P\ (ts + 0.753) + 0.17\ (P1 + P2 -2)).$$

The scaling law: the total parallel execution time Tp is represented by

$$Tp = T1 + T2 + T3 + T4,$$

where Tc = T1 + T4, Tw = 0, and the efficiency by

$$\alpha = 1 - Tc/Tp.$$

These are calculated for several parameter values, as shown in Table I.

Table I. Measured execution time and efficiency in parallel Monte Carlo simulation of amorphous material (Lagrangian scheme) on PAX-32 array.

N	Tp (sec)	Td (sec)	α=1-Td/Tp (%)
192	6.77 (6.87)	0.23 (0.24)	96.60 (96.48)
224	9.29 (9.29)	0.27 (0.28)	97.09 (97.00)
256	12.20 (12.08)	0.31 (0.32)	97.46 (97.38)
288	15.49 (15.23)	0.34 (0.35)	97.80 (97.68)

N = Number of particles, Tp = Total execution time (sec), Td = Data move and idling time (sec), α = 1 - Td/Tp = Efficiency (%). The values in ( ) are calculated from the scaling law.

Since we assume that number of particles per PU n = N/P is a constant parameter determined by the memory capacity of PU, the intra-PU calculation has the same dependence O(N) as the inter-PU communication. As the number of particles N increases, these terms with the dependence O(N) become dominant in the total execution time Tp. This makes the efficiency asymptotic to a constant factor, 30.7 nc /(30.7 nc + 1.2), as P tends to infinity.

Again it must be noted that the inter-PU communication does not dominate over the intra-PU calculation.

3.3 Linear Equation Solving by Gauss-Jordan and Conjugate-Gradient Schemes

3.3.1 Job Partitioning into Parallel Tasks

Parallel processing of linear equation has been extensively studied and parallelism of several levels has so far been exploited [11], [12]. However, it is still important to evaluate the parallel scheme of these well-known algorithms such as Gauss-Jordan and conjugate gradient methods, by implementing on the machine actually operating. It is because, as pointed out in [3], not only the intra-PU operational tasks, but also the inter-PU data move may take major part of the time, leading to the very low efficiency.

As briefly mentioned in the introduction, we made an approach to the matrix processing different from those assumed in Ref. [3]. Our partitioning of the linear equation problem is such that each PU takes care of one or several successive row(s) of matrix and corresponding elements of unknown and constant vectors, as shown in Fig. 5. Major reason is that the balancing is possible between work loads of intra-PU and inter-PU processings as pointed out in the introduction; $O(N^2)$ work load of intra-PU process vs. $O(N^2)$ load of broadcasting among PUs, where N = matrix size. This row-wise partitioning is appropriate as well in constructing the stiffness matrix in finite element analysis. If each PU takes care of the unknown variables associated with nodes or mesh points of the physical structure, the matrix generation itself can be made with high parallelism.

For the practical use of the elimination schemes, the pivoting is important to get the reliable solutions. The maximum element being the next pivot element is found by the comparison among the column elements. Since each row vector is stored in each PU, this maximum finding is carried out by cascade comparison method, in which the data are routed similarily to the cascade sum method, but the comparison is made instead of summation. When a few rows are assigned to each PU, the cascade comparison scheme is made after the intra-PU comparsion.

The inter-PU communication is made by utilizing the broadcasting under the control of CU, from local memory of any single PU to local memories of all other PUs. The circulation of data through the CM memory is not used for this linear equation solving, except the norm calculation over the distributed data in the array. This is because the work load for broadcasting uniformly distributed data to all PUs (i.e. making all combinations of data in the PU array) is the same as that for the same work by the data circulation via nearest data move. Suppose we have N data in P PUs, i.e. N/P data for a PU. The broadcasting N/P data P times takes time proportional to N, while the data circulation via nearest data move takes (N/P)(P1-1) + (N/P2)(P2-1)=N(P-1)/P, where P = P1 · P2. In the present PAX system, single data move in both broadcast/nearest move takes two steps; data write to and read from the CU/CM memories. For P equal to or greater than 32, both works take nearly the same time.

It must be noted that both methods do not give the same work load in the case of Gauss-Jordan elimination, because only the pivot row data are broadcasted to all other rows. This is the case where the broadcast method is superior to the circulation method. It must also be noted that there is no need to exchange the whole row vector data, because only the index number that tells where the row is located in the matrix must be altered.

3.3.2 Parallel Scheme

Variable assignment is made such that the matrix and vectors are divided into P blocks, each having N/P rows, and the i'th block is taken care of by the i'th PU. For simplicity, the explanation of algorithm below assumes N = P, i.e. one-row-to-one-PU correspondence. Figure 6 illustrates how the scheme goes in parallel in PUs.

Parallel Gauss-Jordan scheme.

For k=1,2,...,N, the following two-steps are sequentially executed.

1. From the k'th PU, the k'th row of the matrix and the k'th element of the constant vector $\vec{b}$ are broadcasted to the all PUs.

2. The k'th PU executes

$$a_{kj}^{(k)} := a_{kj}^{(k-1)} / a_{kk}^{(k-1)} , \ (k \le j \le N), \qquad b_{k}^{(k)} := b_{k}^{(k-1)} / a_{kk}^{(k-1)} .$$

The rest of the PUs execute

$$a_{ij}^{(k)} := a_{ij}^{(k-1)} - a_{ik}^{(k-1)} \ a_{kj}^{(k-1)} \ / \ a_{kk}^{(k-1)} ,$$

$$b_{i}^{(k)} := b_{i}^{(k-1)} - a_{ik}^{(k-1)} \ b_{k}^{(k-1)} \ / \ a_{kk}^{(k-1)} ,$$

$$(k \le j \le N, \ 1 \le i \le N, \ i \neq k).$$

When the partial pivoting is made, the k'th row is one that has the largest absolute value among $a_{\ell k}$, $(k \le \ell \le N)$.

Parallel conjugate gradient scheme.

The conjugate gradient method searches the solution vector that gives the minimum value of the error norm. The scheme starts with the initialization,

$$\vec{r}^{(0)} := \vec{b}^{(0)} - \mathbb{A} \vec{x}^{(0)} , \ \vec{p}^{(0)} := \vec{r}^{(0)} , \ k = 0.$$

The following steps are repeated until either k>N or $|\vec{r}|<e$, where e = error criterion. The matrix $\mathbb{A}$ is assumed to be sysmetric and positive definite.

$$\alpha^{(k)} := (\vec{r}^{(k)} \cdot \vec{p}^{(k)})/(\vec{p}^{(k)} \cdot \mathbb{A} \vec{p}^{(k)}),$$

$$\vec{x}^{(k+1)} := \vec{x}^{(k)} + \alpha^{(k)} \vec{p}^{(k)} ,$$

$$\vec{r}^{(k+1)} := \vec{r}^{(k)} - \alpha^{(k)} \mathbb{A} \vec{p}^{(k)},$$

$$\beta^{(k)} := (\vec{r}^{(k+1)} \cdot \vec{r}^{(k+1)})/(\vec{r}^{(k)} \cdot \vec{r}^{(k)}),$$

$$\vec{p}^{(k+1)} := \vec{r}^{(k+1)} + \beta^{(k)} \vec{p}^{(k)} ,$$

$$k := k + 1.$$

Parallelization of the scheme follows the same partitioning of the variable as parallel Gauss-Jordan scheme. The vector and matrix calculations are executed as follows.

1. Vector addition or subtraction is made independently in each PU.

2. Scalar product of vector is made such that the partial product is calculated independently in each PU, and then the partial product is summed up by cascade sum method.

3. The product of matrix $\mathbb{A}$ and vector $\vec{x}$ is made by such a way that,

(a). the j'th element of $\vec{x}$, x_j is broadcasted from the j'th PU, and that,

(b). in the i'th PU, $a_{ij} \ x_j$ is summed up into the partial sum.

These steps (a) and (b) are repeated sequentially for j=1,2,...,N.

3.3.3 Execution Time and Scaling Law

Implementation of these two schemes on PAX-32 system was made successfully for the problem sizes, N=32, 64, 96, 128, and 160. For the conjugate gradient method, the experiment was made for two matrices, one giving good convergence CASE (A), and the other poor convergence CASE (B). The scaling law is the function that expresses the time for parallel calculation in terms of the problem size N, the number of processors P, and the number of iterations M in the conjugate gradient method. The computation time was actually measured and summarized in Table II. The scaling law was derived from these measured data with good accuracy as shown below.

Let us define several operation times for elementary processings; ts = time for taking synchronization, tr = time for real data move between nearest neighbor PUs, ti = similar time for integer move, ta = time for data addition. The measured times for PAX-32 PU array are ts=0.107, tr=0.161, ti=0.126, ta=0.412 (msec).

Time for cascade sum is expressed by

$$Tcsum = (ts + ta) \log_2 P + (P1 + P2 -2) \ tr.$$

Time for broadcasing k data from any PU to all other PUs takes

$$Tbrd(k) = (0.160k + 0.02 \ k/64 + 0.261) + ts.$$

Time for partial pivoting Tpiv can be obtained by a similar expression for Tcsum except that the time for addition ta is replaced by the time for compare and substitution, i.e. ta'=0.870 (msec). The data moved are one real number (an element of matrix) and two integers (PU and row numbers).

The scaling law for the Gauss-Jordan scheme is expressed by using these measured total and elementary times, as follows.

$$Tgj(N,P) = 0.011 \ (N/P+1) + 1.047 \ N /P + (Tpiv + 0.495)N + \sum_{k} Tbrd(k+1) + 0.157 \ N \ (N+1)/P.$$

The estimated time by this scaling law can reproduce the measured values as shown in Table II.

The scaling law can be derived similarily for the conjugate gradient method. The time for the product of matrix and vector Tmat is derived as $Tmat(N,P) = 0.033 \ N/P + 2.43 + P \ (0.222 + Tbrd(N/P) + 0.094 \ (N/P-1) + 0.224 \ (N/P) \ )$

The total parallel computation for conjugate gradient method takes

$$Tcg(N,P,M) = (0.1 + 0.725 \ M) \ N/P + 0.379 \ M + (M+1) \ Tmat(N,P) + M \ Tbrd(1) + (3M + 1)(Tcsum + 0.173) + 0.5,$$

where M=number of iterations. The estimated times by this expression is compared in Table II with the measured values.

The efficiency defined in 3.1 can be applied to these parallel executions as well. For the Gauss-Jordan scheme, the idling time immediately before the synchronization Tw is almost zero, because that the difference between the processings for pivot PU and non-pivot PU is small, and that the idling occurs in only a single PU, i.e. pivot PU. The efficiency is therefore approximated by $\alpha gj = (Tgj - Tc)/Tgj$, where Tc is the time for data move, which is represented by $Tc = N (3\ ts\ \log_2 P + (P1 + P2 -2)(tr + 2ti)$ for Gauss-Jordan scheme. Similar expression holds for the conjugate gradient scheme: $\alpha cg = (Tcg-Tc)/Tcg$, where $Tc = P\ Tbrd(n) + M\ Tbrd(1) + (3M + 1)(ts \log_2 P + tr\ (P1 + P2 - 2))$.

The evaluation of the efficiency is made assuming that n = N/P, the number of rows per PU is a fixed parameter. Asymptotic behavior of the efficiency is such that

$$\lim_{P\to\infty} \alpha gj = n/(n + 0.510), \text{ for Gauss-Jordan},$$

$$\lim_{P\to\infty} \alpha cg = (n^2 + 0.419\ n + 0.573)/(n^2 + 1.14\ n + 2.22), \text{ for conjugate-gradient.}$$

Efficiencies of both schemes become asymptotic to the values ranging from 46 % for n=1 of conjugate gradient scheme (the worst case) to 95 % for n=10 of Gauss-Jordan scheme, and get to 100 % as $n\to\infty$.

Again the inter-PU communication does not devastate the efficiency of these parallel linear equation schemes.

Table II. Measured execution time and efficiency in linear equation solving on PAX-32 array.

<< GAUSS-JORDAN SCHEME >>

N	Tp (sec)	α (%)	Case
32	0.585 (0.640)	55.44	(A)
64	2.558 (2.485)	70.47	(A)
96	6.636 (6.502)	79.28	(A)
128	13.853 (13.656)	84.44	(A)
160	25.676 (24.912)	87.69	(A)

<< CONJUGATE-GRADIENT SCHEME >>

N	Tp (sec)	α (%)	Ni	Case
32	0.509 (0.473)	50.69	9	(A)
64	0.831 (0.777)	63.33	9	(A)
96	1.286 (1.223)	72.52	9	(A)
128	1.879 (1.812)	78.63	9	(A)
160	2.602 (2.545)	82.77	9	(A)
32	1.651 (1.586)	50.69	32	(B)
64	5.337 (5.106)	63.22	64	(B)
96	12.297 (11.957)	72.39	96	(B)
128	23.884 (23.514)	78.51	128	(B)
160	41.341 (41.151)	82.67	160	(B)

N = Matrix size, Tp = Total execution time (sec), α = Efficiency (%), Ni = Number of iterations. Values in ( ) are calculated from the scaling law.
Case (A) is a matrix example that gives good convergence in the conjugate-gradient scheme, while Case (B) is one that gives poor convergence.

3.4 Fast Fourier Transform (FFT)

We have also implemented a Parallel FFT (Fast Fourier Transform) on PAX, and have estimated the efficiency by observing the time of execution. Note that an FFT algorithm is in need of data exchange between distant PUs.

Despite the existence of the specialized hardware for FFT, we were motivated to try the parallel processing of FFT on PU array, because that single processor array should work in two phases of interacting particle transport problems by the particle-mesh-method [13], where the potential field is solved Eulerian by using FFT, and the particles are pushed Lagrangian. If we use two systems: special FFT hardware and the processor for particle push, there must be wide band-width among these processors and common memory. If the PAX is proven fast enough in FFT, the whole problem can be solved on the processor array of PAX.

3.4.1 Parallel FFT Algorithm

Using the Cooley-Tukey [14] notation, FFT algorithm is written as follows. Consider the problem of calculating the complex Fourier series,

$$\varphi (j) = \sum_{k=0}^{N-1} X(k)\ W^{jk} \quad , \ j = 0,1,\ldots,N-1 \quad , \qquad (1)$$

where the given Fourier coefficients X(k) are complex and W is the principal N'th root of unity, $W = \exp(2\pi i/N)$. Suppose N is a power of 2, i.e. $N=2^n$. Then let the indices in (1) be expressed as

$$j = j_{n-1}\ 2^{n-1} + j_{n-2}\ 2^{n-2} + \ldots + j_1\ 2 + j_0 \ ,$$

$$k = k_{n-1}\ 2^{n-1} + k_{n-2}\ 2^{n-2} + \ldots + k_1\ 2 + k_0 \ ,$$

where j_v and k_v are equal to 0 and 1. This expression gives a unique representation of j and k. So, this allows Eq.(1) to be written as

$$\varphi (j_{n-1}, j_{n-2}, \ldots, j_1, j_0) = \sum_{k_0} \sum_{k_1} \ldots \sum_{k_{n-1}} X(k_{n-1}, k_{n-2}, \ldots, k_1, k_0)\ W^{jk} .$$

The innermost sum, over k_{n-1}, can be written as

$$A_1(j_0, k_{n-2}, \ldots, k_1, k_0) = \sum_{k_{n-1}} X(k_{n-1}, k_{n-2}, \ldots, k_1, k_0)\ W^{j_0 k_{n-1} 2^{n-1}} .$$

Proceeding to the next innermost sum, over k_{n-2}, and so on, one obtains recursive notation,

$$A_\ell(j_0, \ldots, j_{\ell-1}, k_{n-\ell-1}, \ldots k_0) = \sum_{k_{n-\ell}} A_{\ell-1}(j_0, \ldots, j_{\ell-2}, k_{n-\ell}, \ldots k_0)\ W^{(j_{\ell-1} 2^{\ell-1} + \cdots + j_0) k_{n-\ell} 2^{n-\ell}} \qquad (2)$$

for l=1,2,...,n. Equation (2) shows that N independent calculations are obtained at each l'th step. In Fig. 7, FFT data flow diagram is shown where N=8, in which

$$W^{(j_{\ell-1} 2^{\ell-1} + \cdots + j_0) k_{n-\ell} 2^{n-\ell}}$$

is multiplied on $\longrightarrow$, and summed, over $k_{n-\ell}$, on $\oplus$. After n steps, the Fourier series can be evaluated at the right most side. Clearly independent N processes, $\longrightarrow$ and $\oplus$, can be separated by $---$.

Each PU(i,j) is corresponded with a non-negative integer p = iP1 + j, where the number of PU's is P, P=P1 · P2 (P1 ,P2 are also power of 2). Then each $\hat{P}(p)$, which means the PU(i,j), may have only one Fourier coefficient X(p) or $A_\ell(p)$, if N=P. Note that $\longrightarrow$ includes a data move operation between $A_{\ell-1}(j_0 , \ldots, j_{\ell-2} , 0, k_{n-\ell-1} , \ldots , k_0)$ and $A_{\ell-1}(j_0, \ldots, j_{\ell-2} , 1, k_{n-\ell-1} , \ldots , k_0)$, and this corresponds to a data routing between $\hat{P}(j_0, \ldots , j_{\ell-2} , 0, k_{n-\ell-1} , \ldots, k_0)$ and $\hat{P}(j_0, \ldots , j_{\ell-2} , 1, k_{n-\ell-1} , \ldots, k_0)$.

Then the interval between these PU's is $2^{n-\ell}$, so this can be easily realized using the routing operation of PAX.

The general case N>P, each PU must have N/P data which is

X(pN/P + q) or A(pN/P + q), for q = 0,1,...,P-1.

First, calculate the pseudo Fourier series $A_{N/P-1}$ for the data distributed over each PU by using a serial FFT algorithm. Second, $A_{N/P-1}$ is considered as N/P sets of Fourier coefficients to be calculated, the q'th set is $A_{N/P-1}$ (pN/P + q), for q = 0,1,...,P-1.

3.4.2 Execution Time and Scaling Law

We rewrite the parallel FFT algorithm as follows in steps 1 and 2.

Parallel FFT.

1. Calculate N/P data at each PU using the serial FFT algorithm, indicated by subscript 'seq'.
2. Execute parallel FFT processing with all PU's;

arithmetic part $\longrightarrow$ operation and $\oplus$ operation,
both indicated by subscript 'op',
routing part indicated by subscript 'r'.

The execution time of each part is written as Tseq(N/P), Top(N), Tr(N,P), then Tpara, the total time of parallel FFT processing, is expressed as Tpara = Tseq(N/P) + Top(N) + Tr(N,P) .

Table III. Measured execution time in parallel FFT algorithm on PAX-32 array, where P=32.

Number of Data per PU N/P	Parallel Execution Time, Tp (msec)	Serial Execution Time, Ts (msec)	Efficiency α =Ts/(Tp P) (%)
1	11.5	258.0	70.00
2	27.2	612.6	70.31
4	54.5	1358.2	77.81
8	109.6	2996.6	85.31

In Table III, the result of measurement is shown where P=32. Tseri, the time of serial FFT processing for N data with 1 PU, is the comparison, and it is clear Tseri = Tseq(N).

There exists an overhead of parallelization while the values for Tseri/Tpara are all less than P. This overhead in the parallel FFT processing is caused by

1. Idle PU's as a result of unbalanced load calculation.
2. Routing.

Cause 1. is a result of measuring the execution time of the PU whose calculated load is the heaviest. Related to Cause 2. is Tr, the total time for routing, and is represented as Tr = ts $\log_2 P$ + tt(N/P) $3\sqrt{P}$, where ts and tt are the unit time for synchronization and data transfer. Top, the time of the arithmetic part, is also represented as Top = top(N/P) $\log_2 P$ + constant, where top is the maximum execution time of one repitition. Moreover, Tseq(n), the total time of the serial FFT processing for n data with 1 PU, is Tseq(n) = t1 n $\log_2 n$ + t2 n + constant.
Define α as the efficiency of parallelization, α = Tseri / (Tpara · P). Assigning the measured parameters ts=0.14, tt=0.18, top=1.80, t1=1.82, and t2=1.20 (msec), we, then, get α and the ratio of routing in Tpara as we increase P, the number of PUs, as shown in Fig. 8. Though $\alpha \rightarrow 0$ as $P \rightarrow \infty$, parallel FFT processing on PAX is well practical, within the currently predicted limit of realization (about 10000 PUs).

4. PERFORMANCE EXTRAPOLATION

It is interesting to wonder whether high efficiency is still maintained, even if the number of processors gets very large, say more than 1000. Proven scaling laws permit us to extend the present performance to the super parallel systems.

The hardware can be speeded up by the factor of 100, which is realized by a microprocessor of 20 MIPS and 45 nano-sec memory devices. Then the system performance could be raised accordingly by the factor of 100, except the synchronization time ts, that consists of a fixed time for detection plus a time proportional to the size of the array P is necessary for the synchronization signal to traverse twice the PU array. However, the size-dependent term does not become great within the implementation limit P<1,000,000 [6]. The signal skew [15] is not the primary limiting factor to the maximum number of processors, either [6].

The extrapolation of performance, therefore, is such that, for number of processors greater than 1,000,000, the efficiency will decrease, while, for the realizable array size (P<10,000), the size-dependent synchronization overhead does not affect the performance extrapolated from that obtained on PAX-32.

The same statement as before must be repeated here: the inter-PU communication does not devastate the practical applicability.

5. CONCLUSIONS AND FUTURE PLAN

The pilot machine PAX-32 demonstrated that the nearest neighbor mesh processor array operable in MIMD mode has been proven capable of executing the wide scientific applications, with the efficiency high enough for the practical use. The applications implemented on PAX machine cover those problems without proximity, such as particle transport problems with general interaction, linear equation solving with matrix and vector calculations, and FFT algorithm. It was proven that, in the practical limit of the size of the array, the efficiency is not devastated by the data move between processors, and the almost linear speed-up can be expected.

The aim of our PAX development project is to demonstrate the practical usefulness of NNM array by physical means, i.e. by constructing the highly parallel NNM-connected PU array, and by solving the end-user's large scale scientific applications on it. The demonstration may only tells that the machine can be sufficiently good in such and such applications, (what we call sufficient condition approach), and it does not say anything about what is necessary for general applicability, (what we call necessary condition approach). Though the general applicability has not been established so far, we are going to pursue the sufficient condition for practical machine, and someday we will ask, "Is such a general-purpose machine really necessary that could cover applications wider than those covered by our PAX system ?" In short we hope to wipe out the widespread pessimism on the NNM array since ILLIAC-IV.

In order to provide super computing power, by far greater than the presently available, in cheap cost for the end users, we have a plan to go through several developmental steps, from the present PAX-128 system, consisting of 128 one-board microcomputers, with approximately 4 MFLOPS, to our final goal PAX-1M, consisting 1 million VLSI-processors with 1 Tera FLOPS speed, that may be realized in early 1990's.

We are optimistic in going our way. Optimism is sometimes dangerous, but no doubt motivates the progress.

ACKNOWLEDGEMENT

The authors are indebted to M. Okazaki and S. Itoh for their help in providing the Monte Carlo simulation program of amorphous material. Also they thank to M. Ejiri for her help in improving English.

REFERENCES

[1] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.J. Slotnick, and R.A. Stokes, "The ILLIAC-IV computer," IEEE Trans. Comput. C-17 (1968), pp. 746-757.

[2] P.M. Flanders, D.J. Hunt, S.F. Reddaway, and D. Perkinson, "Efficient high speed computing with the distributed array processor," High Speed Computer and Algorithm Organisation, Academic Press, London (1977), pp. 113-128.

[3] W.M. Gentleman, "Some complexity results for matrix computations on parallel processors," J. ACM, 25 (1978), pp. 112-115.

[4] T. Hoshino, T. Kawai, T. Shirakawa, J. Higashino, A. Yamaoka, H. Ito, T. Sato, and K. Sawada, "PACS, A parallel microprocessor array for scientific calculations," to be published in ACM Trans. Computer Systems, 1, (May 1983).

[5] T. Hoshino, T. Shirakawa, and T. Kawai, "Parallel processing for scientific applications," International Symp. on Applied Mathematics and Information Sciences, Kyoto University, (March 29-31, 1982), Kyoto, Japan, pp. 7-17--7-26.

[6] T. Hoshino, T. Shirakawa, Y. Oyanagi, K. Takenouchi, and T. Kawai, "Super Freedom Simulator PAX," The 16'th IBM Computer Science Symp. Working Conf. VLSI enigneering, IBM Japan, (October 1-3, 1982), Hakone, Japan, pp. 43-55. (Proceedings will be published as a Lecture Note by Springer Verlag.)

[7] T. Hoshino, and T. Shirakawa, "Load follow simulation of three-dimensional boiling water reactor core by PACS-32 parallel microprocessor system," Nuclear Technology, 56 (1982), pp. 465-477.

[8] R.W. Hockney, "Optimizing the FACR(1) Poisson-solver on parallel computers," Proc. IEEE 1982 International Conf. Parallel Processing, (August 24-27, 1982), pp. 62-71.

[9] T. Fujiwara, S. Itoh, and M. Okazaki, "Structural Model of Amorphous As2S3," J. Non-Crystalline Solids, 45 (1981), pp. 371-378.

[10] K. Takenouchi, "Monte Carlo Simulation Using Parallel Computer PACS," unpublished bachelor thesis, (in Japanese), College of Information Science, University of Tsukuba, (February 1983).

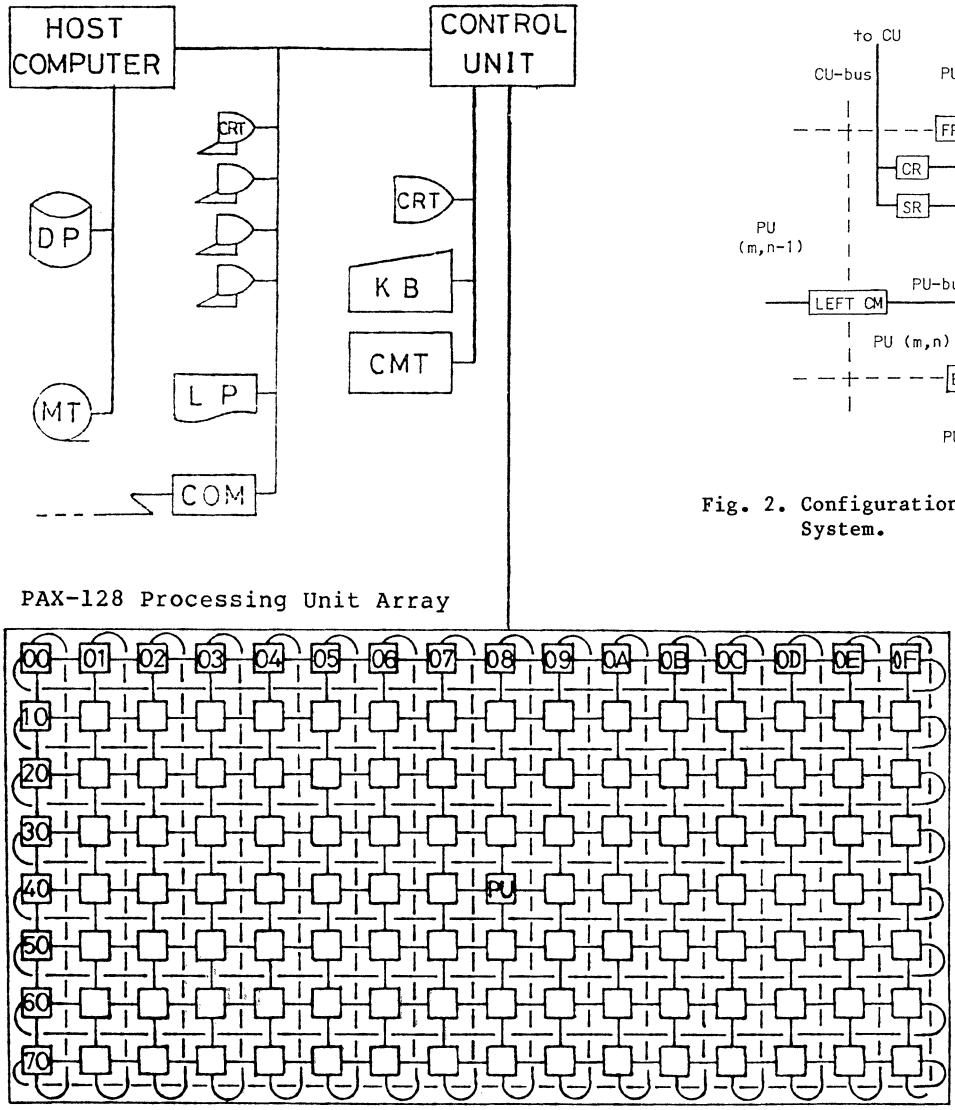
[11] R.W. Hockney, and C.R. Jesshope, Parallel Computers - Architecture, Programming and Algorithms, Adam Hilger, Bristol (1981), 423 pp.

[12] D. Heller, "A survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, 20 (1978), pp. 740-777.

[13] R.W. Hockney, and J.W. Eastwood, Computer Simulation Using Particles, McGraw-Hill, (1982), pp. 540.

[14] J.M. Cooley, and J.W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. Comp. 19 (1965), pp.297-301.

[15] A.L. Fisher, and H.T. Kung, "Synchronizing Large Systolic Arrays," private communication (1982).



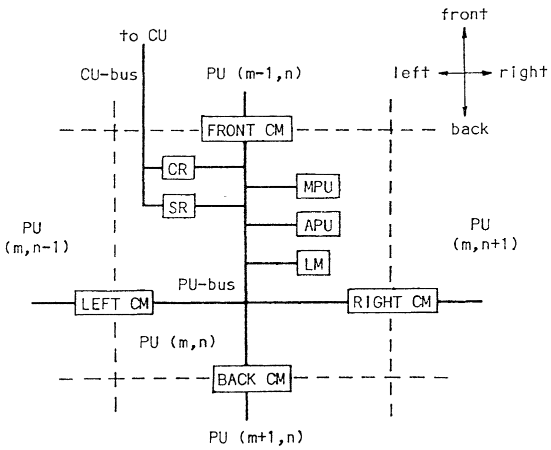


Fig. 2. Configuration of Processing Unit of PAX System.

Fig. 1. Configuration of PAX system.

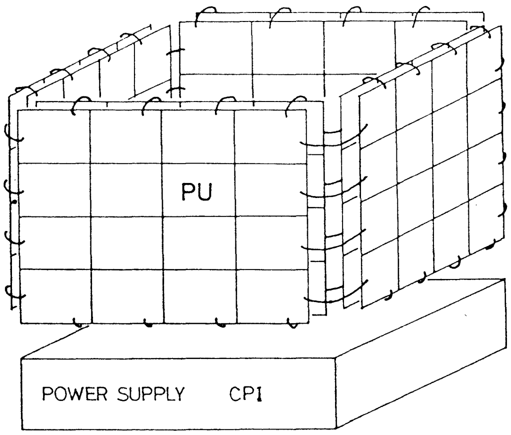


Fig. 3. Hardware Implementation of PAX-128 System.

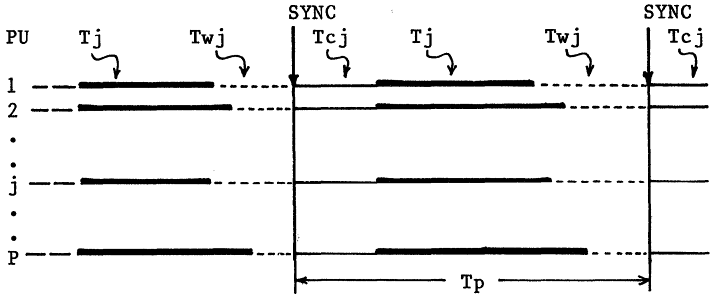
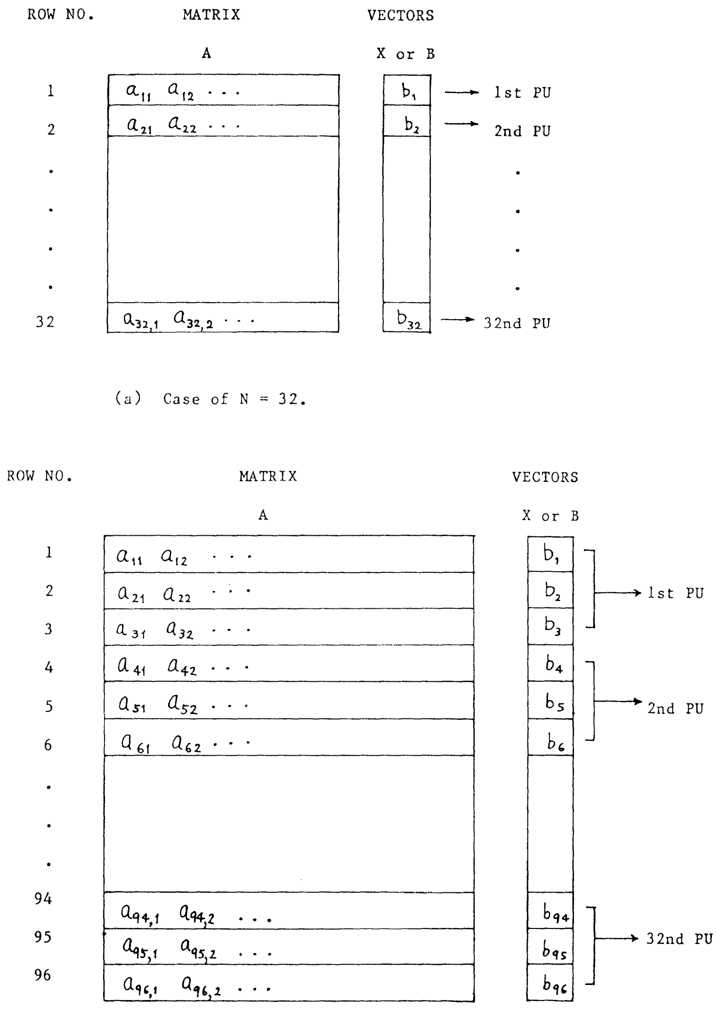


Fig. 4. Time Chart of Typical Iterative Scheme. Symbols Tcj, Tj, and Twj indicate the time intervals for communication, net calculation, and wait, respectively. Synchronization of all PUs is taken at SYNC.



(a) Case of N = 32.

(b) Case of N = 96.

Fig. 5. Partitioning of Matrix and Vectors in Gauss-Jordan Scheme.

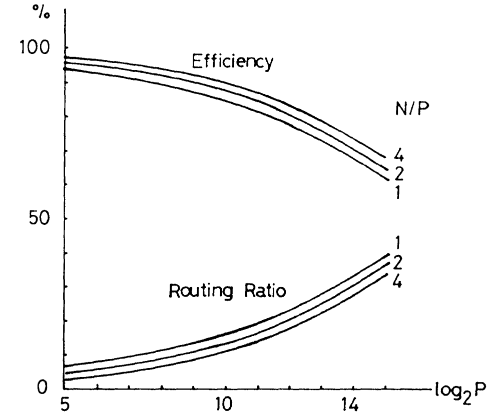


Fig. 8. Efficiency and Routing Ratio in Parallel FFT Algorithm.

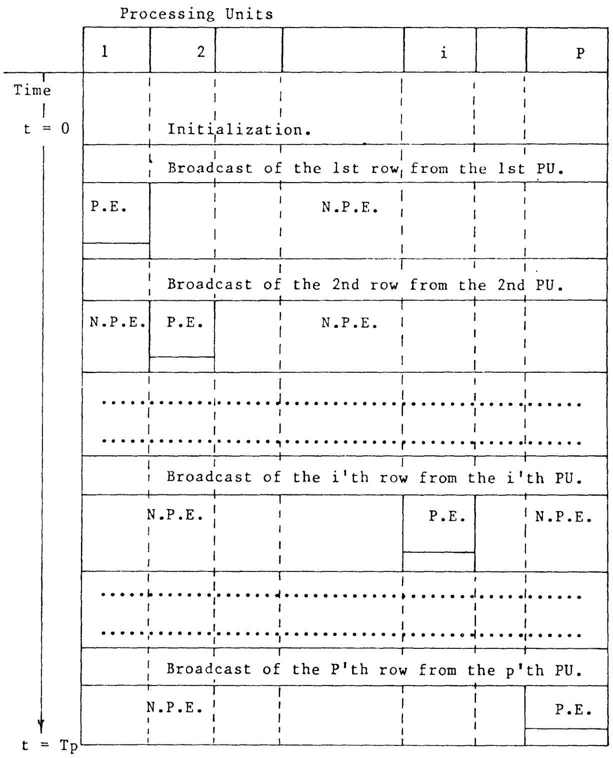

Processing Units
1
2
i
P
Time
t = 0
Initialization.
Broadcast of the 1st row from the 1st PU.
P.E.
N.P.E.
Broadcast of the 2nd row from the 2nd PU.
N.P.E.
P.E.
N.P.E.
Broadcast of the i'th row from the i'th PU.
N.P.E.
P.E.
N.P.E.
Broadcast of the P'th row from the p'th PU.
N.P.E.
P.E.
t = Tp

Fig. 6. PU-Time Diagram in Parallel Gauss-Jordan Scheme for the case of Matrix size = Number of PUs = P. Symbol P.E. stands for the pivot-row elimination and N.P.E for the non-pivot-row elimination.

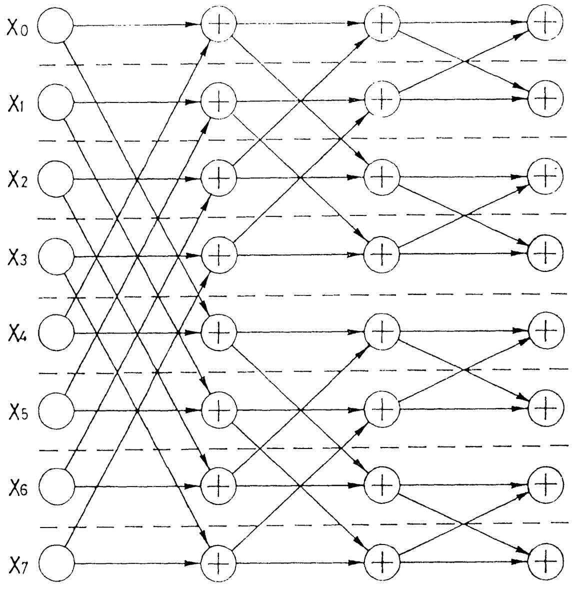


Fig. 7. Data Flow Diagram in Parallel FFT Algorithm.

PARTITIONING JOB STRUCTURES FOR SW-BANYAN NETWORKS

Doug DeGroot
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

Abstract

Large multiprocessor systems interconnected by multistage SW-banyan networks may suffer from communication blockage if resources are not adequately allocated to processes within jobs. This communication blockage can quickly lead to performance degradation. Given a set of job structures, the problem is to map these structures onto the network in such a way that the amount of communication blockage induced by the mappings is held to a minimum. In general, this problem is very hard, bearing resemblance to the graph isomorphism problem. It has been solved for certain structure types. This paper describes one general purpose mapping method that is suitable for structures that can be partitioned in a particular manner. The structures are mapped onto regular SW-banyans in such a way that no communication blockage can occur.

INTRODUCTION

In highly parallel multiprocessor MIMD systems interconnected by large, complex multistage interconnection networks which possess the blocking characteristic, assignment of resources to processes becomes a problem of primary significance. It is important, whenever possible, to assign the resources in such a way that the amount of communication interference is held to a minimum, both on a global scale (among all jobs) and on a local scale (among the processes within a job). The problem is an extremely complex one, one which includes a great number of orthogonal variables. These variables include job sizes and structures, job mix, system organization, network topology, and whether resource allocation is to be made on a dynamic or static basis, to name just a few.

So that the systems being considered can be allowed to be scaled up to larger and larger sizes, it is necessary that resource allocation schemes for such systems exhibit satisfactory performance, preferably linear in the number of resources and job size, and certainly not much worse than logarithmic. It is also extremely important for resource allocation schemes to exhibit as much flexibility as reasonably possible. The number of different job structures that will be presented to a system is usually potentially very large. It is impractical to develop and maintain a large collection of specialized resource schedulers, each of which is suited to only one or some small number of job structures. Instead, a small number of general purpose resource schedulers with great flexibility is desired.

This paper discusses a resource scheduling scheme for use in such a scheduler. As previously mentioned, any resource scheduler must incorporate specific values of many variables. The resource scheduling scheme described here is useful for mapping process-structured computations onto multistage regular SW-banyan networks, both rectangular and non-rectangular. All processors are located on one side of the network. Communication paths pass from one processor all the way to the other side and then back through the network to the desired other processor. All communication paths are bidirectional. The nodes on the network side opposite that of the processors can be connected to memory modules, as in TRAC [Sejnowski] or the Ultra computer [Gottlieb], or they can simply be "bounce back points." It is intended that they will be memory modules, capable of buffering communication. Communication can be either by packet switching or circuit switching - the resource allocation is the same. (However, as will be explained below, additional enhancements are possible in packet switching systems.) The jobs are assigned all resources at once but jobs may be scheduled dynamically. The structure of a job is arbitrary but must be strongly partitionable. This property is defined below. Applications of the resource scheduler to structures which are not strongly partitionable are discussed at the end of this paper.

DESCRIPTIONS OF THE VARIABLE SPACE

This section defines the variable values for which the described resource allocation scheme is suited. Following this section, the allocation scheme itself is presented.

SW-banyans

SW-banyans are formally defined in [Goke]. Figure 1 illustrates an SW-banyan. In this figure, the nodes of the banyan are arranged in a number of distinct levels, with level 0 being at the top, level 1 at the level below level 0, and so on. The last level is denoted "level L." Nodes in adjacent levels are connected together through a set of banyan crossbars. Within each level x, the fanout of a node is denoted f(x); the spread of a node is denoted s(x). The fanout of a node is the number of edges exiting below the node. The spread of a node is the number of edges exiting upwards. Nodes with fanout of zero are called bases; nodes with spread of zero are called apexes. There is exactly one path between any base and any apex; this is the "unique path" property of banyans. The nodes in each level are numbered from left to right, from 1 to n(x), where n(x) is the number of nodes in the level. SW-banyans are three dimensional structures that can be represented in many ways in two dimen-

0190-3918/83/0000/0106$01.00 © 1983 IEEE

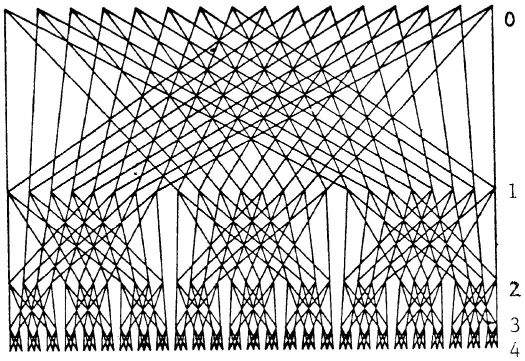


Figure 1

sions. Throughout this paper, all SW-banyans will be represented in a base-distance decomposition manner [DeGroot]. In such a decomposition, each level x of the banyan can be seen to contain f(x) sub-banyans (component banyans) below that level. Thus in Figure 1, f(0), the fanout of the nodes at level 0 (the apex nodes) is 3; and at level 1 there are three component SW-banyans. Although the class of banyans includes busses, trees, and crossbars, we are interested here only in multistage SW-banyans (i.e., SW-banyans with $L > 1$) that have $f(x) \geq 2$ and $s(x) \geq 2$ for all levels x. This class includes such well known networks as the Omega [Lawrie], binary n-cube [Pease], Flip [Batcher], Baseline [Wu], and Generalized Cube [Siegel]. SW-banyans are called regular if for every level x, f(x) is the same for all nodes within level x, and so is s(x). If n(x) is the same for all levels x within an SW-banyan, the banyan is called rectangular. If f(x) and n(x) are the same for all levels x, then the banyan is called strongly rectangular; if all values for n(x) are the same but the values for f(x) differ between levels, the banyan is weakly rectangular. The resource scheduling scheme presented here is applicable to regular SW-banyans, and to either rectangular or non-rectangular ones. The scheduling scheme is first described using only rectangular SW-banyans. It is shown later how non-rectangular SW-banyans may also be used.

Process-Structured Computations

All computations are assumed to be process-structured computations. Such computations consist of a number of disjoint processes (or tasks), each with its own memory and processing element. The processes do not share memory, but they may communicate over logical communication channels [Kahn]. It is possible that these channels may be based in memory, as is the case, for example, in TRAC [Sejnowski]; but this is not necessary. In TRAC, the memory serves to buffer communication. A graphical representation for a process-structured computation would appear simply as a set of nodes with each node representing one process. If two processes are to communicate with each other, then there will be an edge connecting their two respective graph nodes. The number of processes must be determinable at compile time, as must the number and distribution of logical communication channels [Kahn, Hoare].

Dynamic Job Mixes

Jobs are assumed to be able to arrive and complete at will, thus the job mix is a dynamic one. However, the mapping scheme presented here presently works best when it has its own partition of the banyan in which to work. If other jobs are already resident in the system, as would be true in any dynamic system, and a separate sub-banyan can not be found, the success of the mapping method presented would most likely suffer. The operating system must then clearly be relied upon to treat sub-banyans as a schedulable resource. Compensating for a partially busy network is considered at the end of this paper.

System Organization

The systems considered consist of a large number of processors (say 100 to 1000 or more), all of which are located at one side, the apex side for example. Nodes on the base side may be connected to memory buffers or may be "bounce back" points. The interconnection network is an SW-banyan, and therefore the network is characterized by the "unique path" property. For processor A to communicate with processor B, A originates a message which travels down the network from the apex to which A is connected to some base node and then back upward through the network to the node to which processor B is connected. Any base node can be used for this communication. Consequently, there are many different possible communication paths between processors A and B, even though the network is a "unique path" network. However, once a base is selected, there is one and only one path from it to each of the processors A and B. The scheduling scheme presented relies heavily on both of these facts.

SW-banyans, like most multistage interconnection networks, are blocking networks. Thus, given a set of communication paths to be established, it may not be possible for all communication paths to be set up in the network without having two or more of the communication paths interfere with each other. In circuit switching paths, this type of blockage may in fact prevent the needed set of communication paths from being set up. Packet switching communication allows interfering communication paths to be set up, but as communication commences, packets from two or more communications may interfere with each other, thereby slowing down the rate of computation of the processes that interfere with each other. How much interference is encountered depends on run-time performance aspects.

To avoid this run time communication delay, it is desirable to allocate communication paths in such a way that they will never interfere with each other. By so doing, process computation speeds can be kept at a maximum. Figure 2 shows the effect of communication blockage in an SW-banyan. In Figure 2.a, the communication path between processors A and B shares links in the

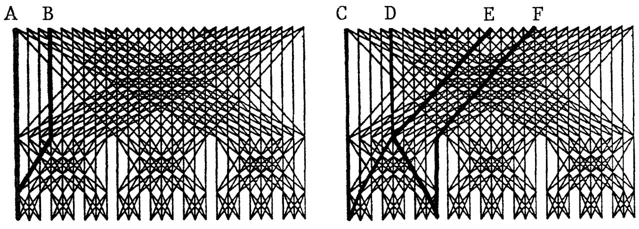


Figure 2

network with itself (the downward path shares links with the upward path), but because these two processors are communicating with each other and are aware of each other's communication activity over this path, they do not interfere with each other. However, processors D and E can interfere with each other; if D wants to talk to C and E wants to talk to F simultaneously, their communications will interfere with each other. The amount of delay experienced by the processors will vary tremendously with the application and timings of individual processes. This is an example of communication blockage. It is this blockage which the mapping method presented herein prevents.

THE PARTITIONING METHOD

The partitioning method is presented in this section. Given a computation structure S, a strong bipartitioning of S into two substructures S1 and S2 is found such that no two edges in the cut set are connected to the same node. This strong partitioning is more restrictive than simple partitioning. The substructures S1 and S2 can then be mapped onto SW-banyans which have $f(x) \ge 3$ for $0 \le x \le L-1$. Such banyans recursively have at least three component SW-banyans at every node level. To map S onto such a banyan, S1 is assigned to one of the three component SW-banyans at level 1, S2 is assigned to another, and a third is used to form the interconnections. Figure 3 illustrates this process. Note that nodes at level 1 are used to form the interconnections between the processors assigned to S1 and S2.

Once the first strong bipartitioning has been performed and the substructures have been assigned to separate level 1 component banyans, the mappings of S1 and S2 are recursively solved using these component banyans. This process continues until the substructures are no longer decomposable, that is, when each substructure consists of only a single processor. Nodes at lower and lower levels will be used to interconnect successive substructure decompositions. It is apparent that this mapping method requires as many levels of nodes as the number of strong partitionings plus one. This mapping method will become clear as a few examples of its application are presented. Following these examples, several ways of extending the method are presented.

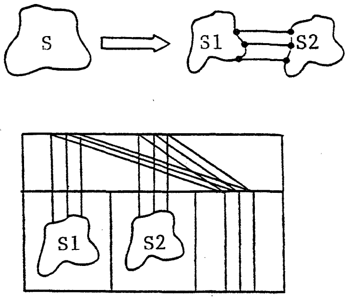


Figure 3

Partitioning the Cube

One simple problem structure is the cube, consisting of eight nodes (processes) and 12 edges (communication channels). The goal of the resource scheduling scheme is to assign one processor (apex) to each process node and one base and communication path to each communication channel. To begin with, the cube can easily be strongly partitioned into two disjoint squares. Each square can then be further partitioned into two lines, and each line can be partitioned into two nodes (processors). See Figure 4. None of these partitionings violate the strong partitioning interconnection constraints, that is, no two edges in any cut set connect to the same node. Using the strong partitioning mapping method, the two squares will be interconnected through nodes at level 1 since they are the substructure results of the first partitioning; the lines of the square will be interconnected through nodes at level 2; and the nodes in the line segments will themselves be interconnected through nodes at level 3. Clearly then, a banyan with at least four node levels (0 through 3) is required for mapping the cube using this method. Since every level x must have $f(x) \ge 3$, $0 \le x \le L-1$, and since four levels of nodes are required, mapping the cube using this method requires a banyan with at least $f^3 = 3^3 = 27$ bases. It is shown later how this number can be lessened.

Following the strong partitioning of a structure, processors and base nodes must be assigned to the decomposed structure. This process is now explained below. Given a base distance decomposition of an SW-banyan [DeGroot], the nodes within a given level x are numbered from left to right, from 1 through n(x). The numbers of the apex nodes of crossbars between levels L-1 and L differ by 1. Crossbars between levels L-2 and L-1 have their apex nodes differing by s(L). Crossbars between levels L-3 and L-2 have apexes

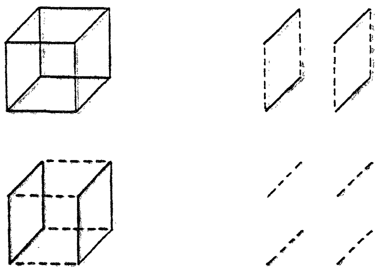


Figure 4

differing by s(L-1)s(L), and so on. In general, crossbars between levels x and x+1 have apex numbers differing by

$$d(x) = \begin{cases} 1, & \text{for } x = L-1 \\ s(x+2)s(x+3)\ldots s(L), & \text{for } 0 \le x \le L-2 \end{cases}$$

Figure 5 illustrates this property. This SW-banyan will be used to demonstrate the mapping. In the cube partitioning, the line segments are formed by connecting together two processors through a node at level 3, and thus crossbars between levels 2 and 3 are used to form the mappings of the nodes in these line segments. In the (3,3) SW-banyan of Figure 5, these crossbars have apex nodes differing by one. The line segments are interconnected into squares through nodes one level up, at level 2, requiring crossbar interconnections between levels 1 and 2. The interconnected processors must thus have numbers differing by d(1) = s(L) = 3. Similarly, the level 1 interconnections of the squares require that the interconnected processors differ by d(0) = s(L-1)s(L) = 9.

The beauty of this mapping method is that only two arbitrary choices are required. For the cube, one line segment is selected and the nodes (processors) connected by it are numbered so that they differ by only one. Clearly many such numberings are possible. Without loss of generality, processors 1 and 2 can be selected. Assignments to the processors in the other line segment in the corresponding square are made simply by assigning them numbers that differ from their nodes to which they are being connected by d(1) = 3. See Figure 6. Node 1 in the first line segment connects to node 1 + 3 = 4; node 2 connects to node 2 + 3 = 5. Note that the new line segment processor assignments, processors 4 and 5, differ from each other by only one as required. Node assignments can now be made in the other square. To do so, nodes are simply chosen which differ by d(0) = 9 from their connecting nodes. The new node assignments clearly maintain the required node relationships, as shown in Figure 7. Once the nodes have all been labeled, the cube can be loaded onto the banyan. Figure 8 shows one possible loading using the assigned processors. It is apparent from the figure that even with the given node labeling many loadings are possible simply by choosing different bases. In the figure, each base selected is only one of three possible.

To see how the interconnections between two substructures are achieved, note that processors 1 and 4 are interconnected at level 2 through node 7, implying the use of base node 7 as well. Node 7 differs from 4 by d(1) = 3, just as does 4 from 1. Similarly, nodes 2 and 5, which differ by 3, are interconnected through node 5 + 3 = 8 at level 2 and at the base. Nodes 1 and 10, which differ by 9, can be interconnected using node 10 + 9 = 19, and so forth. The fact that the third node is always available for interconnecting the other two nodes is due to the constraint that f(x) ≥ 3, 0 ≤ x ≤ L-1. This ensures that there will be at least three base nodes in every crossbar within the banyan. Each of the three nodes will naturally reside within three different component banyans. By having assigned node numbers in the above manner, the availability of the third node for interconnecting the other two is ensured.

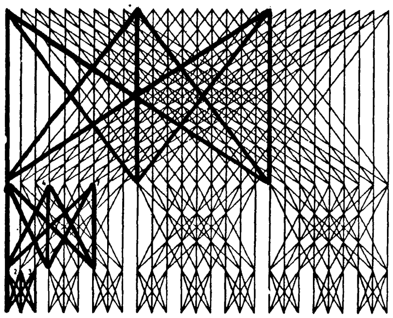


Figure 5

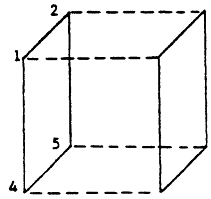


Figure 6

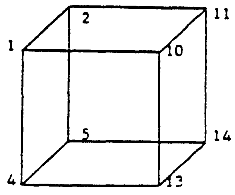


Figure 7

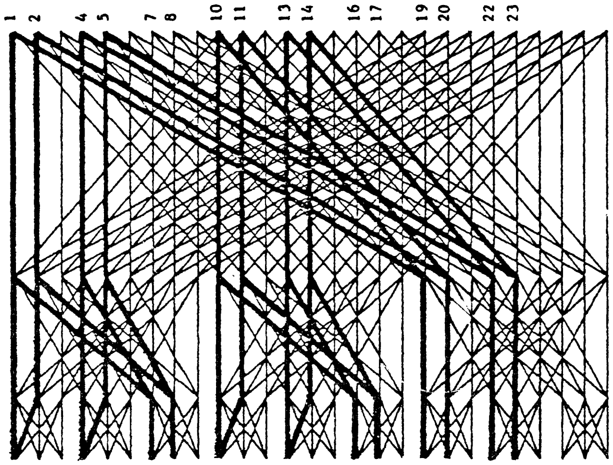


Figure 8

A most important point to notice about the loaded structure is that given any communication path, if the two processors on this path are communicating, there can be no other two communicating processors that interfere with the first two. This is true whether the communication is accomplished with packet switching or circuit switching. An important point to notice about the mapping method itself is that it takes time linear with the number of processors and communication channels to be mapped once the partitionings have been determined. Determining the partitionings may be much more complicated, and in fact, it is unclear at present how effectively this could be automated.

4-Nearest Neighbors

Another common problem structure is the 4-nearest neighbors structure. In this structure, the nodes are organized in rows and columns, as elements of a matrix. Except for the outside elements, node (i,j) is connected to nodes (i+1,j), (i-1,j), (i,j+1), and (i,j-1). Strong partitioning can be performed on this structure just as well as it was on the cube. Figure 9 shows a set of suitable strong partitionings of the 4x4 4-nearest neighbors structure. Because 4 levels of partitioning are required, 5 levels of nodes are required. And again, because fanouts of 3 or more are required, at least 81 bases are required with this method. Node assignments can be made as before, with the restriction that the fourth level partitionings must have nodes that differ by 1, third level nodes must have nodes differing by s(L), and so on. In Figure 10, the needed differences are labeled along the connecting edges. Figure 11 shows one possible node labeling using the required node differences. Finally, Figure 12 shows this structure loaded onto the (3,3) SW-banyan. Again, it is important to notice that no two active communication paths can possibly interfere with each other.

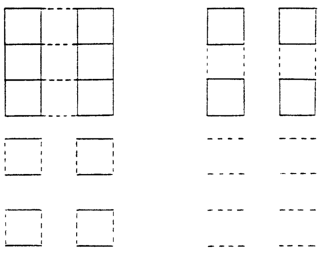


Figure 9

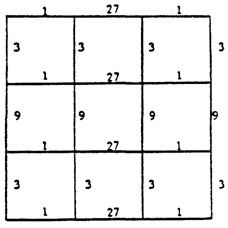


Figure 10

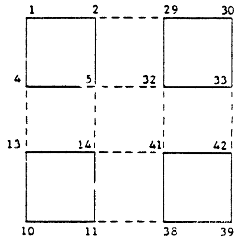


Figure 11

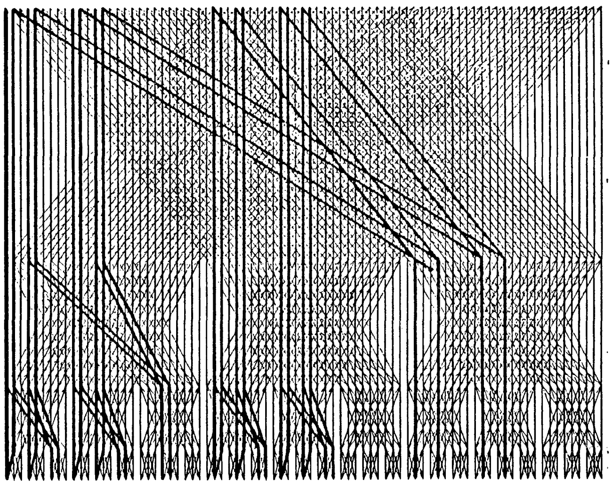


Figure 12

EXTENDING THE PARTITIONING METHOD

This section describes several ways in which the partitioning method can be extended to make it more powerful.

Higher Level Partitionings

So far, example applications have recursively strongly bipartitioned a structure into two substructures at each step. These two substructures were assigned to two separate component SW-banyans and a third was used to interconnect the two. This process required at least three component SW-banyans at each step, or equivalently, f(x) ≥ 3 for 0 ≤ x ≤ L-1. When f(x) > 3 for some node level, it might be possible to strongly partition a given structure into more than two substructures. If a structure is strongly partitioned into n substructures and mapped onto separate level x+1 component SW-banyans, n-1 other component SW-banyans at the same level can be used to form the necessary interconnections between the substructures. The total number of required component SW-banyans at level x+1 is then 2n - 1. Thus the fanouts of the nodes at level x must be 2n - 1 or more. Note that 2n - 1 is always odd. Equivalently, if f(x) = n for some level x, 0 ≤ x ≤ L-1, a given structure can be strongly partitioned into (n+1)/2 substructures, if possible. Figure 13 illustrates the process of strongly partitioning a structure into more than two substructures. As an example of a structure that may be strongly partitioned into more than two partitions, consider the 6x6 4 nearest neighbors structure. A set of suitable strong bipartitionings and tripartitionings of this structure is illustrated in Figure 14.

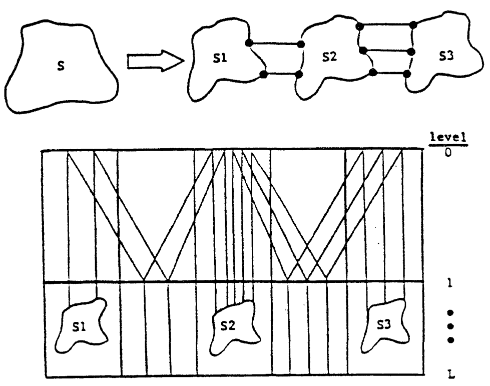


Figure 13

In some cases, n substructures do not require n - 1 component SW-banyans for interconnection, thus allowing fanouts of less than 2n - 1. For instance, three substructures might easily be mapped onto three component SW-banyans and interconnected through only a fourth. As an example, consider the structure shown in Figure 15 and the accompanying strong partitioning and node labelings. This structure is clearly the 6-processor pipeline. Both the strong partitionings and the node labelings obey the constraints described above. Figure 16 shows this structure loaded onto the strongly rectangular (4,2) SW-banyan. Note that at level 1 the structure is strongly tripartitioned but that only four component SW-banyans are used to map the three substructures and interconnect them, instead of 2x3 - 1 = 5. The mapping method is not formally described here for use with fewer than 2n - 1 component SW-banyans, but it is apparent that it is sometimes possible.

Sometimes a substructure is simple enough or small enough to be mapped by other methods than the partitioning method and may consequently require component SW-banyans with fewer levels than might be required by the partitioning method. One trivial instance involves substructures with n or fewer processors interconnected by n or fewer communication paths. Such substructures can easily be mapped onto any crossbar of size nxn or more. For example, the cube can be strongly bipartitioned into two squares. Each square contains 4 processors and 4 communication paths. Clearly these squares (or rings) can be mapped onto 4x4 crossbars. Since only one strong partitioning is required to reduce the cube to substructures that can be mapped onto crossbars, only three levels of nodes are required. Figure 17 shows the cube loaded onto such a banyan.

Interior Bounce-Back Points

It is possible to design communication networks in which all communication turns around in the middle of the network where the downward paths meet the upward paths instead of traveling all the way through the network to a base before turning around. In such cases, it may be possible to omit the allocation of certain base and intermediate level nodes. Since it is the availability of nodes and links that determines in large part the success of a potential mapping in a busy system, the more nodes and links that are available, the better the chance of success for the mapping. In addition, a larger number of structures will be able to coreside on the network at a given time. The mapping method presented is presently unable to take advantage of idle nodes that have busy nodes above them, and thus is presently incapable of satisfactorily dealing with such networks except as regular networks.

Packet Switching Enhancements

When packet switching is used as the form of communication, greater flexibility is allowed in the mappings. If it is impossible to map a given struc-

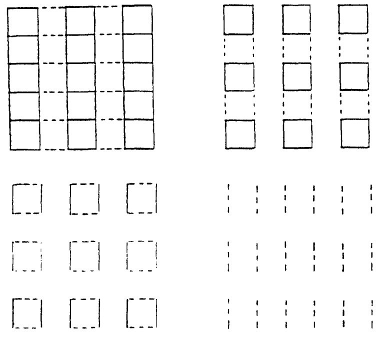


Figure 14

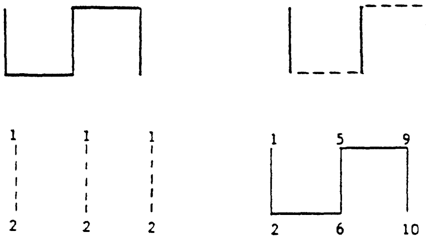


Figure 15

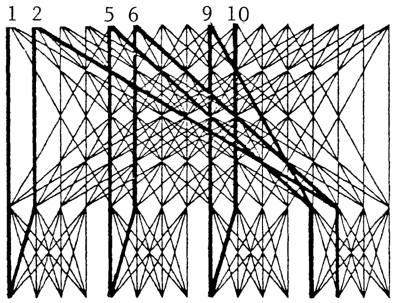


Figure 16

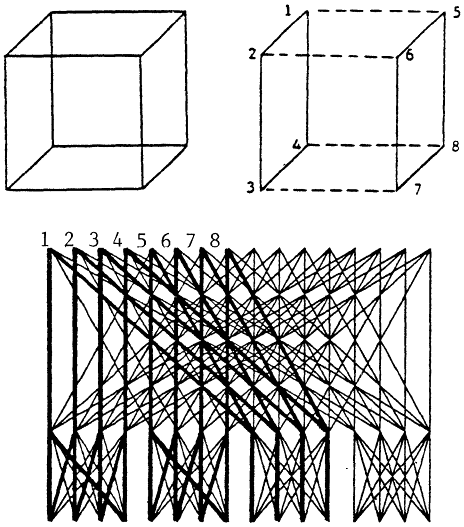


Figure 17

ture, say because no suitable partitionings may be found, then communication channels may be dropped from the computation structure graph one by one until partitionings are possible. This reduced graph may then be mapped onto the network, and those communication channels which were dropped may be arbitrarily assigned. These arbitrarily assigned channels may result in interference with the regularly mapped communication channels, but the number of interfering channels will hopefully be minimized by the partitioning mapping method.

Loading Onto Busy Networks

If the system already has some jobs loaded onto it, a given node labeling for a partitioned structure may not be loadable onto the network due to the indicated nodes, links, or bases being in use. In such cases, adding 1 modulo the number of apexes to every labeled node produces another labeling that still satisfies the needed labeling constraints. Each such labeling can be tried until all have been tried but failed or until one succeeds. In addition, it is possible to try for interconnections of substructures at higher levels. For example, when a cube is decomposed into two squares, these squares can be interconnected through many different levels. Although doing so is easy, how this is done has not been explained in this presentation.

Mapping Onto Non-rectangular SW-banyans

As mentioned at the beginning of this paper, the partitioning mapping method is suitable for both rectangular and non-rectangular SW-banyans. So far, all the examples have dealt solely with rectangular SW-banyans. Figure 18 however shows a cube mapped onto an 8x27 non-rectangular SW-banyan. Given the particular orientation of the banyan in the figure, it is easy to see how certain processors in rectangular SW-banyans are unnecessary for the partitioning method. For instance, the rightmost component SW-banyan at level 1 is used solely to interconnect processors which are above the leftmost and middle level 1 component SW-banyans. Therefore, the processors above this third (rightmost) component are not needed, and indeed, in most cases, will probably be rendered useless by the mapping. The same situation exists recursively within each component banyan. As a consequence, it appears that fanouts of 3 and spreads of 2 are best suited to this mapping scheme. A network with this topology was serendipitously selected as the TRAC network early in the project.

Non Strongly-Partitionable Structures

Recent studies have shown that the partitioning method is extendable to structures that are not strongly partitionable. Thus the number of structures that may be mappable with the partitioning method is larger than first believed. Further research is needed in order to determine the limitations of the partitioning mapping method for non-strongly partitionable structures.

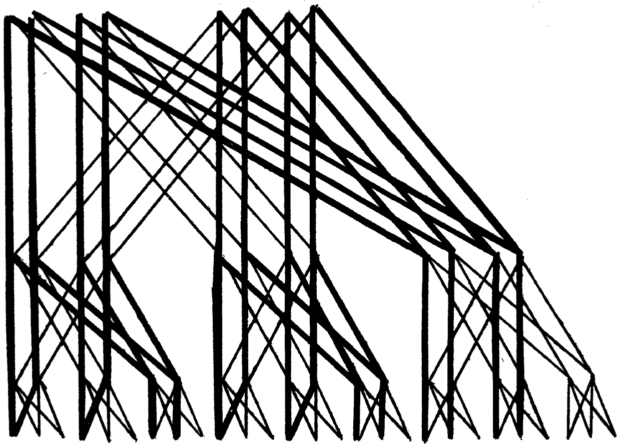


Figure 18

OTHER PARTITIONABLE STRUCTURES

As noted near the beginning of this paper, the partitioning method as presented is applicable to only a very restricted class of computation structures. These structures must be able to be partitioned into a number of disjoint substructures such that no two edges in any cut set are connected to the same node. Fortunately, there are many useful structures in this class. Figure 19 illustrates just a few of the many such structures. Figure 20 shows several structures that do not fall into this class. These structures will be mappable with an extended partitioning method.

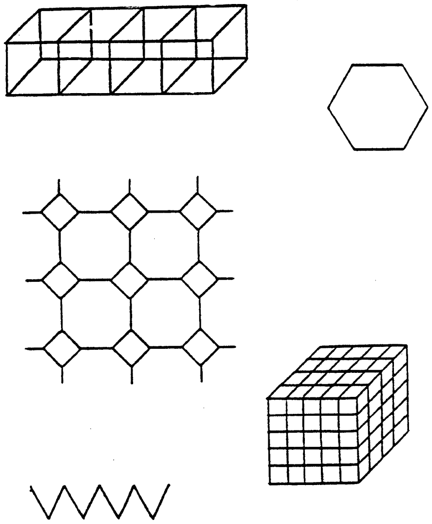


Figure 19

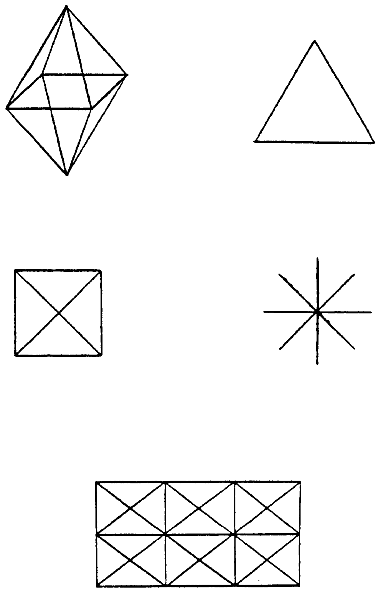


Figure 20

SUMMARY

A resource allocation scheme has been presented for mapping certain classes of job structures onto regular SW-banyan networks, both rectangular and non-rectangular. This method works best when the job structures to be mapped are strongly partitionable. For a given partitioning of a structure, an assignment of processors and base nodes can be made in linear time. Many possible resource assignments are possible for a given partitioning, allowing greater probabilities of loading a structure onto a busy system. Several ways in which the basic partitioning method can be extended have been discussed.

Acknowledgements

Thanks to Dr. James C. Browne and Dr. Miroslaw Malek for their support in this work.

Bibliography

[Batcher] "The Flip Network in Staran," Kenneth E. Batcher, Proc. 1976 International Conference on Parallel Processing, Aug. 1976, pp. 65-71.

[DeGroot] Mapping Computation Structures Onto SW-banyan Networks, Doug DeGroot, Doctoral Dissertation, Department of Computer Sciences, The University of Texas, Austin, 1981.

[Goke] Banyan Networks for Partitioning Multiprocessor Systems, Rodney L. Goke, Doctoral Dissertation, Univ. of Florida, 1976.

[Gottlieb] "Networks and Algorithms for Very-Large-Scale Parallel Computation," Allan Gottlieb and Jack Schwartz, Computer, Vol. 15, No. 1, Jan. 1982, pp. 27-36.

[Hoare] "A Calculus of Total Correctness for Communicating Processes," C.A.R. Hoare, Science of Computer Programming, Vol. 1, No. 1, pp. 49-72.

[Kahn] "The Semantics of a Simple Language for Parallel Programming," Giles Kahn, Inf. Proc. '74, pp. 471-475, IFIP, 1974.

[Lawrie] "Access and Alignment of Data in an Array Processor," Duncan H. Lawrie, Trans. on Computers, IEEE, Vol. C-24, No. 12, Dec. 1975,

[Pease] "The Indirect Binary n-Cube Microprocessor Array," M.C. Pease, Trans. on Computers, IEEE, Vol. C-26, No. 5, May 1977, pp. 458-473.

[Sejnowski] "An Overview of the Texas Reconfigurable Array Computer," Matt Sejnowski, et al, AFIPS Conf. Proc., Vol. 49, 1980, NCC, pp. 631-641.

[Siegel] "The Multistage Cube: A Versatile Interconnection Network," H.J. Siegel and R.J. McMillen, Computer, Vol. 14, Dec. 1981, pp. 65-76.

[Wu] "On a Class of Multistage Interconnection Networks," C.L. Wu and T.Y. Feng, IEEE Transactions on Computers, Vol. C-29, August 1980, pp. 694-702.

CONFIGURING COMPUTATION TREE TOPOLOGIES
ON A DISTRIBUTED COMPUTING SYSTEM

Woei Lin and Chuan-lin Wu
Department of Electrical Engineering
The University of Texas at Austin
Austin, Texas 78712

Abstract

This paper describes an approach to connecting hardware resources for high-performance computation. Two basic algorithms are designed for configuring binary tree topologies. The configuring command can be issued from any processing mode. The algorithms can select proper modes for connection while maintaining good utilization of processing nodes.

I. Introduction

Recently, due to VLSI technology, the cost of hardware has been drastically decreased. Researchers attempt to add more and more hardware to computing systems. In order to improve computing system, they are using multiple processor instead of single processor to achieve higher performance. And generally these processors are interconnected by a communication network. However, an improper communication structure is liable to incur excessive interprocessor communication that is referred to as saturation effect [1]. And this excessive interprocessor communication would seriously degrade the overall performance of multiple processor systems. Therefore, in designing a multiple processor system, communication structure is a major factor as well as to be considered.

In order to minimize interprocessor communication and improve resource utilization, several different techniques have been employed to configure processors into some topologies, such as linear array, star, loop, cube, binary tree, etc. [2-6]. However, a single topology is only suitable for some specific tasks. And the performance will be improved provided that processors can be dynamically reconfigured into suitable topologies needed in the computation. The main idea behind this work is to design two basic configuration algorithms to connect processors into binary tree topology through a communication subnet - starnet, which is able to support some other configuration topologies [7]. And configuration commands could be issued by any processor. The remainder of this paper is organized as follows. Section II presents both system overview and mathematic models. In Section III the configuration algorithms of binary trees are presented. Section IV contains the conclusions.

II. System Overview and Models

A. Introduction to Star

Star is a collection of N heterogeneous processing mode as well as a communication subnet-starnet through which processing modes are able to communicate with each other. Generally, the number of processing nodes (the size of starnet) is a power of two, $N=2^n$. The starnet is a cascade of a baseline network and a bit reversal permutation P. In star, processing nodes are numbered from 0 to N-1. For convenience, each node is labeled with an n-bit binary number, which is referred to as address.

The switching methodology employed in the starnet is circuit switching, that is, a physical path is actually established between processing nodes.

To a specific destination node, all source nodes issue the same routing tags no matter where they are. And the routing tag is exactly the address of destination. More precisely, let source address $S = S_{n-1}\ S_{n-2} \ldots S_{\emptyset}$ and destination address $D = d_{n-1}\ d_{n-2} \ldots d_{\emptyset}$ respectively. At stage i, the requested switching element can be described by

$$(d_{n-1} \ldots d_{n-i}\ S_0 \ldots s_{n-i-2})_i,$$

where $0 \leq i \leq n-1$. And the requested link can be described by

$$(d_{n-1} \ldots d_{n-i}\ s_0 \ldots s_{n-i-1})_i,$$

where $0 \leq i \leq n$.

B. Description of Functions

In this subsection, we examine the problem of conflict-free mapping on the Star. Before the formal description of conflict-free mapping is presented, two basic functions ϕ and ψ are introduced, which facilitates the detection of conflicts among connections of starnet.

DEFINITION. Let $U = U'M + u, V = V'M + v$ where U, V, U', V', M' u, v, are n-bit binary numbers and $u, v < M$, $M = 2^m$, $0 \leq m \leq n$. A function ϕ is defined as

$$\phi(U, V) = \max\ [m],$$

where m makes U and V have $u = v$.

DEFINITION. Let U, V be n-bit binary numbers, function ψ is defined as

$$\psi(U, V) = \phi(P(U), P(V)).$$

For example, if $U = \emptyset 11 \emptyset 11 \emptyset$ and $V = \emptyset 1 \emptyset \emptyset 1 \emptyset$ then $\phi(U, V) = 3$ and $\psi(U,V) = 2$. Two extremes of ϕ are (1) $m = m$, it leads to $U = V$ (2) $m = \emptyset$, one of them is an even number and the other is an odd number. Functions and ψ also have the following properties. Note that the following bit manipulation is based on module-n.

LEMMA 1. If $\phi(U, V) = k$, then for any integer C, $\phi(U + C) = k$.

LEMMA 2. Let $U = U'M + u$ and $V = V'M + v$. If $V > U$, for any integer number $M = 2^m$, then $V' \geq U'$. Morevoer, if $V > U$ and $u \geq v$, then $V' > U'$.

LEMMA 3. If U, V are n-bit binary numbers and $\phi(U, V) = k$, then $\psi(U, V) < n - k$.

0190-3918/83/0000/0114$01.00 © 1983 IEEE

And if ψ (U, V) = k', then
$\phi(U, V) < n-k'$.

LEMMA 4. If $\phi(U,V) = k < n$, then
$\phi(2U, 2V) = k + 1$.

LEMMA 5. If $\phi(U, V) = k$ and $0 < k < n$, then
$\psi(U+1, V) < n-k$.

For convenience, let an ordered pair (S,D) represent a connection between two nodes S and D, where S is source node and D is destination node. In starnet, a conflict of a 2x2 switching element is defined as a situation in which two connections from different input ports and with different routing tags compete the same output port. Therefore, if a set of connections $(S_1,D_1),\ldots(S_k,D_k)$ do not cause any conflict, then they are mappable on star.

THEOREM 1. In star, a set of connections (S_1,D_1), $\ldots(S_k,D_k)$ are mappable if and only if for any two connections (S_i,D_i) and $S_j,D_j)$ where $S_i \neq S_j$ and $D_i \neq D_j$, the following inequality is satisfied

$$\phi(S_i, S_j) + \psi(D_i, D_j) < n.$$

III. Binary Tree

The following algorithm is used to configure a number of processors into an n-level full binary tree which could be rooted from any arbitray node R. And the left son of a predecessor S_i is denoted as $D_{i,1}$ and right son is denoted as $D_{i,2}$.

ALGORITHM 1. (A1)

```
Procedure top-down tree (R, n);
begin
        for i:  = 1 to n-1 do
        begin
        k:  = $2^{i-1}$;
        for j:  k to 2k - 1 do
        begin
        $S_j$:  = j + R - 1;
        for j:  = k to 2k - 1 do
        begin
           $S_j$:  = j + R - 1;
           $D_{j,1}$:  = 2j + R - 1;
           $D_{j,2}$:  = 2j + R;
        end
        end;
     end; (* end of A3 *)
```

An example of a binary tree, generated by A, is shown in Figure 1(a). The following theorem demonstrates the mappability of A1 on Star.

THEOREM 2. Any full binary tree which is generated by A1 is mappable on Star.
(For a proof, see Appendix).
The mapping of the previous example is shown in Figure 1(b). An alternative algorithm which generates an n-level binary tree in a bottom-up fashion is described as follows.

ALGORITHM 2 (A2).

```
Procedure bottom-up tree (R,n);
begin
   for i :  = 1 to n-1 do
   begin
      k :  $2^{i-1}$;
      for j :  = k to 2k - 1 do
      begin
        $S_j$ :  = -j + R - 1;
        $D_{j,1}$ :  = -2j + R - 1;
        $D_{j,2}$ :  = -2j + r;
      end;
   end; (*end of A4 *)
```

An example of binary trees which is generated by A2 is shown in Figure 2(a).

THEOREM 3. Any binary tree which is generated by A2, is mappable on Star.
(Proof is similar to Theorem 1). The mapping of the previous example is illustrated in Figure 2(b).

IV. Conclusions

The main idea behind this work is to design two configurabion algorithms for n-level binary tree. First of all, the sufficient and necessary condition of mappability is found. Then, by means of the detection function, the mappability of A1 and A2 are demonstrated. The result is flexible enough to allow any mode to be the root without centralized control. Hence, A1 and A2 provide a solution to configuring processors into tree topology in a distrubted fashion.

References

1 W. Chu, D. Lee and B. Ittla, "A Distributed Processing System for Naval Data Communication Networks", AFIPS Conf. Proc., Vol. 47, NCC 1978, pp. 783-793.

2 J. Deminet, "Experience with multiprocessor algorithms," IEEE TRans. on Computers, Vol. C-13, No. 4, Apr. 1982, pp. 288-295.

3 L. Goke and G. Lipovski, "Banyan networks for partitioning multiprocessor systems," Proc. 1st Annual Symp. on Computer Arch., Dec. 1973, pp. 21-28.

4 Y. Paker and M. Bozyigit, "Variable topology multicomputer," 2nd Symp. on Micro Arch., 1976, pp. 141-151.

5 T. Feng, "A survey of interconnection networks," Computer, Dec. 1981, pp. 12-27.

6 M. Liu, "Distributed loop computer networks," Advances in Computers, Vol. 17, Academic Press, New York, 1978, pp. 163-221.

7 C. Wu, T. Feng, and M. Lin, "Star - A Local network system for real-time management of imagery data", IEEE Transactions on Computers, Oct. 1982, pp. 923-933.

Appendix

Proof of Theorem 1:

Let (U, W) and (V, X) be two connections of a binary tree generated by A1. In the following discussion, only the condition that $U \neq V$ is considered.

(i) W, Z are left sons of U, V.

Let

$$U' = U - R + 1$$
$$V' = V - R + 1$$

and

$$\phi(U', V') = k.$$

From A1,

$$W = 2U' + R - 1$$
$$Z = 2V' + R - 1.$$

From LEMMA 1,

$\phi(u', V')$

$= \phi(U'+R-1, V'+R-1)$

$= \phi(U, V)$

$= k.$

From LEMMA 4,

$\phi(2U', 2V') = k + 1$

Again,

$= \phi(2U', 2V')$

$= \phi(2U' + R-1, 2V' + R-1)$

$= \phi(W, Z)$

$= k + 1.$

From LEMMA 3,

$\psi(W, Z) < n - k - 1.$

Combine the above two inequalities,

$\phi(U, V) + \psi(W, Z) < n - 1 < n.$

(ii) W, Z are the right sons of U, V.

In a similar way, it can be derived that

$\phi(U, V) + \psi(W, Z) < n.$

(iii) W is the left son of U and Z is the right son of V.

Let

$$U' = U - R + 1$$
$$V' = V - R + 1$$

and

$$\phi(U', V') = k.$$

From A1,

$W = 2U' + R - 1$

$Z = 2V' + R.$

From LEMMA 1,

$\phi(U', V')$

$= \phi(U'+R-1, V'+R-1)$

$= (U, V)$

$= k.$

From LEMMA 4,

$\phi(2U',2V')$

$= \phi(2U'+R-1, 2V'+R-1)$

$= k + 1.$

From LEMMA 5,

$\psi(2U'+R-1, 2V'+R) < n - k.$

Hence,

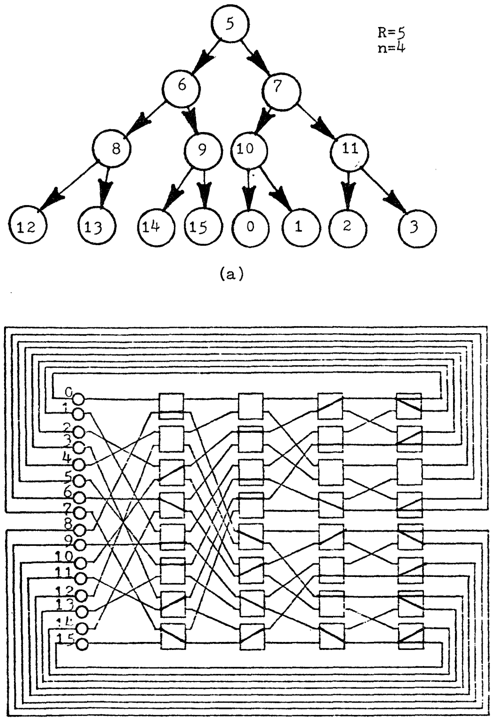
$\psi(W, Z) < n - k.$

Combine the above inequalities,

$\phi(U, V) + \psi(W, Z) < n.$

As a result, any two connections of T are conflict-free in Starnet. Therefore, T is mappable on Star.

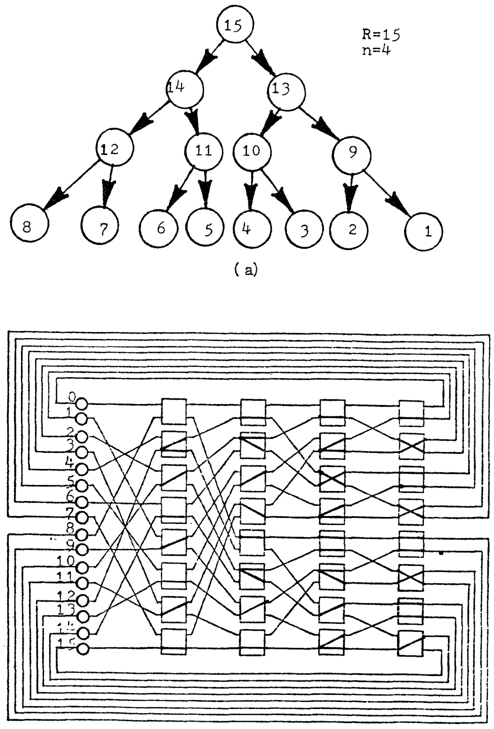
Q.E.D.



(a)

(b)

Figure 1. An example of top-down tree



(a)

(b)

Figure 2. An example of bottom-up tree

Performing the Shuffle with the PM2I and Illiac SIMD Interconnection Networks

Robert R. Seban
Howard Jay Siegel

Purdue University
School of Electrical Engineering
West Lafayette, Indiana 47907

Abstract--Three SIMD single stage interconnection networks which have been proposed and studied in the literature are the Illiac, PM2I, and Shuffle-Exchange. Here the ability of the Illiac and PM2I networks to perform the shuffle interconnection in an SIMD machine with N processors is examined. A lower bound of $3\sqrt{N}/2$ transfers for the Illiac to shuffle data is derived. An algorithm to do this task in $2\sqrt{N}-1$ transfers is given. A lower bound of $\log_2 N$ transfers for the PM2I to shuffle data has been published previously. An algorithm to do this task in $\log_2 N+1$ in transfers is presented here.

1. Introduction

This paper extends SIMD interconnection network studies presented in [28, 31]. In particular, the ability of the PM2I and Illiac single stage interconnection SIMD machine networks to perform the shuffle interconnection is examined. In [28] it is shown that a lower bound on the number of transfers needed for the PM2I network to perform the shuffle is $\log_2 N$, where N is the number of processing elements in the SIMD machine. The algorithm presented here requires only $(\log_2 N)+1$ transfers. This algorithm is used as basis for an algorithm to do the shuffle with the Illiac network in $(2\sqrt{N})-1$ transfers. This compares favorably an earlier result of $4(\sqrt{N}-1)$ in [25]. In addition, a lower bound $3\sqrt{N}/2$ on the number transfers required for Illiac to do shuffle is proved.

The model of SIMD machines used is described in Section 2. In Section 3 the interconnection networks are formally defined. An algorithm to shuffle data using the PM2I network is given in Section 4. The lower bound analysis and algorithm for performing the shuffle with the Illiac network is presented in Section 5.

2. SIMD Machine Model

Typically, an *SIMD (single instruction stream - multiple data stream) machine* [12] is a computer system consisting of a control unit, N processors, N memory modules, and an interconnection network. The control unit broadcasts instructions to the processors, and all active processors execute the same instruction at the same time. Each active processor executes the instruction on data in its own memory module. The interconnection network, sometimes referred to as an alignment or permutation network, provides for communications among the processors and memory modules. Examples of SIMD machines that have been constructed are the Illiac IV [6] and STARAN [2, 3].

One way to view the physical structure of an SIMD machine is as a set of N processing elements interconnected by a network, where each *processing element (PE)* consists of a processor with its own memory. This type of configuration is shown in Fig. 1. It is called the PE-to-PE organization. The network is unidirectional and connects each PE to some subset of the other PEs. A transfer instruction causes data to be moved from each PE to one of the PEs to which the PE is connected by the network. (Here only one-to-one communications will be considered, i.e., broadcasting (one-to-many) connections are not considered.) To move data between two processing elements that are not directly connected, the data must be passed through intermediary processing elements by executing a programmed sequence of data transfers. An alternative to the PE-to-PE SIMD machine organization is to position a bidirectional network between the processors and the memories. The PE-to-PE paradigm will be used here, however, the results presented will be applicable to the other organization also.

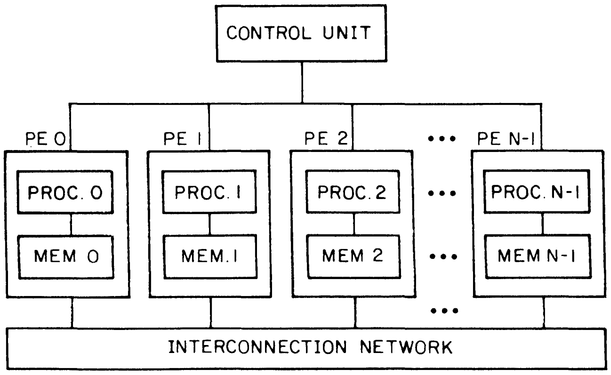


Fig. 1: PE-to-PE SIMD machine configuration, with N PEs.

The formal model of an SIMD machine used here consists of five parts: processing elements, control unit instructions, processing element instructions, masking schemes, and interconnection functions. It is a mathematical model that provides a common basis for evaluating and comparing the various components of different SIMD machines. This model is based on the one presented in [31].

Each *processing element (PE)* is a processor together with its own memory. There are *N* PEs, addressed (numbered) from 0 to N–1, where $N = 2^m$. It is assumed that the processor contains a fast access general purpose register A and a *data transfer register (DTR).* When data transfers among PEs occur, it is the DTR contents of each PE that are transferred. At any point in time, each PE is either in the active or the inactive mode. If a PE is *active,* it executes the instructions broadcast to it by the control unit. If a PE is *inactive,* it will not execute the instructions broadcast to it.

The *control unit* stores the SIMD programs, executes control of flow instructions, and broadcasts pro-

This material is based upon work supported by the National Science Foundation under Grant ECS-8120896.

0190-3918/83/0000/0117$01.00 © 1983 IEEE

cessing element instructions to the PEs. An example of a control of flow instruction is the loop statement "*for* i = 0 *until* N–1 *do...*"

The *processing element instructions* consist of those operations that each processor can perform on data in its individual memory or registers. It is assumed the set of processing element instructions includes the capability to move data among the registers. The notation "Z ← Y" means the contents of register Y are copied into register Z. The notation "Z ⟷ Y" means two registers exchange their contents.

A *masking scheme* is a method for determining which PEs will be active at a given point in time. The *PE address masking* scheme uses an m-position mask to specify which PEs are to be activated, each position of the mask corresponding to a bit position in the binary addresses of the PEs [28]. Each position of the mask will contain either a 0, 1, or X ("don't care"). The only PEs that will be active are those that match the mask for all i, $0 \leq i < m$: if the mask has a 0 in the i-th position, then the PE address must have a 0 in the i-th position; if the mask has a 1 in the i-th position, then the PE address must have a 1 in the i-th position; and if the mask has an X in the i-th position, then the PE address may have either a 0 or 1 in the i-th position. For example, if N = 8 and the mask is 1X0, then only PEs 6 and 4 are active. Superscripts are used as repetition factors, e.g., $X^3 01^2$ is XXX011. Square brackets will be used to denote a mask. Each PE instruction and interconnection function (defined below) will be accompanied by a mask specifying which PEs will execute that command. For example, executing "A ← DTR $[X^{m-1}0]$" means that each even numbered PE is active and loads its A register from its DTR. Each odd numbered PE is inactive and does nothing. Further information about the use and implementation of PE address masks is in [18, 28, 31, 34].

An *interconnection network* can be described by a set of interconnection functions, where each *interconnection function* is a bijection (permutation) on the set of PE addresses [28]. When an interconnection function f is applied, PE i sends the contents of its DTR to the DTR of PE f(i). This occurs for all i simultaneously, for $0 \leq i < N$ and PE i active. Saying that an interconnection function is a bijection means that every PE sends data to exactly one PE, and every PE receives data from exactly one PE (assuming all PEs are active). In this model, it is assumed that an inactive PE can receive data from another PE if an interconnection function is executed, but an inactive PE cannot send data. To pass data from one PE to another PE a programmed sequence of one or more interconnection functions must be executed. This sequence of functions moves the data from one PE's DTR to the other's by a single transfer or by passing the data through intermediary PEs.

In summary, an SIMD machine can be formally represented as the five-tuple *(N,C,I,M,F)*, where:

(1) N is a positive integer, representing the number of PEs in the machine;

(2) C is the set of control unit instructions, i.e., instructions that are executed by the control unit in order to control the flow of the program;

(3) I is the set of processing element instructions, i.e., instructions that can be executed by each active PE and act on data within that PE;

(4) M is the set of masking schemes, where each mask partitions the set {0, 1, ..., N–1} into two disjoint sets, the enabled PEs and the disabled PEs; and

(5) F is the set of interconnection functions (i.e., the interconnection network), where each function is a bijection on the set {0, 1, ..., N–1}, which determines the communication links among the PEs.

A particular SIMD machine architecture can be described by specifying N, C, I, M, and F. In this paper, $N = 2^m$; C includes "for ... until ... do" instructions for controlling the flow of loops in the program; I includes instructions for moving data among the registers of a given PE; M includes PE address masks; and F is varied. The assumptions made about the SIMD machine to be used as the model are intentionally minimal so that the material presented is applicable to a wide range of machines.

3. The Interconnection Networks

A. Introduction

In this paper, three networks which can be constructed from a single stage of switches are examined. In a single stage network, data items may have to be passed through the switches several times before reaching their final destinations. Conceptually, a single stage network can be viewed as N input selectors and N output selectors, as shown in Fig. 2 [30]. The way in which the input selectors are connected to the output selectors determines the allowable interconnections.

The following notation will be used: let $N = 2^m$, let the binary representation of an arbitrary PE address P be $p_{m-1}p_{m-2}\cdots p_1p_0$, let $\overline{p}_i$ be the complement of p_i, and let the integer n be the square root of N. It is assumed that –j mod N = N–j mod N, for j > 0.

B. The Illiac Network

The *Illiac* network consists of four interconnection functions defined as follows:

$$\text{Illiac}_{+1}(P) = P+1 \bmod N$$

$$\text{Illiac}_{-1}(P) = P-1 \bmod N$$

$$\text{Illiac}_{+n}(P) = P+n \bmod N$$

$$\text{Illiac}_{-n}(P) = P-n \bmod N$$

where *n* is assumed to be an integer. For example, if N = 16, $\text{Illiac}_{+n}(0) = 4$. This network allows PE P to send data to any one of PEs P+1, PE P–1, PE P+n, or PE P–n, arithmetic mod N. This is often referred to as

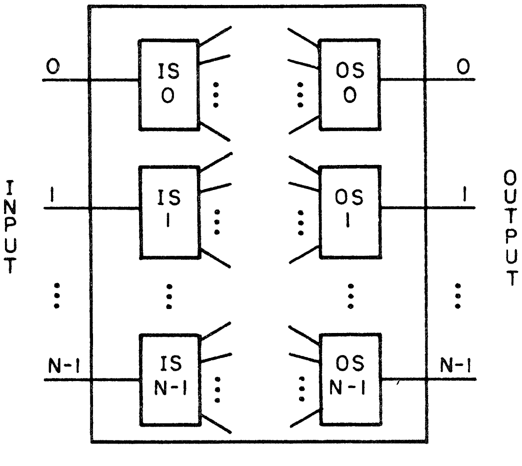


Fig. 2: Conceptual view of a single-stage network. "IS" is input selector, "OS" is output selector.

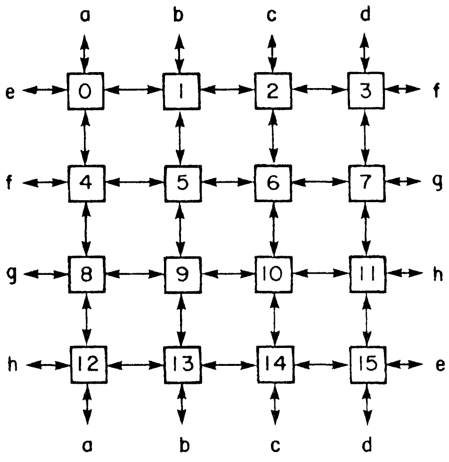


Fig. 3: Illiac network for $N = 16$. (The actual Illiac IV SIMD machine had $N = 64$). Vertical lines are $+\sqrt{N}$ and $-\sqrt{N}$. Horizontal lines are $+1$ and -1.

a four nearest neighbor connection pattern, as shown for $N = 16$ in Fig. 3. This network was implemented in the Illiac IV SIMD machine, where $N = 64$ [1, 6].

Relating this to the conceptual model of a single stage network shown in Fig. 2, for each i, $0 \leq i < N$, input selector i has lines to output selectors $i+1$, $i-1$, $i+n$, and $i-n$, mod N. For each j, $0 \leq j < N$, output selector j gets its inputs from input selectors $j-1$, $j+1$, $j-n$, and $j+n$, mod N. Since there is a single instruction stream in an SIMD machine, all active PEs must use the same interconnection function (connection) at the same time. For example, if PE 0 is sending data to PE 1, then all active PEs must send data using the Illiac_{+1} connection.

This type of network is included in the MPP [4, 5] and DAP [16] SIMD systems. Various properties and capabilities of the Illiac network are discussed in [6, 13, 25, 28, 31, 32].

C. The Plus-Minus 2^i (PM2I) Network

The *Plus-Minus* 2^i *(PM2I)* network consists of 2m interconnection functions defined by:

$$PM2_{+i}(P) = P+2^i \bmod N$$

$$PM2_{-i}(P) = P-2^i \bmod N$$

for $0 \leq i < m$. For example, $PM2_{+1}(2) = 4$ if $N > 4$. Since $P+2^{m-1} = P-2^{m-1}$, mod N, for all P, $0 \leq P < N$, the interconnection functions $PM2_{+(m-1)}$ and $PM2_{-(m-1)}$ are equivalent. Fig. 4 shows the $PM2_{+i}$ interconnections for $N = 8$. Diagrammatically, $PM2_{-i}$ is the same as $PM2_{+i}$ except the direction is reversed. This network is called the Plus-Minus 2^i since, in terms of mapping source addresses to destinations, it can add or subtract 2^i from the PE addresses, i.e., it allows PE P to send data to any one of PE $P+2^i$ or PE $P-2^i$, arithmetic mod N, $0 \leq i < m$.

In terms of the conceptual model of a single stage network (Fig. 2), for the PM2I network, for each j, $0 \leq j < N$, input selector j is connected to output selectors $j+2^i$ and $j-2^i$ mod N, for all i, $0 \leq i < m$. For each j, $0 \leq j < N$, output selector j gets its inputs from input selectors $j-2^i$ and $j+2^i$ mod N, for all i, $0 \leq i < m$. As with the Illiac network, all active PEs must use the same PM2I interconnection function at the same time.

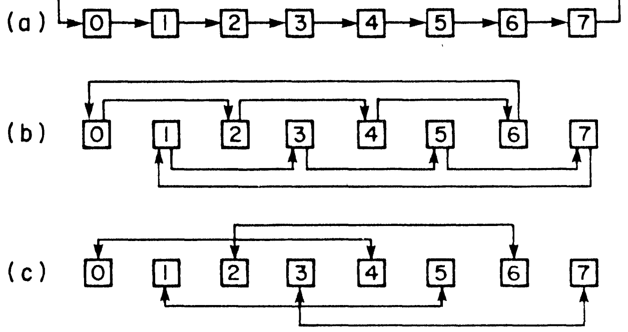


Fig. 4: PM2I network for $N = 8$. (a) $PM2_{+0}$ connections. (b) $PM2_{+1}$ connections. (c) $PM2_{+2}$ connections. For the $PM2_{-i}$ connections, $0 \leq i \leq 2$, reverse the direction of the arrows.

A network similar to the PM2I is used in the "Novel Multiprocessor Array" [24] and is included in the network of the Omen computer [15]. The concept underlying the SIMDA machine's interconnection network is similar to that of the PM2I [36]. The PM2I connection pattern forms the basis for the data manipulator [10], ADM [33], and gamma [26] multistage networks. Various properties of the PM2I are discussed in [11, 27, 28, 29, 31, 32].

D. The Shuffle-Exchange Network

The *Shuffle-Exchange* network consists of a shuffle function and an exchange function. The *shuffle* is defined by:

$$\text{shuffle}(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-2}p_{m-3}\cdots p_1p_0p_{m-1}$$

and the *exchange* is defined by:

$$\text{exchange}(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-1}p_{m-2}\cdots p_1\bar{p}_0.$$

For example, shuffle(3) = 6 and exchange(6) = 7, for $N \geq 8$. This network is shown in Fig. 5 for $N = 8$,

Consider the conceptual model of single stage networks shown in Fig. 2. For the Shuffle-Exchange single stage network, input selector $P = p_{m-1}\cdots p_1p_0$ is connected to output selectors $p_{m-2}\cdots p_1p_0p_{m-1}$ (= shuffle(P)) and $p_{m-1}\cdots p_1\bar{p}_0$ (= exchange(P)). Output selector $r_{m-1}\cdots r_1r_0$ gets its inputs from input selectors $r_0r_{m-1}\cdots r_2r_1$ and $r_{m-1}\cdots r_1\bar{r}_0$. As with the other networks, all active PEs must use the same interconnection function at the same time.

Mathematical properties of the shuffle are discussed in [14, 17]. The multistage omega network is a series of m Shuffle-Exchanges [21]. The shuffle is also included in the networks of the Omen [15] and RAP [9] systems.

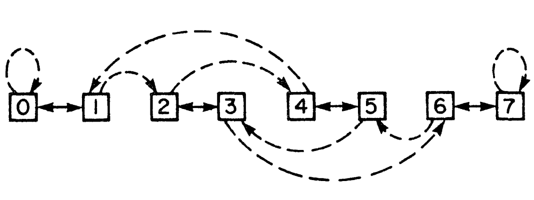


Fig. 5: Shuffle-Exchange network for $N = 8$. Solid line is exchange, dashed line is shuffle.

Features of the Shuffle-Exchange are discussed in [7, 8, 11, 13, 19, 20, 22, 23, 27, 28, 31, 32, 35, 37]. (The ability of each of the PM2I and Illiac networks to perform the exchange function in just two transfers was presented in [31] and is not considered here.)

4. Shuffling with the PM2I Network

The following ground rules will be used in the design and analysis of the algorithm to perform the shuffle with the PM2I network.

(1) The model and definitions presented in Sections 2 and 3 will be the formal basis for the results.

(2) When simulating the shuffle, the data that is originally the DTR of PE P must be transferred to the DTR of shuffle(P), for all P, $0 \leq P < N$.

(3) The time for each algorithm is in terms of the number of executions of interconnection functions required to perform the simulation.

The reason for (3) can be seen by considering the way in which various instructions can be implemented. The instructions in the simulation algorithms can be divided into three categories: control unit operations (in C), register to register operations (in I), and interprocessor data transfers (in F). Control unit operations, such as incrementing a count register in the control unit for a "for loop," can, in general, be done in parallel (overlapped) with the previously broadcast PE instruction, thus taking no additional time. Register to register operations within a PE will probably involve a single chip or, at worst, adjacent chips. The inter-PE data transfers will involve setting the controls of the interconnection network and passing data among the PEs, involving board to board, and probably rack to rack, distances. Thus, unless the number of register to register operations is much greater than the number of inter-PE data transfers, the time for the interprocessor transfers will be the dominating factor in determining the execution time of the simulation algorithm.

In the algorithm below ":" indicates a comment. When discussing the algorithms, "Li" is used as an abbreviation for "line i of the algorithm."

To understand the concept underlying the algorithm to perform the shuffle, consider the "distance" the shuffle moves a data item. The data item in the DTR of PE P, $0 \leq P < N/2$, is moved to shuffle(P) = 2P, a distance of shuffle(P) − P = P. The data item in the DTR of PE P′, $N/2 \leq P' < N$, is moved to shuffle(P′) = 2P′ +1 mod N, a distance of shuffle(P′) − P′ = P′ +1. This is shown in Fig. 6 for N = 8. The algorithm uses the $PM2_{+0}$, $PM2_{+1}$, $PM2_{+2}$, ..., $PM2_{+m-1}$, and $PM2_{+0}$ functions, in that order, to move the DTR data from PE P a distance of P, $0 \leq P < N/2$, and the DTR data from PE P′ a distance of P′ +1 mod N, $N/2 \leq P' < N$. This is also shown for N = 8 in Fig. 6. Note that in L3 to L5, for $1 \leq j < m-1$, all of the data of interest is in even numbered PEs.

Algorithm to perform the shuffle with the PM2I:

```
(L1) $A \leftarrow DTR\ [X^{m-1}0]$        :even PEs save DTR in A
(L2) $PM2_{+0}\ [X^{m-1}1]$              :odd PEs send to even PEs
(L3) for $j = 1$ until $m-1$ do
(L4)     $A \longleftrightarrow DTR[X^{m-j-1}1X^{j-1}0]$ :do if $p_j = 1, p_0 = 0$
(L5)     $PM2_{+j}\ [X^{m-1}0]$          :even PEs send $+2^j$
(L6) $PM2_{+0}\ [X^{m-1}0]$              :even PEs send to odd PEs
(L7) $DTR \leftarrow A\ [X^{m-1}0]$        :reload DTR in even PEs
```

This algorithm used m+1 inter-PE data transfers and m+1 register to register moves. The operation of this algorithm for N = 8 is shown in Tab. 1. For example, consider the data item initially in the DTR of PE 5 (= 101). PE 5 does not match the mask in L1 ([XX0]). PE 5 does match the mask in L2 ([XX1]) and the data is moved to PE $PM2_{+0}(5)$ = 6 (= 110). PE 6 does match the mask in L4 when j = 1 ([X10]) and the data is moved to the A register of PE 6. The data is unaffected by L5 when j = 1 (since it is not in the DTR). PE 6 does match the mask in L4 when j = 2 ([1X0]) and the data is moved to the DTR of PE 6. PE 6 does match the mask in L5 when j = 2 ([XX0]) and the data is moved to the DTR of PE $PM2_{+2}(6)$ = 2. PE 2 does match the mask in L6 ([XX0]) and the data is moved to the DTR of PE $PM2_{+0}(2)$ = 3. PE 3 does not match the mask in L7 ([XX0]). Thus, the data from PE 5 is moved to PE 3 = shuffle(5). This is shown by the dotted line in Tab. 1.

To prove the algorithm is correct, induction will be used (assume all arithmetic is mod N). The induction hypothesis (proven correct below) is that after executing $PM2_{+j}$ in L1 (for j = 0) or L5 (for $1 \leq j < m$) the data originally in the DTR of PE $G = g_{m-1} \cdots g_1 g_0$ will currently be in PE $P = p_{m-1} \cdots p_1 p_0 = (g_{m-1} \cdots g_{j+2} g_{j+1}) * 2^{j+1} + (g_j \cdots g_1 g_0) * 2$. (When j = 0, $P = (g_{m-1} \cdots g_2 g_1) * 2 + (g_0) * 2$.) The data will be in the A register if $g_j = 0$ and in the DTR if $g_j = 1$.

Thus, when j = m−1, the data originally from PE G is in PE $(g_{m-1} \cdots g_1 g_0) * 2$. The data item from the DTR of PE $(g_{m-1} \cdots g_1 g_0) * 2$ is moved to PE $(g_{m-1} \cdots g_1 g_0) * 2 + 1$ by L6; which is correct since this data item is from a PE where $g_j = g_{m-1} = 1$, so shuffle(G) = 2*G + 1. The data item from the A register of PE $(g_{m-1} \cdots g_1 g_0) * 2$ is moved to the DTR of that PE by L7; this is correct since this data item is from a PE where $g_j = g_{m-1} = 0$, so shuffle(G) = 2*G.

To complete the correctness proof it must be shown that the induction hypothesis is true.

Basis: j = 0.

origin PE number	distance moved by shuffle		distance moved by PM2I			
0 = 000	+0	--	--	--	--	+0
1 = 001	+1	+1	--	--	--	+1
2 = 010	+2	--	+2	--	--	+2
3 = 011	+3	+1	+2	--	--	+3
4 = 100	+5	--	--	+4	+1	+5
5 = 101	+6	+1	--	+4	+1	+6
6 = 110	+7	--	+2	+4	+1	+7
7 = 111	+0	+1	+2	+4	+1	+0
		$PM2_{+0}$	$PM2_{+1}$	$PM2_{+2}$	$PM2_{+0}$	Total

Fig. 6: The idea underlying the algorithm for the PM2I to perform the shuffle, shown for N = 8.

Tab. 1: Example of the algorithm for performing the shuffle using the PM2I when $N = 8$. It is assumed that initially the DTR of PE P contains the integer P, $0 \leq P < 8$.

PE	Initial DTR	L1 A	L2 DTR	L4 j=1 A	L4 j=1 DTR	L5 j=1 DTR	L4 j=2 A	L4 j=2 DTR	L5 j=2 DTR	L6 DTR	L7 DTR	PE
000	000	000	111	000	111	110	000	110	100	100	000	000
001	001	-	-	-	-	-	-	-	-	100	100	001
010	010	010	001	001	010	111	001	111	101	101	001	010
011	011	-	-	-	-	-	-	-	-	101	101	011
100	100	100	011	100	011	010	010	100	110	110	010	100
101	101	-	-	-	-	-	-	-	-	110	110	101
110	110	110	101	101	110	011	011	101	111	111	011	110
111	111	-	-	-	-	-	-	-	-	111	111	111

Case 1: The data item from the DTR of PE $G = g_{m-1} \cdots g_2 g_1 0$. This data item is moved to the A register of that PE by L1. Since $g_0 = 0$, $G = (g_{m-1} \cdots g_2 g_1)*2 + (g_0)*2 = P$. This data is not moved by L2. It remains in the A register and $g_0 = 0$. Thus, the induction hypothesis is true for $j = 0$ for this case.

Case 2: The data item from the DTR of PE $G = g_{m-1} \cdots g_2 g_1 1$. This data item is not moved by L1. It is moved to the DTR of PE $P = G + 1$ by $PM2_{+0}$ in L2. Since $g_0 = 1$, $G + 1 = g_{m-1} \cdots g_2 g_1 1 + 1 = (g_{m-1} \cdots g_2 g_1)*2 + 2 = (g_{m-1} \cdots g_2 g_1)*2 + (g_0)*2 = P$. The data item is in the DTR and $g_0 = 1$. Thus, the induction hypothesis is true for $j = 0$ for this case.

Induction Step: Assume true for $j = k - 1$ and show true for $j = k$.

Case 1: The data item from the DTR of PE $G = g_{m-1} \cdots g_2 g_1 g_0$, where $g_{k-1} = 0$. From the induction hypothesis when $j = k-1$, this data item is in the A register of PE $P = (g_{m-1} \cdots g_{k+1} g_k)*2^k + (g_{k-1} \cdots g_1 g_0)*2$.

Subcase 1a: $p_k = 1$. The A register data is moved to the DTR of PE P by L4 and then to the DTR of PE $P + 2^k$ by L5. Recall $P = p_{m-1} \cdots p_1 p_0 = (g_{m-1} \cdots g_{k+1} g_k)*2^k + (g_{k-1} \cdots g_1 g_0)*2$. Since $g_{k-1} = 0$, $(0 g_{k-2} \cdots g_1 g_0)*2 < 2^k$. Thus, if $p_k = 1$, it must be that $g_k = 1$. Since $g_k = 1$, $P + 2^k =$
$$(g_{m-1} \cdots g_{k+1} 1)*2^k + (g_{k-1} \cdots g_1 g_0)*2 + 2^k$$
$$= (g_{m-1} \cdots g_{k+1})*2^{k+1} + 2^k + (g_{k-1} \cdots g_1 g_0)*2 + 2^k$$
$$= (g_{m-1} \cdots g_{k+1})*2^{k+1} + (1 g_{k-1} \cdots g_1 g_0)*2$$
$$= (g_{m-1} \cdots g_{k+1})*2^{k+1} + (g_k g_{k-1} \cdots g_1 g_0)*2.$$
Furthermore, the data is in the DTR and $g_k = 1$. Thus, the induction hypothesis is true for $j = k$ for this subcase.

Subcase 1b: $p_k = 0$. The A register data is kept in the A register of PE P and not moved by L4 or L5. As in Subcase 1a, since $g_{k-1} = 0$, $(0 g_{k-2} \cdots g_1 g_0)*2 < 2^k$. Thus, if $p_k = 0$, it must be that $g_k = 0$. Since $g_k = 0$, $P =$
$$(g_{m-1} \cdots g_{k+1} 0)*2^k + (g_{k-1} \cdots g_1 g_0)*2$$
$$= (g_{m-1} \cdots g_{k+1})*2^{k+1} + (g_k \cdots g_1 g_0)*2.$$
Furthermore, the data is in the A register and $g_k = 0$. Thus, the induction hypothesis is true for $j = k$ for this subcase.

Case 2: The data item from the DTR of PE $G = g_{m-1} \cdots g_1 g_0$, where $g_{k-1} = 1$. From the induction hypothesis when $j = k-1$, this data item is in the DTR of PE
$$P = (g_{m-1} \cdots g_{k+1} g_k) + 2^k + (g_{k-1} \cdots g_1 g_0)*2,$$

Subcase 2a: $p_k = 1$. The DTR register data is moved to the A register of PE P by L4 and is not moved by L5. Recall $p_{m-1} \cdots p_1 p_0 = (g_{m-1} \cdots g_{k+1} g_k)*2^k + (g_{k-1} \cdots g_1 g_0)*2$. Since $g_{k-1} = 1$, $(g_{k-1} \cdots g_1 g_0)*2 = 2^k + (g_{k-2} \cdots g_1 g_0)*2$. Thus, if $p_k = 1$, it must be that $g_k = 0$. Since $g_k = 0$, $P =$
$$(g_{m-1} \cdots g_{k+1} 0)*2^k + (g_{k-1} \cdots g_1 g_0)*2$$
$$= (g_{m-1} \cdots g_{k+1})*2^{k+1} + (g_k \cdots g_1 g_0)*2.$$
Furthermore, the data is in the A register and $g_k = 0$. Thus, the induction hypothesis is true for $j = k$ for this subcase.

Subcase 2b: $p_k = 0$. The DTR register data is kept in the DTR register of PE P (not moved by L4). It is then moved to the DTR of PE $P + 2^k$ by L5. Since $g_{k-1} = 1$, $(g_{k-1} \cdots g_1 g_0)*2 = 2^k + (g_{k-2} \cdots g_1 g_0)*2$. Thus, if $p_k = 0$, it must be that $g_k = 1$. Since $g_k = 1$, $P + 2^k = (g_{m-1} \cdots g_{k+1})*2^{k+1} + (g_k g_{k-1} \cdots g_1 g_0)*2$ as in Subcase 1a. Furthermore, the data is in the DTR and $g_k = 1$. Thus the induction hypothesis is true for $j = k$ for this subcase.

5. Shuffling with the Illiac Network

In this section the use of the Illiac network to perform the shuffle will be examined. First, it will be shown that a lower bound on the number of transfers (executions of Illiac interconnection functions) needed is 3n/2. Then, an algorithm requiring 2n–1 transfers will be presented.

To show that a lower bound on the number of transfers is 3n/2, four of the N data moves which the shuffle performs will be considered. These are:

(a) from PE (N/4 – n/4) to PE (N/2 – n/2)
(b) from PE (N/2 – n/2) to PE (N – n)
(c) from PE (N/2 + n/2 – 1) to PE (n – 1)
(d) from PE (3N/4 + n/4 – 1) to PE (N/2 + n/2 – 1)

For N = 64 these correspond to: (a) 14 → 28, (b) 28 → 56, (c) 35 → 7, and (d) 49 → 35. All four of these moves are done simultaneously when the shuffle interconnection function is executed. It will now be shown that the Illiac *cannot* do all four in *less than* 3n/2 transfers, i.e., at least 3n/2 transfers are needed. To simplify the presentation, the N = 64 values will be used to demonstrate the bound. The result obtained in this way is directly generalizable by substituting (n–1) for 7, (N/4 – n/4) for 14, (N/2 – n/2) for 28, (N/2 + n/2 – 1) for 35, (3N/4 + n/4 – 1) for 49, (N – n) for 56, Illiac_{+n} for Illiac_{+8}, and Illiac_{-n} for Illiac_{-8}.

In order to more easily visualize the data movements in the Illiac network the "wrap-around" connections (e.g., 7 to 8, 56 to 0) have been "unwrapped" by drawing eight projections of the network, as shown in Fig. 7. The actual network is labeled "C" for center, and the eight projections are labeled NW (north west), N (north), NE (north east), W (west), E (east), SW (south west), S (south), and SE (south east). Thus, each PE is represented nine times: once in the original (center) network, and once in each projection.

For example, consider the data movement from PE 7 to PE 8 using the Illiac$_{+1}$ function. Normally, PE 7, which is in the rightmost column of the Illiac network, connects to PE 8, which is in the leftmost column, using a "wrap-around" connection. For purposes of this discussion, the data from PE 7 in C will be moved to C's PE 8 equivalent in the E projection.

In order to draw the projections, two constraints must be satisfied.

(1) Each projection has to be topologically isomorphic to the Illiac network.

(2) Each projection must have the proper adjacency to the C network and the other projections.

Proper adjacency means that two PEs, each from different projections, are drawn adjacent to one another if and only if they are connected in the original network. As an example of this, consider 7 in C, 63 in N, 0 in NE, and 8 in E.

One could continue generating more of these projections "ad infinitum" to represent all possible implementations of all possible moves. However, the goal here is the show that the set of moves (a) through (d) above cannot be done in less than 3n/2 steps. Therefore, projections which would involve more than 3n/2 steps to do any of (a) through (d) individually are not of interest and are unnecessary.

The lower bound proof is organized as follows. First it will be shown that there are only five sets of Illiac function executions that can perform both the 28 → 56 and 35 → 7 moves in less than 3n/2 steps (Fig. 7 and Tab. 2). Then it will be shown that there are only five sets of Illiac function executions (which happen to be different from the first five sets) that can perform both the 14 → 28 and 49 → 35 moves in less than 3n/2 steps (Fig. 8 and Tab. 2). Finally, it will be shown that no single set of less than 3n/2 Illiac function executions can perform all four moves (Tab. 3).

Tab. 2: All possible combinations of 28 → 56 and 35 → 7 paths that can be done individually in less than 3n/2 steps.

28 → 56 / 35 → 7		C (4,0,0,4)	E (3,4,0,0)	NE (0,4,5,0)	N (0,0,4,4)
C	(0,4,4,0)	(4,4,4,4)	(3,4,4,0) 1 √	(0,4,5,0) 2 √	(0,4,4,4)
W	(0,0,3,4)	(4,0,3,4) 3 √	(3,4,3,4)	(0,4,5,4)	(0,0,4,4) 4 √
SW	(5,0,4,0)	(5,0,4,4)	(5,4,4,0)	(5,4,5,0)	(5,0,4,4)
S	(4,4,0,0)	(4,4,0,4)	(4,4,0,0) 5 √	(4,4,5,0)	(4,4,4,4)

Tab. 3: All possible combinations of 14 → 28 and 49 → 35 paths that can be done individually in less than 3n/2 steps.

14 → 28 / 49 → 35		C (2,0,0,2)	N (0,0,6,2)	E (1,6,0,0)
C	(0,2,2,0)	(2,2,2,2) 1 √	(0,2,6,2) 2 √	(1,6,2,0) 3 √
W	(0,0,1,6)	(2,0,1,6) 4 √	(0,0,6,6)	(1,6,1,6)
S	(6,2,0,0)	(6,2,0,2) 5 √	(6,2,6,2)	(6,6,0,0)

Fig. 7 shows all the paths from the source PE 28 in the C network to its associated destination PE 56 in the C network and in the eight projections. Also shown is the source PE 35 in the C network and its associated destination PE 7 in the C network and in the eight projections. There are only four ways to go from 28 to 56 in less than 3n/2 = 12 moves and these are shown at the top of Tab. 2. The four ways to go from 35 to 7 in less than 12 moves are shown on the side of Tab. 2. The four-tuple (w, x, y, z) means that the path consists of w Illiac$_{+8}$ executions (moves), x Illiac$_{+1}$ executions, y Illiac$_{-8}$ executions, and z Illiac$_{-1}$ executions. Note that for the purposes here the order of execution is irrelevant. For example, 28 in C can go to 56 in the NE projection by (0, 4, 5, 0), i.e., the path consists of four Illiac$_{+1}$ moves and five Illiac$_{-8}$ moves. Any path between 28 in C and 56 in NE must include these moves. This is true in general, i.e., if the path from PE A to PE B is given as (w, x, y, z) then (1) the moves specified by the four-tuple will send data from A to B, and (2) any path from A to B must include the moves specified by the four-tuple. In what follows {·} will denote the generalization of the path from n = 8 to any n.

Each square in Tab. 2 shows the set of moves needed to do both the 28 → 56 and 35 → 7 moves for all possible combinations of the individual moves which can be done in less than 12 {3n/2} steps. The five combinations which can be done in less than 12 {3n/2} steps are marked by a check (√). For example, the 28 → 56 path (4, 0, 0, 4) {(n/2, 0, 0, n/2)} and 35 → 7 path (0, 0, 3, 4) {(0, 0, (n/2)–1, n/2)} can be combined to require the moves (4, 0, 3, 4) {(n/2, 0, (n/2)–1, n/2)}. These two paths can both use the same four executions of Illiac$_{-1}$ and so these four moves are counted only once. Therefore, the total is $4 + 0 + 3 + 4 = 11 < 12$ $\{n/2 + 0 + (n/2)–1 + n/2 = (3n/2)–1 < 3n/2\}$ moves. An example of a combination involving more than 12 {3n/2} moves is the 28 → 56 (4, 0, 0, 4) {(n/2, 0, 0, n/2)} path and the 35 → 7 (0, 4, 4, 0) {(0, n/2, n/2, 0)} path. This combination requires the execution of $4 + 4 + 4 + 4 = 16 > 12$ $\{n/2 + n/2 + n/2 + n/2 = 2n > 3n/2\}$ Illiac functions.

The analysis for the 14 → 28 and 49 → 35 transfers shown in Fig. 8 and Tab. 3 is similar. The five sets of Illiac functions which can do both of these transfers in less than 12 moves are checked in Tab. 3.

The final step to the proof is to examine all combinations of the five sets found in each of Tabs. 2 and 3 to see if there exists any set of transfers which can perform all four transfers (28 → 56, 35 → 7, 14 → 28, and 49 → 35) in less than 12 moves. This is shown in Tab.

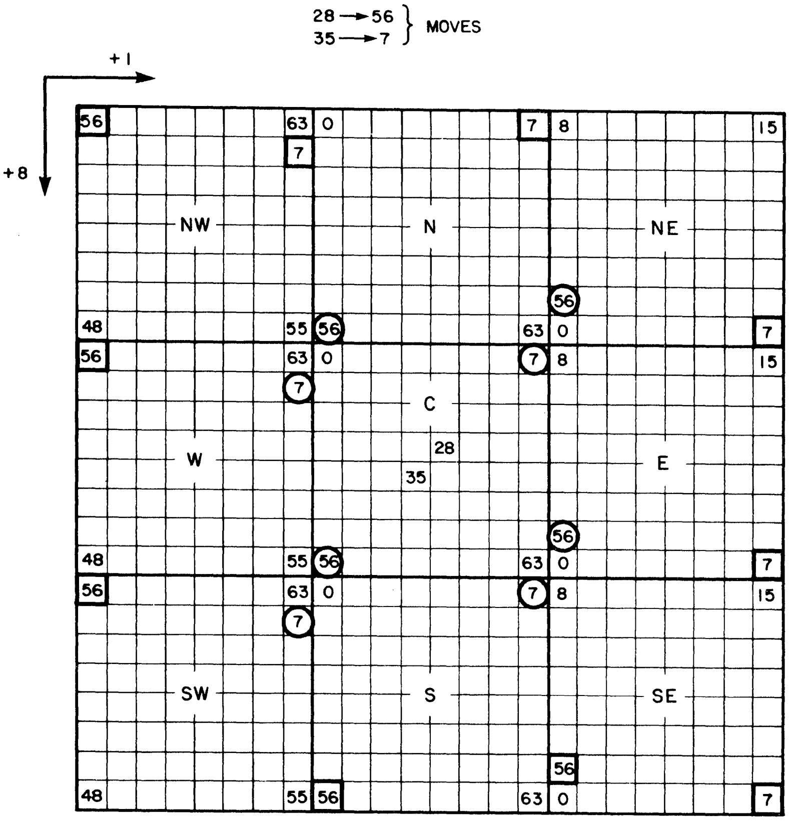


Fig. 7: The source/destination relationship for the moves 28 → 56 and 35 → 7 in an "unwrapped" Illiac network. The circle denotes a destination which can be reached in less than 3n/2 steps.

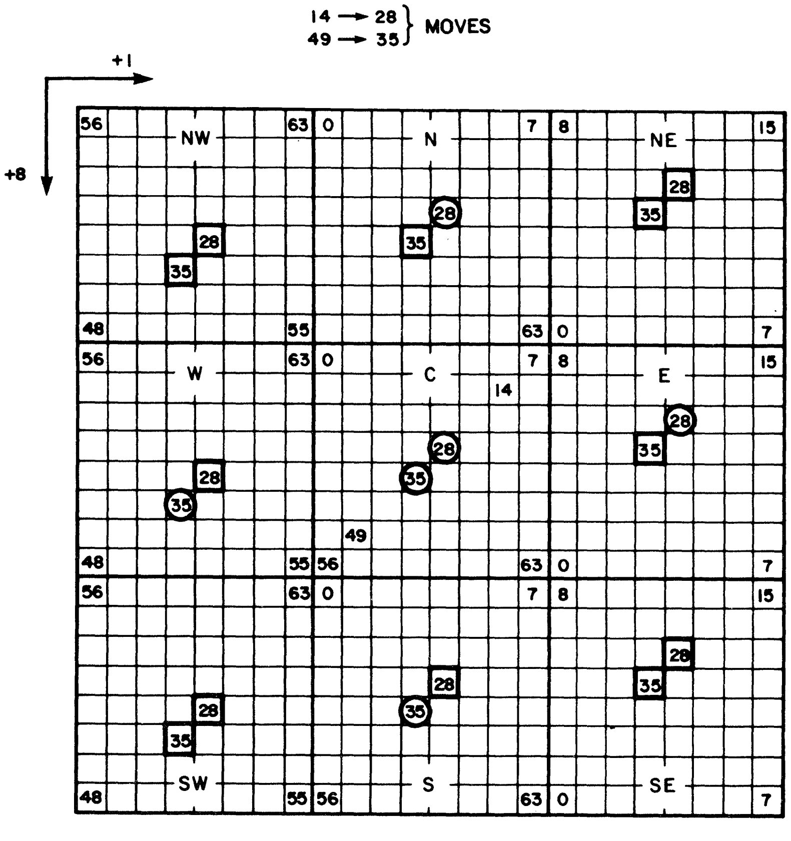


Fig. 8: The source/destination relationship for the moves 14 → 28 and 49 → 35 in an "unwrapped" Illiac network. The circle denotes a destination which can be reached in less than 3n/2 steps.

Tab. 4: Combination of relevant paths from Tabs. 2 and 3.

14 → 28 49 → 35 \ 28 → 56 35 → 7	1 (3,4,4,0)	2 (0,4,5,0)	3 (4,0,3,4)	4 (0,0,4,4)	5 (4,4,0,0)
1 (2,2,2,2)	(3,4,4,2)	(2,4,5,2)	(4,2,3,4)	(2,2,4,4) ✓	(4,4,2,2) ✓
2 (0,2,6,2)	(3,4,6,2)	(0,4,6,2) ✓	(4,2,6,4)	(0,2,6,4) ✓	(4,4,6,2)
3 (1,6,2,0)	(3,6,4,0)	(1,6,5,0) ✓	(4,6,3,4)	(1,6,4,4)	(4,6,2,0) ✓
4 (2,0,1,6)	(3,4,4,6)	(2,4,5,6)	(4,0,3,6)	(2,0,4,6) ✓	(4,4,1,6)
5 (6,2,0,2)	(6,4,4,2)	(6,4,5,2)	(6,2,3,4)	(6,2,4,4)	(6,4,0,2) ✓

4. As demonstrated, there is no such set. There are seven sets which require exactly 12 moves (indicated by checks), but none which requires less than 12. For example, 28 → 56 and 35 → 7 can be done using (3, 4, 4, 0), and 14 → 28 and 49 → 35 can be done using (2, 2, 2, 2), however, the combination of these two sets yields (3, 4, 4, 2), which is greater than 12 moves.

In summary, four of the moves performed by shuffle (28 → 56, 35 → 7, 14 → 28, and 49 → 35) have been examined. It has been shown that no set of Illiac function executions can do this in less than $3n/2 = 12$ moves. As indicated above, this argument can be generalized directly using the substitutions listed.

Consider an algorithm for performing the shuffle interconnection function with the Illiac network. This will be done by replacing each PM2I interconnection function in the above algorithm with Illiac interconnection functions. For L2, use "Illiac_{+1} $[X^{m-1}1]$," since $\text{Illiac}_{+1} \equiv \text{PM2}_{+0}$. Similarly, for L6, use "Illiac_{+1} $[X^{m-1}0]$." To do L5, first recall that only the even numbered PEs contain the data of concern (after L2 is executed and before L6 is executed). Therefore, it is acceptable to use "PM2_{+j} $[X^m]$" in L5, since any data movement among the odd numbered PEs is ignored (and overwritten by L6). To perform "PM2_{+j} $[X^m]$," for $1 \leq j < m$, with the Illiac network the algorithms presented in [31] can be used. Specifically, to perform "PM2_{+j} $[X^m]$" for $1 \leq j < m/2$ use:

```
for i = 1 until $2^j$ do $\text{Illiac}_{+1}$ $[X^m]$
```

since 2^j execution of Illiac_{+1} is equivalent to $+2^j = \text{PM2}_{+j}$. Analogously, to perform "PM2_{+j} $[X^m]$" for $m/2 \leq j < m$ use:

```
for i = 1 until $2^j/n$ do $\text{Illiac}_{+n}$ $[X^m]$
```

since $2^j/n$ executions of Illiac_{+n} is equivalent to $+2^j = \text{PM2}_{+j}$. The total number of Illiac transfers needed is:

for L2: 1

for L6: 1

for L5, $1 \leq j < m/2$: $\sum_{j=1}^{(m/2)-1} 2^j = 2^{m/2} - 2 = n-2$

for L5, $m/2 \leq j < m$:

$$\sum_{j=m/2}^{m-1} 2^j/n = \sum_{j=m/2}^{m-1} 2^{j-(m/2)} = \sum_{j=0}^{(m/2)-1} 2^j = n-1$$

Thus, the grand total is 2n−1 transfers. As mentioned in Section 1, this compares favorably with the earlier result of 4n−4 in [25].

6. Conclusions

The ability of the PM2I and Illiac single stage interconnection SIMD machine networks to perform the shuffle interconnect was examined. In [28] is was shown that a lower bound on the number of transfers needed for the PM2I network to perform the shuffle is $\log_2 N$. The algorithm described here and proven correct required only $(\log_2 N)+1$ transfers. This algorithm was used as basis for an algorithm to do the shuffle with the Illiac network in $(2\sqrt{N})-1$ transfers. This compares favorably an earlier result of $4(\sqrt{N}-1)$ in [25]. A lower bound analysis was presented showing that at least $3\sqrt{N}/2$ transfers are required for this task.

These results are of both theoretical and practical value. Theoretically, they add to the body of knowledge about the properties of the PM2I and Illiac networks. Practically, the algorithms presented could actually be used to perform the shuffle interconnection on a system that has implemented the PM2I or Illiac network. Furthermore, the lower bound proof shows that it is impossible to do the shuffle with the Illiac in any fewer than $3\sqrt{N}/2$ steps.

Acknowledgements: Some of the figures and tables in this paper are from "Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies," by H. J. Siegel, to be published by D. C. Heath and Co.

References

[1] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV computer," *IEEE Trans. Comput.*, Vol. C-17, Aug. 1968, pp. 746-757.

[2] K. E. Batcher, "STARAN parallel processor system hardware," *AFIPS Conf. Proc. 1974 Nat'l. Computer Conf.*, May 1974, pp. 405-410.

[3] K. E. Batcher, "STARAN series E," *1977 Int'l. Conf. Parallel Processing,* Aug. 1977, pp. 140-143.

[4] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.*, Vol. C-29, Sept. 1980, pp. 836-840.

[5] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Trans. Comput.*, Vol. C-31, Mar. 1982, pp. 377-384.

[6] W. J. Bouknight, S. A. Denneberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proc. of the IEEE,* Vol. 60, Apr. 1972, pp. 369-388.

[7] P-Y. Chen, D. H. Lawrie, P-C. Yew, and D. A. Padua, "Interconnection networks using shuffles," *Computer,* Vol. 14, Dec. 1981, pp. 55-64.

[8] P-Y. Chen, P-C. Yew, and D. H. Lawrie, "Performance of packet switching in buffered single-stage shuffle-exchange networks," *3rd Int'l. Conf. Distributed Computer Systems,* Oct. 1982, pp. 622-627.

[9] G. R. Couranz, M. S. Gerhardt, and C. J. Young, "Programmable RADAR signal processing using the RAP," *1974 Sagamore Computer Conf. Parallel Processing,* Aug. 1974, pp. 37-52.

[10] T. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Comput.*, Vol. C-23, Mar. 1974, pp. 309-318.

[11] J. P. Fishburn and R. A. Finkel, "Quotient networks," *IEEE Trans. Comput.,* Vol. C-31, Apr. 1982, pp. 288-295.

[12] M. J. Flynn, "Very high-speed computing systems," *Proc. of the IEEE,* Vol. 54, Dec. 1966, pp. 1901-1909.

[13] W. M. Gentleman, "Some complexity results for matrix computations parallel processors," *Journal of the ACM,* Vol. 25, Jan. 1978, pp. 112-115.

[14] S. W. Golomb, "Permutations by cutting and shuffling," *SIAM Review,* Vol. 3, Oct. 1961, pp. 293-297.

[15] L. C. Higbie, "The Omen computer: associative array processor," *IEEE Computer Society Compcon 72,* Sept. 1972, pp. 287-290.

[16] D. J. Hunt, "The ICL DAP and its application to image processing," in *Languages and Architectures for Image Processing,* M. J. B. Duff and S. Levialdi, eds., Academic Press, London, England, 1981, pp. 275-282.

[17] P. B. Johnson, "Congruences and card shuffling," *American Mathematical Monthly,* Vol. 63, Dec. 1956, pp. 718-719.

[18] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 Int'l. Conf. Parallel Processing,* Aug. 1982, pp. 353-362.

[19] T. Lang, "Interconnections between processors and memory modules using the shuffle-exchange network," *IEEE Trans. Comput.,* Vol. C-25, May 1976, pp. 496-503.

[20] T. Lang and H. S. Stone, "A shuffle-exchange network with simplified control," *IEEE Trans. Comput.,* Vol. C-25, Jan. 1976, pp. 55-66.

[21] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.,* Vol. C-24, Dec. 1975, pp. 1145-1155.

[22] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," *IEEE Trans. Comput.,* Vol. C-30, Feb. 1981, pp. 101-107.

[23] D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network," *Journal of the ACM,* Vol. 29, July 1982, pp. 642-667.

[24] Y. Okada, H. Tajima, and R. Mori, "A reconfigurable parallel processor with microprogram control," *IEEE Micro,* Vol. 2, Nov. 1982, pp. 48-60.

[25] S. E. Orcutt, "Implementation of permutation functions in Illiac IV-type computers," *IEEE Trans. Comput.,* Vol. C-25, Sept. 1976, pp. 929-936.

[26] D. S. Parker and C. S. Raghavendra, "The gamma network: a multiprocessor interconnection network with redundant paths," *9th Annual Symp. Computer Architecture,* Apr. 1982, pp. 73-80.

[27] D. K. Pradhan and K. L. Kodandapani, "A uniform representation of single- and multistage interconnection networks used in SIMD machines," *IEEE Trans. Comput.,* Vol. C-29, Sept. 1980, pp. 777-791.

[28] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Trans. Comput.,* Vol. C-26, Feb. 1977, pp. 153-161.

[29] H. J. Siegel, "Partitionable SIMD computer system interconnection network universality," *16th Annual Allerton Conf. Communication, Control, and Computing,* Univ. Ill., Oct. 1978, pp. 586-595.

[30] H. J. Siegel, "Interconnection networks for SIMD machines," *Computer,* Vol. 12, June 1979, pp. 57-65.

[31] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Trans. Comput.,* Vol. C-28, Dec. 1979, pp. 907-917.

[32] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comput.,* Vol. C-29, Sept. 1980, pp. 791-801.

[33] H. J. Siegel and R. J. McMillen, "Using the Augmented Data Manipulator network in PASM," *Computer,* Vol. 14, Feb. 1981, pp. 25-33.

[34] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.,* Vol. C-30, Dec. 1981, pp. 934-947.

[35] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.,* Vol. C-20, Feb. 1971, pp. 153-161.

[36] A. H. Wester, "Special features in SIMDA," *1972 Sagamore Computer Conf.,* Aug. 1972, pp. 29-40.

[37] C. Wu and T. Feng, "The universality of the shuffle-exchange network," *IEEE Trans. Comput.,* Vol. C-30, May 1981, pp. 324-332.

A CLASSIFICATION OF CUBE-CONNECTED NETWORKS WITH A SIMPLE CONTROL SCHEME

A. Yavuz Oruc
Electrical, Computer and Sys.tems Engineering Department
Rensselaer Polytechnic Institute
Troy, New York 12181

Abstract -- The paper presents three classes of cube-connected networks with individual stage control based on a group theoretic representation of interconnection networks. It is shown that these classes of networks have non-isomorphic group properties. Although permutations realizable by such networks are rather limited in number, the simplicity of their control scheme makes them attractive for VLSI implementation. Moreover, the interconnection power of these networks can be enhanced by simulating the networks which belong to one class by any member of that class. Thus it becomes important to derive conditions under which these networks become equivalent. The paper provides a characterization for each class of networks from which isomorphism maps can easily be obtained. Methods are also presented to construct networks belonging to each class.

I. Introduction

Interconnection networks for multi-processor computers have drawn much attention from researchers in recent years [1,2,3]. Much of the previous work concentrated on proposing networks to implement a specific class of data routing algorithms in a parallel processing environment. Examples of such networks can be found in [4,5,6,7, 8,9,10,11,12,13]. The availability of several networks for similar tasks prompted further studies to determine the interconnection power and capability of certain networks to simulate various other networks. These fall into two categories. The first category deals with proving asymptotic bounds on the number of distinct interconnections and steps (passes) to realize them by a given network [14,15, 16]. Such bounds describe the worst case behavior of a network and hence form a useful basis for its evaluation. However, they do not reveal any information as to what class of interconnections a network can realize. To provide a comparative evaluation of interconnection networks, others investigated the capabilities of certain networks to simulate various other networks [11,17,18,19,20, 21]. These led to a class of multistage interconnection networks, here called the edge-wise cube-connected networks, simply because their interconnection functions can be associated with the edges of a cube. We demonstrate that under individual stage control, edge-wise cube-connected networks manifest themselves by pairwise commuting permutations which represent their stages. This observation essentially paves the way for obtaining two new classes of cube-connected networks with the same control scheme . We provide characterizations of these classes of networks in terms of the cycle structures of their permutations. These indicate that an equivalence among two cube-connected networks can be attributed to (1) the existence of a group isomorphism among the permutations of two networks and/or (2) a one-to-one onto correspondence between the permutations representing the stages of the two networks. The paper is organized as follows. In section II we introduce a network model which describes the behavior of interconnection networks under arbitrary control schemes. In section III cube-connected networks are described using this model. In section IV the group properties of cube-connected networks with individual stage control have been explored. It is shown that there exist at least three classes of such networks with non-isomorphic group properties. In section V the characterization of each class of networks is provided in terms of the cycle structures of their permutations. In section VI an example is given for each class of networks.

II. Network Model and Definitions

In this section we introduce an interconnection network model and basic definitions about interconnection networks. These will serve as a basis for the analysis and presentation of cube-connected networks. To this end, we define an interconnection network IN, as the five tuple (S, D,M,F,g) where:

(a) S and D are finite sets whose elements are called the source nodes and destination nodes respectively,

(b) M is a finite set whose elements are called the control inputs,

(c) F is a set of mappings f_i, called the interconnection functions such that $f_i:S_i \rightarrow D_i$ where S_i and D_i belong to some partitions of S and D respectively,

(d) g, called the control function, is a surjection from F to M.

For convenience, the elements of S(D) are assumed to be integers modulo $|S|(|D|)$ where $|S|(|D|)$ is the cardinality of S(D).

IN is called a permutation network if f_i is a bijection for all $f_i \in F$ and $S_i=D_i$ for all $S_i \in S$ and $D_i \in D$. All the networks described in this paper are permutation networks unless otherwise stated. The cycle notation will be used to represent f_i of a permutation network. Thus we shall write $f_i=(s_1 s_2 \ldots s_r)$ to imply $d_2=(s_1)f_i$, $d_3=(s_2)f_i, \ldots,$ $d_1=(s_r)f_i$ where $s_i \in S$ and $d_i \in D$. A cycle of r elements is called an r-cycle, in particular, a cycle 2 elements is called transposition. The set of all permutations on n symbols forms the symmetric group S_n. A group G is a set with a binary operation '•' on G, where '•' is associative, there is an identity element e in G such that $e \cdot p = p \cdot e = p$ for all $p \in G$ and for each $p \in G$, there is an inverse element, denoted p^{-1}, in G with the property that $p^{-1} \cdot p = p \cdot p^{-1} = e$. The group generated by permutations $p_1, \ldots, p_r$ will be denoted by $<p_1, \ldots, p_r>$. G is called commutative if $p \cdot q = q \cdot p$ for all $p,q \in G$ and it

0190-3918/83/0000/0126$01.00 © 1983 IEEE

is called cyclic if all of its elements can be generated ~~by one~~ element in G.

A network is said to generate a subgroup of S_n in k passes if the permutations realized by k consecutive applications of its interconnection functions over its source nodes form a subgroup of S_n. Since every finite group is a closed set, once a network generates a group, it exhausts all the interconnections it can realize. It follows that groups play important roles in determining the interconnection power of permutation networks.

Control inputs are integer-valued variables. Each f_i is further refined by the values of the control input with which it is associated by g. Thus, if $m=(f_i)g$ and $m\epsilon\{0,1\}$ then $f_i(m=0)$ and $f_i(m=1)$ denote the two cycles that f_i will designate when m=0 and m=1 respectively. The map g coordinates the composition of f_i. It partitions F into disjoint subsets and assigns to each subset a unique control input. Thus if for IN=(S,D,M,F,g), $F=\{f_1,f_2,f_3\}$, $m_1,m_2\epsilon\{0,1\}$, $(f_1)g=(f_2)g=m_1$ and $(f_3)g=m_2$ then IN realizes the composition $f_1(m_1=0)\cdot f_2(m_1=0)\cdot f_3(m_2=1)$ when m =0 and m =1. As an example, let $f_1(m_1=0)=(021)$, $f_2(m_1=0)=(354)$ and $f_3(m_2=1)=(67)$. The network with $F=\{f_1,f_2,f_3\}$ and the required setting to realize $f_1(m_1=0)\cdot f_2(m_1=0)\cdot f_3(m_2=1)$ is depicted in Fig. 1.

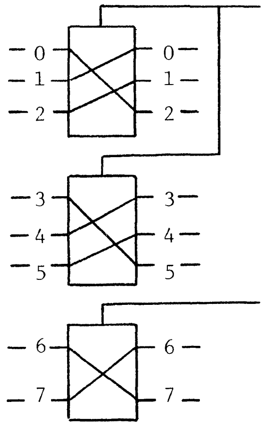


Fig. 1. Network IN Realizing f=(021)(354)(67).

Note that the order of the composition is not critical since f_i are pairwise disjoint cycles. Also note that the number of interconnections IN can realize is the number of values that m_1 can take times the number of values m_2 can take, which is 4. In general if the number of values which m_i can take is n_i then the number of interconnections realizable by IN is given by $\prod_{i=1}^{|M|} n_i$.

Interconnection networks can be cascaded together to form multistage networks. An n-input, r-stage network, hereafter called an (n,r)-network, is a cascade of r networks $IN_i=(S_i,D_i,M_i,F_i,g_i)$ such that S_i and D_i are sets of integers modulo n and every interconnection function f of the network is defined as the composition $f=f_1\ldots f_r$, that is, $(s_1)f=(s_1)f_1\ldots f_r$, where $(s_1)f_1=d_1=s_2,\ldots,$ $(s_r)f_r=d_r$, $s_i\epsilon S_i$, $d_i\epsilon D_i$ and f_i is a composition of the interconnection functions of IN_i, for all i; $1\le i\le r$. As an example, let IN be a (5,2) network where $F_1=\{f_{11},f_{21}\}$, $F_2=\{f_{12},f_{22}\}$. IN is shown in Fig. 2. The control inputs are not specified although the rule described earlier applies to each stage. Hence each permutation of IN must be of the form $f=(f_{11}\cdot f_{21})\cdot(f_{12}\cdot f_{22})$

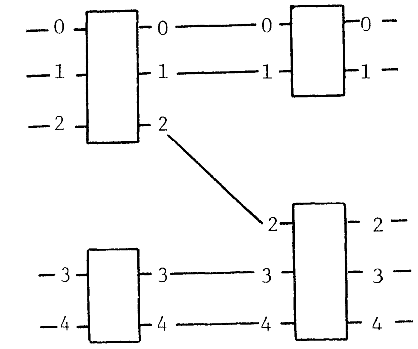


Fig. 2. An (5,2) Interconnection Network.

We note that our definition of a multistage interconnection network differs from that of Benes [22] and many others in that the links between the stages are not included in our model. This is due to the fact that we assign the same label to a source and destination node if and only if there is a path between them when all interconnection nodes involving them are set to the identity connections. Thus we do not need to use interconnection functions to describe the links explicitly.

It is of interest to compute the number N of distinct control schemes for an (n,r)-network. By counting the number of surjections from F_i to M_i [23], it can be shown that,

$$N = \prod_{i=1}^{r} \{(-1)^{M_i}\cdot \sum_{k=0}^{|M_i|}(-1)^k\cdot \binom{|M_i|}{k}\cdot k^{|F_i|}\}/|M_i|!$$

In particular, if $|M_i|=m$, $|F_i|=p$ for all i=1,...,r, then

$$N = r\{(-1)^m\cdot \Sigma_{k=0}^{m}(-1)^k\cdot \binom{m}{k}\cdot k^p\}/m!$$

or,

$N=r\cdot S(p,m)$ where S(p,m) denotes the Stirling number of the second kind [23].

III. Cube-Connected Networks

In this section, interconnection networks with cube topology are presented. These constitute an important class of networks since their interconnection functions are transpositions which can easily be implemented by 2-input/output crossbar switches. Various such networks can be found in [5,6,9,11,12,13,19]. In what follows, we derive the conditions to generate a subgroup of S_n through these networks in one pass under individual stage control (ISC).

Definition 1. An (n,r) network with stages $IN_{ik}=(S_i,D_i,M_i,F,g)$ is called cube-connected if $n=2^k$ for some positive integer k and $F_i=\{f_{ij}:j=1,\dots,n/2\}$, where f_{ij} is either the identity function or the transportation function from 2-subset S_{ij} of S_i to 2-subset D_{ij} of D_i for all i; $1\le i\le r$.

As an example, a (4,2) cube-connected network can be specified by the partitions $P(S_1)=\{\{0,1\},\{2,3\}\}$ and $P(S_2)=\{\{0,3\},\{1,2\}\}$. The elements of $P(S_i)$ can be viewed as pairing the nodes of the 2-cube labelled by integers modulo 4. The same analogy holds between an (n,r) cube-connected network and the k-cube where $k=\log_2 n$. In particular if the end nodes of the edges of the k-cube are labelled by integers whose binary expansions differ in exactly one place, then the elements of $P(S_1)[P(S_2)]$ correspond to pairing the end nodes of the edges (diagonals) of the 2-cube.

IV. Cube-Connected Networks With ISC

In this section we present the group properties of cube-connected networks with ISC. An (n,r)-network is said to be using ISC if $|M_i|=1$ for all i; $1\le i\le r$. As a rule, such networks can not generate large subsets of S_n since the stage IN_i of these networks can realize only two permutations, namely, $f_i(m_i=0)=e_i$ and $f_i(m_i=1)=c_i=c_{i1}\dots c_{in/2}$ where $M_i=\{m_i\}$. Thus the set of permutations realizable by an (n,r) cube-connected network with ISC in one pass is given by the product $<c_1>\dots<c_r>$ where $<c_i>\cdot<c_j>$ $\{x\cdot y:x\epsilon<c_i>,y\epsilon<c_j>\}$. Clearly, the number of permutations generated this way can not exceed 2^r. Despite this shortcoming, ISC is a very simple control scheme and hence networks with ISC are appealing in at least realizing subgroups of small order of S_n. We now state some conditions for generating such subgroups by these networks.

Theorem 1. Let IN be an (n,r) cube-connected network with ISC. IN generates a subgroup of order 2^r of S_n in one pass if $c_i\ c_j=c_j\ c_i$ for all i,j; $1\le i\le j\le r$ and $c_{i+1}\notin<c_1>\dots<c_i>$ for all i; $1\le i\le r-1$.

Proof. The set of permutations G generated by IN is the product $G=<c_1>\dots<c_r>$. Let $x,y\epsilon G$, where $x=x_1\dots x_r$, $y=y_1\dots y_r$ and $x_i,y_i\epsilon<c_i>$. Then $x\cdot y=(x_1\dots x_r)\cdot(y_1\dots y_r)$. Since $c_i\cdot c_j\cdot c_i$, $x_i\cdot x_j=x_j\cdot x_i$ for all i,j; $1\le i$, $j\le r$, and hence $x\cdot y=(x_1\cdot y_1)\dots(x_r\cdot y_r)$. But $x_j\cdot y_i\epsilon<c_i>$. Thus $(x\cdot y)\epsilon G$, and G is closed. Also for any $x\epsilon G$, $x^{-1}=x_r^{-1}\dots x_1^{-1}$ or $x^{-1}=x_r\dots x_1=x_1\dots x_r=x$. Finally $e=e_1\dots e_r$. Thus G is a group. To complete the proof, we need to show $|G|=2^r$. First, $|G|\le 2^r$ since G is a product of r 2-sets. The fact that $|G|=2^r$ follows inductively from the observation that $G_i=<c_1>\dots<c_i>$, $1\le i\le r$ is a group of order 2^i and for each $x\epsilon G_i$, $x\cdot c_{i+1}\notin G_i$ since if $x\cdot c_{i+1}=y\epsilon G_i$ then $c_{i+1}=x^{-1}\ y\epsilon G_i$, which contradicts the hypothesis.

Theorem 1 provides a sufficient condition for the realizability of a subgroup of S_n in one pass by an (n,r) cube-connected network with ISC. However, the commutativity condition among the c_i of an (n,r) cube-connected network is a strong condition to realize a subgroup of S_n by the network. As such it always leads to commutative subgroups. It is noted that all edge-wise cube-connected networks which have appeared in [5,6,11,12,19] generate isomorphic copies of the commutative group of order 2^r. The following therem is based on a weaker condition and it indicates the presence of non-commutative subgroups realizable by an (n,r) cube-connected network.

Theorem 2. Let IN by an (n,r) cube-connected network with ISC. IN generates a group G in one pass if for every index pair i,j, where i<j there exists k<j such that $c_i=c_j\cdot c_k\cdot c_j$.

Proof. (By induction).

Basis. r=2. In this case, $G=<c_1>\cdot<c_2>$. Clearly, e,c_1^{-1}, $c_2^{-1}\epsilon G$. To prove that $(c_1\cdot c_2)^{-1}=c_2\cdot c_1\epsilon G$, let i=1 and j=2. Then by the hypothesis, there exists k=1<j such that $c_1=c_2\cdot c_1\cdot c_2$ or $c_1\cdot c_2=c_2\cdot c_1$. Thus $(c_1\cdot c_2)^{-1}=c_2\cdot c_1=c_1\cdot c_2\epsilon G$ and G is a group.

Induction Step. r=i>2. Suppose $G_i=<c_1>\dots<c_i>$ is a group. We wish to show that $G_{i+1}=G_i\cdot<c_{i+1}>$ is also a group. Thus we must prove that $(c_{i+1}\cdot x)\epsilon G_{i+1}$ for each $x\epsilon G_i$. Let $x=x_1\dots x_i$ where $x_i\epsilon<c_i>$. Then $c_{i+1}\cdot x=c_{i+1}\cdot(x_1\dots x_i)$. Now since i+1>1 by hypothesis there exists a $k_1<i+1$ such that $x_1=c_{i+1}x_{k_1}c_{i+1}$ or $c_{i+1}\cdot x_1=x_{k_1}\cdot c_{i+1}$. Thus $c_{i+1}\cdot x=x_{k_1}\cdot c_{i+1}\cdot x_2\dots x_i$. After repeating the same argument i-1 more times we obtain $c_{i+1}\ x=(x_{k_1}\cdot\dots x_{k_i})\cdot c_{i+1}$. Now since G_i is a group $y=x_{k_1}\dots x_{k_i}\epsilon G_i$ and hence $c_{i+1}\cdot x=y\cdot c_{i+1}\epsilon G_{i+1}$. The assertion follows by the principle of induction.

A similar result can be stated as follows.

Corollary. 1. Let IN be an (n,r) cube-connected network with ISC. IN generates a group in one pass if for each i>j there exists k>j such that $c_i=c_j\cdot c_k\cdot c_j$.

Proof. The proof easily follows from Theorem 2.

The previous results amount to providing conditions for realizing subgroups of S_n in one pass by an (n,r) cube-connected network with ISC. It is also important to find conditions for which such networks fail to realize a group since such conditions will lead to the construction of cube-connected networks with multi-pass features. The next result establishes such a condition. It is still a conjecture for which a proof is under development.

Conjecture 1. Let IN be an (n,r) cube-connected network with ISC and c_i such that $c_i\cdot c_j\ne c_j\cdot c_i$ for all i,j; $1\le i\ne j\le r$. IN can not realize a subgroup of S_n in one pass.

The previous results assert that there exist at least three classes of cube-connected networks with non-isomorphic group properties:

(a) Networks which generate commutative subgroups of S_n of order 2^r in one pass,

(b) Networks which generate non-commutative subgroups of S_n of order 2^r in one pass,

(c) Networks which fail to realize any subgroup of S_n in one pass.

In the following section the characterization of each class of networks is obtained.

V. Characterization of Network Classes

In this section we give two results to identify the c_i of a network belonging to one of the three classes.

Theorem 3. Let c_i and c_j be produces of n/2 pairwise disjoint transposition in common, where $n=2^m$ for some positive integer m. Then $c_i \cdot c_j = c_j \cdot c_i$ if for some partition of the transpositions of c_i into pairs $(a_k b_k)(c_k d_k)$ there exists a partition of the transpositions of c_i into pairs $(a_k c_k)(b_k d_k)$.

Proof. Suppose, $c_i = p_1 \ldots p_{n/4}$ and $c_j = q_1 \ldots q_{n/4}$, where $p_k = (a_k b_k)(c_k d_k)$ and $q_k = (a_k c_k)(b_k d_k)$. Then $p_i \cdot q_j = q_j \cdot p_i$ since p_i and q_j are products of disjoint pairs of transpositions whenever $i \neq j$. Thus $c_i \cdot c_j = (p_1 \cdot q_1) \ldots (p_{n/4} \cdot q_{n/4})$. But $p_k\ q_k = q_k\ p_k$, and $c_i \cdot c_j = (q_1 \ldots q_{n/4}) \cdot (p_1 \ldots p_{n/4})$ or $c_i \cdot c_j = c_j \cdot c_i$. Conversely, let (ab) be a transposition of c_i and (bc) of c_j. Then $(a) c_i \cdot c_j = c$. Now suppose (cd) is a transposition of c_i and (ad) is not of c_j. Then $(a) c_j\ c_i \neq c$. Hence $c_i\ c_j \neq c_j\ c_i$ and the assertion follows by contradiction.

Theorem 4. Let c_i, c_j and c_k be products of n/2 pairwise disjoint transpositions where $n=2^m$ for some positive integer $m \geq 3$, and suppose that there is no transposition common to all of c_i, c_j and c_k. Then $c_i = c_j \cdot c_k \cdot c_j$ if when;

$$c_i = \prod_{s=1}^{2^{m-2}} (a_s b_s)(c_s d_s)(e_s f_s)(g_s h_s) \text{ and}$$

$$c_j = \prod_{s=1}^{2^{m-2}} (a_s c_s)(b_s f_s)(d_s g_s)(e_s h_s), c_k \text{ has the form,}$$

$$c_k = \prod_{s=1}^{2^{m-2}} (a_s g_s)(b_s h_s)(c_s f_s)(d_s e_s).$$

Proof. The condition given for $c_i = c_j \cdot c_k \cdot c_j$ is obviously sufficient. To prove that it is also necessary, suppose (xy) is a transposition of c_i such that $(x) c_j = w$ and $(y) c_j = z$. Now suppose $(w) c_k = z' \neq z$. Then $(x) c_i = (x) c_j \cdot c_k \cdot c_j = (w) c_k \cdot c_j = (z') c_j \neq y$. But this contradicts the assumption that (xy) is a transposition of c_i. Therefore the assertion must be true.

VI. Examples

In this section we use the characterizations given in the earlier section to construct three CCN's with ISC one from each class. Theorem 3 can be used to construct the networks asserted in Theorems 1 and 3, while Theorem 4 can be used to construct the networks asserted in Theorem 2. Three (8,3)-networks, IN_A, IN_B and IN_C are depicted in Figs. 3, 4 and 5. To construct IN_A, its c_i are so chosen that they satisfy the hypothesis of Theorem 3, that is, $c_1 = p_1 \cdot p_2$, $c_2 = q_1 \cdot q_2$, where $p_1 = (01)(23)$, $p_2 = (45)(67)$ and $q_1 = (02)(13)$, $q_2 = (46)(57)$. Similarly, $c_1 = p_1 \cdot p_2$ and $c_3 = u_1 \cdot u_2$, where $p_1 = (01)(45)$, $p_2 = (23)(67)$ and $u_1 = (04)(15)$, $u_2 = (26)(37)$. Also, $c_2 = p_1 \cdot p_2$, $c_3 = u_1 \cdot u_2$, where $p_1 = (02)(46)$, $p_2 = (13)(57)$ and $u_1 = (04)(26)$, $u_2 = (15)(37)$. Thus by Theorem 3, c_1, c_2 and c_3 pairwise commute and IN_A generates a commutative subgroup of order 8, as shown in Fig. 3. On the other hand, IN_C is constructed such that no two of its c_i commute and hence it fails to generate a group as indicated by - entries in Fig. 5. Finally, the c_i network IN_B are chosen according to the hypothesis of Theorem 4 so that $c_1 = c_2 c_1 c_2$ and $c_1 = c_3 \cdot c_2 \cdot c_3$ or $c_2 = c_3 \cdot c_1 \cdot c_3$. Thus the hypothesis of Theorem 2 is satisfied and IN_B generates the non-commutative group of order 8, which is shown in Fig. 3.

Note that Theorems 3 and 4 can also be used to test if two networks belong to the same class. If the networks are specified topologically as in Figs. 3, 4 or 5, first construct the permutations corresponding to the stages of each network. Then test if the permutations of both networks have the same characterization in the sense of either of Theorems 3 and 4. For example, if both have pairwise commuting permutations, it is clear that both belong to the class of commutative cube-connected networks and hence they are equivalent. The explicit construction of equivalence maps will be deferred to another place.

VII. Conclusions

The paper has introduced a network model and dealt with the classification of cube-connected networks with ISC using this model. It has been shown that there exist at least three classes of cube-connected networks with non-isomorphic group properties. The characterization of each class has also been provided. These results have three implications. First, there are cube-connected networks which generate all of their interconnections in one pass. Moreover, some generate commutative while some others generate non-commutative groups. Therefore, a network in one class can not simulate a network in the other class. Finally, there are cube-connected networks which fail to realize any group in one pass. Hence, they have the potential to generate groups of larger order if they are operated under a multi-pass scheme. As a further research, these results can be used to obtain network synthesis techniques for the three classes of networks presented in this paper.

Acknowledgements

I would like to thank Prof. N. Balabanian and M. Yaman-Oruc for their help and suggestions.

References

[1] Feng, T., "A Survey of Interconnection Networks," Computer, Vol. 14, No. 12, Dec. 1981, pp. 12-27.

[2] Siegel, H.J., Mcmillen, R.J. and Philip, T.M.,

T.M., Jr., "A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems," National Computer Conference, 1979, pp. 529-542.

[3] Thurber, K.J., "Interconnection Networks - A Survey and Assessment," AFIPS Conf. Proc., Vol. 43, 1974, pp. 909-919.

[4] Stone, H.S., "Parallel Processing with the Perfect Shuffle," IEEE Trans. Computers, Vol. C-20, No. 2, Feb. 1971, pp. 153-161.

[5] Pease, M.C., "The Indirect Binary N-cube Microprocessor Array," IEEE Trans. Computers, Vol. C-26, No. 5, May 1976, pp. 458-473.

[6] Batcher, K.E., "The Flip Network in STARAN," Proc. 1976 Int'l Conf. Parallel Processing, Aug. 1976, pp. 65-71.

[7] Feng, T., "Data Manipulating Functions in Parallel Processors and Their Implementations," IEEE Trans. Computers, Vol. C-23, No. 3, Mar. 1974, pp. 309-318.

[8] Barnes, G.H., et al., "The Illiac IV Computer," IEEE Trans. Comp., Vol. C-17, Aug. 1968, pp. 746-757.

[9] McMillen, R.J. and Siegel, H.J., "The Hybrid Cube Network," Proc. Distributed Data Acquisition, Computing and Control Symp., Dec. 1980, pp. 11-22.

[10] Harada, K., "Sequential Permutations Networks," IEEE Trans. Computers, Vol. C-21, No. 5, May 1972, pp. 472-479.

[11] Wu, C. and Feng, T., "The Reverse-Exchange Interconnection Network," IEEE Trans. Comp., Vol. C-29, No. 9, Sept., 1980, pp. 801-811.

[12] Lawrie, D.K., "Access and Alignment of Data in an Array Processor," IEEE Trans. Computers, Vol. C-24, Dec. 1975, pp. 1145-1155.

[13] Tripathi, A.R. and Lipovski, G.J., "Packet Switching Banyan Networks," Sixth Annual Symposium on Computer Architecture, June 1979, pp. 160-167.

[14] Orcutt, S.E., "Implementation of Permutation Functions in an ILLIAC IV Type Computer," IEEE Trans. Computers, Vol. C-25, No. 9, Sept. 1976, pp. 929-936.

[15] Lang, T., "Interconnections between Processors and Memory Modules Using the Shuffle-Exchange Network," IEEE Trans. Computers, Vol. C-25, No. 5, Mar. 1976, pp. 496-503.

[16] Adams III, G.B. and Siegel, H.J., "On the number of Permutations Performable by the Augmented Data Manipulator Network," IEEE Trans. Computers, Vol. C-31, No.4, Apr. 1982, pp. 270-277.

[17] Siegel, H.J., "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks," IEEE Trans. Comp., Vol. C-28, No. 12, Dec. 1979, pp . 907-917.

[18] Siegel, H.J. and Smith, S.D., "Study of Multistage SIMD Interconnection Networks," 5th Annual Symposium on Computer Architecture, Apr. 1978, pp. 223-229.

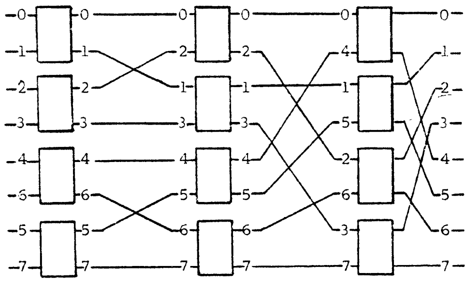
[19] Wu, C. and Feng, T., "On a Class of Multistage Interconnection Networks," IEEE Trans. Comp., Vol. C-29, No. 8, Aug. 1980, pp. 694-702.

[20] Wu, C. and Feng, T., "The Universality of Shuffle-Exchange Network," IEEE Trans. Comp., Vol. C-30, No. 5, May 1981, pp. 324-332.

[21] Fisburn, J.P. and Finkel, R., "Quotient Networks," IEEE Trans. Computers, Vol. C-31, No. 4, Apr. 1982, pp. 288-295.

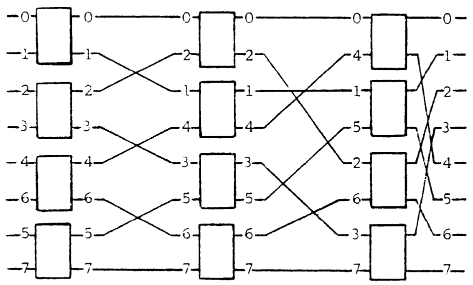
[22] Benes, V.E., "Mathematical Theory of Connecting Networks and Telephone Traffic," New York Academic Press, 1965, pp. 99-102.

[23] Graver, J.E. and Watkins, M.E., "Combinatorics with Emphasis on the Theory of Graphs," New York, Springer-Verlag, 1977, p. 25.



	e	c_1	c_2	c_3	c_4	c_5	c_6	c_7
c_1=(01)(23)(46)(57)	c_1	e	c_4	c_5	c_2	c_3	c_7	c_6
c_2=(02)(13)(45)(67)	c_2	c_4	e	c_6	c_1	c_7	c_3	c_5
c_3=(04)(15)(26)(37)	c_3	c_6	c_5	e	c_7	c_2	c_1	c_4
c_4=(03)(12)(47)(56)	c_4	c_2	c_1	c_7	e	c_6	c_5	c_3
c_5=(0536)(1427)	c_5	c_7	c_3	c_1	c_6	c_4	e	c_2
c_6=(0635)(1724)	c_6	c_3	c_7	c_2	c_5	e	c_4	c_1
c_7=(07)(16)(25)(34)	c_7	c_5	c_6	c_4	c_3	c_1	c_2	e

Fig. 3. Network IN_A and its Permutations



e

c_1=(01)(23)(46)(57)

c_2=(02)(14)(35)(67)

c_3=(04)(15)(26)(37)

c_4=(0473)(1256)

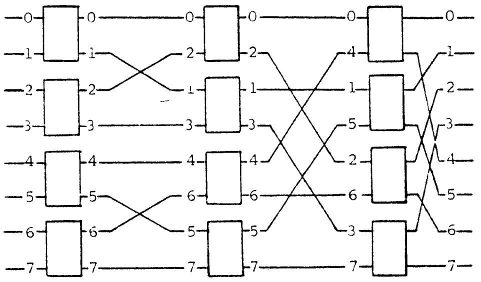
c_5=(0536)(1427)

c_6=(0631)(2457)

c_7=(1652)(34)

e	c_1	c_2	c_3	c_4	c_5	c_6	c_7
c_1	e	c_4	c_5	c_2	c_3	c_7	c_6
c_2	-	e	c_6	-	-	c_3	-
c_3	-	-	e	-	-	-	-
c_4	-	c_1	c_7	-	-	c_5	-
c_5	-	-	c_1	-	-	-	-
c_6	-	-	c_2	-	-	-	-
c_7	-	-	c_4	-	-	-	-

Fig. 4. Network IN_B and its Permutations



e

c_1=(01)(23)(45)(67)

c_2=(02)(13)(46)(57)

c_3=(04)(15)(26)(37)

c_4=(03)(12)(47)(56)

c_5=(05)(14)(27)(36)

c_6=(06)(17)(24)(35)

c_7=(07)(16)(25)(34)

e	c_1	c_2	c_3	c_4	c_5	c_6	c_7
c_1	e	c_4	c_5	c_2	c_3	c_7	c_6
c_2	c_4	e	c_6	c_1	c_7	c_3	c_5
c_3	c_5	c_6	e	c_7	c_1	c_2	c_4
c_4	c_2	c_1	c_7	e	c_6	c_5	c_3
c_5	c_3	c_7	c_1	c_6	e	c_4	c_2
c_6	c_7	c_3	c_2	c_5	c_4	e	c_1
c_7	c_6	c_5	c_4	c_3	c_2	c_1	e

Fig. 5. Network IN_C and its Permutations

THE FEM-2 DESIGN METHOD

Terrence W. Pratt
Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville, VA 22901

Loyce M. Adams
Piyush Mehrotra
John Van Rosendale
Robert G. Voigt
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665

Merrell Patrick
Department of Computer Science
Duke University
Durham, NC 27706

Abstract — The FEM-2 parallel computer is being designed using methods differing from those ordinarily employed in parallel computer design. The major distinguishing aspects are: (1) a top-down rather than bottom-up design process, (2) the design considers the entire system structure in terms of layers of virtual machines, and (3) each layer of virtual machine is defined formally during the design process. The result is a complete hardware/software system design. The basic design method is discussed and the advantages of the method are considered. A status report on the FEM-2 design is included.

Introduction

The Finite Element Machine [1,2] is an array of microprocessors, originally designed as a special purpose parallel computer for solution of problems in structural analysis using finite element methods. The authors are currently in the process of designing a successor, FEM-2, aimed at essentially the same applications.

Parallel Machine Design

In most parallel machine design, the basic hardware decisions are fixed at an early stage of the design, long before the software organization and external environment have been considered in detail. This approach often leads to major problems at later stages, where the software and external supporting environment must be distorted to match the already fixed hardware organization. The general approach of early decision on hardware, followed by later detailed software design is seen in the original FEM [1,2], and in most other designs reported in the literature, e.g., Blue CHiP [3], TRAC [4], MPP [5] to name a few. This design approach is basically a "bottom-up" approach.

In the FEM-2 design, an alternative "top-down" approach has been adopted. While the use of a top-down approach to system design is not novel, the particular form this has taken in the FEM-2 design is novel, in the context of parallel computer design. Two aspects are of note:

a. FEM-2 is considered to be composed of layers of virtual machine. Each layer defines the view of the system available to one class of users. Four layers of virtual machine are currently conceived: (1) The applications user's machine (e.g., as defined by the interactive command language), (2) the applications programmer/numerical analyst's machine (e.g., as defined by the applications language), (3) the systems programmer's machine (e.g., as defined by the operating system structure), and (4) the hardware itself (which if microprogrammed may include another layer of virtual machine).

b. Each layer of virtual machine is formally specified during the design process, using the methods of H-graph semantics [6] to construct a formal model of each layer. The advantages of this formal specification are explained below.

A virtual machine is composed of (1) various types of data objects, (2) various operations on those data objects, (3) various sequence control mechanisms for specifying the order of the operations, (4) various data control mechanisms for controlling access to data objects by the operations, and (5) storage management mechanisms for determining the placement and movement of data and code during program execution.

The FEM-2 Virtual Machines

Although complete virtual machine descriptions cannot be given here, a brief sketch will indicate the general type of results from this design approach. Considering each of the four levels of virtual machine, some typical data objects, operations, control mechanisms, and storage management methods are listed below.

This work supported in part by NASA Contracts NAS1-17130 and NAS1-17070 while the authors were in residence at ICASE. The first author was also supported in part by NSF Grant MCS78-00763.

0190-3918/83/0000/0132$01.00 © 1983 IEEE

Application User's Virtual Machine

The FEM-2 user would typically be a structural engineer using the system as an interactive workstation that allows him to store the description of a structural model, to invoke applications packages to analyze the model, and to display the results. The following is a partial list of the virtual machine components at this level.

Data objects:

Structure/substructure model
Grid description
Node/element description
Load set
Displacements of nodes
Stresses on elements

Operations:

Define structure model
Generate grid
Define elements
Solve structure model/load set for displacements
Calculate stresses
Data base operations (store model in DB/retrieve)

Sequence control:

Direct interpretation of user commands

Data control:

Workspace (user local data)
Data base (long-term storage; shared data)

Storage management:

Dynamic storage allocation for models, results, workspaces, etc.
Data movement between data base and workspace

Numerical Analyst's Virtual Machine

The numerical analyst is a research user who views the machine in terms of a high-level language that allows him to specify directly the data structures, operations and their sequences, and the parallelism in the linear algebra necessary to implement efficiently a structural engineer's application. We assume as a base a sequential language such as Fortran, Pascal, or Ada, and only mention some of the new constructs needed for effective control of the parallel processing and data distribution in the parallel system.

Data objects:

Windows on arrays (e.g., row, column, block descriptors, for remote access to non-local data)

Operations:

Tasks (programmer-defined parallel procedures)
Window operations: create window, access/assign data visible in a window
Broadcast data to a set of tasks
Linear algebra operations: inner product, vector operations, etc.

Sequence control:

Forall loops — do all iterations in parallel if possible
Pardo...end — do all statements in parallel
Task control: initiate a task, pause, resume a paused task, terminate
Remote procedure call - location determined by location of data visible in a window

Data control:

All data owned by a single task
Data accessible non-locally only via windows
Windows may be transmitted as parameters, further partitioned, stored as values of variables, etc.
Tasks may communicate through windows

Storage management:

Dynamic creation of data objects by a task
Data lifetime = lifetime of owner task
Dynamic creation of multiple task replications
Local data of a task retained over pause/resume

System Programmer's Virtual Machine

By specifying the run-time representation of tasks, their scheduling, the communication between them, and the storage representation of the data, the system programmer's virtual machine is used to implement the numerical analyst's virtual machine. The following is a partial list of the virtual machine components.

Data objects:

Code blocks/constants blocks
Task/procedure activation records (local data)
Window descriptors
Storage representations for scalars, arrays, etc.
Messages from tasks:
initiate K replications of a task of type T
pause and notify parent task
resume a child task
terminate and notify parent
remote procedure call
remote procedure return
load code/constants

Operations:

Usual sequential operations: arithmetic, procedure call, etc.
Library routines for linear algebra operations
Format and send message (one of the 7 types above)
Decode and execute message (e.g., an initiate task message may require the following steps: find code for task, allocate an activation record, copy parameters from the message queue into activation record, enter task in ready queue)

Sequence control:

Usual sequential language control structures

Data control:

Usual sequential language structures

Storage management:

General heap with variable size blocks

Hardware architecture

The requirements imposed by the upper levels of virtual machine suggest that the architecture should be chosen to effectively support:

Large scale dynamic task initiation
Remote access to local data (through windows)
Large messages (between tasks, and from a task to the operating system)
Irregular communication patterns
Large storage requirements; dynamic allocation
Fast linear algebra operations (to extract the low-level parallelism available in these operations)

In addition, several additional requirements are imposed independently:

Use off-the-shelf hardware/software if possible
Provide a way to extend the system to larger configurations easily
Provide reconfigurability to isolate faulty hardware components
Provide multi-user access

From these requirements, an architecture is evolving that is configured as clusters of processing elements organized around a shared memory. Sets of clusters communicate through a common communication network. Within each cluster, one PE runs the operating system kernel, which fields incoming messages and assigns available PE's to process them. Messages arriving in the input queue of any cluster can be processed by any available PE. Since this architecture will be described at length in other papers, no detailed design is given here.

Formal Specification of Virtual Machines

By formally specifying the data objects, operations on those data objects, control mechanisms, and storage management techniques of each virtual machine level, a detailed software/hardware design can be obtained that specifies the function of each level as well as its implementation on the next lower lever. Our research uses the methods of H-graph semantics [6] for making this formal specification. H-graph semantics is a mathematical modeling method for software/hardware systems that can be used to construct a precise mathematical model of each virtual machine level. The data objects are modeled as hierarchies of directed graphs (H-graphs) in which the nodes represent abstract storage locations and the arcs represent access paths. Data types are modeled using formal "H-graph grammars," a type of BNF grammar in which the "language" defined is a set of H-graphs representing a class of data objects. Operations (procedures) on the data objects are modeled as "H-graph transforms," which are functions defining transformations on the H-graph models of data objects. H-graph transforms may invoke each other in the usual manner of subprogram calling hierarchies to determine the overall flow of control in a model of a virtual machine.

In the FEM-2 design process, each layer of virtual machine is designed first, starting with the top layer and considering each layer as defining the requirements that must be satisfied by the design at the level below. Several iterations through the four levels are made, adjusting the design to find an appropriate mix of hardware and software at each level. As the design begins to "firm up", the individual virtual machines are defined formally. The precise formal definitions are then used as the basis for simulations of the various virtual machine levels. Simulations to measure the storage, processing, and communication patterns in typical FEM-2 applications and to determine the ease of programming the machine at the various levels are of particular importance. The ultimate result is to be a detailed design of the hardware and software, completely specified at each level in terms of its function and its implementation on the next lower level of virtual machine.

Conclusion

A major advantage of the top-down, layers of virtual machine, design approach is that it forces a design of the entire system structure, including I/O (virtual) devices, global control strategies, interfaces with the outside environment, etc. at an early design stage. It also allows the potential parallelism at various levels to be considered in detail: parallelism in user requests for simultaneous solution of several independent problems, parallelism in the substructure analysis of a larger structure, parallelism in the finer structure of solution of a particular system of simultaneous equations, etc. A third advantage is that the entire design process may be iterated, adjusting the design of each virtual machine level, until the proper match of hardware and software organizations is found.

Current Status

The FEM-2 design effort has been underway since December 1982. At present the first iteration of the design of the four layers of virtual machine is nearing completion. Several scenarios of use of the numerical analyst's virtual machine have been carried out in detail, using a detailed design of a typical algorithm to get quantitative estimates of processing requirements, storage requirements, and communication requirements for a typical large-scale application. One such analysis is reported in [7]. H-graph semantics definitions of the various levels are being constructed.

References

[1] H. Jordan, "A Special Purpose Architecture for Finite Element Analysis," Proc. 1978 IEEE Conf. on Parallel Proc.

[2] O. Storaasli, et al. "The Finite Element Machine: An Experiment in Parallel Processing," Research in Struct. & Solid Mechanics, NASA Conf. Pub. 2245, Wash. D.C., 201-217, October 1982.

[3] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," IEEE Computer, Jan. 1982.

[4] M. Sejnowski, at al. "An Overview of the Texas Reconfigurable Array Computer", AFIPS Proc. 1980 NCC, 631-641.

[5] K. Batcher, "Design of a Massively Parallel Processor," IEEE Trans. on Comps., Sept. 1980, 836-840.

[6] T. Pratt, "Formal Specification of Software Using H-graph Semantics," Rept. 83-2, Appl. Math & Comp. Sci., U. of Va., Jan. 1983.

[7] L. Adams and R. Voigt, "A Methodology for Exploiting Parallelism in the Finite Element Process," Proc. NATO Advanced Research Workshop on High Speed Computation, Julich, West Germany, June 1983, Springer-Verlag.

A MULTI-MICROPROCESSOR SYSTEM FOR CONCURRENT LISP

Shigeo Sugimoto*, Kiyoshi Agusa*, Koichi Tabata**, Yutaka Ohno*

* Department of Information Science
Kyoto University
Kyoto 606, JAPAN

** University of Library and Information Science
Yatabe-cho, Niihari-gun
Ibaraki 305, JAPAN

abstract

Recent advances of VLSI technologies have made multi-microprocessor systems feasible to construct. This paper presents a multi-microprocessor system for a LISP-based concurrent programming language, Concurrent LISP. Concurrent LISP is designed for user oriented concurrent programs, especially for artificial intelligence programs. The authors had developed Concurrent LISP on single processor systems. The multi-microprocessor system proposed here is constructed on the basis of these experiences. The multi-microprocessor system is constructed using general purpose microprocessors and it has the language oriented system configuration.

The multi-microprocessor system presented has the nine processor elements and the large common memory area. Reflecting the types of the data to be stored and their access mechanisms, each processor element has the specialized memory interface circuits, and the common area is separated into three sub-areas. The system software is distributed to all the processor elements and has the hierarchical configuration. The system software, especially the operating system, is simplified well to reduce the system overhead.

1 Introduction

This paper describes a multi-microprocessor system which has specialized memory interface circuits for list processing and multiprocessing. We have developed a LISP-based concurrent programming language, called Concurrent LISP (C-LISP) [5]. C-LISP has been developed to make use of multi-process description mechanism for the artificial intelligence problems instead of conventional description mechanisms such as backtracking and coroutines. C-LISP is user oriented, and it has simple yet flexible facilities to describe concurrent processes. We had developed the C-LISP interpreter on a large scale computer (FACOM M-200) [5] and on an MC68000 system [3]. Based on the experiences on these interpreters, we are at work on the development of a C-LISP machine composed of multiple 16-bit microprocessors (nine MC68000's) and a large common memory area (8 MB) [4].

C-LISP has flexible facilities to describe explicit parallelism. A typical example program is a multi-process search program for a game problem: multiple cooperative processes search their own paths in the search space for the game. During the execution of C-LISP programs, many concurrently executable processes are normally created. C-LISP programs need large computation power, as most of LISP programs need much computation capacity rather than I/O capacity. Multi-processor system configuration is fruitful for C-LISP, since in such configuration every processor element will be utilized well by C-LISP processes.

The multi-microprocessor system presented is an MIMD type system consisting of two different types of processor elements and a very large common memory area. C-LISP programs are stored on the common memory area and executed by the processor elements in parallel. The two types of processor elements are Master Processor (MP) for the management of the whole system and Interpreter Processors (IP's) for interpretation of C-LISP programs. The system monitor is distributed to MP and IP's. Each IP has the interpreter which is controlled by the system monitor. All IP's have the same program and have no data of processes except certain portion of processes.

We paid our attentions to two key problems for designing our system. One problem is to make the processor elements be dedicated to interpretation of C-LISP programs. Since many concurrent processes are normally created, all the processor elements are fully utilized by the processes. The other problem is the well balanced design of the system software and hardware. On the general purpose system, the software bridges the gap between the speciality of the applications and the generality of the system. Therefore, the overhead of the software is usually heavy. On the other hand, we may get powerful C-LISP system if we can construct the machine using special hardware architecture or firmware. However, it is expensive and time consuming to construct such specialized systems. We designed this system using general purpose microprocessors with small scale additional hardware.

2 Overview of the System

2.1 Concurrent LISP

Concurrent LISP is a concurrent programming language based on LISP 1.5. C-LISP completely includes LISP 1.5 as its sequential part. C-LISP has simple yet powerful concurrent functions to create a process explicitly and to write inter-process communication. The definition of a process of C-LISP is:

0190-3918/83/0000/0135$01.00 © 1983 IEEE

"A process is a self-contained entity which evaluates a given form."

Processes are activated at top-level and at STARTEVAL functions. The process activated at top-level is called the main process, and other processes activated at STARTEVAL functions are called sub-processes. A process which creates a new process is called the parent process of the created process. On the contrary, the created process is called the son process. Processes have properties such as identifiers, relationships among processes, status, mailboxes, evaluation results and so on.

The concurrent functions include three primitive concurrent functions, which are STARTEVAL for process activation, CR and CCR for interprocess communication, and also includes the basic functions for manipulation of process properties. All of these functions are designed to be fruitful for the language feature of LISP.

This definition allows users to use processes as program components in their programs like variables and procedures. Since C-LISP is designed for writing problem solving programs which require flexibility of both control and data structure, C-LISP is useful not only for writing application programs in itself but also for constructing application oriented languages on it.

The primitive concurrent functions STARTEVAL, CR and CCR are presented below. The language feature is described in detail in the references [4][5].

* starteval[$\underline{proc_1}$;$\underline{proc_2}$;$\cdots$;$\underline{proc_n}$]

$\underline{proc_i}$ = list[$\underline{name_i}$;$\underline{form_i}$;$\underline{shared_i}$],
$\underline{name_i}$ = name of i'th son process,
$\underline{form_i}$ = form to be evaluated by i'th son process.
$\underline{shared_i}$ = shared variables available for i'th son process.

When a process executes STARTEVAL, the process may activate n son processes. Each son process has its own name specified by <u>name</u> and evaluates <u>form</u>. The value of STARTEVAL is a list of names of son processes;

starteval[$\underline{proc_1}$;$\underline{proc_2}$;$\cdots$;$\underline{proc_n}$]
= list[$\underline{name_1}$;$\underline{name_2}$; $\cdots$;$\underline{name_n}$].

* cr[<u>var</u>;<u>form</u>]

A process evaluates <u>form</u> with the exclusive right to access the variable <u>var</u>. During evaluation of <u>form</u>, the process keeps the right. The value of cr[<u>var</u>;<u>form</u>] is the value of <u>form</u>;

cr[<u>var</u>;<u>form</u>] = <u>form</u>.

* ccr[<u>var</u>;<u>condition</u>;<u>form</u>]

A process waits until <u>condition</u> is neither NIL nor F, and evaluates <u>form</u> with the exclusive right to access the variable <u>var</u> shared among processes. The value of ccr[<u>var</u>;<u>condition</u>;<u>form</u>] is the value of <u>form</u>;

ccr[<u>var</u>;<u>condition</u>;<u>form</u>] = <u>form</u>.

C-LISP processes communicate with each other via shared objects. These two primitives guarantee mutual exclusion for the shared objects among the processes. (The shared objects are variables shared among processes, certain properties on property lists, and mailboxes of processes.)

The following example program shows a program for Fibonacci number. This program is not a typical one but includes several concurrent functions in a few lines. This function also shows processes can be activated recursively.

```
(FIBONACCI (LAMBDA (N)
 (COND ((LESSP N 2) 1)
       (T
        ((LAMBDA (X)
          (PLUS (FIBONACCI (SUB1 N))
                (CCR X (TERMP X) (PROCVAL X))))
         (CAR
          (STARTEVAL
           ((GENSYM)(FIBONACCI (SUB2 N)) NIL))))
 )) )
```

The meaning of this function is as follows:
If n<2 then fibonacci[n]=1.
Otherwise, create a new process which executes fibonacci[n-2]. The new process is given a name generated by gensym[] and no shared variables in the initial environments. The creating process computes fibonacci[n-1] by itself. The creating process waits until the created process terminates. (termp[x] becomes true if process x has terminated.) The creating process gets the result of the created process by procval[x]. The creating process adds these values and returns it.

2.2 Overview of the System Configuration

2.2.1 C-LISP Interpreter

Fig.1 shows the overview of the configuration of the interpreter on single processor systems [3][5]. The interpreter has two program modules, the schedule module and the interpret module. The former manages all processes, i.e., management of process activation, process switching and process termination. The latter interprets given C-LISP programs under the control of the former. For quick process switching, the interpret module is designed to load no private data of processes in itself. We call this feature "transparency" of the interpret module. Each process has its own private data on the process control block (PCB), the control stack, and the environment realized using association list (A-list) method. For the realization of the multiple control stacks and environments, linked structure is utilized well in the interpreter. Though continuous memory allocation is usually more efficient in both

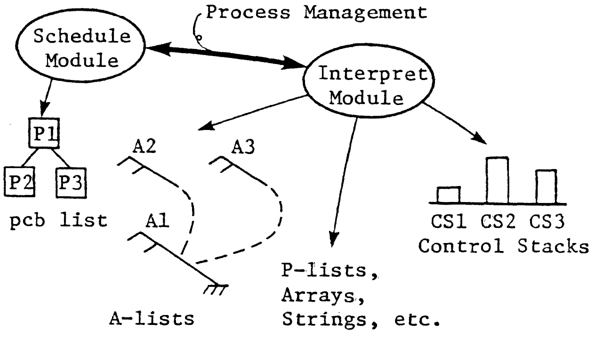


Fig.1 Overview of the C-LISP Interpreter

aspects of access speed and memory size than the linked structure, it is difficult to arrange the data of multiple processes into continuous memory space. The performance of the interpreters already developed is not so high because of the memory management task for multiple processes. The interface circuits described in this paper are designed to solve this problem.

Based on the configuration of the interpreter, we determined the basic design of the multi-microprocessor system. The followings are the basic concepts for the design.

1. C-LISP processes should be loaded on one common space.
2. Processors which interpret programs should be transparent for processes, i.e., C-LISP engine.

Consequently, followings are the basic problems which must be solved.

1. The bus bottleneck problem must be solved to connect considerable number of processors to the common bus.
2. The access methods for specialized memory areas should be reflected on the hardware configuration to improve the access speed.
3. Each processor should determine its tasks by itself to reduce overhead for the processor-processor interaction.

2.2.2 The Hardware Configuration

The multi-microprocessor system consists of nine 16-bit microprocessors, MC68000 (M68K), and a large common memory area (8MB). The processor called Master Processor (MP) manages the entire system, and other eight processors called Interpreter Processors (IP's) interpret C-LISP programs under the control of MP (Fig.2). According to the access method and access frequency, the common memory is separated into three parts, each of which has the independent common bus, to avoid the bus bottleneck.

Each processor element consists of the processor part and interface part. The former is designed as a general purpose single board computer with one M68K (8 MHz), RAM (256 KB), ROM (2/4 KB), one communication port, and IEEE 796 bus interface. Each processor has its own programs on the local memory, i.e., the system monitor functions, the garbage collectors, and interpreter functions. The latter includes intelligent interface circuits to the common memory areas, and interrupt interface circuits between processor elements. The interface circuits play the very important role in this system because they bridge the gap between the specialized information structure of C-LISP and the general purpose microprocessor.

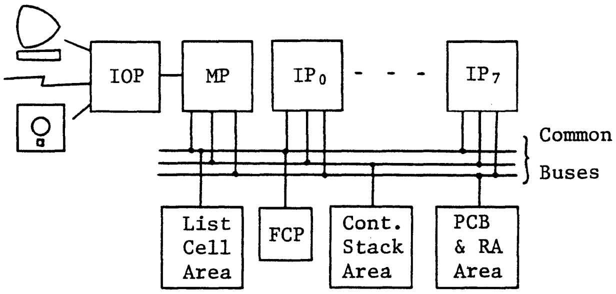


Fig.2 Overview of the System Configuration

The three common memory areas are as follows.

1. Control Stack area: Control stacks of all processes are stored. The control stacks contain control information of processes such as return address and temporary variables. The stack area is divided into 1 KB blocks. Each process has logically continuous control stack space which is composed of one or more physical blocks.
2. List Cell area: List cells are stored. This area is designed to have 1 Mega Cells. Each cell has 48 bits, 20 bits each for CAR and CDR, and 8 bits for attributes.
3. PCB and Random Access (PCB & RA) area: Non list data, such as character strings and arrays, are stored. C-LISP system also uses this area as working space for system management.

In the case of a multi-M68K system with one common bus, no more than two processors can be connected to the bus, since 62.5 % of one machine cycle is necessary for memory access. In our system, all M68K programs are stored on local memory, whose access time is shorter than the common memory, to decrease the access frequency to the common area to half or less.

For inter-processor communication, this system has the interrupt signal lines between MP and IP's, i.e., star-connection configuration whose center is MP. The interrupt lines are used for synchronization of the processors, and the communication messages are put on the interrupt message buffers on the PCB & RA area. The usage of the interrupt lines are restricted to several purposes, which are described in the later section, because of the overhead for the synchronization.

2.2.3 The Software Configuration

The software which works on the multi-microprocessor system is composed of

1) User Programs,
2) Interpreter,
3) Garbage Collector, and

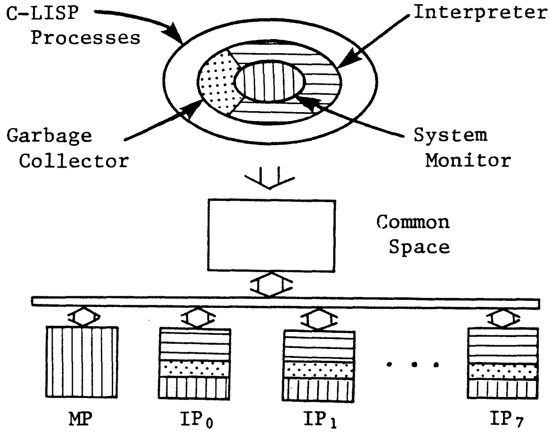


Fig.3 Layered Configuration of the Software

4) System Monitor.
Fig.3 shows the layered configuration of the software and the relationships bewtween the software components and hardware components.

1) User Programs

User programs are put on the list cell area. During their execution, the interpreter creates and puts information necessary to execute user processes on the common memory, i.e., association lists and property lists on the list cell area, control stacks on the stack area, and arrays, strings, large numbers and process control blocks on the PCB & RA area.

2) Interpreter

The interpreter on every IP executes users' programs on the common memory. The interpreter should not possess data of user processes on the local memory for quick process switching. This feature is called "transparency" of the interpreter.

3) Garbage Collector

This system has the garbage collectors for every common memory area. The garbage collectors are invoked by the events indicating shortage or exhaustion of memory cells. The PCB & RA area garbage collector squeezes garbages out of allocated area, and reclaims free area. The stack area garbage collector is invoked by exhaustion of the stack blocks, and finds the garbage blocks among the allocated ones. The list cell area garbage collector is invoked by the FCP interrupt which indicates that the exhaustion of list cells will come soon.

List cell area garbage collection is designed to be performed by all processors in parallel. We use a modified mark-and-collect algorithm for our system. The garbage collection is performed in two phases: in the first phase, each IP marks active cells from the roots of the processes allocated to itself, and in the second phase, each IP collects unused cells in its allocated portion, which is an eighth part of the whole area. MP arranges the synchronization of these activities at each phase, and restores the collected cells to FCP. All IP's execute their tasks in parallel under the control of MP. In both phases, the load of the tasks is distributed to all processor elements, so that the response time of the garbage collection is improved. Since no bad effect is caused by overwriting marks on the same cells, IP's need not access cells exclusively for marking and may mark the same cells twice or more. As the exclusive access usually takes a long time, this is an advantageous feature. The on-the-fly garbage collection algorithm [2] is not introduced, because it obliges IP's to load no roots for marking at all. Such complete transparency is considered harmful for the system performance.

4) System Monitor

The functions of the system monitor (or the operating system) are distributed to MP and IP's. MP portion mainly performs housekeeping tasks, and IP portion performs monitoring of user processes. The MP portion manages state transition of waiting and suspended processes, receives requests from IP's and IOP (I/O Processor), and executes the requested functions. On the other hand, the IP portion selects a process and executes it. Communication between these portions, i.e., intra-OS communication, is performed via either interrupt interfaces between MP and IP's or message buffers on PCB & RA area. The key problem for the design of the system monitor is to let IP's work as freely as possible. Therefore, the direct intra-OS communication via the interrupt interfaces should be restricted only to real time communication to stop or to suspend execution of running processes. The strategies to manage processes and processors are presented in another section.

3 Intelligent Interface Circuits

The multi-microprocessor system is composed of general purpose microprocessors. Though memory area is separated to avoid bus bottleneck, we must provide specialized interface circuits between the processors and the common memory components because objects stored in the common memory have specialized data structure and access mechanism which may be different from those of the conventional microprocessor.

3.1 List Cell Interface

3.1.1 Basic Idea

The followings are the basic requirements for the C-LISP machine for efficient list cell access.

Address Translation: Since a list cell usually has three portions in it, CAR, CDR and attributes, the length of a cell is rather longer than the bus width of processors. A list cell should be accessed using cell address to decrease the overhead for address translation by processors.

Quick List Read/Write: Quick list read/write operation is indispensable for quick access to a variable on an association list (A-list) and fast list manipulation. For quick list manipulation, the overlapped list read operation, i.e., list pre-fetching, will work well, since the next cell to be operated may be found on the interface registers.

Free Cell Pointer Circuit (FCP): The pointer to the top of free cell list must be accessed exclusively. Since heavy overhead is inherent in the arbitration of the pointer by software, we should provide special purpose circuit, which always possesses the current top of the free cell list and automatically updates it to avoid the overhead.

Quick list cell manipulation is important for the C-LISP system to reduce the system overhead for the memory allocation problem. In sequential LISP systems, continuous data structures are used for efficiency, e.g., CDR-coding [1], shallow binding and deep binding implemented using stacks. However, to manage continuous structures is difficult in the case of multiple process/processor systems. For example, our system uses A-list method to make environments of processes. Though A-list method is said inefficient compared with other sophisticated methods, it is considered more efficient to put multiple environments on the common area. Moreover, if we can get quick list access facilities, we can make other components of the interpreter in the form of lists because of the flexibility of list structure.

3.1.2 the List Cell Interface Circuit

The cell interface circuit has 16 IFR's, the sequence control logic, and bus interface logic. Fig.4 shows the concepts of the overlapped operation and the configuration of the interface circuits. The width of an IFR is 24 bits consisting of 20 bits cell address (i.e., up to 1 MCell) and 4 bits attributes. Since one cell consists of 48 bits, two IFR's are occupied by one cell. The IFR's are composed of high speed TTL memory chips. The cell interface command is encoded as an absolute address in the M68K's memory space. Fig.5 shows the instruction schema. The cell command has two attributes and four IFR fields. The first IFR field, i.e., to-CPU field, specifies the IFR to/from which M68K transfers data. The second field, i.e., Cell-Address field, specifies the IFR which contains the cell address to be accessed. The third and forth fields, i.e., CAR and CDR fields, specify IFR's to/from which CAR and CDR data are transfered from/to the list cell area. The R/W field specifies the direction of data transfer between IFR's and the list cell area. (R/W=R means that data is read from the list cell area, and R/W=W means the reversed direction.) The M/N field specifies that the attributes of a half cell are masked at the data bus buffer of the local bus. (M/N=M means "mask", and M/N=N means "no-mask".) The following example is an M68K's move instruction used for data transfer on the list cell interface circuit.

MOVE.L [N,R,1,2,3,4],destination

The meaning of this instruction is:

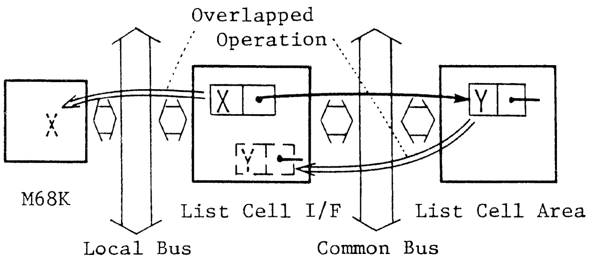
The contents of IFR1 is moved to the destination, and CAR and CDR data of the cell specified by

bit

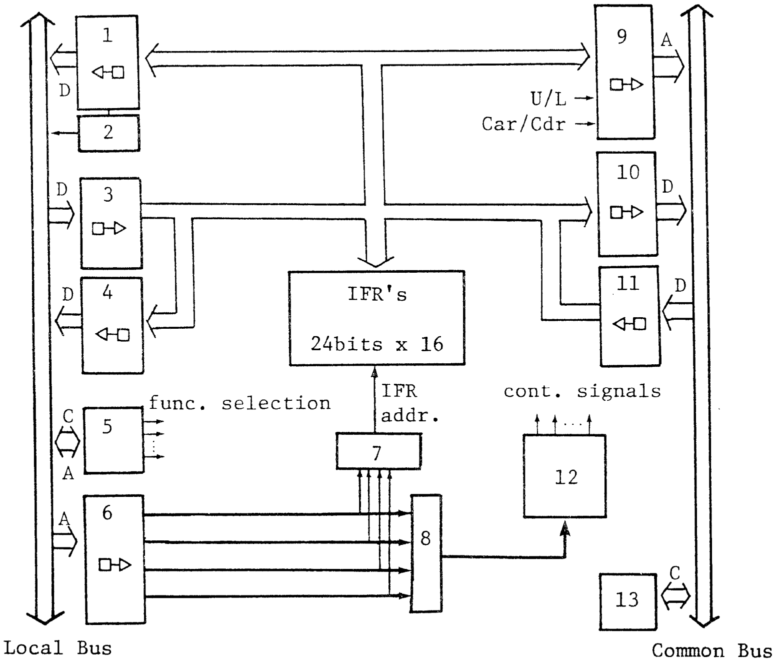
23 - 20	19	18	17 - 14	13 - 10	9-6	5-2	1,0
Sel.Cell	M/N	R/W	to-CPU	Cell-Addr.	CAR	CDR	00

23-20 - Select Cell Area
19 - Mask or Non-mask
18 - Read or Write
17-14 - To-CPU Register Field
13-10 - Cell Address Register Field
9 - 6 - Car Register Field
5 - 2 - Cdr Register Field
1 - 0 - Always zero for long word operation
* If zero, no data is transfered between IFR's and cells.
** If zero, Test-and-Set operation is executed on the target cell.

Fig.5 Instruction Schema of the List Cell I/F



a. Concepts of the Overlapped Operation



1.Interrupt Address Buffer
2.Interrupt Signaling Logic
3.Local Output Buffer
4.Local Input Buffer
5.Local Bus Control Logic
6.Local Address buffer
7.Register Address Multiplexer
8.Zero Detecter
9.Common Address Buffer
10.Common Output Buffer
11.Common Input Buffer
12.Sequence Control Logic
13.Common Bus Control Logic

Note:
*U/L=Upper word/Lower word
*Both buses satisfy IEEE 796 bus specification.
*A=Addr.,D=Data,C=Command

b. Block Diagram of the List Cell Interface Circuit

Fig.4 The List Cell Interface Circuit

IFR2 are read and moved to IFR3 and IFR4 respectively.

The interface register 0 (IFR0) has the special role. If to-CPU field contains zero, it means Test-and-Set operation is executed on the cell specified by the Cell-Address field. If other fields contain zeroes, it means those fields are not used in the operation. In Fig.6, several instructions are presented. In those instructions except the Test-and-Set instrusction, the operation on the common bus starts just after the operation on the local bus has finished. Thus, this interface manages the overlapped operation of data transfer on the local bus and the common bus.

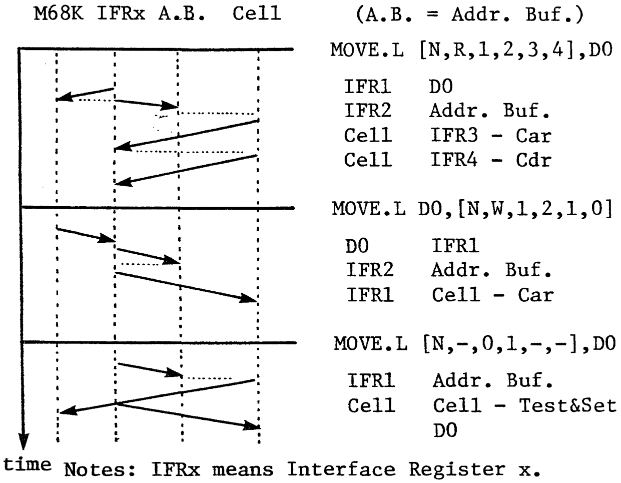
This interface circuit has the update-interrupt facility which is provided to detect an update event on a shared variable. The address of the updated cell is passed to M68K. The system monitor receives this event and executes the relevant tasks.

3.1.3 the Free Cell Pointer Circuit

This system is designed to have only one free cell list in the common cell memory. The exclusive access to the top of free cell list should be maintained to deliver a new free cell to each processor consistently. We provide the FCP circuit for quick CONS operation. It is the simplified circuit which has the register holding the next free cell address, the register holding the remaining free cell number and the interrupt logic. (Fig.7 shows the configuration.) The former register is called the free cell pointer register (FCPR) and the latter register is called the free cell counter (FCC). At each time a processor element reads FCPR to get a new cell, FCP automatically updates FCPR just after the read operation. This schema is guaranteed by giving the highest priority of the list cell bus to FCP. FCC and the interrupt logic are provided for initiating the list cell garbage collection. FCP decrements FCC whenever FCPR is read. When FCC indicates that the remaining cells are less than certain amount, FCP interrupts MP to request garbage collection. Exhaustive use of free cells is inhibited to guarantee correct response for read FCPR operation. FCP activities are summarized as follows:

1. arbitration for exclusive access to the top of the free cell list,
2. automatic update of the top of the free cell list, and
3. detection of shortage of the free cells.

These activities are simple but heavy if executed by software. Since CONS operation and the list cell garbage collection are primitive operations of LISP and CONS is executed frequently, FCP is indispensable for our system.



Notes: IFRx means Interface Register x.
"-" means "don't care".

Fig.6 Flow of Data on the List Cell Interface

3.2 Control Stack Interface

3.2.1 Basic Idea

Since the access frequency to control stack is quite high, for example about 1/5 of the whole memory access in the C-LISP interpreter on the single M68K system [3], efficient access mechanism is indispensable for this system.

The stack area is divided into 1 KB blocks, which are allocated to processes block by block. In the interpreter on the single M68K, since this memory management is performed by software, it has quite heavy overhead to test illegal access to outside area of allocated blocks. Therefore, we need memory management facilities on the processor elements to provide logically continuous space for each process. In addition to the memory management problem, local buffer memory should be provided on the processor elements to accelerate the access speed to the control stacks.

3.2.2 the Control Stack Interface Circuit

The stack interface circuit consists of three major portions; the limit registers, the buffer memory (4 KB), and the DMA control logic. The limit registers specify the upper and lower boundaries of the portion of the control stack

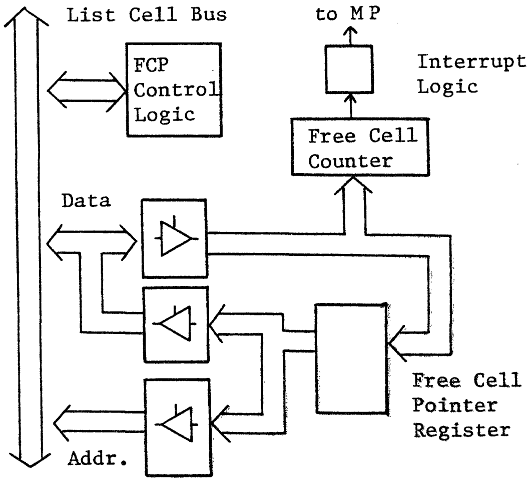
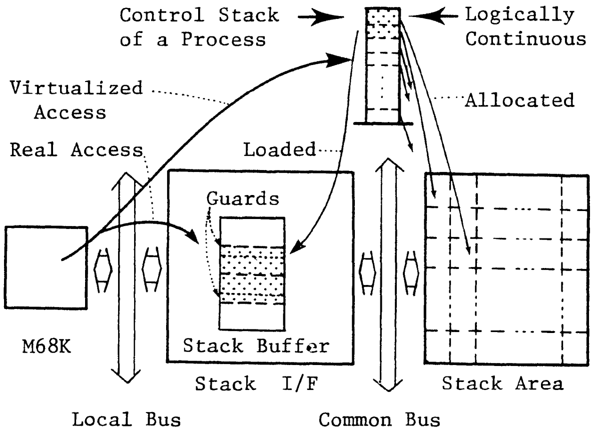


Fig.7 Free Cell Pointer Circuit

loaded on the buffer. The DMA controller transfers blocks between the buffer and the stack area. Fig.8 shows the concepts of the operation and the configuration of the stack interface circuit.

Each process is given its logically continuous stack space. The interpreter accesses to a control stack of a process using the logical address. The accessed location is always found on the stack buffer. This is guaranteed by the guard areas located at the both ends of the loaded portion on the buffer. (When either of the guard areas is accessed, the system monitor makes the next block available on the buffer.) When the system monitor transfers a block from the buffer to the common memory, it allocates a physical block to a logical block. Consequently, the interpreter is freed from the heavy overhead for stack manipulation. We use this pseudo-page-fault manipulation mechanism, since M68K has no page-fault facilities. Our method has the restriction that the processor cannot access distant location from the location accessed currently. (Since we assume the width of the guard area is 256 bytes, the processor cannot access the location whose offset from the location accessed currently is more than 255 bytes.) However, this restriction has little effect on the software, since the control stack space has the very strict locality.



a. Concepts of the Stack I/F Operation

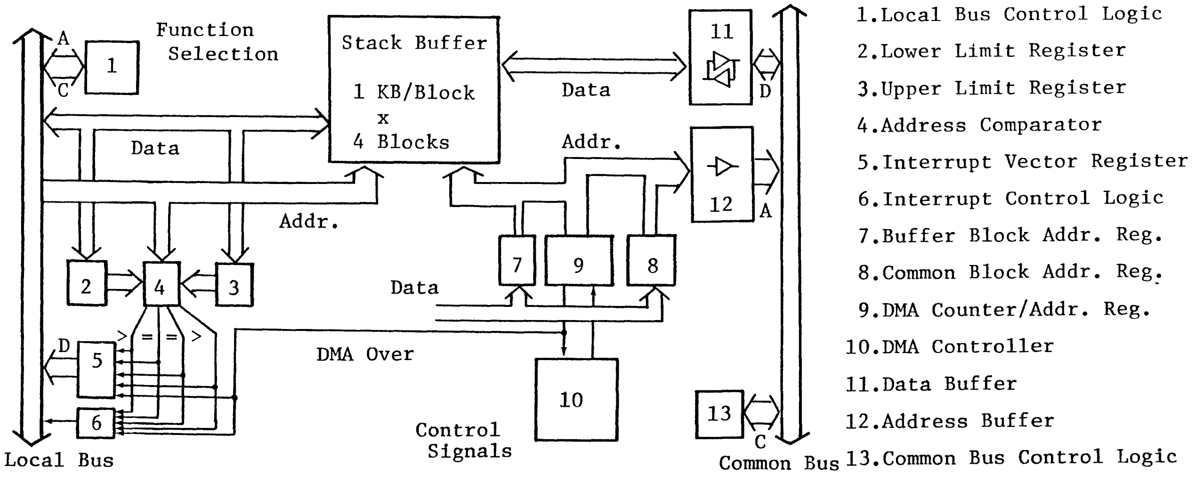
As described above, the stack interface circuit has two major functions, i.e., memory management for the logical stack spaces of processes and buffering of the active portion of the stacks for the common memory. The former feature is indispensable to realize the multiple process environments, i.e., virtualization of the users' memory spaces. The latter has the buffering effect but it disrupts the transparency of IP's. By this disruption, process switching overhead becomes rather heavy to swap out/in blocks. However, if processor elements have no buffer on the interface, the low access speed to the stack area will cause severe effect on the system performance since access frequency of the stack area is quite high. In addition, the process switching overhead is negligible, because it needs about 2 ms to switch processes while switching interval is designed to be long. (Notes: DMA controller consumes 0.5 ms to transfer one block, and four blocks are transfered in the average for each process switching. LISP interpreter usually consumes long CPU time, so that we assume one or more seconds for the interval timer which triggers process switching.)

3.3 Example Procedures

Fig.9 presents an APPEND and a SASSOC procedure, which are typical procedures for list manipulation. These procedures utilize the interface circuits well.

4 System Monitor

This system 'has the distributed system monitor. The tasks of the monitor are process management, processor management, and memory management. I/O management, which is one of the major tasks of operating systems, is executed by



b. Block Diagram of the Stack Interface Circuit

Fig.8 The Stack Interface Circuit

the I/O processor (IOP), so that it is excluded in this paper.

4.1 State Transition of C-LISP Processes

Fig.10 shows the state transition diagram of C-LISP processes. All processes are monitored and moved from state to state by the system monitor. To move a process from the waiting state to the ready state, the system monitor must test the waiting condition written in C-LISP: the system monitor evaluates the second parameter of CCR by itself. Therefore, the state transition diagram from the system monitor's view is different from the users' view as shown in Fig.10.

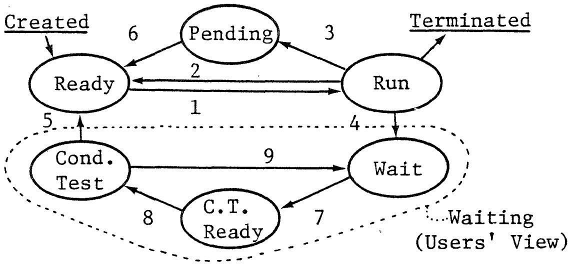
```
   MOVE.L  a,[M,-,1,0,-,-]        ; read list a
   MOVE.L  [-,R,n,1,2,3],dummy ; load first elem.
A1:MOVE.L  [M,R,2,3,2,3],A7@-   ; move elem. to CPU
   CMP.L   #NIL,[M,-,3,0,-,-]  ; end of list ?
   BNE     A1
   MOVE.L  b,[M,-,3,0,-,-]        ; set b on IFR3
A2:MOVE.L  A7@+,[N,-,2,0,-,-]  ; move elem. of a
   MOVE.L  FCP,[M,-,1,0,-,-]    ; get a cell
   MOVE.L  [M,W,1,1,2,3],[N,-,3,0,-,-]
             ; CONS & set current list top on IFR3
   CMPA.L  #BOTTOM,A7             ; termination test
   BEQ     A2
     finished
```

a. Append - append list b to a.

```
   MOVE.L  a,[M,-,1,0,-,-]        ; set A-list
   MOVE.L  [-,R,n,1,3,4],dummy ; get first pair
S1:CMP.L   #NIL,[M,R,4,3,1,2]  ; termination test
   BEQ     S2
   CMP.L   [M,R,1,4,3,4],x       ; variable match ?
   BNE     S1                      ; not match
   MOVE.L  [N,-,2,0,-,-],value ; get value
     finished
S2:CMP.L   [M,R,1,0,-,-],x       ; last var. match ?
   BNE     error                   ; not match -> error
   MOVE.L  [N,-,2,0,-,-],value ; get value
     finished
```

b. Sassoc - find a dotted pair whose CAR is x in a.
("-" means don't care.)

Fig.9 Example Procedures



1.Selected for execution 2.Timeout
3.Suspended for I/O, etc. 4.CCR wait
5.Condition satisfied 6.I/O completed, etc.
7.Shared var. updated, etc. 8.Selected for testing
9.Condition unsatisfied

Fig.10 State Transition Diagram of User Processes

4.2 Allocation of Processes to Processors

C-LISP processes are created by MP and reside in the common memory area. For the process management task, the system monitor has process queues and process pools for every state. Fig.11 shows the relationships of those queues and the system monitor. It shows that MP executes the housekeeping tasks, and IP's execute the interpretation tasks. MP creates new processes, watches events, moves a process from pending/wait to ready/condition-test-ready, and monitors process termination, and IP's select processes and execute them.

4.3 Inter-process Communication

C-LISP processes communicate with each other via shared objects. All the shared objects are put on the common memory, so that they are always visible from MP and IP's. Communication messages are, therefore, buffered on the objects, and no direct communication between processor elements is necessary. Mutual exclusion for the shared objects is controlled by flags located at the shared objects: whenever IP's interpret CR or CCR, they test-and-set those flags.

The system monitor synchronizes the processes for the inter-process communication. The synchronization operation is triggered by event messages put on the request message queue. IP's put the event messages on the queue when they recognize the local events such as update events of shared objects and release events of shared objects.

4.4 Inter-processor Communication

Since the system monitor is distributed to MP and IP's, the components communicate with each other for cooperation. As this communication is performed inside the system monitor, it is called Intra-OS communication. This system has two types of the intra-OS communication; the indirect communication via the request message queue and the direct communication via the interrupt interface circuit. (Fig.12 and Table 1 summarize the intra-OS communication.) The former is mainly used to transfer request messages of processes and event messages to MP. This type of communication

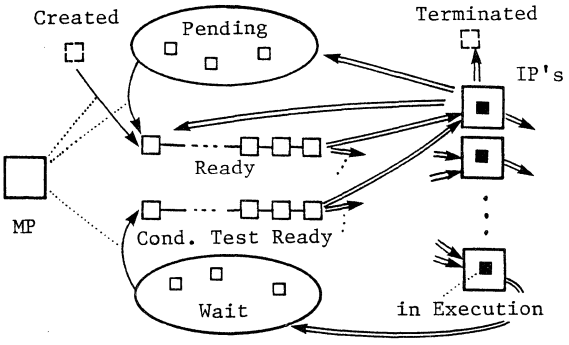


Fig.11 Process Management

is one directional, since MP directly replies to the relevant processes. The latter is used for very restricted purposes, while the former is used for many purposes. To let IP's be dedicated to interpretation, the direct communication is used only for the messages which must be transfered as soon as possible.

4.5 Memory Management

The memory management is very simple because no protection mechanism is needed. This system has the simple memory allocation and reclamation facilities. Each common area has memory allocation status descriptor which has the master information necessary for memory management, e.g. FCP. C-LISP processes get memory cells under the control of the managers of these descriptors. On the other hand, the system has the garbage collection facilities to reclaim the garbages in the allocated area as described earlier.

5 Discussion and Conclusion

In this paper, we presented the special purpose machine which comprises general purpose microprocessors. C-LISP was designed to apply multi-process description techniques to artificial intelligence problems instead of the conventional techniques such as backtracking and coroutines. From the experience on the interpreters developed earlier, the authors found that C-LISP programs usually have enough parallelism for implementing them on multiprocessor systems, in addition to the fact that LISP programs usually require very large computation power.

The hardware configuration of this system strongly reflects the language feature of C-LISP. However, our system includes several key problems common among multi-microprocessor systems, such as the bus bottleneck problem and the multiprocess environment problem. We chose general purpose microprocessors, since it was considered expensive and time-consuming to make special purpose processors by hardware or by firmware. Thus, the system is constructed using the general purpose microprocessor with small yet powerful circuits for special purposes.

Special purpose multi-processor systems consisting of general purpose processors will become more popular. We consider specialized small scale circuits may bridge the gap between the generality of the processors and the speciality of users' application on such multi-processor systems.

Table 1 Intra-OS Communication

Direct Communication (Interrupt Driven Com.)
From MP to IP Synchronization request for Garbage Collection Kill request of running processes
From IP to MP Synchronization acknowledgement of G.C. Process termination report G.C. request (intentional G.C.)
Indirect Communication (Buffered Communication)
From MP to IP none
From IP to MP I/O request Pending report of CR and CCR Event messages (Release shared objects, Update shared objects, and etc.)

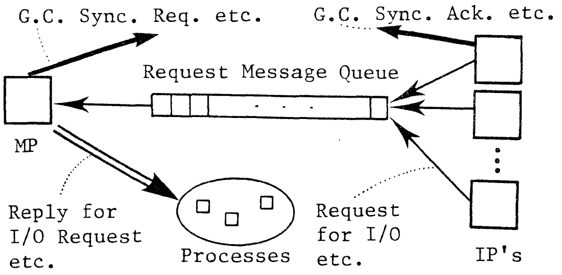


Fig.12 Intra-OS Communication

Acknowledgements

The authors thank Mr.N.Sonobe, Mr.A.Yamamura and Mr.T.Okada of Kyoto University for their contribution to this paper. The authors also thank Mr.T.Fukuda, Senior Engineer of Nihon Denki Kagaku Inc. for his help and contribution to this research.

References

[1] CLARK, D.W. and GREEN, C.C., An empirical study of list structure in LISP, Comm. ACM Vol.20, No.2, Feb. 1977

[2] DIJKSTRA, E., LAMPORT, L., MARTIN, A.J., SCHOLTEN, C.S., and STEFFENS, E.F.M., On-the-fly Garbage Collection: an exercise in cooperation, Language Hierarchies and Interfaces (Lecture Note on Computer Science 46) Springer-Verlag, 1976

[3] SONOBE, N., SUGIMOTO, S., AGUSA, K., TABATA, K., and OHNO, Y., Concurrent LISP on a Personal Computer, 24th Annual Convention of IPSJ, Mar. 1982 (in Japanese)

[4] SUGIMOTO, S., TABATA, K., AGUSA, K., and OHNO, Y., Concurrent LISP on a Multi-Micro-Processor System, IJCAI 81, Aug. 1981

[5] TABATA, K., SUGIMOTO, S., and OHNO, Y., Concurrent LISP and Its Interpreter, Journal of Information Processing, Vol.4, No.4, Feb. 1982

A Multi-Micro System for I/O Intensive Applications

F. M. Tse

American Bell
Denver, Colorado 80234

Abstract -- During the acceptance test of a computerized product, it is desirable to be able to test the load handling capability of the system. This paper describes a load generation system using a multiprocessor architecture to generate the large amount of data needed to load-test the DIMENSION® AIS™/System 85.

The system described in this paper consists of a total of 25 processors and employs a unique bussing scheme. The system uses a master-slave organization and includes several levels of hierarchically ordered busses. The host processor is used for high-level task execution and overall timing control. The 24 slave processors are used for parallel high-speed data collection and compression.

INTRODUCTION

In today's business environment, the goal of ensuring the product quality is becoming more challenging and important. During the initial acceptance test or subsequent reissues of a computerized product, it is often necessary to characterize or verify the performance of the system under heavy load. However, as semiconductor technology advances, systems are getting faster, more powerful and harder to test. Very often, it is quite difficult to generate enough load to test the performance of a system. Manual load testing has become a thing of the past and automated testing has become a must.

The DIMENSION® AIS™/System 85[a] is the latest product of the DIMENSION® family. It includes a sophisticated digital switch capable of providing both Voice and Data Management services. In order to verify the operation of the switch under heavy usage, a load-generation environment is required. The Multi-Micro System is a part of this environment - it is designed for load and system testing of telephony features associated with electronic voice terminals. To simulate the actual operating environment, the system is connected to the switch at places where electronic voice terminals are normally connected.

The system performs two main functions: 1) simulates heavy user activities on 96 electronic voice terminals and 2) verifies the responses. It implements these functions by performing the following low-level I/O operations: Voice terminal refresh data in the form of bit pulses is collected from 96 data channels (at 63Kb/s burst rate). This data is then collated, compressed, and stored into a data base for query. In addition, bit pulses simulating user activities must be transmitted back to the 96 data channels at the same time but at half the input data rate.

As part of the design objective, it is desired that the system be fairly compact, modularly structured, and easily expandable. The Multi-Micro System satisfies these requirements. It employs a distributed-intelligence multiprocessor architecture with a total of 25 processors in the system. There is one host processor responsible for high-level task execution and overall timing control. In addition, there are 24 slave processors responsible for parallel high-speed data collection and compression.

SYSTEM ORGANIZATION

The reason for choosing a multiprocessor organization is simply because of necessity. First, due to the complexity of the data compression operation, it is obvious that random logic cannot be used to implement the system and keep the size reasonably small. Second, the use of a single microprocessor is deemed inadequate because most microprocessors do not have the throughput to handle such a large amount of data. Finally, the use of a single microprocessor to interface with each of the channels would be excessive. Based on these limitations, the choice of using a microprocessor to interface with four data channels is a reasonable compromise - it requires each of the processors to process a reasonable amount of data, but not at an unattainable rate.

While there are many different approaches to building a multiprocessor system (references [1] - [3] describe several different alternatives), an interesting hierarchically ordered interconnection scheme is used. Figure 1 shows a block diagram of the system organization and the following list summarizes some of the important characteristics of the system:

- The system uses a tightly coupled, master slave type multiprocessor organization. There is one Intel 8086 host processor and twenty-four Intel 8089 slave processors[4] in this system. Each of the processors has a private local control store for efficient, independent, and parallel operation.
- There are two levels of hierarchically ordered busses in this system - one trunk bus and two branch busses. These busses are used for interprocessor communication only.
- The host processor resides on the trunk bus and the slave processors are connected by branch busses. A global memory communication scheme is used between the host processor and the slave processors.
- Each of the slave processors operates two direct memory access (DMA) channels simultaneously. One reads the input data stream while the other transfers data to the

[a] ® Registered Trademark of AT&T. ™ Trademark of AT&T.

0190-3918/83/0000/0144$01.00 © 1983 IEEE

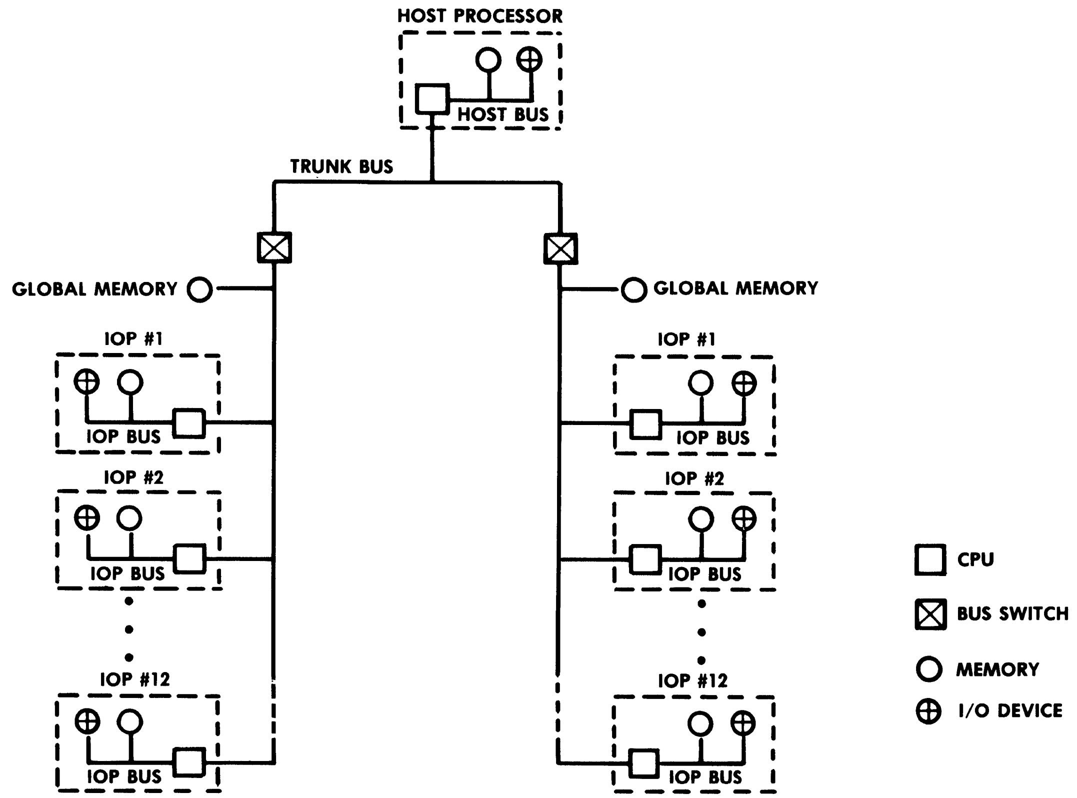


FIGURE 1. SYSTEM ORGANIZATION AND BUSSING SCHEME

output. Furthermore, there are four data data streams multiplexed into each of the DMA channels.

With these characteristics in mind, the system architecture can be described as follows. There is one host processor in this system and it is connected to a trunk bus. The trunk bus is in turn connected to two branch busses via branch bus gateways. For simplicity, the branch bus uses a simplified version of the Intel MULTIBUS[b] interface. It is a multi-master global bus and there are 12 slave processors connected to each of the branch busses. Moreover, either the host processor or one of the 12 slave processors residing on that bus can become the bus master. All data transfers on this bus are 16 bits wide and it has a worst-case transfer rate of over 800K words per second.

On each of the branch busses, there is a block of 16K bytes of global memory. Since this memory can be accessed by any of the bus masters, it can be accessed by either the host or any of the 12 slave processors. To simplify the system hardware, a simple software communication scheme is used - the host processor is restricted to communicate with the slave processors through the global memory only.

During normal op.eration, the slave processors continuously update the data base stored in the global memory. When the host processor needs information about the system, it looks up the proper location in the global memory. Likewise, when the host processor wants to issue an I/O command to a slave processor, it writes the command into specific locations in the global memory.

This system architecture was chosen for this application because it exhibits the following characteristics:

- Information compression - one of the inherent characteristics of a hierarchically ordered system is the ability to filter out unnecessary information before passing it up the hierarchies. In this system, the slave processors perform this filtering function. In fact, the ratio of data read from the channel versus those written into the global data base is approximately 1000 to 1. As a result of this information compression action, the host processor control program is simplified because it does not have to perform the time critical tasks.

- Efficient communication mechanism - in this application, the combination of global memory and dual branch busses offers more than adequate communication bandwidth and requires little protocol overhead.

[b] MULTIBUS is a patented Intel bus.

- System modularity - the bussing scheme is expandable. In fact, the system is actually equipped with four branch busses, of which only two are currently used. If even further expansion is needed, the implementation can be reconfigured into a multi-dimensional bussing arrangement interconnecting multiple host processors.

SYSTEM HARDWARE

There are a total of 28 circuit boards in the system, including a CPU board, a bus control board, two branch bus gateways, and 24 IOP boards.

HOST PROCESSOR

The host CPU is an Intel 8086 operating in maximum mode[4]. It is connected to two different busses - an internal private bus called the host bus and an external bus called the trunk bus. To simplify the system timing control mechanism, to allow parallel operations, and to provide some data privacy among the processors, the host CPU performs all of its instruction fetches and local data accesses from on-board memories. The host CPU accesses the trunk bus only when it has to communicate with the IOPs.

BRANCH BUS GATEWAY

For each of the Branch Busses in the system, there is an associated circuit board called the branch bus gateway. The gateway board can be divided into three sections: the trunk bus to branch bus interface buffers, branch bus arbitration circuit, and the global memory.

I/O PROCESSOR BOARD

The IOP board contains all the circuitry of a slave processor and its associated peripherals. The processor section contains an Intel 8089 I/O Processor (IOP) operating in remote mode[4]. The IOP can access two different busses - an internal private IOP bus or the external global branch bus. Normally, the IOP performs instruction fetches and DMA operations on its local IOP bus. The branch bus is used for data base updates and communicating with the host processor only.

There are several advantages in using the 8089 IOP to control the operation of the IOP board. The IOP is specifically designed to operate as a slave to the 8086 processor. Also, the IOP has two built-in DMA controllers to facilitate high speed data transfers. Finally, the IOP can actually execute programs residing in either its private store or the global memory. During the program debugging phase, the machine code for the IOP is downloaded through the host processor into the global memory. The IOP program is transferred into EPROMs only after it has been stabilized. This feature is quite useful for the program debug procedure.

Actually, the IOP can be considered as two processors in one. There are two identical channels inside the IOP and each of the channels can operate in either programmed mode or DMA mode. In the programmed mode, a channel can operate as if it is a normal microprocessor running under the control of a channel program. Moreover, a channel can enter into DMA mode by the execution of a single instruction. In the DMA mode, the channel stops the execution of the channel program and operates as a DMA controller. It can perform port-to-port, port-to-memory, or memory-to-memory transfers at high speed.

In the current design, both channels and both modes are being used. One of the channels is dedicated to collecting data from the input and the remaining channel is dedicated to sending data to the output.

The input data stream has several unique characteristics. The data arrives in bursts at 25-millisecond intervals and the amount of data in each burst alternates between two different formats. For this reason, both of the channels are designed to operate on 25-millisecond frames. At the beginning of each frame, both channels operate in DMA mode - one channel collects data into its local memory while the other channel sends data stored in local memory to the output circuits.

When a sufficient amount of data has been collected (usually between 5 to 20 milliseconds into each frame), the system hardware generates an interrupt signal to force the IOP to leave the DMA mode. The IOP then enters into the programmed mode, runs under the control of the channel program, performs the collating and compression operations, updates the global data base, executes I/O commands from the host processor, and reenters the DMA mode again.

SYSTEM SOFTWARE

An important part of this system is the global memory communication structure. The host processor can communicate with the IOPs only through these software structures. As with all multi-processor systems that use global memory for interprocessor communication, it is important to maintain process synchronization and data privacy between the various processors. In this system, privacy is ensured by having a separate local and global memory.

The global memory consists of three major data communication structures: a system status database storage, a host-to-IOP communication interface, and an IOP-to-host processor interface. To assure data synchronization between the host processor and the IOP, several simple but effective techniques are employed. For example, some of the data structures are restricted to be unidirectional only. Other techniques employed include semaphore constructs and event counters.

The source program for the 8086 CPU was developed on a UNIX™-based[c] VAX 11/780 minicomputer. Except for the I/O drivers, most of the routines are written in the "C" programming language[5]. The

[c] ™ UNIX is a trademark of Bell Laboratories.

system is written to be task-oriented. In the current implementation a task can be in one of five states: it can be waiting for execution, in execution, sleeping, waiting for response from the IOP, or waiting on the output device.

To minimize execution time and program size, the program for the 8089 IOP is written in assembly language. It consists of two separate control programs, one for each of the channels of an IOP. Although these two channels can operate independently, the DMA input and output transfers are always kept in synchronization by signals from the external hardware. As mentioned earlier, both control programs are written to run on 25-millisecond frames. Moreover, some of the processing, such as the compressing function, is spread over several frames. This approach reduces the processing time within each individual frame and enables the IOP to satisfy real-time requirements.

SUMMARY AND CONCLUSIONS

The Multi-Micro System is currently being used for in-house system testing. We believe that the architecture described in this paper is well suited for real-time critical and I/O intensive applications. The use of private local storage in each processor and multiple global busses in the system are essential for efficient, independent, parallel operations. Additionally, the highly modular organization allows the users to add or delete test resources easily. The built-in DMA controller feature of the slave processor enhances the throughput of the system significantly and enables the designers to implement such a compact tool.

In the future, more fault-tolerant features should be added to the system. For example, overall system reliability can be further increased by adding a redundant host processor. Moreover, the host processor software should be expanded to include periodic maintenance routines to audit the health of an IOP board and restart those that have failed.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the contribution of C. A. Conkey for implementing the program for the IOP, C. Martin for sharing his experience on data structures, and D. L. Fairchild for his encouragement throughout this project.

REFERENCES

1. G. A. Anderson, and E. D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics and Examples," Computing Surveys, Vol 7, No 4, December 1975, pp. 197-213.

2. A. M. Despain, D. A. Patterson, "X-tree: A Tree Structured Multiprocessor Computer Architecture," The 5th Annual Symposium on Computer Architecture Conference Proceeding, April 1978, pp. 144-151.

3. D. Katsuki, et al, "Pluribus - An Operational Fault-Tolerant Multiprocessor," AFIPS Conference Proceedings, Vol 46, 1977, pp. 637-644.

4. The 8086 Family User's Manual, Intel Corporation, October 1979.

5. B. W. Kernighan, and D. M. Ritchie, The C Programming Language, Prentice-Hall, 1978.

PIPELINE AND PARALLEL ARCHITECTURES FOR COMPUTER COMMUNICATION SYSTEMS

ARUMALLA V. REDDI

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Powai, Bombay-400076, India

ABSTRACT

Various existing communication processor systems (CPSs) at different nodes in computer communication systems (CCSs) are reviewed for distributed processing systems. To meet the increasing load of messages, pipeline and parallel architectures are suggested in CPSs. Finally, pipeline, array, multi and multiple-processor architectures and their advantages in CPSs for CCSs are presented and analysed, and their performances are compared with the performance of uniprocessor architecture.

INTRODUCTION

Over the past few years, we have seen the emergence of a new generation of computer technology. This causes to increase the availability of low cost mini/microcomputers, intelligent terminals, etc., and further greately reduces the cost for unit of computing power. This leads to tremendous increase of computer users and their requirement for enormous computing power to assist in day-to-day human life and solve a number of problems in several areas of science and technology. The best way to meet all the requirements of present user population is through Distributed Processing Systems (DPSs).

Distributed Processing Systems

These DPSs bring the computing power nearer to the computer users irrespective of their geographical distribution. Unlike centralized computer systems, these systems are less expensive, and increase processing power and shorten instruction execution time with the addition of new computer users/terminals. The main principle of DPSs is to do the processing where the data is. This leads to a reduction of the data communication traffic, since data processing usually involves data reduction. These DPSs increase the redundancy, fault-tolerance, reliability and sharing of expensive resources. They are more flexible, versatile, dynamic, reconfigurable and expandable, encourage construction of dedicated systems, distributed databases and multi-microprocessor systems, give more throughput due to inherent pipelining and parallilsm and require less incremental cost than centralized systems. They further provide remote access to a variety of resources, access to databases, a facility for exchanging personal messages, etc. In these systems with distributed control it is more natural to communicate through messages rather than through synchronized signals. With proper autonomy and storage in the message transport system, processors do not communicate in future systems with many processors.

Computer Communication Systems

Seeing the advantages of DPSs, the demand for DPSs is growing very fast along with the computer user population. This trend increases the number of processing nodes as well as the complexity of DPSs. At the same time the exchange of messages between nodes is also increasing. To streamline the passage of messages in between processing nodes, the requirement for advance computer communication systems (CCSs) is also increasing.

With the recent advances in computer communications, protocols, computer networks, communication software, operating systems for networks, etc., many CCSs like ARPANET, ALOHA, ETHERNET, CYCLADES, etc. have come into operation and they are becoming very popular with their use in DPSs. With the present advances in CCSs, the development of DPSs is also progressing.

As the computer user population and the demand for DPSs is increasing, the demand for high throughput CCSs is also increasing. New techniques like multiprocessing [1], parallel processing [2,3] have been proposed to improve the throughput of CCSs and provide service facilities to all the computer users. At the same time the demand for high throughput communication processor systems (CPSs) [4,5] is also increasing to meet the demand for processing all the messages, at any node, within reasonable time. Recent advances in computer architectures, LSI, VLSI, VHSI has caused significant changes in the area of computer architecture. These new developments are encouraging the growth of high throughput and reliable CPSs for CCSs in DPSs [5]. Many CPSs capable of performing a wide range of communication applications including circuit switching and store-and-forward switching or a combination of these two in CCSs are under operation [5].

COMMUNICATION PROCESSOR SYSTEMS

Bell Labs have developed many systems [6-9] for stored program controlled

0190-3918/83/0000/0148$01.00 © 1983 IEEE

switching. The GTE Sylvania, Inc. has proposed a CPS for combination of circuit and store-and-forward switching [5]. A multiprocessor, ETS-4[10] for circuit switching and another multiprocessor, Cmmp [11,12] as a switching node of the ARPA network for combination of circuit and store-and-forward switching have been developed. Applying associative and parallel processing techniques the Honeywell research group [13] proposed an architecture for a combination of circuit and store-and-forward switching to implement demultiplexer and multiplexer functions. The North Electric company has proposed CPS [14] for distributed processor system. The Pluribus system [15,16] is used as a modular switching node for the ARPA network.

Looking at the evolution of the CPS architecture, we can find that the trend has been from uniprocessor architecture to multiple-processor architecture. As the communication requirements continuously increase, the complexity of CPSs will keep growing and more functional modules would be included in the system[17-19].

PROCESSOR ARCHITECTURES

At every node in a CCS, the CPS receives the messages and stores them in its memory and routes them to their next destination to reach within the shortest possible time. Before routing, the CPS performs operations like frame decomposition, control packet processing synchronization, field processing, control packet formulating, receive and transmit tables updating, and frame composition, on each message. If a single processor is used to do all these operations, it will take longer time to process all the incoming messages. The processing time can be reduced by using more than one processor arranged in pipeline and parallel processor architectures. Eventhough, theoretically, the performance of n processor system will have throughput equal to n times that of uniprocessor, due to practical limitations it will always be less than the theoretical value. The CPSs having the pipeline and parallel architectures handle very high loads and will have very high throughput rate. In this paper, some resource sharing architectures having n processors to increase the throughput of CPS are proposed and their performances over a uniprocessor are analysed.

Uniprocessor

A uniprocessor CPS performs all the operations on the messages in a sequential manner. For example, if the CPS receives m messages, each message requires n steps to complete the execution of all operations and each step requires an average execution time t_s seconds then the time required to process all the m messages on a uniprocessor is

$$t_u = mnt_s \text{ sec.} \qquad (1)$$

If different operations on each message or different messages are simultaneously executed on different processors then the overall time required to process all the messages will be reduced. This increases the speed of processing and throughput. To achieve this goal, the following pipeline, array, multi and multiple-processor architecture schemes, more suitable to execute different operations or messages simultaneously in a number of processors, to increase the overall throughput of CPSs are proposed.

Pipeline Processor

In this scheme all the operations to be performed on each message are partitioned into n steps of equal average execution time t_s and all the steps are executed in n different processors on n different messages simultaneously. All the n processors are arranged in series, like a pipeline and the messages pass through all the n processors one after the other in nt_s seconds, completing the execution of all the n steps of operations on all n processors. The total time required to process m messages in a pipeline processor architecture having n processors is

$$t_p = (m+n)t_s \text{ sec.}$$

$$\approx mt_s \text{ sec. for } m >>> n \qquad (2)$$

Comparing equations (1) and (2), one can see that the pipeline processor requires only 1/n of the time required by the uniprocessor. The average message process delay, in this scheme, is equal to 1/n of the message process delay caused by a uniprocessor and its throughput will be n times that of a uniprocessor. Since the pipeline scheme has n processors and all are connected in series, a processor failure will effect the whole scheme and so its reliability is much less than uniprocessor. The higher reliability and throughput can be achieved by applying multiprocessor technique.

Multiprocessor

In this scheme, all the operations to be performed on each message are partitioned in to independent steps and all the possible independent parallel steps are simultaneously executed in n different processors on the same message. Since the execution time of all the steps are not equal it requires complicated scheduling to get optimum performance from all the processors. In this case, throughput will be nearly equal to n times that of uniprocessor, slightly less than that of pipeline processor. This is because the non-utilization of full n processors capacity due to practical limitations. In this case the failure of one processor will not effect the functioning of other processors.

Hence, its reliability is much better than pipeline and uniprocessors. In pipelineand multiprocessors, the expandability cost will be higher. A suitable architecture for fast increasing loads with lower additional cost for expandability is array processor.

Array Processor

In this all the n processors are arranged in the form of an array and n different messages are processed in n different processors simultaneously. An instruction is executed simultaneously in all the n processors on all the n messages present in them. The throughput of an array processor will be equal to n times than that of uniprocessor. In this case any one processor failure will not effect the functioning of other processors. This increases the reliability and it is nearly equal to that of multiprocessor. Unlike multiprocessors, in this case, addition of additional processors is easier and it does not take complicated scheduling to share all the resources optimally. The main drawback in this scheme is, at any time, two or more processors can try to access same resource. This creates additional operating system design problems. This can be overcome by using multiple-processor technique.

Multiple-processor

In this scheme all the n processors shares all the resources and operates independently. At any time, n messages are processed independently on n processors and whenever two or more processors try to access a single resource, then all those requests are put in a queue and they will be served on FCFS queue discipline. This delays only those messages whose processor requests are in the waiting queue and all the other processors continue executions without waiting. Due to this fact, the throughput of a multiple-processor will be slightly less than that of pipeline processor, but its reliability and expandability are greater than the others. The average message processing delay, in this case, is nearly equal to that of the delay caused in case of pipeline and array processors.

CONCLUSION

Out of all the above schemes, pipeline scheme has got higher throughput and least reliability. If high reliable pipeline processors are available then this scheme is very suitable where the throughput of CCS is almost steady and failure of CPS will not cost much. On the other hand, array , multi and multiple-processors have high reliability and expandability than pipeline scheme and their throughput is also nearly equal to that of a pipeline processor scheme. If one considers throughput, reliability and expandability, then the best suitable scheme will be multiple-processor architecture.

REFERENCES

[1] F.E. Heart, et al., A new minicomputer/multiprocessor for ARPA network, AFIPS Conf.Proc.(June 1973),529-537.

[2] D. Cohen,et al., A parallel processing approach to computer communication, in R.L. Grimsade and F.F. Kuo (eds.), Computer Communication Networks (1975), 181-194.

[3] A. Faro, et al., Parallel processing in computer communications, Proc. Int. Conf. on Parallel Processing (August 1981), 294-296.

[4] C.B. Newport, et al., Communication processors, Proc. IEEE, Vol.60, (Nov.1972),1321-1332.

[5] D.P. Agrawal, et al., A survey of communication processor systems, Proc. COMPSAC (Nov.1978),668-673.

[6] W. Keister, et al., No.1 ESS: system organization and objectives, BSTJ, Vol.43 (Sep.1964), 1831-1844.

[7] P.C. Richards, et al., No.2, ESS: An electronic switching system for the suburban community, Bell Lab. Record, Vol.51 (May 1973), 130-135.

[8] E.A. Irland, et al., New developments in suburban and rural ESS (No.2 and No.3 ESS), Int. Switching Symp. Rec. (Sep.1974),512/1 - 512/6.

[9] A.E. Ritchie, et al., No.4 ESS system objectives and organization, BSTJ, Vol.56 (Sep.1977), 1017-1028.

[10] J.J. Dankoweki, et al., ETS-4 System overview, IEEE Int. Conf. on Comm. Rec., Vol.1 (June 1975),14/1-14/6.

[-1] M. Barbacci, et al., The application of multiple processor computer systems to digital communication network, CMU report (June 1976).

[12] W.A. Wulf, et al., Cmmp: A multiprocessor, Proc. of FJCC(1972),765-777.

[13] H.G. Schmitz, et al., Application of associative processing techniques to an integrated voice/data switching network, Honeywell systems and research centre report, (1976).

[14] North Electric Company, Communications processor system, Vol.I-VIII,TR (1977).

[15] S.M. Ornstein, et al., Pluribus-a reliable multiprocessor, Proc. AFIPS Nat. Comp. Conf. (May 1975),551-559.

[16] J.L. Baer, Multiprocessing systems, IEEE Trans. on Comp. Vol.C-25 (Dec.1976),1271-1277.

[17] C.H. Sequin, Message switching circuits for multi-microprocessors, Proc. COMPCON Spring (Feb.1980),328-334.

[18] K. Hwang, et al., Computer architectures for parallel processing, McGraw-Hill, NY (1983).

[19] E.T. Fathi, et al., Multiple microprocessor systems: what, why, and when, IEEE computer, Vol.16,(March 1983), 23-32.

AN INTERFACE MESSAGE PROCESSOR
WITH A MULTIPROCESSING ARCHITECTURE

Krish Purswani
Department of Computer Science
Bijan Jabbari
Department of Electrical Sciences and Systems Engineering
Southern Illinois University, Carbondale, Il-62901.

ABSTRACT

This paper describes the architecture of a multiprocessing system for voice/data switching. The system consists of a large number of units called "Processing Elements," which have specialized and dedicated functions. Due to their parallel operation, a high degree of concurrency can be achieved.

The architecture of the system is described. Simulation results pertaining to memory access times are presented. The operating system is described. Certain analytical results pertaining to blocking probabilities of packets, and utilization of the Processing Elements are discussed.

INTRODUCTION

The last decade has witnessed a tremendous growth in the amount of information that is transmitted digitally. Besides the digital traffic generated as a result of computer communication, applications like digitized voice, electronic mail, etc. have further intensified the need for efficient transmission of digital data. Integrating these various kinds of data onto a single channel is an attractive solution for an Integrated Services Digital Network due to the greater channel utilization that is thus accrued [1]. However, due to a variety of switching schemes that are found to be efficient for different types of data [2], and due to the differences in protocols used by different hosts, a fairly sophisticated switching and routing center is required. Advances in VLSI have made it feasible to realize systems with distributed architectures, making integrated switching schemes economically viable [3],[4].

SYSTEM ARCHITECTURE

It is desired to have an Interface Message Processor capable of handling data generated by sources of different kinds. These different kinds of data need to be integrated in a way that would optimize certain system parameters (cost, for instance) subject to certain system constraints, such as delay, error rate, etc. To achieve high throughput, and to permit many different switching strategies to be implemented concurrently, a multiprocessing architecture is desirable [4].

The system that we describe has a large number of units called "Processing Elements" (PE's). A set of PE's forms a "cluster." There are 25 "clusters" in the system. Each "cluster" consists of a PE called the "Circuit Switching Element" (CSE) and four other PE's called the "Packet Processing Elements" (PPE's). These five PE's in a "cluster" are interconnected over a bus, called the intra-cluster bus. Each "cluster" has a local memory called the "cluster memory," which it uses for storing instructions and data. Each "cluster" exercises direct control over one Switching Matrix. The CSE is responsible for controlling the switching operations in the Switching Matrix, whereas the PPE's are in charge of buffering and transmitting packets.

The "clusters" are again connected, via a global bus, to three memory banks (shared memory) and a CPU (Fig. 1). Two devices, called the Bus and Interrupt Controller, and a Bus Multiplexor, are used for efficient management of the global bus and the memory banks respectively. The PE's in a cluster use a token ring protocol to govern the use of the intra-cluster bus.

The CPU acts primarily to disburse resources for the different tasks in an orderly manner. It constructs the "Switch Control Words," which are executed by the CSE's. These are similar to the Channel Commands in a general purpose machine. The Switch Control Words are prefetched by the CSE into its cluster memory, which, thus, acts as a cache memory for the CSE. In executing the Switch Control Words, the CSE connects one line to another (circuit switching) or a line to a Packet Processing Element (when packets need to be buffered). The number of lines is more than the number of Packet Processing Elements. This is due to the fact that sometimes, certain kinds of data may be circuit switched. A set of Switch Control Words forms a "Switching Program," which, when executed, implements a specific switching scheme. The CSE also determines the manner in which packets would be received by the PPE, and so on.

PERFORMANCE ANALYSIS

The system was simulated using GPSS-V. Specifically, it was desired to estimate the time required for an access from the shared memory. It was observed that, for a memory with a cycle time of 250 ns., the CPU, operating in a high-priority mode, was able to access the memory within 300 ns. 77% of the time, and the Processing Elements only 33% of the time [4].

Certain analytical results pertaining to the probability of blocking were derived. For the

0190-3918/83/0000/0151$01.00 © 1983 IEEE

purpose of analysis, it was assumed that a switching matrix had N lines, and the cluster that controlled it had M PPE's. It was further assumed that the traffic could be divided into two broad categories, viz., circuit switched traffic (henceforth referred to as voice-streams) or packet-switched traffic (henceforth referred to as data packets). Fig. 2 shows the blocking probabilities for circuit-switched traffic (voice-streams) and packet-switched traffic (data packets), as a function of the number of PPE's per cluster. It is observed that as the number of lines present increases, the blocking probability of a data packet increases. This is because as the number of lines increases, the blocking probability of voice-streams (i.e. circuit-switched traffic) decreases, and this makes the lines less available for data packets. Blocking could occur for both kinds of traffic. Fig. 3 shows the average number of PPE's in use as a function of the number present in a cluster. The ratio of the average number of PPE's in use and the number of PPE's present is the utilization of the PPE's. It is observed that as the number of lines and the amount of data traffic increases, the average number of PPE's utilized increases, and approaches the ideal limit depicted as a straight line in the figure. This situation corresponds to 100% utilization of the PPE's. For other cases, the average number of PPE's utilized increases with the number of PPE's present, but only upto a certain point.

SOFTWARE

A group of users who wish to communicate are normally allocated lines in one Switching Matrix. This is to reduce the overhead of moving packets from one cluster memory into another. The operating system creates a series of processes for each set of users that use a particular protocol, or constitute a group whose members communicate primarily with other members within the group. Different system utilities serve to implement different protocols, which are executed to create a particular pattern of "Switch Control Words" necessary to implement a particular protocol.

A routine called the Line Allocation Module performs line allocation. Users that need to communicate frequently are allocated contiguous lines in the same switching matrix (to the extent possible). Allocation of contiguous lines to similar users and the implementation of a specific protocol by the CSE creates the effect of a dedicated system.

Another important module called the "Memory Management Module" is in charge of allocating/de-allocating the shared memory. It uses the Partitioned memory management scheme, and the best-fit algorithm. It allocates the memory so as to balance the occupancy in each bank, thus increasing concurrency in memory accesses.

CONCLUSIONS

This paper discussed the architecture of a multiprocessing system for circuit/packet switching. Some of the simulation results were also discussed. An overview of a part of the necessary switching software was presented.

APPENDIX

We address the problem of determining the PPE utilization. Specifically, if a Switching Matrix has N lines and the "cluster" that controls it has M Packet Processing Elements (M < N), what is the average number of PPE's being utilized at any instant? We also attempt to find the probability of a data packet or a voice-stream being blocked. For the purpose of analysis, a stream that needs to be circuit switched could also be viewed as a packet to be transmitted instantaneously (without buffering and without engaging a PPE). Assume that the arriving traffic can be divided into two broad categories, i.e., "packets" that need to be circuit switched (voice-streams) or those which need to be buffered (data packets). A voice-stream, hence, needs two lines, but a data packet requires a line and a PPE. We further assume that the arrival pattern is poisson, with a mean arrival rate of λ_v for voice-streams, and a mean arrival rate of λ_d for data packets. Each packet engages the line(s) or the PPE for the length of time determined by the bit rate, and its size. This is the service time, and is assumed to be $1/\mu v$ for voice-streams, and $1/\mu d$ for data packets. The parameter of interest is r, the number of lines that are tied up at any instant; and s, the number of PPE's engaged in service. This represents the situation when s data packets and (r-s)/2 voice-streams are being serviced.

Let p(n,q) be the probability of a cluster being in a state (n,q), which corresponds to 'n' voice-streams, and 'q' data packets currently being serviced by the cluster. This corresponds to the case of r = 2(n) + q and s = q. These states are depicted in the Markov-chain in Fig. 4, for the case of N = 8, and M = 4. A transition can occur from the state (n,q) to the state (n+1, q) with the probability λ_v (dt) in a time interval dt, provided (n+1, q) is an accessible state. This corresponds to an arrival of a voice-stream. Similarly, a transition could occur from the state (n,q) to the state (n-1, q) with a probability of $n\mu_v$(dt). This is because the cluster might complete servicing any one of the n voice-streams. A similar argument holds for data packets. Hence, in Fig. 4, voice-streams cause transitions along the columns, and data packets cause transitions along the rows. Each of the accessible states (n,q) satisfies the following condition:

$$0 \leq q \leq M, \text{ and } 0 \leq 2(n) + q \leq N \qquad (1)$$

Solving for the state probabilites yields the following equation for p(n,q), the state probabilites for states (n,q) that satisfy (1):

$$p(n,q) = \frac{\left(\frac{\lambda_v}{\mu_v}\right)^n \left(\frac{\lambda_d}{\mu_d}\right)^q \frac{1}{n!} \frac{1}{q!}}{K} \qquad (2)$$

$$\text{where } K = \sum_{j=0}^{M} \sum_{i=0}^{\left\lfloor \frac{N-j}{2} \right\rfloor} \left(\frac{\lambda_v}{\mu_v}\right)^i \left(\frac{\lambda_d}{\mu_d}\right)^j \frac{1}{i!} \frac{1}{j!} \qquad (3)$$

Hence E(s), the average number of PPE's utilized is:

$$E(s) = E(q) = \frac{\displaystyle\sum_{q=1}^{M}\sum_{n=0}^{\left\lfloor\frac{N-q}{2}\right\rfloor}\left(\frac{\lambda_v}{\mu_v}\right)^{n}\left(\frac{\lambda_d}{\mu_d}\right)^{q}\frac{1}{n!}\,\frac{1}{(q-1)!}}{K} \qquad (4)$$

and, E(r), the average number of lines utilized is:

$$E(r) = E(2n+q) = \frac{\displaystyle\sum_{q=0}^{M}\sum_{n=0}^{\left\lfloor\frac{N-q}{2}\right\rfloor}(2n+q)\left(\frac{\lambda_v}{\mu_v}\right)^{n}\left(\frac{\lambda_d}{\mu_d}\right)^{q}\frac{1}{n!}\,\frac{1}{q!}}{K} \qquad (5)$$

V, the probability of a voice-stream being blocked is the sum of state probabilities of states at the bottom of each column in the Markov-chain. Hence:

$$V = \frac{\displaystyle\sum_{q=0}^{M}\left(\frac{\lambda_v}{\mu_v}\right)^{\left\lfloor\frac{N-q}{2}\right\rfloor}\left(\frac{\lambda_d}{\mu_d}\right)^{q}\frac{1}{q!}\,\frac{1}{\left(\left\lfloor\frac{N-q}{2}\right\rfloor\right)!}}{K} \qquad (6)$$

and d, the probability of a data packet being blocked is the sum of the state probabilites of states in the last column (q = M) and the states for which N - (2n + q) = 0 and q < M. Thus:

$$d = \frac{\displaystyle\sum_{q=0}^{M-1}\left\{\left(\frac{\lambda_v}{\mu_v}\right)^{\left\lceil\frac{N-q}{2}\right\rceil}\left(\frac{\lambda_d}{\mu_d}\right)^{q}\frac{1}{N!}\,\frac{1}{q!}\left(1+\left\lfloor\frac{N-q}{2}\right\rfloor-\left\lceil\frac{N-q}{2}\right\rceil\right)\right\}}{K} \qquad (7)$$

$$+\frac{\displaystyle\sum_{n=0}^{\left\lfloor\frac{N-M}{2}\right\rfloor}\left(\frac{\lambda_v}{\mu_v}\right)^{n}\left(\frac{\lambda_d}{\mu_d}\right)^{M}\frac{1}{n!}\,\frac{1}{M!}}{K}$$

REFERENCES

[1] M. J. Ross and C. M. Sidlo, "Approaches to the Integration of Voice and Data Telecommunications," Nat. Telecommun. Conf.,Nov. 1979.

[2] M. J. Ross and Osama A. Mowafi, "Performance Analysis of Hybrid Switching Concepts for Integrated Voice/Data Communications," IEEE Trans. on Comm.,Vol. Com-30,No. 5,May 1982, pp. 1073.

[3] M. J. Ross, and K. A. Garrigus, "A Distributed Processing Architecture for Voice/Data Switching," NAECON 1981, Vol. 1, pp. 350-356.

[4] Purswani, K., and Jabbari B., "A Distributed Architecture System for Switching High Volume Integrated Voice/Data Traffic," Phoenix Conference on Computers and Communications, Phoenix, 14-16 March, 1983 pp. 26.

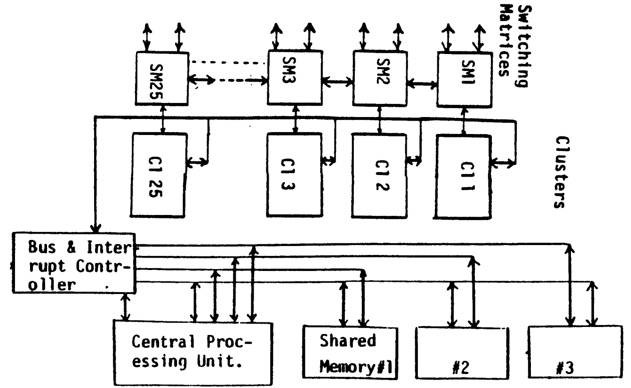


Fig.1. Architecture of the Interface Msg. Proc.

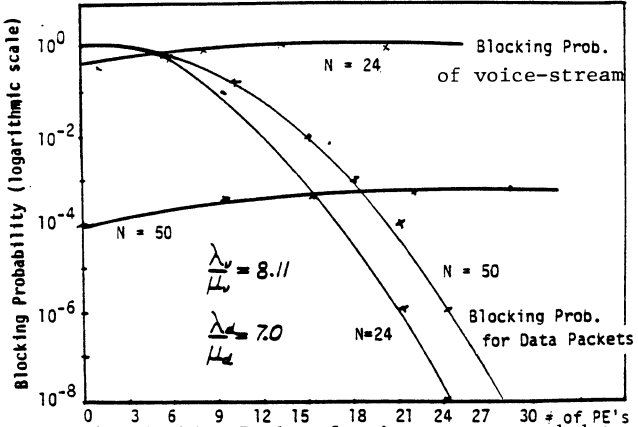


Fig. 2. Blocking Prob. of voice-streams and data.

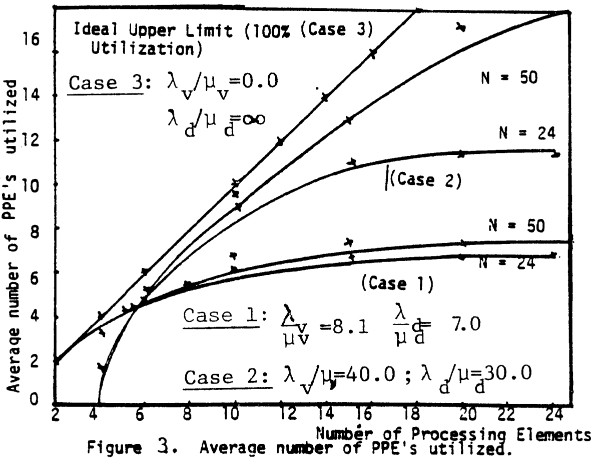
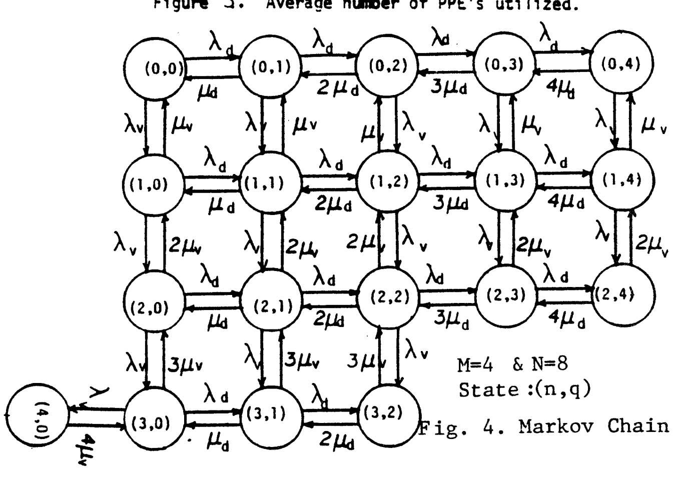


Figure 3. Average number of PPE's utilized.



M=4 & N=8
State :(n,q)

Fig. 4. Markov Chain

A CLASS OF GRAPHS FOR PROCESSOR INTERCONNECTION[+]

S.M. Reddy[†], P. Raghavan[*] and J.G. Kuhl[†]

Abstract -- Two classes of graphs with $N = 2^n$ nodes, diameter $\log_2 N$, $\approx$ N or < 1.5N links and node connectivity of 2 are presented. These graphs appear to be suitable for interconnection applications in computing networks. Even though nodes in the graphs have different degrees (at most n+1 different degrees), the graphs have structured interconnection patterns. The maximum degree of the nodes in the graphs is n.

I. Introduction

Several researchers have studied the problems of network topology for computing networks based on the graph model [1-13]. Some of the properties of graphs that are important to this application are diameter, node and link connectivity, modularity, maximum degree of a node, the number of links, routing algorithms and the effect on diameter when a node(s) or link(s) is removed. Most of the current research in network topology has dealt with regular graphs (graphs in which the degrees of all nodes are equal) or "essentially regular" graphs [1-13]. In this paper we present two classes of graphs with 2^n nodes, diameter n and node connectivity of 2. These graphs have nodes with varying degrees (2 through n). In one class of graphs, on an average, there are less than 1.5 links per node and in the other class of graphs, on an average, there is one link per node.

A brief definition of the terms used follows. The distance between two nodes, say i and j, of a graph is the number of links in a shortest path between i and j and the diameter of a graph G is the maximum of the distances between pairs of nodes of the graph. The node (link) connectivity of a graph G is the minimum number of nodes (links) that have to be removed for G to become disconnected or reduce to a single node. It is known that the link connectivity of a graph is less than or equal to its node connectivity. The t-node-deleted (t-link-deleted) diameter of a graph G is the maximum of the diameters of the subgraphs of G that are obtained by removing some t nodes (t links) [16]. The size of a graph is the number of links (edges) in it.

The diameter of a graph G gives a measure of the worst case delay in a message switched network and the connectivity of G gives a measure of the fault-tolerance of the network modeled by G.

II. Graph Construction

The degree of a node in a graph G gives the number of communication links (physical or logical) incident on a node. Clearly it is important to keep the number of communication links small to achieve the desired diameter and connectivity. Most researchers have concentrated on the construction of regular graphs with small degrees. One of the most widely studied class of regular graphs is for degree 3 [3-5,7-9,11]. A simple lower bound by Moore [14] shows that the diameter of degree 3 regular graphs of N nodes should be $\approx \log_2 N$. The best known construction achieves a diameter of $1.472 \log_2 N$ [8,11]. Note that the size of a N node regular graph of degree 3 is 1.5N. A binary n-cube, which has also been proposed as a possible topology for computing networks [13], is a regular graph of degree n with 2^n nodes and node connectivity of n. However the size of the binary n-cube is $n*2^{n-1}$. Another class of popular graphs for computing networks is loops [15]. These graphs are regular with degree 2, N links and have node connectivity of 2. The diameter of the loops is however N/2. The popularity of loops stems from the simple routing, modularity (i.e. nodes can be added and deleted without changing much of the network connections) and tolerance to single faults [15]. For the number of links, maximum degree of a node, diameter and node connectivity it can be seen that the proposed graphs fall between loops and binary n-cubes. We summarize this comparison in Table I. Before we present the details of the construction it is important to consider reasons to construct graphs presented here.

If one wants to construct graphs of small diameter and minimum number of links, one excellent solution is the graph shown in Figure 1. Diameter of this graph is 2 and its size is N-1. However the node connectivity of this graph is 1 and hence the modeled computing network has a single point failure. A graph with node connectivity 2 would require that the degree of each node be at least 2. If it is exactly 2, the graph would be a loop, with O(N) diameter. Hence to achieve smaller diameters one must allow the minimum degree of the nodes to be greater than 2. However the maximum degree of nodes and the total number of links

[+] The research reported was supported in part by Airforce Office of Scientific Research Grant AFOSR-78-3582 and NSF Grant ECS-8205188.

[†] S.M. Reddy and J.G. Kuhl are with Electrical and Computer Engineering Department, University of Iowa, Iowa City, Iowa 52242.

[*] P. Raghavan is with the Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, California.

0190-3918/83/0000/0154$01.00 © 1983 IEEE

should be kept low, to keep the complexity of the individual nodes and the complexity of the interconnections reasonable. Systematic methods to interplay maximum degree, size, diameter and connectivity of the graphs appear to be extremely difficult to obtain (this research topic is called Extremal Graph Theory [14]). The work presented in this paper can be viewed as an emperical solution to this problem.

2.1 Class I Graphs

Next we give a class of graphs with $N = 2^n$ and size less than 1.5 N. We call this class of graphs Class I Graphs. The construction procedure we give uses three basic graphs with diameters 1, 2 and 3, shown in Figure 2. The graphs with higher diameters are constructed iteratively using these graphs. Since the graphs we would construct have 2^n nodes we can use a binary n-tuple to label the nodes. The labels to be used for the first three graphs, are shown in Figure 2.

Let the labels of the nodes of the graph to be constructed be x_{n-1} x_{n-1} $x_{n-2} \cdots x_1 x_0$, $x_i \in \{0, 1\}$. The set of nodes which have identical values in some k fixed positions constitute an (n-k)-cube and is a subcube of the binary n-cube with 2^{n-k} vertices. The subcubes are denoted by placing dashes (-) in the (n-k) positions where the values are not fixed. For example 00--- represents the 3-cube {00000, 00001, 00010, 00100, 00001, 00101, 00110, 00111}. This notion of the subcube is useful in following the algorithm given next.

Algorithm Graph Construct:

Let G_n be the graph to be constructed.

1. If n = 1, 2, 3, then G_n is defined by the graphs in Figure 2. If n > 3, then for each 3-cube $x_{n-1}x_{n-2} \cdots x_3$ --- include the edges of the graph shown in Figure 2(c). Note that there will be 2^{n-3} subcubes and the edges in each subcube are placed by referring to Figure 2(c) and neglecting the first (n-3) components of the node labels in each component 3-cube.

2. Apply step 1 recursively to the (n-3)-cubes $\overset{\leftarrow\, n-3\, \rightarrow}{---\ldots-}000$ and $\overset{\leftarrow\, n-3\, \rightarrow}{---\ldots-}100$.

Figure 3 illustrates the construction procedure proposed. In Figure 4 we give an example of constructing the 32 node graph by applying algorithm Graph_Construct. Some of the basic properties of the graphs constructed above are investigated next.

2.1.1 Properties of Class I Graphs

Theorem 1: For any $n \geqslant 1$, algorithm Graph_Construct produces a 2^n node graph with diameter n.

Theorem 2: For $n \geqslant 1$, the number of links in G_n, the graph constructed by applying algorithm Graph_Construct, is

$$\frac{3}{2}\left[2^n-2^{(n(\mathrm{mod}3)) + [n/3]}\right]+ 2^{[n/3]} * (n(\mathrm{mod}3))^2,$$

where [x] is the interger part of x.

Corollary 1: The number of links in a N node graph constructed by algorithm Graph_Construct, is less than 1.5 N.

Theorem 3: In a $N = 2^n$ graph constructed by algorithm Graph_Construct, there are nodes with different degrees and these degrees range from 2 through n.

Theorem 4: For n > 1, the node connectivity of a graph constructed by algorithm Graph_Construct, is 2.

Theorem 5: For $n \geqslant 2$, the 1-deleted diameter of a $N = 2^n$ graph constructed by algorithm Graph_Construct, is at most n+2 if $n \neq 4$ and for n=4 it is 7.

2.2 Binomial Graphs

The second class of graphs being proposed are called Binomial Graphs. The construction of binomial graphs is illustrated in Figure 5. The number of nodes N in the graphs is 2^n. Two nodes, called the super nodes, have degree n and the other nodes have degrees of 2 through (n+1)/2. The nodes are arranged into (n+1) levels, with $\binom{n}{i}$ nodes in ith level, $0 \leqslant i \leqslant n$. Edges connect nodes in ith to nodes in (i+1)th and (i-1)th level only, $1 \leqslant i \leqslant (n-1)$. Furthermore every node in the ith level is connected to exactly one node in the (i-1)th level, $1 \leqslant i \leqslant [n/2]$. Every node in (n-1-j)th level is connected to exactly one node in the (n-j)th level, $1 \leqslant j \leqslant [n/2]$. For n an odd integer a node in [n/2]th level is connected to exactly one node in ([n/2] +1)th level. As long as the connection rules given above are satisfied, the details of which node is connected to which particular node(s) does not have an impact on the diameter and connectivity of the Binomial Graphs. However to keep the maximum degrees of nodes small, we also require that the edges be distributed such that the maximum difference between the degrees of nodes, at the same level, be 1. Examples of Binomial Graphs, for n=4 and 5, with the additional restriction on the distribution of edges, are given in Figure 6. Some of the properties of the Binomial Graphs are given next.

Theorem 6: The diameter of the Binomial Graphs with $N=2^n$ is n and the 1-deleted diameter is 2n-1.

Theorem 7: The node connectivity of a Binomial Graph is 2.

Theorem 8: The size of the Binomial Graph with $N=2^n$ is $N-2+\binom{n}{[n/2]}$.

Corollary 2: The size of the Binomial Graphs with $N=2^n$ nodes approaches N as $n \to \infty$.

III. Remarks

We have investigated several variations and extensions of the proposed graphs. For example it can be shown that graphs with maximum degree 5, diameter $\log_2 N$, connectivity 2 and size approximately 1.3N can be constructed. We have also constructed graphs with connectivity greater than 2.

The regular graphs studied earlier [1-13] other than loop and binary n-cube lack structure for conveniently adding and deleting nodes. Loop has the best properties for this problem, since deleting or adding a single node only perturbs at most two links. A network based on a binary n-cube can be made to have such properties by assigning longer labels, such that many labels are possibly unused. Then one can connect new nodes and delete old nodes without perturbing more than approximately $\log_2 N$ connections. The modularity properties of the proposed graphs are good. For example many nodes (3/4 nodes in Class I graphs and at least $\binom{n}{[n/2]}$ nodes in Binomial Graphs)in the proposed graphs can be deleted by perturbing at most two links. Many nodes can also be added while increasing the diameter by at most 1. These results for Class I graphs are reported in [17].

References

[1] S. Akers, "On the Construction of (d,k) Graphs," IEEE Trans. Electron. Comput., vol. EC-14, 1965, p. 448.

[2] B. Elspas, "Topological Constraints on Interconnection-Limited Logic," Switching Theory Logic Design, vol. S-164, 1964, pp. 133-147.

[3] E.P. Preparata, and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," Communications of ACM, vol. 24, no. 5, May 1981, pp. 300-309.

[4] B. Arden, and H. Lee, "Analysis of Chordal Ring Network," IEEE Trans. Comput., vol. C-30, 1981, pp. 291-295.

[5] B. Arden, and H. Lee, "A Regular Network for Multicomputer Systems," IEEE Trans. Comput., vol. C-31, 1982, pp. 60-69.

[6] D.K. Pradhan, and S.M. Reddy, "A Fault-Tolerant Communication Architecture for Distributed Systems," IEEE Trans. Comput., vol. C-31, no. 9, 1982, pp. 863-870.

[7] M. Imase, and M. Itoh, "Design to Minimize Diameter on Building-Block Network," IEEE Trans. Comput., vol. C-30, 1981, pp. 439-442.

[8] W.E. Leland, "Density and Reliability of Interconnection Topologies for Multi-Computers," Ph.D. dissertation, Department of Computer Sciences, University of Wisconsin-Madison, May 1982.

[9] K.W. Doty, "Large Regular Interconnection Networks," Proc. of the 3rd International Conference on Distributed Computing Systems, October 1982, pp. 312-317.

[10] D.K. Pradhan, "On a Class of Fault-Tolerant Multiprocessor Network Architectures," Proc. of the 3rd International Conference on Distributed Computing Systems, October 1982, pp. 302-311.

[11] M. Jerrum, and S. Skyum, "Families of Fixed Degree Graphs for Processor Interconnection," Report CSR-121-82, Department of Computer Science, University of Edinburgh, July 1982.

[12] L.N. Bhuyan, and D.P. Agrawal, "A General Class of Processor Interconnection Strategies," Proc. 9th Symposium on Computer Architecture, April 1982, pp. 90-98.

[13] H. Sullivan, T. Bashkov, and D. Klappholz, "A Large-Scale Homogeneous, Fully Distributed Parallel Machine," Proc. Fourth Annual Symposium on Computer Architecture, March 1977, pp. 105-124.

[14] B. Bollobás, "External Graph Theory," London Math. Soc. Monographs No. 11, Academic Press, London, 1978.

[15] M.T. Liu, et al., "System Design of the Distributed Double-Loop Computer Network," Proc. 1st International Conference on Distributed Computing Systems, October 1979, pp. 95-105.

[16] J.G. Kuhl, and S.M. Reddy, "Distributed Fault-tolerance for Large Multiprocessor Systems," in Proc. Seventh International Symposium on Computer Architecture, June 1980, pp. 23-30.

[17] V.R. Kode, "A Class of Inhomogeneous Graphs for Processor Interconnections," M.S. Thesis, Electrical and Computer Engineering, University of Iowa, Iowa City, Iowa 52242.

TABLE 1

COMPARISON BETWEEN BINARY N-CUBES, LOOPS AND THE PROPOSED GRAPHS

			PROPOSED GRAPHS	
	BINARY N-CUBE	LOOP	CLASS I	BINOMIAL GRAPHS (CLASS II)
NUMBER OF NODES	$N = 2^N$	N	$N = 2^N$	$N = 2^N$
NUMBER OF LINKS	$nN/2$	N	$<1.5N$	$N-2+\binom{N}{\lfloor \frac{N}{2} \rfloor}$
MINIMUM DEGREE	N	2	·2	2
MAXIMUM DEGREE	N	2	N	N
DIAMETER	N	N/2	N	N
NODE CONNECTIVITY	N	N-1	2	2
1-DELETED DIAMETER	N (FOR $N\geq 8$)	N-2	N+2	2N-1

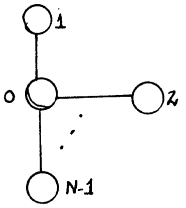


Figure 1: The graph with diameter 2 and size N-1.

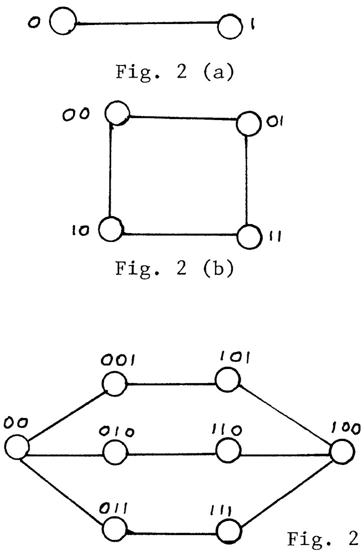


Fig. 2 (a)

Fig. 2 (b)

Fig. 2 (c)

Figure 2: Basic graphs for Class I graphs

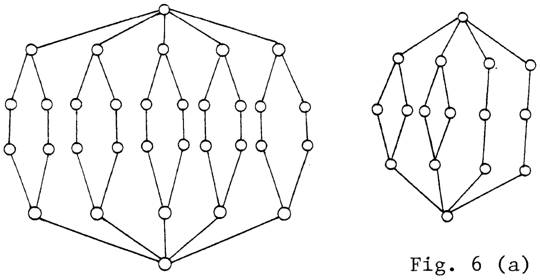


Fig. 6 (a)

Fig. 6 (b)

Figure 6: Binomial Graphs with N = 16,32

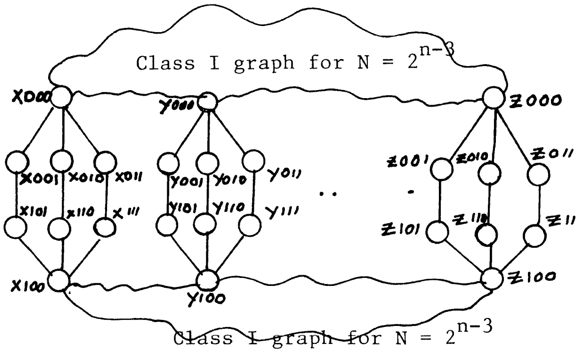


Figure 3: Class i Graph construction.

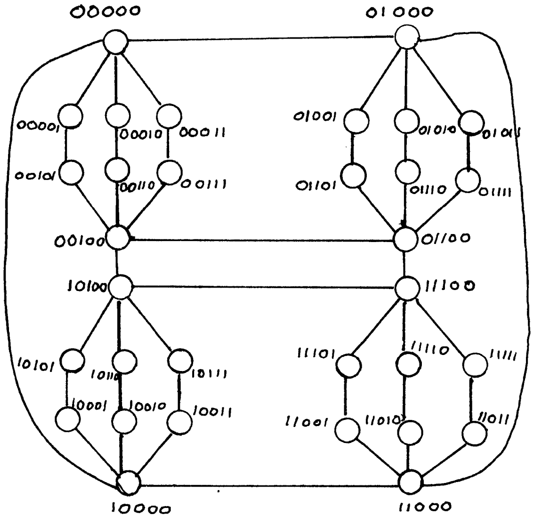


Figure 4: N = 32 Class I Graph

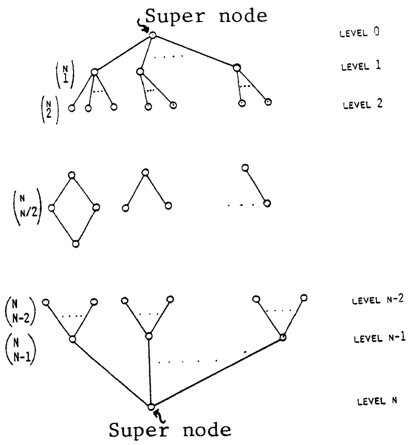


Figure 5: Binomial Graph

DENSE BUS CONNECTION NETWORKS

Karl W. Doty
Systems Control Technology, Inc.
Palo Alto, California 94304

Abstract -- In designing interconnection networks, two important criteria are minimization of the "distance" between nodes and minimization of the number of interconnections that must be made among the nodes. When direct node-to-node connections are modeled, one problem which has been extensively examined is maximizing the number of nodes subject to restrictions on the maximum internode distance and the maximum number of connections per node. This paper explores the same problem when more general bus-like connections among more than two nodes are allowed. New designs, including a generalization of de Bruijn networks, are presented with more nodes than previously proposed designs.

Introduction

There are many important factors involved in the design of a computer interconnection network. The distances between processors should be minimal, in order to have short delay times. Physical limitations usually restrict the number of connections for each processor. Network performance should not be radically affected by processor or link failures. Cost considerations may reduce the possible number of connections. Appropriate tradeoffs need to be made among these factors.

A graph theory model of a computer network, with nodes representing processors and edges representing links, has been extensively used to look at some of these factors. One problem which has been examined by several authors is the problem of finding the graph with the most nodes, given constraints on the graph's degree (the maximum number of nodes adjacent to one node) and diameter (the maximum number of edges which must be traversed between two nodes) [1-3]. In some computer systems, however, the model of having two nodes connected by an edge is not very realistic. Instead, several processors may be connected by a bus and may be thought of as being equally distant from each other. At the same time, each processor may be connected to several buses. The result is a bus connection network.

The same factors are relevant in the construction of bus connection networks as in the construction of standard networks. We will focus on the property of the maximum distance between processors, or the diameter. Specifically, we will be looking for bus connection networks with as many nodes as possible as a function of the diameter, subject to constraints on the number of connections for each node and bus. Such networks should have smaller maximum communication delays than networks of comparable size.

A Bus Connection Network Model

We will use a model discussed by Mickunas [4] for a bus connection model. In this model, each node is incident on a certain number of buses, which we will call the node's degree. The maximum of the degrees of the nodes will be called the graph's nodal degree. Each bus has a certain number of nodes on it, which will be called the degree of the bus. The maximum of the degrees of the buses will be called the graph's bus degree. For notation, we will denote the nodal degree by d, the bus degree by b, the diameter by k, and the number of nodes by n.

Two nodes will have a distance of one if they are incident on a common bus. In general, the distance between two nodes is the minimum number of buses which must be passed through to get between them.

We will use a representation of a bus connection network as a bipartite graph. One type of node represents the original nodes, while the other type represents the buses. In the figures in this paper nodes are filled-in circles and buses are empty circles. An edge in a figure represents a node incident on a bus.

Moore Graphs for Bus Connection Networks

For standard graphs, the bus degree $b = 2$. In most of this paper we will assume $b > 2$, since the other case is covered elsewhere [1-3]. When $b = 2$, the maximum possible number of nodes a graph can have is

$$\frac{d(d-1)^k - 2}{d - 2}$$

A graph with this number of nodes is called a Moore graph.

An analogous concept can be defined for the case of $b > 2$. From each node at most d buses can be reached, from which at most $d(b-1)$ nodes can be reached. Thus the maximum number of nodes in a diameter one graph is $1 + d(b-1)$. Similarly in exactly two steps at most $d(d-1)(b-1)^2$ nodes can be reached, and in exactly j steps at most $d(d-1)^{j-1}(b-1)^j$ nodes can be reached. If we define $n(d,b,k)$ as the maximum number of nodes in a graph with nodal degree d, bus degree b, and diameter k, we have

$$n(d,b,k) \leq 1 + d(b-1)\frac{(d-1)^k(b-1)^k - 1}{(d-1)(b-1) - 1}$$

A graph with this number of nodes is called a Moore geometry.

Since there are very few Moore graphs it is natural to ask if there are any Moore geometries. A Moore geometry with diameter 1 has $1 + d(b-1)$ nodes. The simplest case here is where $d = b$ (the node degree equals the bus degree), so there are $d^2 - d + 1$ nodes and the same number of buses. Every pair of nodes is on exactly one common bus. The existence of such a graph

0190-3918/83/0000/0158$01.00 © 1983 IEEE

depends on the existence of a mathematical object called a finite projective plane, a subject which has been extensively examined. It can be shown that such a plane exists if d-1 is a power of a prime number [5].

For d ≠ b, diameter 1 Moore geometries can be based on balanced incomplete block designs, or BIBDs. A design arranges n objects (analogous to the nodes) into v blocks (analogous to the buses) so that every object is in d blocks, every block contains b objects, and every pair of objects occurs together in exactly one block. Several examples of BIBDs are given in [6].

The question of the existence of Moore geometries with diameters greater than one is more complicated. None are known if b > 2. Bose and Dowling [7] give necessary conditions for existence when k = 2, although they could find no graphs satisfying those conditions. Fuglister [8] showed that there are no Moore geometries with k = 3. Another result which is easily proved is that there are no Moore geometries with d = 2 and b > 2.

Previously Proposed Bus Connection Topologies

Several construction methods for bus connection networks have been proposed in the literature. Perhaps the simplest bus connection network is a hypercube. Such a network can be thought of as a cube in d-dimensional space. Each row, column, etc. of nodes corresponds to a bus. Each node is on d buses, and each bus has b nodes. The graph has a total of b^d nodes.

A structure proposed by Wittie [9] is called a dual bus hypercube. This is a hypercube with many of the buses removed. Each node is connected to two buses. One bus can be thought of as a line in the same direction for all nodes. The other bus is in the same direction for all nodes in a single plane. A node corresponds to a vector of the form $(x_1, x_2, \ldots, x_b, z)$ where the x's and z take values from 1 to b. Thus there are b^{b+1} nodes. One bus that each node is on connects all nodes with the same z values and the same x values except one--the z^{th}. The diameter of the graph is 2b. Other proposed bus connection networks include the "snowflake" and "star" graphs introduced by Finkel and Solomon [10].

New Bus Connection Network Topologies

The first method we will consider is a generalization of the hinging graphs of Friedman [11]. In a standard hinging nodes are arranged in a hierarchical fashion, then the nodes at the bottom level of several hierarchies are connected. These networks are bipartite so we can call one class of the original nodes to be the new "nodes" and the other class the "busses." The graphs can be generalized so that the two classes have different degrees. As an example, Figure 1 shows a hinging for k = 3, d = 3, b = 4, and n = 40. It can be proven that it is optimal (in the sense of having more nodes) for the object (node or bus) with the higher degree to be on the hinge. Formulas can be easily derived expressing n as a function of d, b, and k, in which

$$n = O([(d-1)(b-1)]^{k/2}).$$

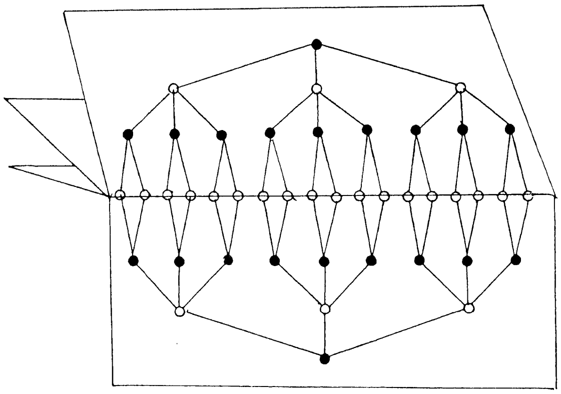


Figure 1
Example of a Bus Hinging Graph

When d = b, large networks can be found by using chordal rings [1]. These are a generalization of the chordal rings of Arden and Lee [12]. A chordal ring is a graph which begins as a ring on n nodes, then chords are added connecting additional pairs of nodes. The chordal lengths form a pattern, which repeats around the ring. Again we need a bipartite structure. With an even number of nodes, the ring part is naturally bipartite. In order to insure the graph remains bipartite, all chord lengths must be odd. Table 1 lists the characteristics of a few of the chordal rings found by the author which can be used for bus connection networks. In most cases these are larger than the graphs which have been produced by other methods. An example with 24 nodes and buses, degree 3, and diameter 2 is shown in Figure 2. The pattern length for this graph is 8.

Degree	Diameter	Number of Nodes	Moore Bound
3	2	24	31
3	3	75	127
3	4	180	511
3	5	455	2047
4	2	72	121
5	2	128	341
6	2	242	781
7	1	35	43

Table 1
Chordal Rings for Bus Connection Networks

A Generalization of de Bruijn Networks

In this section we introduce a new contruction which generalizes the de Bruijn networks [2]. In a de Bruijn network, each node is associated with a vector of k components, each being one of the integers between 1 and d/2 (k is the graph diameter and d is the degree, which is assumed to be even). The node $(v_1, v_2, \ldots, v_k)$ is connected to all nodes of the form $(x, v_1, v_2, \ldots, v_{k-1})$ and all nodes of the form $(v_2, v_3, \ldots, v_k, x)$, where x is an integer between 1 and

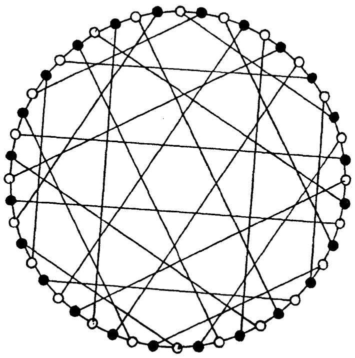


Figure 2
Example of a Chordal Ring Bus Network

d/2. The graph has $(d/2)^k$ nodes.

In order to generalize this to bus connection networks, let us examine the node connection rule more carefully. All nodes of the form $(v_1, v_2, \ldots, v_{d-1}, x)$ are connected to all nodes of the form $(y, v_1, \ldots, v_{d-1})$. With the bipartite representation of graphs, the interface between two of these sets of nodes with d = 6, k = 3 looks like Figure 3. When higher degree buses are allowed, the appearance of the interface will change. The goal will be to choose the interface so that the number of nodes per set (the number of values per digit), which will be denoted h, is as large as possible. This is because the total number of nodes in the graph is h^k. In the standard case, h = d/2. An example of a more complex interface is shown in Figure 4 for d = 4, b = 6, k = 3.

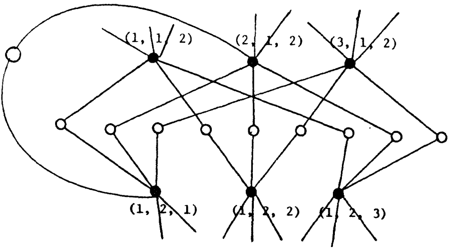


Figure 3
Example Interface--Standard de Bruijn Graph

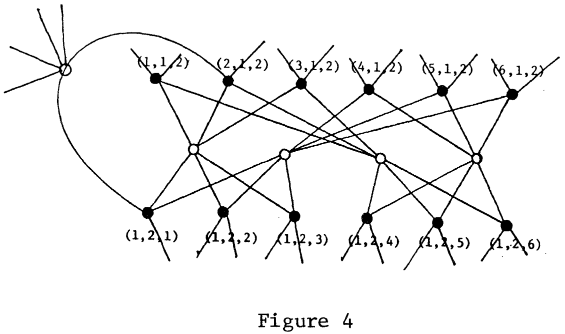


Figure 4
Example Interface--Generalized de Bruijn Graph

The same form of construction as in Figure 4 works whenever d and b are both even. We will have h = (d/2)(b/2), and $(d/2)^2$ buses for each interface. The first b/2 nodes on the "bottom" level of an interface (the ones whose vectors will be shifted forward) whould be connected to the first d/2 buses, the second b/2 nodes to the second d/2 buses, etc. The first b/2 nodes on the "top" should be connected to those buses which, if numbered left to right, are congruent to 1 (mod d/2). The second should be connected to those congruent to 2 (mod d/2), etc. This makes every bottom node connected to every top node via a bus. A similar construction is used if either d or b is odd. Notice that the generalized de Bruijn graphs are of size $(db/4)^k$. This is larger than $O([(d-1)(b-1)]^{k/2})$ when db > 16. The generalized de Bruijn graphs also inherit all of the advantageous properties of the standard de Bruijn graphs. For example, a simple algorithm allows one to bypass a faulty node by taking only four additional steps.

References

[1] K. Doty, "Large Regular Interconnection Networks," Proc. 3rd Int. Conf. Dist. Comp. Sys. Miami, 1982, pp. 312-317.

[2] W. Leland et al, "High Density Graphs for Processor Interconnection," Inf. Proc. Let., v. 12 (1981) pp. 117-120.

[3] J.-C. Bermond et al, "Tables of Large Graphs with Given Degree and Diameter," Inf. Proc. Let., v. 15 (1982) pp. 10-13.

[4] M. Mickunas, "Using Projective Geometry to Design Bus Connection Networks," Proc. Wkshp. Int. Net., 1980, West Lafayette, IN, pp.47-55.

[5] O. Veblen and W. Bussey, "Finite Projective Geometries," Trans. Amer. Math. Soc., v. 7 (1906), pp. 241-259.

[6] R. Fisher and F. Yates, Statistical Tables for Biological, Agricultural and Medical Research, Hafner Publ. Co., New York, 1963.

[7] R. Bose and T. Dowling, "A Generalization of Moore Graphs of Diameter Two," J. Comb. Thry. v. 11 (1971) pp. 213-226.

[8] F. Fuglister, "On Finite Moore Geometries," J. Comb. Thry (A) v. 23 (1977) pp.187-197.

[9] L. Wittie, "Communication Structures for Large Networks of Microcomputers," IEEE Trans. Comput. v. C-30 (1981) pp. 264-272

[10] R. Finkel and M. Solomon, "Processor Interconnection Strategies," IEEE Trans. Comput. v. C-29 (1980) pp. 360-371.

[11] H. Friedman, "A Design for (d,k) Graphs," IEEE Trans. El. Comp. v. 15 (1966) pp.253-254.

[12] B. Arden and H. Lee, "Analysis of Chordal Ring Network," IEEE Trans. Comput., v. C-30, (1981) pp. 291-295.

A SIMULATION STUDY OF MULTIMICROCOMPUTER NETWORKS

Daniel A. Reed[†]

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

Abstract: Recent developments in VLSI have made it feasible to interconnect large numbers of single chip computers to form a multimicrocomputer network. Using task precedence graphs to represent the time varying behavior of parallel computations, we investigate the performance of interconnection topologies for multimicrocomputer networks.

Introduction

Current evolutionary trends in integrated circuit fabrication suggest that it will soon be cost effective to consider a new parallel processing paradigm based on large networks of interconnected single chip computers. The single VLSI chip comprising each network node would contain a processor with a modicum of locally addressable memory, a communication controller capable of routing internode messages without delaying the processor, and a small number of connections to other nodes.

Among the suggested application areas for these multimicrocomputer networks are partial differential equations solvers and divide and conquer algorithms. The cooperating tasks of a parallel algorithm for solving one of these problems would execute asynchronously on different nodes and communicate via internode message passing. The limited node fanout implied by the VLSI implementation, as well as the absence of shared memory, make it crucial to select an interconnection network capable of efficiently supporting message passing. .In this paper we discuss computation paradigms for multimicrocomputer networks and techniques for assessing the performance of network interconnections.

Models of Computation

In one view of parallel computation, all parallel tasks are known *a priori* and are statically mapped onto the network nodes before the computation begins. In this case, queueing theoretic models can be used to estimate the performance of a given multimicrocomputer network executing a particular algorithm [4].

In the alternate view, a parallel computation is defined by a dynamically created task precedence graph. Tasks are created and destroyed as the computation proceeds, and the mapping of tasks onto network nodes is done dynamically.

Because most queueing theoretic models assume steady state behavior, they are not generally applicable to study of time dependent parallel computations. In particular, models of time dependent computation must account for time varying workloads, distribution of data to multiple tasks, and dynamic mapping of tasks onto network nodes using only partial knowledge of the global network state. Because we know of no analytic technique capable of accurately representing this behavior, we have adopted simulation as a means of study.

Subsequent sections of this paper present five multimicrocomputer interconnection networks, outline a task precedence model of time dependent computation, and discuss the results of a parametric simulation study of these interconnection networks when supporting time dependent computations.

Interconnection Networks

Because of the computational expense of simulation, we limited our study to five interconnection networks that earlier analysis suggested were worthy of further investigation: the *2-D* spanning bus hypercube [6], *2-D* toroid [4], cube-connected cycles [3], 2-ary N-cube [1], and the complete connection. We have included the complete connection to determine the performance degradation attributable to incompletely connected networks.

Task Precedence Graphs

As stated earlier, our model of time varying computation is the task precedence graph. A precedence graph represents a computation as a series of dependencies. The results of all computations providing input to a task, its antecedents, must become available before the task is eligible for execution.

In each precedence graph, three types of tasks can be distinguished: fork tasks, join tasks, and regular tasks. A *fork* task has a single antecedent task and one or more consequent tasks; it represents the computation prior to initiation of parallel subtasks to solve a problem. A *join* task has one or more antecedent tasks and a single consequent task; it represents the combination of subproblem solutions to yield a solution to an entire problem. Finally, a *regular* task is any task that is not a fork or join task; it represents a simple computation. If we interpret the juxtaposition AB of tasks to mean "A is an antecedent of B", a task precedence graph can be formally defined by the following grammar.

<precedence graph> ::= *<regular task>* |
<fork task> *<precedence graph>*$^+$ *<join task>*

As summarized in Table I, the characteristics of a precedence graph are determined by several parameters. Because the number of possible graph parameterizations is so large, we have somewhat arbitrarily selected a set of values, also given in Table I, to be used as a reference point in our study. By systematically varying subsets of these parameters, we obtain different performance results. By comparing these results to those obtained using the reference parameters, we can estimate the effect of the variations.

Simulation Methodology

For comparative purposes, we generated twenty five task precedence graphs using the reference parameters shown in Table I. All service times were drawn from negative exponential distributions, the number of consequents of each fork task was uniformly distributed between B_{min} and B_{max}, and all graphs were

[†] Present address: Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina 27514

constrained to have between *Maxtasks* / 2 and *Maxtasks* tasks. Each node was assumed to possess complete knowledge of the network state, and each task eligible for execution was scheduled on the idle node nearest its location. We will return to this assumption when discussing distributed scheduling algorithms. Finally, to model the fact that each network node is a single chip with fixed bandwidth, we scaled the mean data communication times by the number of link connections to each node.

The average parallelism P attained when evaluating a precedence graph on a network has been taken as the measure of performance. This is

$$P = \frac{\sum_{i=1}^{Numtasks} S_i}{parallel\ execution\ time} \qquad where \quad S_i \in \{ S_F, S_R, S_J \} .$$

Simulation Experiments

Using the assumptions discussed above, we explored five different variations of precedence graph parameters and network characteristics and their effect on network performance: precedence graph structure, the event horizon of a distributed task scheduler, the maximum task branching factor, the mean computation time/communication time ratio, and the number of network nodes. The first two of these are discussed below; an analysis of the other variations can be found in [5].

Precedence Graph Structure

Figure I shows the graph parallelism when each of the twenty five graphs derived from the reference graph parameters was simulated on the five networks with 64 nodes. The precedence graphs were sorted in increasing order of parallelism on the complete connection. Table II shows the average parallelism over the set of graphs using each network.

Two features of Figure I are of particular interest. The first is the way networks other than the complete connection exhibit the same performance trends from precedence graph to graph. This suggests that something inherent to the graphs is affecting the time required for their evaluation. To determine what this might be, we examined two precedence graphs, numbers nine and eleven in the figure, that represented two extremes of behavior. Figure II shows the time varying parallelism when the two graphs were evaluated on a 2-*D* toroid with 64 nodes. The simulation of precedence graph nine exhibits a striking decrease in the number of parallel tasks near time 90. Because a similar simulation on the complete connection exhibits no such decrease, we can only conclude that this variation is caused by the collapse of a parallel subgraph requiring the transmission of results across several communication links. During the delay caused by this transmission, tasks otherwise eligible for execution were forced to wait for these results.

Figure I also points out the performance differential between the spanning bus hypercube and the networks using dedicated links. Although, this behavior may appear somewhat anomalous in light of the apparently greater communication bearing capacity of the dedicated link networks, this is not the case. A detailed examination of the simulation results shows that tasks generally execute on nodes near their point of origin. In other words, the precedence graph evaluation exhibits considerable communication locality. For this communication pattern and the given ratio of computation time to communication time for tasks, the utilization of the communication links is low. Because of this, the buses of the spanning bus hypercube permit more rapid distribution of tasks to other nodes than the dedicated links of the other networks. For the same reason, distinct differences among the dedicated link networks are also not apparent.

Event Horizon of a Distributed Scheduler

Heretofore we have assumed that the task scheduler at each node always possesses complete knowledge of the global network state. In practice, only limited information is available, and it is often no longer completely accurate when it is received.

To determine a scheduler's operation in the face of partial knowledge, we postulated the existence of an *event horizon* for each network node. We assume the scheduler at each node has no knowledge of network activity at any nodes beyond its event horizon and that it must schedule all eligible tasks on nodes within its event horizon. Using the reference precedence graph parameters, Figure III shows the average graph parallelism as a function of the distance to the event horizon from a node. Similar results are obtained when the ratio of computation times to communication times varies from 1:1 to 100:1. Based on this limited evidence, it appears that state knowledge of nodes within a small distance from each source node is sufficient to achieve reasonable results. This is encouraging because it suggests that efficient distributed schedulers can be constructed for multimicrocomputer networks.

Two final observations about distributed schedulers should be made. First, this dynamic scheduling strategy does not use the precedence graph structure to aid its decisions. It should be possible to design heuristics that take advantage of some graph specific information.

Second, the acquisition of state information from nodes within an event horizon is decidedly more difficult for networks connected by buses than for those using dedicated communication links. This is primarily because so many more nodes are within a small number of bus crossings from a source node. Communicating state information to other nodes on the same bus could conceivably consume a significant portion of the available communication bandwidth. Additional work is needed to determine the cost of acquiring state information.

Summary

We have presented a model of time dependent parallel computation and studied the behavior of five multimicrocomputer interconnection networks supporting computations similar to those of the model. Among the issues considered were the relative performance of interconnection networks and the efficacy of distributed scheduling using incomplete information.

For small, dynamically created tasks, the spanning bus hypercube appears to have better performance than the dedicated link networks because it can diffuse work more rapidly. This is not always true; if message routing does not exhibit enough locality (i.e., messages must cross many links to reach their destination), the smaller communication bearing capacity of the spanning bus hypercube will be saturated, and the dedicated link networks will be preferred. Clearly, the selection of a network must be made with knowledge of communication patterns and task sizes required by an algorithm.

Finally, dynamic task scheduling using only local information seems successful for the class of algorithms represented by precedence graphs, suggesting that efficient distributed schedulers can be designed.

References

[1] F. W. Burton and M. R. Sleep, "Executing Functional Programs on a Virtual Tree of Processors," *Proc of the 1981 Conf on Functional Prog Lang and Computer Arch*, Oct. 1981, pp. 187-194.

[2] M. C. Pease, "The Indirect Binary n-cube Microprocessor Array," *IEEE Trans on Comput*, Vol. C-26, No. 5, May 1977, pp. 458-473.

[3] F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Comm of the ACM*, Vol. 24, No. 5, May 1981, pp. 300-309.

[4] D. A. Reed and H. D. Schwetman, "Cost-Performance Bounds for Multimicrocomputer Networks," *IEEE Trans on Comput*, Vol. C-32, No. 1, Jan. 1983, pp. 83-95.

[5] D. A. Reed, "A Simulation Study of Multimicrocomputer Networks," *Tech Rep CSD-TR-435*, Department of Computer Sciences, Purdue University.

[6] L. D. Wittie, "Communication Structures for Large Multimicrocomputer Systems," *IEEE Trans on Comput*, Vol. C-30, No. 4, Apr. 1981, pp. 264-273.

Table I Precedence graph parameters

Quantity	*Definition*	*Reference Value*
B_{min}	minimum number of consequents of a fork task	1
B_{max}	maximum number of consequents of a fork task	4
C_F	mean data communication time to initiate a fork or regular task	1
C_J	mean data communication time to initiate a join task	1
Maxpath	maximum length path through the graph	60
Numtasks	number of tasks in the graph	1024
S_F	mean fork task service time	10
S_R	mean regular task service time	10
S_J	mean join task service time	10

Table II
Average graph parallelism for 64 node networks using the reference precedence graph parameters

Network	*Average parallelism*	*Fraction of complete connection*
Complete Connection	22.17	1.00
Cube-connected Cycles	15.33	0.69
2-ary 4-cube	15.42	0.70
2-D Spanning Bus Hypercube	18.04	0.81
2-D Toroid	14.75	0.67

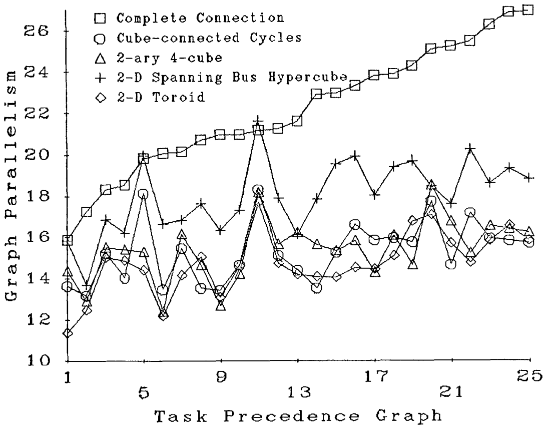


Figure I
Graph parallelism for 64 node networks using the reference precedence graph parameters

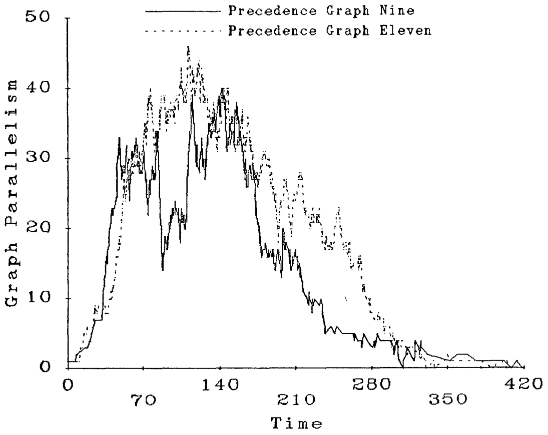


Figure II
Time varying parallelism for two precedence graphs on a 64 node toroid

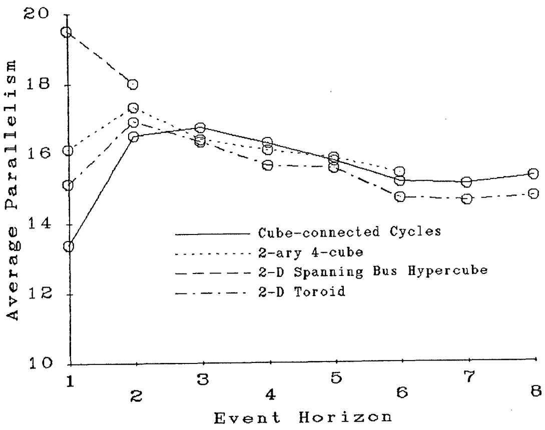


Figure III
Average parallelism for 64 node networks with varying amounts of scheduler information

Evaluation of Multiprocessor Interconnect Structures With the Cm* Testbed

Andrew Wilson, Dan Siewiorek and Zary Segall
Department of Electrical Engineering and Computer Science Department
Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

Abstract

This paper presents a method for emulating multiprocessor architectures on Cm*, a 50 processor multiprocessor at Carnegie Mellon University. Combined with other instrumentation tools already developed at CMU the result is a flexible testbed for multiprocessor architecture evaluation. An experiment to demonstrate the usefulness of this testbed is presented along with results for three architectures: ring, nearest neighbor and fully connected. These results are used to show how the testbed could be used to aid in multiprocessor design. It is shown that for this particular real-time applicaton a three processor fully connected structure provides more usable compute power than a six processor ring.

Introduction

Large computational problems such as weather forecasting, fusion modeling, and aircraft simulation demand computing power far in excess of that supplied by uniprocessors. Thus researchers have been seeking alternative architectures to solve these pressing problems. Many of these proposals involve the construction of multiprocessors, systems in which a shared address space is provided for interprocessor communication and synchronization. A very popular form of multiprocessor is the Multiple Instruction stream, Multiple Data stream (MIMD) computer [5].

All multiprocessors require an interconnection mechanism which physically implements the shared address space. Numerous proposals for such structures appear in the literature, covering a wide range of performance, cost, and reliability [2, 4, 6, 15]. Unfortunately few working multiprocessors have actually been built, so the principle sources of comparative information come from modeling, simulations, and educated guesses. Due to the complexity of parallel program interaction, the complexity of modeling the hardware systems and the large size of the design space, much of the results to date are inadequate.

To help alleviate this situation, the construction of a multiprocessor test bed which could closely emulate a number of architectures would be desirable. Several testbeds are currently under development by other researchers, such as the network testbed at Mitre [3] and the multiprocessor testbed at TRW [8]. With a versatile emulator, accurate measurements of the performance of these architectures under both synthetic and actual workloads could be obtained. Fortunately, as will be shown, Cm* [13, 14] can function as such a test bed and be used to answer some of the questions plaguing multiprocessor research.

The message passing portion of the Medusa operating system [9] is quite suitable for emulation of alternate architectures through software routing schemes. The time cost of the routing software is generally less than the time to send the message, as would be the case in a real system. To see how useful such an emulation system might be, an experiment to model the communications demands of an actual multiprocessor was implemented. Four versions, one using full interconnection directly implemented with Medusa communication primitives, one using emulated full interconnection, one using emulated nearest neighbor communication, and one using an emulated ring were built. The ability to remove unwanted Cm* characteristics and quantify emulation overhead was demonstrated. To distinguish the different experiments, the direct implementation will be refered to as the basic experiment, while the other three as direct connect, nearest neighbor and ring experiments.

Description of the Cm* Testbed Environment

The testbed described in this paper is being developed on Cm*, a 50 processor multiprocessor operating at Carnegie-Mellon University [13, 14]. The processors are Digital Equipment Corporation LSI-11's arranged in a two level hierarchy of Computer Modules and Clusters, as shown in Figure 1. The custom built communications mapping processors (Kmaps) handle interprocessor memory references and provide low level operating system support. For simple memory references between Cms within the same cluster access times are about a factor of three longer than those of intra-Cm references, while between cluster references take nine times as long. For messages the distinction between intercluster and intracluster transit times disappears, as will be demonstrated shortly.

All of the experiments were performed using the Medusa operating system. In Medusa, an experiment consists of a group of cooperating activities (processes) which are termed a task force. Each activity is assigned to a Computer Module (Cm), in which it alone executes, and given its own copy of code. Communication between activities can be by shared memory, with the shared memory resident on any Cm in the system, or by message passing, using a block transfer mechanism implemented in Kmap microcode. Messages are passed through Pipes which queue up waiting messages in FIFO order, up to the capacity of the Pipe. In this respect they are much like Unix Pipes, except that they maintain the identity of each message and will only deliver them as separate units. Routines running on the LSI-11s invoke Kmap operations to send and receive messages, with the choice of suspending execution when an operation cannot be completed or quiting with an error indication. The experiments make use of these facilities to emulate a multicomputer system.

The research described in this document is funded in part by RCA, by the Balistic Missile Defense Agency under contract DASG60-80-C-0057, by the National Science Foundation under contract number ECS-8120270, and also by the Defense Advanced Research Agency (DOD), ARPA Order No. 3579, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. We would also like to acknowledge the Digital Equipment Corporation for Equipment grants related to Cm*.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Agency or the US Government, or the Digital Equipment Corporation.

0190-3918/83/0000/0164$01.00 © 1983 IEEE

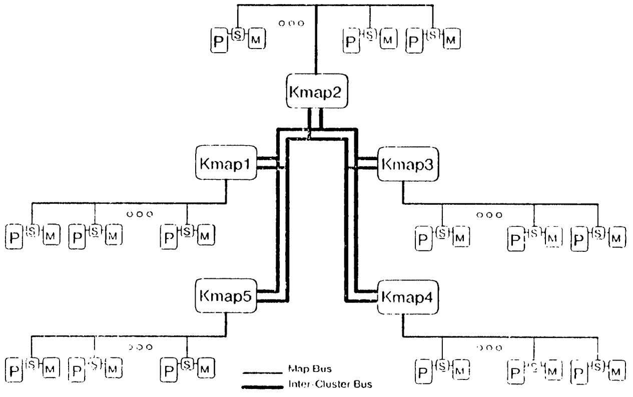


Figure 1: Cm* Multiprocessor Structure[1]

In emulating different architectures it is desirable to eliminate any biases caused by the underlying system. Cm* has a hierarchical communication structure which can cause variations in system performance depending on the relative locations of the communicating processes. It turns out though that the Medusa message passing system exhibits approximately constant delays, regardless of subtask location, provided the Pipes are properly located, as discussed below.

In the Medusa message system, greatest message throughput is obtained when the Pipe is on a different cluster than the Sender and Receiver subtasks. This surprising result was discovered while measuring message system throughput for the emulations. It appears to be an effect of Kmap contention caused by the heavy processing load placed on the Kmaps by the communication mechanism. Placing the Pipe on a different cluster distributes this load over several Kmaps resulting in the observed speedup. If several messages are being transmitted by the system simultaneously there may be less impact from distributing the workload, but the effect has been observed even under those conditions.

Figure 2 shows the effect of placing the Pipe on the same cluster. For all three curves in the figure the Pipe was on cluster one. The cluster one transfers were considerably slower than those on clusters two and three. If the cluster one transfer measurements are redone with the pipe on another cluster the transfer rate is identical to those of the other two clusters. There is a small decrease in transfer rates when the Sending and Receiving processes are both on the same cluster, as shown in Figure 3. The decrease is not significant, however. If Pipes are properly placed, a nearly flat communication structure results, providing a base for accurate emulation.

Multiprocessor Emulations Possible on Cm*

The space of possible multiprocessor designs is large. Dimensions on which multiprocessors may vary include the number of independent instruction streams, speed of processors, bandwidth of interconnection links, and degree of connectivity. The specific attributes of these dimensions as regards to Cm* determine the space of multiprocessors which the Cm* testbed can emulate.

A major area of research in multiprocessor architectures involves the design and evaluation of the interconnection mechanisms used in them. A few of the possibilities include multistage networks such as the Augmented Data Manipulator [1] and Omega [7], point-to-point networks such as Cube Connected Cycles [10] and nearest neighbor meshes, and shared bus networks, such as the mesh scheme originally proposed for Cm* [12]. Interprocessor communication may be through direct memory references or through explicit messages. The multistage networks tend to favor direct memory references or short messages while the more sparsely connected point-to-point networks favor long messages with store and forward operation at the nodes. Since the testbed described in this paper uses software routing and explicit message passing, it is better suited to emulating the point-to-point networks and certain types of shared bus networks. However, special cases of multistage networks may also be feasible to emulate. Some examples of the multiprocessor interconnection networks suitable for emulation on Cm* are shown in Figure 4. Research is continuing on developing a testbed which can accurately emulate the multistage networks as well.

Since Cm* is a MIMD machine with asynchronously operating computer modules, it would be inappropriate to attempt to emulate multiprocessors without those features. In normal operation code and local data are kept in the memory of the computer module using them and are accessed directly rather than through the interconnection network. While it is possible to force all accesses through the network, it is probably preferable to concentrate experimentation on architectures which feature the same computer module structure. The results reported in this paper all assume local access to code and non-shared data.

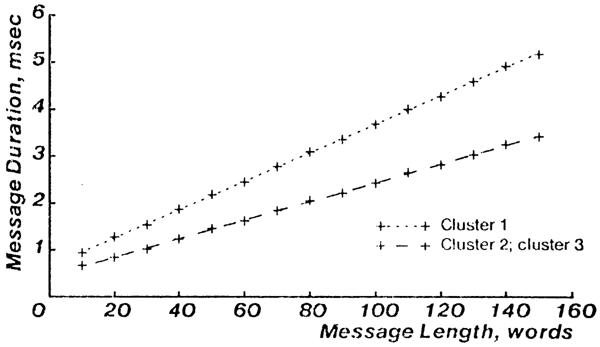


Figure 2: Effect of poorly placed Pipes on throughput (Pipe on Cluster 1)

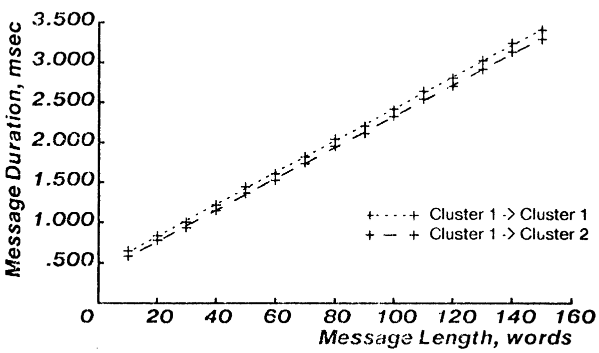


Figure 3: Message transfer, Intercluster vs Intracluster

[1] Figure courtesy of Pradeep Sindhu

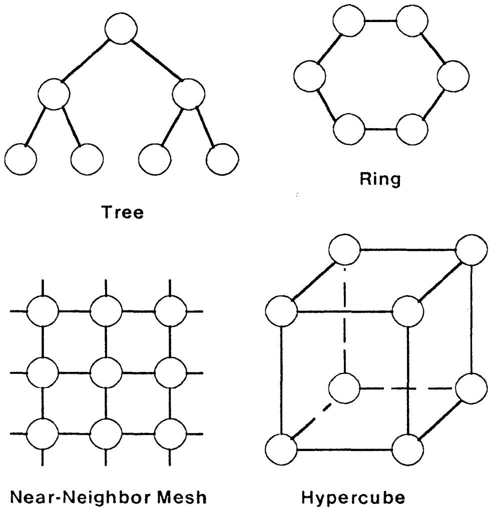


Figure 4: Typical Interconnection Networks suitable for Cm* Testbed Emulation

Methods used to Emulate Multiprocessor Architectures

The emulation package was implemented by adding a subroutine package to each of the subtasks to perform message routing and delivery. In this scheme each subtask is only allowed to communicate with logically adjacent subtasks as determined by routing tables. Messages for nonadjacent subtasks have to be forwarded by intermediate subtasks.

Subtasks communicate through a set of virtual buffers corresponding to the physical buffers used in the basic experiment. Messages sent over these buffers are routed by information contained in the first word of the message. This header word contains source, destination and logical buffer indices. Routing tables computed partly at compile time and partly during initialization determine the exact path taken by each message.

The actual operation of the message system starts when a subtask calls the SNDMESS subroutine in the routing package to send a simulated message. This routine computes the routing word for the message and calls FRWRDMSS to initiate its transport through the network. FRWRDMSS determines the appropriate physical output buffer and invokes the Kmap to send the message through it. Each subtask repeatedly calls the CHKBUFS routine to determine if any messages have arrived and to deliver or forward them as appropriate. If the message is for the local subtask INCRMESS is called to increment a received message counter and check for overflow. A flow control system is provided to prevent deadlock which limits the number of messages in transit through the emulated interconnection networks to a safe number. Buffer overflow is determined by examining the message counts for each virtual buffer maintained by INCRMESS. Buffer overflow in this scheme is analogous to buffer overflow in the basic experiment.

Using the emulation package described above, a large variety of interconnection structures can be modeled by simply changing parameters in the routing tables. Since real messages are passed between processors, actual working programs can be used to test the structures. This allows the data dependencies often associated with real multiprocessor algorithms to be fully reflected in the observed interconnection structure behavior.

Description of Modeled System

The experiments consist of simulating the high level behavior of a real-time multiple computer system on different emulated interconnection structures. The actual system consists of three clusters of minicomputers, with four processors in each cluster. Within each cluster communication is through shared memory, making each cluster a small multiprocessor. Between clusters point-to-point communication paths are used, resulting in a multicomputer structure.

The multicomputer system is driven by inputs from external sensors. Its outputs consist of status displays and actuators. The three computer clusters perform distinct functions in the overall system from which their mnemonics are derived. The Actuator Control System (ACS) has primary control over the mechanical devices used in this system. A second computer cluster, termed the SPU, controls a signal processing unit. The final cluster handles overall control and information display, earning it the title of Control and Display (C&D).

This is a real-time system which must respond immediately to any significant event. Thus there are minimum throughput requirements as well as constraints on the maximum latency of certain operations. The object of the experiment is to emulate the system behavior, with Cm* used as a test bed comparing the performance of different architectures. The performance is measured by varying the synthetic workload (which simulates the high level system behavior) and message length while determining the maximum sustainable communications rate.

These experiments were based on a high level description of the system. The communications requirements at the multicomputer level, are shown in Figure 5. the requirements included information on the average message rate between clusters and external devices, between different clusters, and the allowed minimum response time for certain high level activities. The response time limits proved to be difficult to measure with the present level of Cm* instrumentation. However the Message Event generator provided on Cm* does facilitate the generation of messages at fixed time intervals which can be used to stress the communications system. This generator is used to simulate external inputs to the control system which would

DATA FLOW DIRECTION	BITS IN A WORD	WORDS IN A MESSAGE	MESSAGES PER SECOND	DATA BITS TRANSFERRED PER SECOND
C&D -- SPU				
C&D out	32	256	2	
C&D in	32	256	14	
				131K
ACS -- SPU				
ACS out	32	138	49	
ACS in	32	132	49	
				424K
C&D -- ACS				
ACS out	32	256	16	
ACS in	32	256	16	
				262K

Figure 5: Communication Rates in Multicomputer System

normally determine total system workload. Failure to meet real-time requirements was indicated by message buffer overflows somewhere in the system.

Description of Experimental Methodology

The actual experiment consists of a task force composed of one or more processors representing each multiprocessor cluster, plus several support activities. The experiment utilizes resources from the Synthetic Workload Generator system [11] and was intended to be implemented with its "B" language. In this environment, Pipes are termed buffers and activities termed subtasks. The support subtasks consist of the Pegasus user interface, the message event generator and a monitoring routine. The user interface provides communication with the user's terminal, controls operation of the message event generator, and allows control of user specified variables in the user's subtasks. The message event generator monitors a real-time clock and sends short (one word) messages to specific subtasks at specified intervals. The monitor subtask has an array of integers where other subtasks may record the occurrence of errors. It repeatedly scans these integers for evidence of changes and reports them to the user. In the present experiment these integers record the occurrence of message buffer overflows, as detected during attempted message sends. Using shared memory instead of short error messages is a much less expensive way of communicating this error information.

The modeled version of the Real-time system is shown in Figure 6. The simulated workload is driven by the message event generator, which periodically sends event tokens to the SPU to initiate a unit of system activity by triggering an SPU to C&D message. This message in turn generates other messages in such a manner that the average communication rate on the data paths in the system is similar to that experienced by the real system. This is done by generating new messages to send to other nodes under the control of a random number generator. The C&D subtask will generate two new messages for each message received from SPU with 7% going back to SPU, 57% to ACS and 36% going nowhere. C&D messages which arrive at the ACS subtask spawn four more messages, 25% of which return to C&D and 75% of which continue on to SPU. The SPU messages generate messages back to ACS, where they are terminated. The resulting message rates are shown in parenthesis in the figure, and correspond well to actual system rates.

In the real system the stated message traffic is only an average and varies with actual workload and computation patterns. This effect is simulated by using a uniform random number generator to determine the routing of messages whenever there is more than one possible destination. The generator provides a degree of random clustering of messages such as a real system might see, allowing the effects of buffer size and message length to be observed. Distributions other than uniform could be obtained if a particular experiment required them.

When a message is received by a subtask it performs some simulated work corresponding to that which the actual system would have to do. This simply consists of executing a null loop a number of times as requested by the experimenter through the Pegasus user interface. The number of loops per received message executed by each subtask is scaled by the average number of messages received per second so that each subtask executes the same average amount of work. For example, when actual system message rates are used, the SPU subtask receives an average of 50 messages a second and executes 256 loops for each message while the ACS subtask receives 64 messages per second and executes 200 loops per message. Thus both subtasks execute 12800 loops per second. The random number schemes could also be used here to provide some variability if desired. The experiments were conducted over a range of such workloads, with all subtasks subject to approximately the same load.

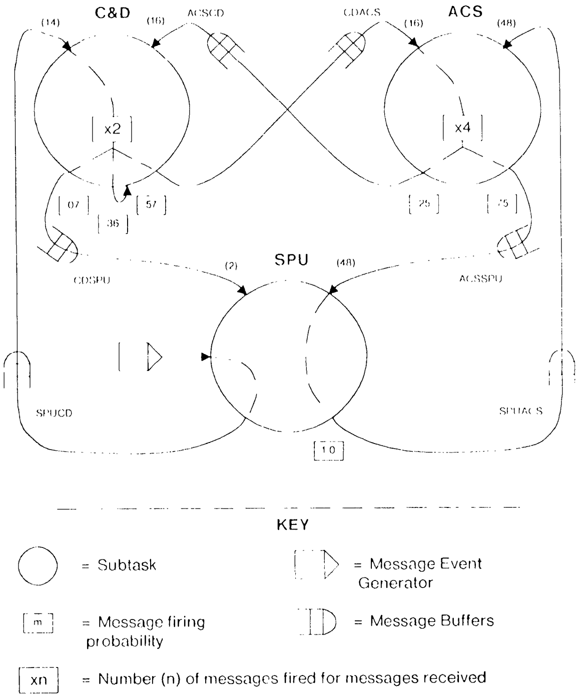


Figure 6: Representation of Data Flow Used in the Experiment

The maximum sustainable message rate for each combination of interconnection structure and synthetic work load was determined by repeated runs at increasing intervals between message events until a sustained period of operation in which no buffer overflows occurred was observed. Processing power was varied by varying the number of processors per subtask. Workload per message and message length were also varied for each structure in order to aid in characterizing its performance. With each processor executing the same amount of work, any overflows recorded were due to message system saturation. The communication abilities of the direct connect, nearest neighbor and ring networks were then compared.

Results

The first experiment consisted of the basic point-to-point, fully connected network similar to that used in the actual system. One subtask was used to represent each computer cluster of the system. Three different message lengths were used, one-half, one-fourth, and one-eighth of those specified in the actual system. Unfortunately the full message length proved to be too much for Cm* to handle, and so was deleted from the experiments. The system was deemed to have saturated when any message buffer overflowed[2] . Message system saturation periods for six different synthetic work loads were found for each message length. The results, averaged over three successive runs, are shown in Figure 7. Note that the three curves are essentially linear. However, small values of simulated work show a disproportionately longer period between messages, due to the increased significance of message activity. As message length

[2] Usually the SPU to WCS buffer overflows first, probably because it is the busiest buffer and the WCS subtask receives the most total messages.

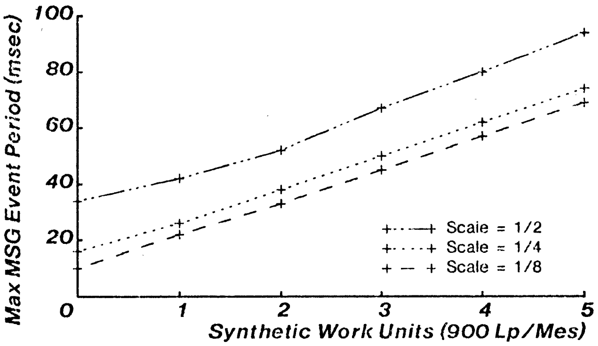


Figure 7: Saturation Curves for the basic Experiment

decreases (decreasing scale factor) the plots become linear even at low synthetic work load, again indicating the reduced effect of messages. The non zero saturation message period observed at zero synthetic work load is due to housekeeping tasks and message send/receive invocations.

As messages get longer there is a disproportionate increase in the small workload message event period. This is most likely due to the operation of the message system. When a process sends or receives a message it is suspended for the entire duration of that event. Since in Medusa messages take about twenty microseconds per word to transport, this value becomes significant at longer message lengths. In these experiments the message rate tends to decrease as the workload increases, since both workload and message passing contribute to message system saturation. Thus the contribution from the message system is large at low simulated workloads and small at high simulated workloads, resulting in the non linear curves evident in the figures.

Before collecting data on the nearest neighbor emulation a calibration experiment was attempted to check the accuracy of the emulation scheme. A fully connected network was implemented with the emulation package developed for the nearest neighbor emulation. The logical interconnection structure is shown in Figure 8. The circles labled S0 to S5 represent the message switching subroutines used to route and forward messages. The other circles represent the activities performing the actual processing. Each pair of switching node and adjacent activity resides on a single Cm. The experiments using one processor per subtask are indicated by the solid components. Additional experiments using two processors per subtask include the dotted components as well.

The resulting message saturation data was compared with the initial, nonemulated experiment. The curves exhibit a small offset from the basic experiment which is due to the extra overhead of the send and receive subroutines, and the periodic polling for incoming messages. As can be seen from the curves presented in Figure 9, this overhead is a constant 4-6 milliseconds over a wide range of workload and message length. Thus its effect can be easily factored out, as will be demonstrated in the conclusions section.

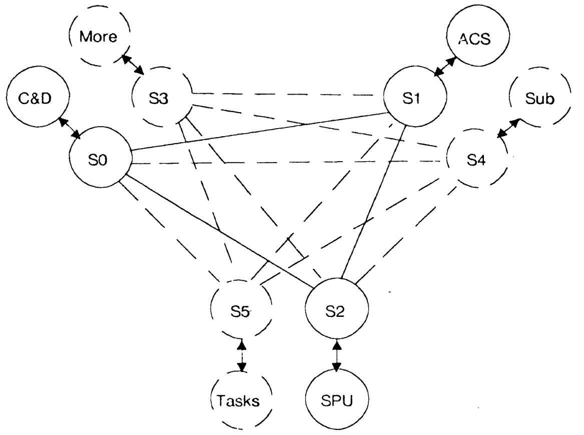


Figure 8: Fully Connected Emulation Experiment

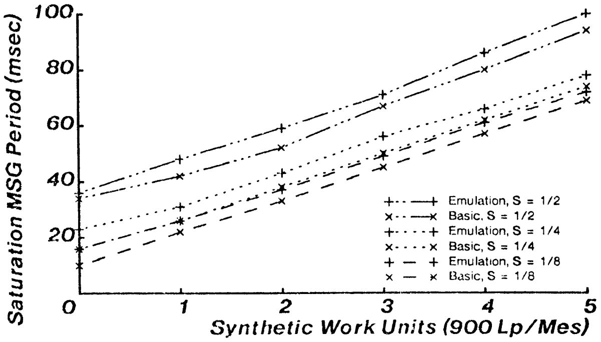


Figure 9: Saturation Curves for the Fully Connected Network Emulation

These experiments were then repeated for a Nearest Neighbor emulation of the multicomputer system. Since there are only three subtasks used in these initial trials, the interconnection structure actually consists of a line rather than a mesh. Figure 10 shows the actual interconnection of the subtasks, where the indicated components correspond to those described above for the fully connected emulation. The subtasks were arranged in the optimal order as determined by their communication behavior. A comparison was made with a less optimal order to see how large the effect would be. As seen in Figure 11 the message saturation period increased dramatically. In future studies with more subtasks, different arrangements will have to be tried to ensure optimality.

The measured workload curves for the nearest neighbor network, as shown in Figure 12, again show a decrease in linearity with increasing message length. The anomalous behavior of the long message length curve at low workloads (i.e. the shorter than expected message period) is currently under investigation and appears to be an isolated, though repeatable case. Even with optimal configurations, the saturation points occur at a significantly lower message rate (i.e. Larger message period) than with the basic experiment. Some of this is due to the increased overhead of the message routing subroutines, as evidenced in the fully connected emulation results, and some to the additional burdens of message forwarding, as would be expected with the poorer connectivity.

After obtaining results for the three processor systems, a similar set of experiments was tried with six processors. In addition a ring network was added. Since the three processor ring configuration is identical to the fully connected network, there was no need to collect separate three processor data. A fully connected network of six processors was included to provide a baseline for the other architectures and to determine the amount of overhead to subtract from the six processor configurations.

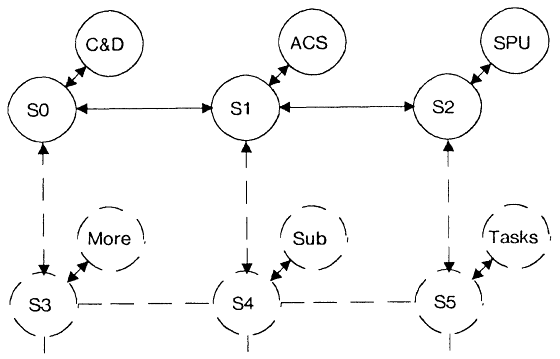


Figure 10: Nearest Neighbor connection scheme

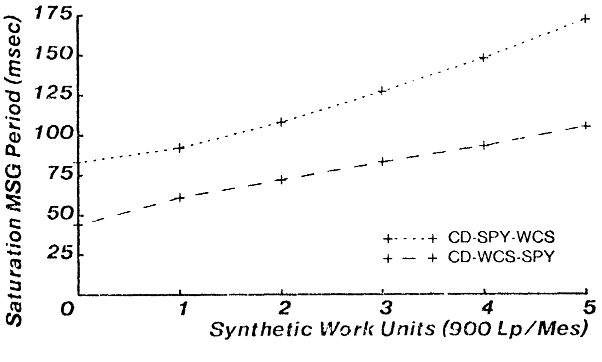


Figure 11: Comparison of different Subtask placement

With the six processor architectures each multiple processor subtask is implemented with two processors. The assumption here is that processors make intersubtask references directly to the intended processor, rather than through an arbitrary link processor in its subtask group. It is also assumed that intrasubtask references go through the shared memory of its computer group and do not enter into these experiments. With increasing numbers of processors, the relative merits of the proposed interconnect structures become apparent.

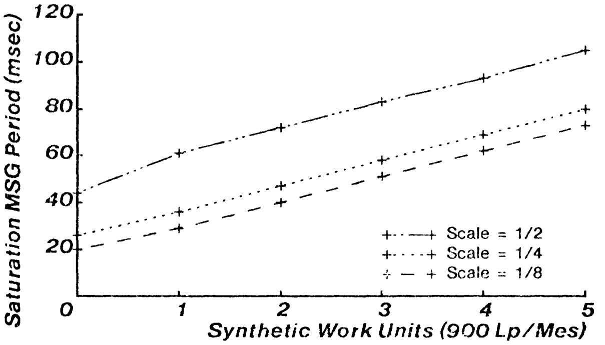


Figure 12: Saturation Curves for the Nearest Neighbor Network

Figure 13 presents both the three and six processor fully connected curves. Note that at large workloads the throughput of the six processor cases are approximately double those of the three processor cases, as would be expected from the dominance of processing over message passing at large workloads. If the overhead correction factors obtained earlier from comparisons of the basic and fully connected three processor architectures are applied, it is found that a correction factor of one-half the three processor correction factor yields almost exactly a factor of two difference for large workloads. This halving of the correction factor is consistent with the notion that the overhead observed is due to the extra computation required by the emulation routines. With six processors instead of three, the emulation overhead per processor is halved for any given message rate. The correction factors derived for three and six processor cases will be applied to the raw data when making detailed comparisons of the interconnection structures later in the paper.

The raw data for the six processor nearest neighbor and ring networks are presented in Figure 14. The reduced connectivity of the ring network is reflected in its poorer performance as compared to the nearest neighbor at long message lengths, though they exhibit similar behavior at shorter message lengths.

Conclusions

This paper has demonstrated a possible method for using Cm* as a multiprocessor testbed. Of particular interest to designers of multiprocessors is the amount of useful work they can obtain for a given application on various architectures. To indicate how this might be done an attempt was made to compare the work available for applications tasks on the various architectures. The emulation overhead was first subtracted out as described in the results section, then the percentage of time the processors were used for actual computation was computed. The computation times were calculated from the number of synthetic work loops executed per external message and the measured external message rates. Finally representative samples were plotted.

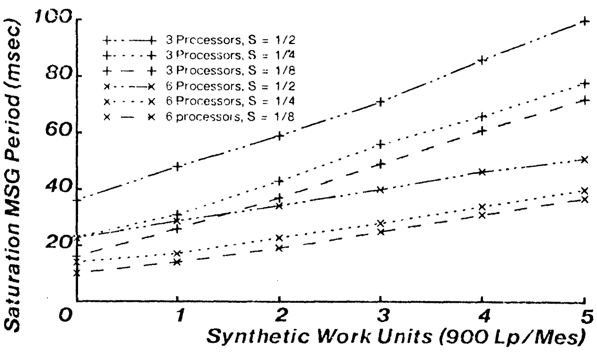


Figure 13: Saturation Curves for Both Fully Connected Networks

After calculating the usable processing time as described above, the various network configurations can be compared. Figure 15 compares the long message length (scale of one-half) results for all network configurations after adjustment for overhead. At first glance it appears that the usable processing power of each processor when two processors per subtask are employed is substantially higher than with only one processor. This is due to the fact that each processor is only involved with half of the messages sent or received by the subtask. Thus if the amount of processing done by each subtask is directly determined by the incoming message rate, as it is in the system modeled in these experiments, the message rate per processor needs to be considered. Figure 16 compares the three processor configuration of the fully connected network with the six processor

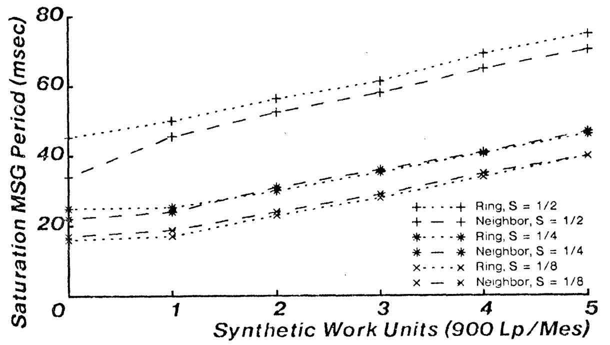


Figure 14: Six Processor Nearest Neighbor and Ring Results

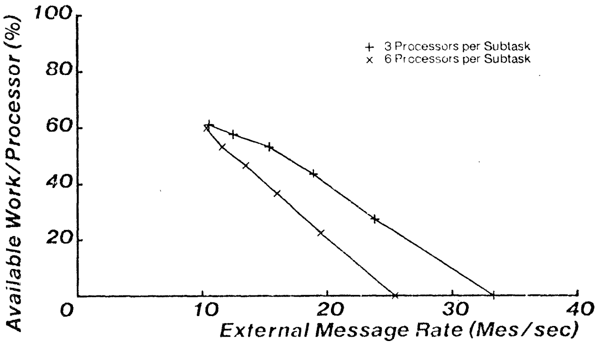


Figure 16: Comparison of potential workload, adjusted for per processor message rates

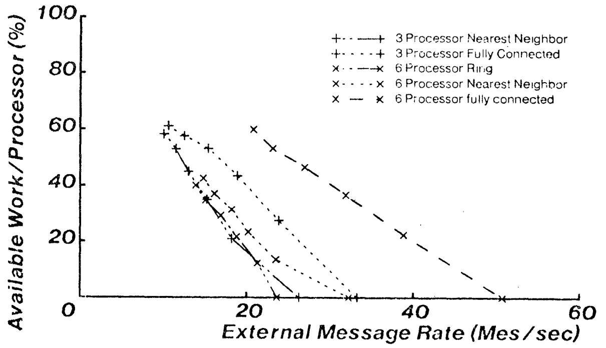


Figure 15: Comparison of Adjusted Results for Scale = 1/2

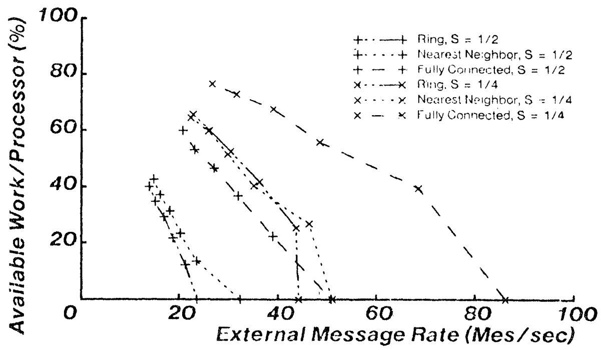


Figure 17: Comparison of the interconnection schemes

configuration, after accounting for the lessened message rate per processor. Note that the six processor architecture is actually using its processors less efficiently than the three processor case.

Even though the experiments involved only a small number of processors, some interesting results can be seen. Again examining Figure 15 it is seen that for this application a six processor ring utilizes its processors no more effectively than a three processor nearest neighbor. Of course more processing power is available since two processors are assigned to each subtask. Notice though that above 20 messages per second a three processor fully connect system has more than twice as much available power per processor than the six processor ring. Thus it would definitely be the prefered system at the high message rates. Below 20 messages a second the six processor ring has more total processing power available and might be prefered if large amounts of processing were required. It would be cheaper than either the six processor nearest neighbor or fully connected systems, provided it supplied sufficient processing power.

Of particular interest is the relative performance of the three multiprocessor architectures when six processors are employed, since the relative connectivity of the interconnection structures will have a more significant effect on the results. Figure 17 compares the curves produced by all three structures when one-half and one-fourth length messages are used. The effects of emulation overhead have been factored out in these graphs, so that direct performance comparison is possible. The ring and nearest neighbor networks provide similar amounts of processing time to their application tasks, though at long message lengths the nearest neighbor does perform slightly better. In all cases the fully connected architecture performs markedly better, performing almost as well at long message lengths as the other two architectures at short messages lengths. At an external message rate of 20 messages per second the fully connected network provides almost 2.5 times the processing potential of the nearest neighbor network. Although the fully connected network requires fifteen interconnection links to the nearest neighbor's seven, it would probably be the better choice.

These experiments demonstrate that it is possible to use Cm* to emulate other multiprocessor architectures. The methods employed can easily be extended to emulate additional multiprocessors covering a large class of interconnection mechanisms as well. Though some of the overhead incured in emulating the routing algorithm's is due to the emulation software, this can be measured by comparison with nonemulated systems such as the basic experiment. Once the overhead is subtracted detailed studies of comparative network performance are possible. It is expected that the Cm* testbed will prove useful for studying general multiprocessor architecture issues as well as designing specific multiprocessors for specific applications.

References

1. G. B. Adams and H. J. Siegel. "On the Number of Permutations Performable by the Augmented Data Manipulator Network." *IEEE Transactions on Computers C-31*, 4 (April 1982), 270-277.

2. G. A. Anderson and E. D. Jensen. "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples." *Computing Surveys* 7, 4 (Dec. 1975), 197-213.

3. K. Brayer and V. Lafleur. "A Testbed Approach to the Design of a Computer Communication Network." *Computer 15*, 10 (October 1982), 14-23.

4. T. Feng. "A Survey of Interconnection Networks." *Computer 14*, 12 (Dec. 1981), 12-27.

5. M. J. Flynn. "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers C-21*, 9 (Sept. 1972), 948-960.

6. L.S. Haynes, R.L. Lau, D.P. Siewiorek, D.W. Mitzell. "A Survey of Highly Parallel Computing." *Computer* (January 1982), 9-24.

7. D. H. Lawrie. "Access and Alignment of Data in an Array Processor." *IEEE Transactions on Computers C-24*, 12 (December 1975), 1145-1155.

8. W. C. McDonald and R. W. Smith. "A Flexible Distributed Testbed for Real-Time Applications." *Computer 15*, 10 (October 1982), 25-39.

9. J.K. Ousterhout, D.A. Scelza, and P.S. Sindhu. "Medusa: an experiment in distributed operating system structure." *Communications of the ACM 23*, 2 (Feb. 1980), 92-105.

10. F. P. Preparata and J. Vuillemin. "The Cube-Connected Cycles: A Versatile Network for Parallel Computation." *Comunications of the ACM 24*, 5 (May 1981), 300-309.

11. A. Singh. Pegasus-A Workload Generator for Multiprocessors. Master Th., Carnegie-Mellon University, Department of Electrical Engineering, 1981.

12. Richard J. Swan. *The switching structure and addressing architecture of an extensible multiprocessor, Cm**. Ph.D. Th., Carnegie Mellon University, August 1978.

13. R.J. Swan, S.H. Fuller, and D.P. Siewiorek. "Cm* -- a modular, multi-microprocessor." *National Computer Conference, Proceedings, AFIPS 46* (1977), 637-44.

14. R.J. Swan, A. Bechtolsheim, K.W. Lai, and J.K. Ousterhout. "The implementation of the Cm* multi-microprocessor." *National Computer Conference, Proceedings, AFIPS 46* (1977), 645-55.

15. K. J. Thurber. Interconnection Networks - A survey and assessment. AFIPS Conf Proceedings, 1974, pp. 909-919.

SLOT-BASED MULTI-ACCESS PROTOCOL FOR LOCAL COMPUTER NETWORK

A.I. Noor — Carleton University
G.S. Hope, O.P. Malik — University of Calgary

ABSTRACT

This paper presents a novel protocol for a Local Area Network with single channel communication. Users are grouped according to their physical location and each group is assigned a channel slot. Conflicts are resolved by an Assigned Slot Carrier Sense Multiple Access Mechanism with Collision Detection Capability (ASCD) protocol. The maximum throughput and average delays are evaluated. The results indicate that the performance of this protocol is better than many contention type protocols widely used in Local Area Networks. This protocol is best suited to transaction oriented messages, which are found in the real-time process control industry.

1. INTRODUCTION

Current trends in hardware encourage the abandonment of a single large computer in favour of a number of small machines. The resulting decentralization of computing power is, for many applications a natural and obvious pattern. In these applications, the information itself is distributed in nature and is best managed by a network of machines. Thus, it has become very attractive to connect a number of microcomputers to form a resource sharing computer network. Reliability and throughput are improved. Small processors can be designed and implemented more quickly thus it becomes practical to update to the latest and most cost-effective hardware technology.

Applications such as plant and laboratory automation are made possible by computer networking. In these applications each microcomputer station monitors and controls an elementary part of the overall plant. Like any other communication network, this network must have a communication medium (channel). The microcomputer stations require an interface in order to transmit/receive messages using the channel. The need for channel access protocol arises because the communication medium is shared by a number of stations. Conflicts which arise when more than one demand is simultaneously placed upon the channel give rise to multi-access protocol. Contention type protocol is attractive if each station independently decides when to transmit. Carrier Sense Multiple Access with Collision Detection (CSMA-CD) [1] is a popular contention protocol. Each user senses the channel and if the channel is idle starts its transmission. When a collision is detected the transmission ceases and the channel is jammed for a period (equivalent to 30 bits in Ethernet) [2] to make sure that all the users are aware of the collision. Colliding users each select a random backoff delay after which retransmission is attempted. Metcalfe and Boggs [2] proposed a binary exponential backoff algorithm. The performance of this algorithm and similar backoff algorithms has been investigated by various authors [3-6].

The main reason for collision is the absence of coordination among users, and long distance physical separation. One user does not know when another user starts message transmission.

In this paper an alternative to CSMA-CD protocol that eliminates the need of a collision enforcement mechanism is proposed. This protocol is designed to minimize the collision resolution time and hence maximize the channel throughput, by eliminating collision enforcement and the probability of repeated collision.

In Section 2 of this paper the alternative protocol is described. Section 3 describes the simulation model. Sections 4 & 5 present the result for simulation tests. Section 6 describes the design of the microcomputer network, based on this protocol.

2. ASSIGNED-SLOT-CARRIER SENSE MULTIPLE ACCESS PROTOCOL WITH COLLISION DETECTION (ASCD)

In this proposed protocol users monitor the communication channel and record its state. A user attempts transmission only when the channel is in the idle state. If a collision is detected, all but one user backs off immediately, the unsuccessful user attempts retransmission when the channel becomes idle.

Under this scheme the channel is divided into time frames each containing an equal number of slots. The users connected to the network are divided into a number of groups. The users are grouped according to their physical separation. The two furthest members of a group can be d kilometers apart

$$d < \frac{2}{3}\frac{c}{f}$$

where c = velocity of light,
f = the operating frequency of the communication medium.

Typically, $f = 10^6$ bits/sec and the distance, d, is less than 0.1 kilometers. If this distance is exceeded complete backoff is required.

Each group is assigned a slot in a time frame. The members sense the channel and transmit in its assigned slot. Sensing and transmission is started at the slot boundary. There is still a chance that two users in the same group sense the channel as idle and start transmission at the same time. In this event collision causes the users to backoff and reschedule their transmission.

Performance Measurement Criteria

The foremost measure of the network's performance is (i) channel utilization, and (ii) average message delay.

The channel utilization is the "ratio" of the time the network is successfully carrying packets and the time the network is busy. The average message delay is defined as the average interval between a user's desire to transmit and the successful reception of the packet by the destined user.

0190-3918/83/0000/0172$01.00 © 1983 IEEE

3. SIMULATION MODEL

A simulation program, to determine the performance of the protocol, was written in SIMULA [7] and used the report generating facilities of DEMOS [8]. The simulation program is based on the following assumptions:

- Poisson message arrival.
- The packet lengths are uniformly distributed.
- The packet has a maximum length of 1024 bits.
- A user either transmits or receives.
- The channel is assumed noiseless.
- Colliding users backoff for a random time.
- A user can migrate to a slot without any delay. Collisions are detected within one bit.

The parameter values used in the simulation are given in Table I.

TABLE I

PARAMETERS	VALUE
Speed of the channel	8 million bits per second
Maximum Packet length	1024 bits
Minimum Packet length	256 bits
Number of users	63
Number of slots	8
Length of the slot	8 bits
Number of Maximum Tries	32

4. SIMULATION RESULTS

Several observations are made about ASCD protocol. Backoff probability plays an important role. Figure 1 shows that channel utilization is improved with higher sensing probability, ρ. As ρ approaches 0, the delay gets arbitrarily large (due to the large retransmission delays). For $\rho > 0.1$ the relative change in performance becomes small.

Average delay are shown in Figure 2. The knee moves to the right as the packet size increases indicating higher utilization. The variable packet has an average delay of less than 0.5 milliseconds for average utilization of up to 0.92.

In the real-time environment the average load is not high, but faster consistent response is required. Figure 3 shows both the mean and the standard deviation of the delay due to uniformly distributed packet size. This indicates that the protocol can perform well without the loss of stability.

5. PERFORMANCE SUMMARY

A number of observations are made in this analysis and they are:

- The protocol is stable and utilization is a decreasing function of load.
- The variance in response time is small.
- The slot switching mechanism improves the channel utilization and reduces the average delay.
- This protocol is better than other contention base protocols, because of its shorter response time and increased effective transmission rate.

Figure 4 compares the delay-load relationship of the ASCD protocol for various packet sizes. For $\beta = 64$ and 128 there is an improvement in performance over other protocols in the 10^4 to 10^5 packets/sec average arrival rate range. For $\beta = 512$ and 1024 there is an improvement throughout the whole average arrival rate range due to the effect of slot switching.

Figure 5 describes the channel utilization for the ASCD over the arrival rate range. This protocol has improved response and stability.

6. THE DESIGN OF THE PROTOCOL HARDWARE

The goal is to design and build an economical, efficient network to interconnect user stations. Each station is likely to have an 8/16 bit microcomputer, mass storage or peripheral controller. Our particular application is the real-time operation of an Electric Substation in which certain messages must have priority. These characteristics are found in other real-time process control applications which require a hierarchical or super user relationship between stations.

The node has two parts. The first part is the user module which is the plant processor part of each station. The second part or Universal Interface Module, (UIB) executes the communication protocol. The UIB is built around an 8-bit microcomputer with special hardware to form a message from a packet. The UIB formats the message from the user's module into a packet with destination address(es) and information identifiers. The message part is submitted directly from the memory of UIB-processor. The remainder of the packet frame and the collision control is provided by the communication medium interface. An 8-bit shift register, identifies its time-slot and also identifies adjacent time-slots for possible migration.

The UIB hardware is based on INTEL-8085 processor with 2K of RAM and 4K of ROM, memory-mapping logic, programmable interrupt controller, and programmable interrupt time. The complete hardware portion uses about 100 IC's.

CONCLUSION

This protocol has been developed for the real-time application where messages are mostly transaction oriented. Advantages as confirmed through simulation are:

- shorter response times
- increased effective channel utilization
- message priority discussion

The cost of the UIB is anticipated to be reasonable since it has been implemented with about 100 IC's and a microcomputer. The general constraints are limited physical distance between stations and short messages.

REFERENCES

(1) R.M. Metcalfe, D.R. Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, Comm. ACM, Vol. 19, pp. 395-404, July 1976.

(2) *The Ethernet - A Local Area Network, Data Link Layer and Physical Layer Specification*, Intel, Dec, publication, Version 1.0, September 30, 1980.

(3) S.S. Lam, *A Carrier Sense Multiple Access Protocol for Local Networks*, Computer Networks, Vol. 4, pp. 21-32, February 1980.

(4) J. Shock and J. Hopp, *Performance of an Ethernet Local Network - A Preliminary Report,* In Proc. Local Area Communications Network Symposium, NBS and THE MITRE Corporation, Boston, May 1979.

(5) F. Tobagi and B. Hunt, *Performance Analysis of Carrier Sense Multiple Access with Collision Detection,* Technical Report 1973, Computer Science Laboratory, Stanford University, June 1979.

(6) W. Bux, *Local-Area Subnetworks: A Performance Comparison,* IEEE Trans. on Communications, Vol. COM-29, No. 10, pp. 1465-1473, October 1981.

(7) O.J. Dahl, B. Myhrhaug and K. Nygward, *SIMULA Information: Common Based Language,* Norwegian Computing Centre, Oslo, Norway, 1970.

(8) A.B. Birtwistle, *DEMOS Implementation Guide and Reference Manual,* Research Report No. 81/70/22, Department of Computing Science, University of Calgary, Calgary, 1982.

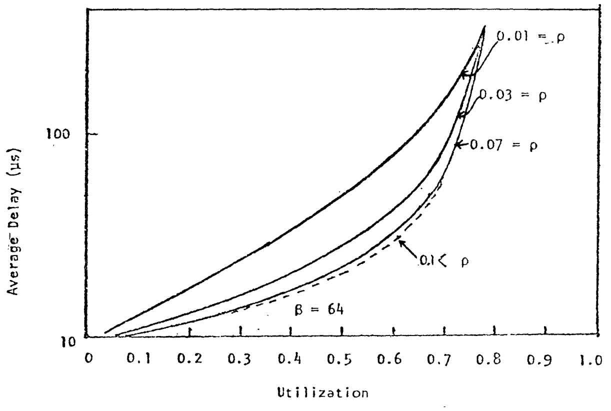


Figure 1. The effect of channel utilization at different backoff rate

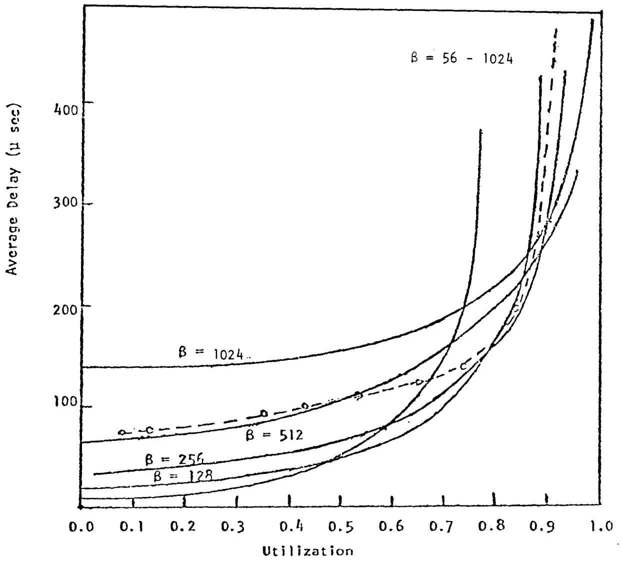


Figure 2. Average delay as seen by fixed and variable size packet

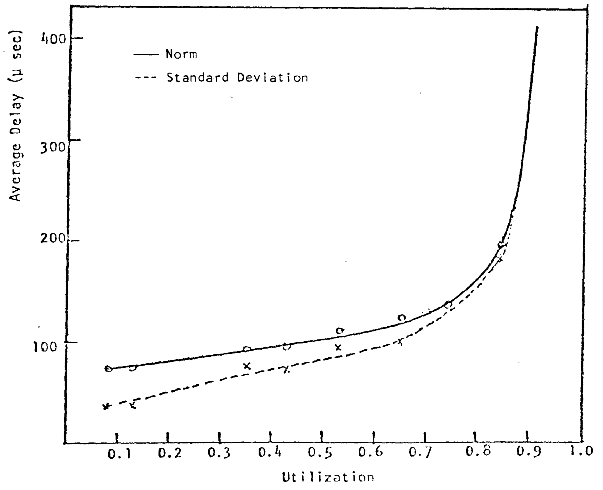


Figure 3. Comparison of norm and standard deviation due to uniformly distributed packets

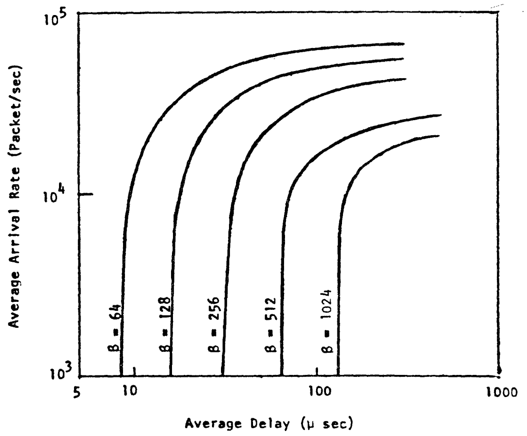


Figure 4. Comparison of delay time against various loads

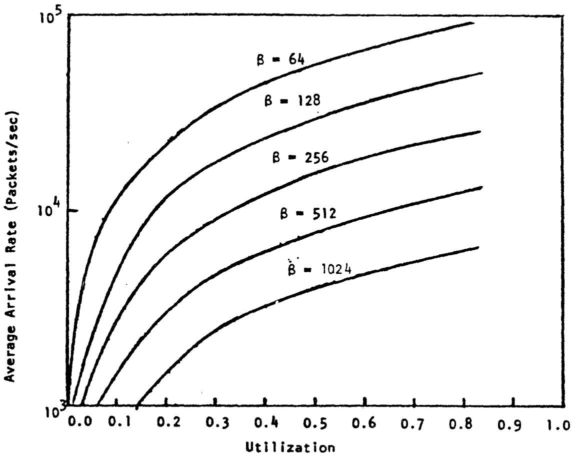


Figure 5. Channel utilization at various network loads

NEW CONNECTIVITY AND MSF ALGORITHMS FOR ULTRACOMPUTER AND PRAM

B. Awerbuch   Y. Shiloach
IBM - Israel Scientific Center
Technion, Haifa, Israel

ABSTRACT

Parallel ULTRACOMPUTER algorithms for finding the connected components and a minimum spanning forest (MSF) of an undirected graph are presented: Both have depth of $O((\log n)^2)$ where n is the number of vertices in the graph and n^2 processors are used. Both algorithms are implementations of PRAM algorithms that are presented first.

The connectivity PRAM algorithm is a simplification of the one appearing in [10]. A modification of this algorithm yields a simple and efficient MSF algorithm. Both have depth of $O(\log n)$ and they use $2E$ processors, where E is the number of edges in the graph.

1. INTRODUCTION

The ULTRACOMPUTER (later denoted as UC) and the PRAM are two models of parallel computation. The first consists of a set of processors, each having a local memory and random-access capabilities. They communicate via a shuffle-exchange network. A detailed description of this model and several basic algorithms in it, can be found in [6] and [7]. In the PRAM model the processors share a common memory and each of them has access to each memory cell. Variants of this model differ in the capability of performing concurrent READs and/or WRITEs from the same memory cell. (See [9].)

In this paper we describe two UC algorithms, for computing connected components and minimum spanning forest in an undirected graph. The first problem has an $O((\log n)^4)$ solution using $n+E$ processors. This algorithm is presented in [7] following a PRAM connectivity algorithm of [3], having depth of $O((\log n)^2)$ UC steps. A faster PRAM algorithm of $O(\log n)$ depth, is given in [10]. This algorithm and its proof are further simplified in this paper. A new technique of PRAM simulation by UC, realizes this algorithm on a UC in $O((\log n)^2)$ time using n^2 processors. Similar ideas lead to PRAM and UC MSF algorithms of the same depth and number of processors as in the connectivity algorithms.

Existing PRAM MSF algorithms have $O((\log n)^2)$ depth ([2] and [9]). These algorithms can be implemented in a UC in $O((\log n)^4)$ time using ideas similar to those described in [7]. No faster UC MSF algorithm is known to us.

2. A SIMPLE PRAM CONNECTIVITY ALGORITHM

Our connectivity algorithm is a modification of that appearing in [10] which has a much simpler proof of logarithmic depth. It deals with exactly the same model as in [10], i.e. simultaneous reading and writing are allowed. In the latter case one processor succeeds but we don't care which. Two processors $P(i,j)$ and $P(j,i)$ are assigned to each edge (i,j).

The notions of 'rooted tree', 'rooted star', 'pointer graph', 'shortcutting' and 'hooking' are the same as in [10]. The father and grandfather (= father of father) or a vertex i in the pointer graph are denoted by $D(i)$ and $G(i)$ respectively. Each step of the algorithm below is executed by each processor $P(i,j)$.

The Algorithm.

Step 1: Conditional hooking, (similar to Step 2 in [10]). If $G(i) = D(i)$ and $D(i) > D(j)$ then $D(D(i)) \leftarrow D(j)$.

Step 2: Unconditional star hooking.
If i belongs to a star and $D(i) \neq D(j)$ then $D(D(i)) \leftarrow D(j)$.

Step 3: If i belongs to a star then STOP.
Else $D(i) \leftarrow G(i)$ (Shortcutting).

The algorithm loops on the three steps above until Step 3 does not produce any non-trivial shortcuts.

COMMENTS:

1. In order to make the first iteration look like its successors, the input graph is slightly modified by connecting a dummy vertex i' to i for all $i=1,\ldots,n$. Moreover, the pointer graph is initialized by: $D(i') = D(i) = i$.
2. The condition 'i belongs to a star' is checked by a simple STARCHECK procedure. A field $ST(i)$ is attached to each vertex i, indicating whether it belongs to a star or not. STARCHECK is based on the observation that a vertex i does not belong to a star iff it has a nontrivial grandfather or grandson or nephew (= grandson of father).

STARCHECK

```
    ST(i) ← TRUE
    IF D(i) ≠ G(i) then ST(i),ST(G(i)) ← FALSE
    ST(i) ← ST(G(i)).
```

<u>Theorem 1</u> (partial correctness):

If the algorithm terminates then

a. The pointer graph consists entirely of stars.

b. The vertices of each star form a connected component of the graph.

0190-3918/83/0000/0175$01.00 © 1983 IEEE

Proof:

a. Follows immediately from the termination condition in Step 3.

b. It is easy to see that the partition defined by the pointer graph is always a sub-partition of that defined by the connected components. Moreover, a proper subset of a connected component cannot form a star in the pointer graph at the beginning of Step 3 since stars are unconditionally hooked in Step 2.

Theorem 2 (logarithmic depth):

a. Stars are not hooked on stars in Step 2.

b. The pointer graph is always a forest of rooted trees.

c. If C is a connected component of G then the sum of the heights of the trees in the pointer graph consisting of vertices of C decreases by a factor of 3/2, at least, after each application of the loop, until these trees form a star.

Proof:

a. A star at the beginning of Step 2 has not been changed in Step 1 since the hooking in Step 1 produces trees of height two at least, (a result of the initialization). At the beginning of Step 2 there exist no edge with both endpoints belonging to different stars. Otherwise one of the edge processors should have attempted to hook the larger root on the smaller one at Step 1 making it a non-root at the beginning of Step 2.

b. The statement is true initially. Steps 1 and 3 never produce any directed cycle. By a. stars are hooked only on non-stars in Step 2. Since non-stars are not further hooked on anything in that step, cycles are not formed in Step 2 either.

c. As a result of the initialization step, all the trees are of height one at least and therefore hooking is never done on leaves. Thus, hooking one tree on another yields a tree of height not greater than the sum of the heights of the original trees. Therefore, Steps 1 and 2 do not increase the sum of heights of any set of trees. Let C be a connected component of G. If the set of trees in the pointer graph consisting of the vertices of C, has not shrunk yet to a single star, then at the beginning of Step 3 none of them is a star. Hence, at Step 3 their sum of heights decreases by a factor of 3/2 at least.

3. A PRAM MSF ALGORITHM

As in [8], our MSF algorithm is also an implementation of Sollin's algorithm that is brought in [1]. Our improvement of their result is similar to the improvement of [10] with respect to the connectivity algorithm of [4]. The depth and number of processors are the same as in the connectivity algorithm above.

This algorithm, however, assumes a stronger model of computation in which the lowest processor writes in case of concurrent writing to the same location. As in the connectivity algorithm above, two processors P(i,j) and P(j,i) are assigned to each edge (i,j) of the graph. The algorithm needs a preprocessing assignment phase in which processors are assigned to edges in such a way that processors with lower identity numbers correspond to edges with lower lengths. In the worst case sorting of the edges with respect to their lengths might be needed which takes O(log n) or $O((\log n)^2))$ time according to the number of available processors, (see [5] and [9]).

Previous results on this problem ([2] and [8]) achieved depth of $(\log n)^2$ with n^2 and $(n^2)/((\log n)^2)$ processors respectively. Both works assume the 'concurrent READ exclusive WRITE' PRAM model.

The following variables are used.

D(i) denotes the father of i in the pointer graph.

TREE (i,j) is a Boolean variable corresponding to P(i,j). By the end of the algorithm an edge (i,j) belongs to the minimal spanning forest iff TREE(i,j) = 1 or TREE(j,i) = 1.

ATTEMPT (i) is an auxiliary variable corresponding to vertex i.

L(i,j) is the length of Edge (i,j).

MSF ALGORITHM

```
Initialization: TREE(i,j) ← 0, D(i) ← i;
     ATTEMPT (i) ← NIL for all  i,j.

Step 1: Star Hooking
          If i belongs to a star and D(i) ≠ D(j)
          Then ATTEMPT (D(i)) ← (i,j)
               If (i,j) = ATTEMPT (D(i))
               Then TREE(i,j) ← 1 and D(D(i)) ← D(j)

Step 2: Tie Breaking
          If i < D(i) and i = G(i)
          Then D(i) ← i

Step 3: If i belongs to a star then STOP.
          Else D(i) ← D(D(i))
```

The algorithm loops on these three steps until Step 3 does not produce any real shortcut.

COMMENT:

All P(i,j) that want to hook D(i), try competitively to write their identities into ATTEMPT (D(i)) where the winner's identity i eventually stores. Only the winner succeeds in hooking and its edge is inserted to FOREST which is the set of all edges (i,j) for which either TREE(i,j) or TREE(j,i) are 1.

In order to prove the correctness and logarithmic depth of the algorithm the following lemmas are needed:

Lemma 1: At each step of the algorithms:

a. FOREST is a subset of an MSF.

b. The connected components of the pointer graph are identical to the connected components of FOREST and constitute a partition of the connected components of the graph.

c. The pointer graph is a directed forest **with** the exception of self-loops at roots and cycles of length 2 that exist only between Steps 1 and 2.

Proof:

a. It can be assumed that edge lengths are all different. This is achieved by taking the length of an edge (i,j) as the triple L(i,j) = (length of Edge (i,j), min (i,j), max (i,j)) and considering the lexicographic order. This yields a unique MSF. Whenever TREE (i,j) ← 1, the edge (i,j) is the shortest edge emanating from the set of vertices determined by the star rooted at D(i,j). Thus (i,j) belongs to the MSF, and therefore FOREST is always a subset of the MSF.

b. This statement can be proved inductively since whenever two connected components of the pointer graph are connected, so do the corresponding components of FOREST and the latter ones are by definition a partition of the connected components of the graph.

c. Assume that the statement is true at the beginning of the current iteration. Consider an application of Step 1. Any new edge added at this step emanates from a star root to a non-leaf vertex in another tree. Suppose that a directed cycle C was created by Step 1. Since C must contain new edges, and since an old edge cannot be followed in C by a new edge, C must contain only new edges stitching the star roots. Thus if C contains a new cycle of length > 2 so does FOREST - contradicting a. Cycles of length 2 are opened at Step 2. The proof is completed by observing that Step 3 does not introduce any new cycles to the pointer graph.

Theorem:

Upon termination FOREST is the MSF of the graph.

Proof:

It is easy to see that by the end of the algorithm the connected components of the graph coincide with those of the pointer graph. Thus by statement B above FOREST spans the graph and by A it is an MSF.

Theorem:

The algorithm terminates after at most log n iterations.

Proof:

The proof is similar to that of the corresponding statement in the connectivity algorithm and therefore omitted.

4. SUPERCOMPUTER IMPLEMENTATION OF THE CONNECTIVITY AND MSF ALGORITHMS

The SUPERCOMPUTER (later denoted as SC) is an intermediate level between the PRAM and the UC. It is a UC with a full communication network enabling direct communication between any pair of processors. Thus, each processor can read and write into each other processor's memory. However, concurrent READs and WRITEs from/to the same processor can be executed only if one memory cell is accessed. In case of WRITE the minimal value is written.

In this section we implement the PRAM algorithms above by an SC with n + 2E processors P(i,j) and P(j,i). for each edge (i,j) and P(i,i) for 1 = 1,...,n. In the next section, the SC algorithms will be further implemented by a UC.

Both SC algorithms use three basic instructions READ, WRITE and TRANSPOSE.

READ has the form: READ (OLDFIELD from P(ADR,ADR) into NEWFIELD). The exact effect of this statement is that P(i,j) reads OLDFIELD in P(ADR,ADR), and copies it into NEWFIELD which is a new field of its own. (ADR itself is a field in P(i,j)).

WRITE has the form: WRITE (OLDFIELD at P(ADR,ADR) into NEWFIELD) which means that P(i,j) writes the value stored in OLDFIELD into the new field NEWFIELD at P(ADR,ADR).

TRANSPOSE has the form: TRANSPOSE (OLDFIELD to NEWFIELD) and it transfers the value stored at P(i,j) in OLDFIELD to P(j,i) into NEWFIELD.

Each processor P(i,j) stores the following variables in the fields:

Variables:	D(i)	D(j)	G(i)	ST(i)	i	ATTEMPT(i)	L(i,j)
Fields:	D1	D2	G	ST	I	ATTEMPT	LENGTH

THE CONNECTIVITY SC-ALGORITHM

```
Step 1:  READ (D1 from P(I,I) into D1)
         TRANSPOSE (D1 to D2)
         READ (D1 from P(D1,D1) into G)
         If G = D1 and D1 > D2
         Then WRITE (D2 at P(D1,D1) into D1)

Step 2:  STARCHECK         /* See routine below */
         TRANSPOSE (D1 to D2)
         If ST = TRUE and D1 ≠ D2
         Then WRITE (D2 at P(D1,D1) into D1)

Step 3:  SHORTCUT
         If ST = TRUE then STOP
         Else READ (D1 from P(I,I) into D1)
              READ (D1 from P(D1,D1) into D1)

STARCHECK
      READ (D1 from P(I,I) into D1)
      READ (D1 from P(D1,D1) into G)
      ST ← TRUE
      If  G ≠ D1
      Then ST ← FALSE

      WRITE (ST at P(G,G) into ST)
      READ (ST from P(G,G) into ST)
```

SC MSF ALGORITHM:

```
Step 1: STARCHECK
        TRANSPOSE (D1 to D2)
        If $D1 \neq D2$ and STAR = TRUE
        Then WRITE (LENGTH at P(D1,D1) into ATTEMPT)
             READ (ATTEMPT at P(D1,D1) into ATTEMPT)
             If LENGTH = ATTEMPT
             Then WRITE (D2 at P(D1,D1) into D1)
                  TREE $\leftarrow$ 1
Step 2: READ (D1 from P(I,I) into D1)
        READ (D1 from P(D1,D1) into G)
        If G = I and $D1 > I$ then D1 $\leftarrow$ I
Step 3: SHORTCUT
        If ST = TRUE then STOP.
        Else READ (D1 from P(I,I) into D1)
             READ (D1 from P(D1,D1) into D1)
```

5. UC IMPLEMENTATION OF SC INSTRUCTIONS READ, WRITE AND TRANSPOSE

In this section the three SC operations above are simulated by an $N = n^2$ - UC. Each operation is simulated in O(log n) steps yielding total depth of $O((\log n)^2)$ for both algorithms. To each ordered pair of vertices <i,j>, a processor P(i,j) is attached regardless whether (i,j) is an edge of G or not. A single bit BIT stored in the local memory of each processor, indicates whether P(i,j) is an edge-processor or not. The edge-processors of UC simulate the corresponding SC processors while the non-edge UC processors serve for communication purposes. Only edge-processors participate in WRITEs. As usual for UC, n is assumed to be equal to 2^k for some integer k, (thus $N = 2^{2k}$). The simulation is still sub-divided into higher and lower level implementations.

5.1 The Higher Level

In this level READ, WRITE and TRANSPOSE are implemented by lower level routines BROADCAST, CHOOSE and TRANS. These low-level routines are implemented in the next subsection by the basic UC routine GROUPSUM.

CHOOSE has the form : CHOOSE (OLDFIELD into NEWFIELD). It chooses a minimal value among all non-NIL values stored at OLDFIELD of P(i,j) for fixed i and all j, and inserts it into NEWFIELD at P(i,i). If the values in OLDFIELD are NIL for all P(i,j), $1 \le j \le n$, then the resulting value of NEWFIELD at P(i,i) does not change.

BROADCAST has the form: BROADCAST (OLDFIELD into NEWFIELD). It inserts the value stored at OLDFIELD of P(i,i) into NEWFIELD of P(i,j) for j = 1,...,n.

TRANS has the form: TRANS (OLDFIELD to NEWFIELD). It inserts the value stored at OLDFIELD of P(j,i) into NEWFILED of P(i,j).

Let FO, FN and AD denote OLDFIELD, NEWFIELD and ADDRESS respectively.

READ (FO from P(AD,AD) into FN):

```
    BROADCAST (FO into F1)
    TRANS (F1 to F2)
    If $j \neq AD$
    Then F2 $\leftarrow$ NIL
    CHOOSE (F2 into F3)
    BROADCAST (F3 into FN)
```

WRITE (FO at P(AD,AD) into FN):

```
    If BIT = 0 then FO $\leftarrow$ NIL
    CHOOSE (FO into F1)
    BROADCAST (F1 into F2)
    If $j \neq AD$
    Then F2 $\leftarrow$ NIL
    TRANS (F2 to F3)
    CHOOSE (F3 into FN).
```

TRANSPOSE (FO to FN):

```
    TRANS (FO to FN)
```

5.2 The Lower Level

Let '+' denote any associative binary operation and let $A=a_1,\ldots,a_N$ be any N-vector where a_i is stored in P_i for all i. The output of (LEFT) SUM is the vector of partial 'sums' $a_1, a_1+a_2, a_1+a_2+a_3, \ldots, a_1+\ldots+a_N$ such that $a_1+\ldots+a_i$ is stored in P_i for all i. The LEFT GROUPSUM (LGS) operation is similar to SUM. The difference is that here the vector $a_1,\ldots, a_N$ is divided into several segments and the result is that of SUM operating on each segment separately. When the summation starts from the rightmost element of each segment we obtain the RIGHT GROUPSUM (RGS).

GROUPSUM of a vector of length N can be done in log N time by an N-UC as shown in SCH-80.

Realization of CHOOSE and BROADCAST by GROUPSUM.

In the following realizations, our GROUPSUM segments consist of all P(i,j) for a fixed i. Our associative binary operation is defined by: A+B = min(A,B) where NIL is considered as bigger than any non-NIL value.

CHOOSE (FO into FN):

```
    F1 $\leftarrow$ LGS (FO)
```

/* At this point the field F1 in P(i,n) contains a minimal non-NIL element if such exists in its segment and NIL otherwise. In the first case this non-NIL element is inserted to FN at P(i,i). */

```
    If P(i,j) = P(i,n) and $F1 \neq$ NIL
    Then FN $\leftarrow$ RGS (F1)
```

BROADCAST (FO into FN):

```
    If $P(i,j) \neq P(i,i)$
    Then FN $\leftarrow$ NIL
    Else FN $\leftarrow$ FO
    LGS (FN)
    If $P(i,j) \neq P(i,n)$
    Then FN $\leftarrow$ NIL
    RGS (FN)
```

TRANS (FO to FN):

```
    FN $\leftarrow$ FO
    DO log n times:
    FN $\leftarrow$ SHUFFLE (FN)
```

/* SHUFFLE is the basic UC operation defined by:

$$\mathrm{SHUFFLE}(i) = \begin{cases} 2i & \text{if } i < N/2 \\ 2i - N + 1 & \text{else} \end{cases}$$

TRANS is exactly the matrix transposition algorithm appearing in [6]. */

REFERENCES

[1] C. Berge and A. Ghouila-Houri, 'Programming, Games and Transportation Networks', Wiley and Sons, New York, 1965.

[2] F.Y. Chin, J. Lam and I. Chen, 'Efficient Parallel Algorithms for Some Graph Problems', CACM, Vol. 25, No.9 (Sept. '82), p. 659.

[3] D.S. Hirschberg, 'Parallel Algorithms for the Transitive Closure and the Connected Component Problem', Proc. of the 8-th Annual ACM Symposium on Thoery of Computing, Hershey, PA 1976, p. 55.

[4] D.S. Hirschberg, A,K. Chandra and D.V, Sarwate, 'Computing Connected Components on Parallel Computers', CACM, Vol. 22 (1979), p. 461.

[5] D. Nassimi and S. Sahni, 'Parallel Permutation and Sorting and an New Generalized Connection Network', JACM, Vol. 29, No.3, (July '82), p. 642.

[6] H. Stone, 'Parallel Processing with a Perfect Shuffle', IEEE Trans. on Computers, Vol. 20, No.2, (Feb. '71) p. 153.

[7] J.T. Schwartz, 'Ultracomputers', ACM TOPLAS 2 (1980), p. 484.

[8] C. Savage and J. Ja'Ja, 'Fast Efficient Parallel Algorithms for Some Graph Problems', SIAM J. on Computing, Vol. 10, No.4 (Nov. '81), p. 682.

[9] Y. Shiloach and U. Vishkin, 'Finding the Maximum, Merging and Sorting in a Parallel Computation Model', J. of Algorithms, Vol. 2, (1981), p. 88.

[10] Y. Shiloach and U. Vishkin, 'An O(log n) Parallel Connectivity Algorithm', J. of Algorithms, Vol. 3 (1982), p. 57.

Bridge-connectivity and Biconnectivity Algorithms for Parallel Computer Models

Yung H. Tsin
Department of Computing Science
University of Alberta
Edmonton,Alberta,Canada T6G 2H1

Abstract

New algorithms for the bridge-connectivity and biconnectivity problems are presented. It is shown that these algorithms can be implemented on *any* parallel computer models on which an ordinary matrix multiplication algorithm exists and that the hardware resources required(in terms of the number of processors and the chip area) are no more than those required by the matrix multiplication algorithm. The time required is at most a factor of $\max(\log d, \log d'')+1$, $1 \le d, d'' \le n$, greater than that needed by the matrix multiplication algorithm. Since most of the existing parallel computer models have efficient ordinary matrix multiplication algorithms, the algorithms presented here turn out to be very efficient.

1. Introduction

The result of this paper is motivated by the following observations.

1. For most of the existing parallel computer models, there exist few algorithms for graph theoretic problems. This is especially true for those models which obey the VLSI design constraints[8,9,14]. On the other hand, an important basic operation, namely the matrix multiplication, has an efficient algorithm on almost every parallel computer model[3,8,9,11]. Since graphs are usually stored in the form of matrices in computers, it is conceivable that matrix multiplication or the techniques it uses may be useful in developing efficient algorithms for graph theoretic problems. As a matter of fact, Dekel, Nassimi and Sahni have exploited this idea[3].

2. It usually happens that whenever an algorithm is presented, it is designed with a particular model in mind, and its complexity analysis is provided for that model only. Extra effort has to be made in order to carry it over to the other models. A typical example is Hirschberg's graph-connectivity algorithm which was originally designed for the SIMD-SM-R model[2,4]. It was then implemented on the SIMD-MCC model by Nassimi and Sahni[6]; on a SIMD-SM-RW by Shiloach and Vishkin[13,17] and finally on the PSN, OTN, and OTC models by Nath, Maheshwari and Bhatt[7,8]. It would be convenient if the complexity analysis of an algorithm could be given in such a way that it would be valid for *any* model satisfying certain reasonably moderate conditions.

In this paper, we shall design parallel algorithms for the bridge-connectivity and biconnectivity problems. We then analyze their complexities for *any* parallel computer models on which an ordinary matrix multiplication algorithm exists. Since almost every existing model has an efficient algorithm for matrix multiplication, the condition imposed is not severe. Let $O(t(n))$ and $H(n)$ denote the time and hardware resources(in terms of number of processors and chip area) required by the ordinary nxn matrix multiplication algorithm. We will show that the time and hardware complexities of our algorithms are *bounded above* by $O(t(n)*(\max(\log d, \log d'')+1))$, $1 \le d, d'' \le n$, and $H(n)$ respectively on those models.

2. Basic Definitions

The definitions of the graph theoretic terms used in this paper are standard and can be found in various texts.

Definition: let $T(V,E')$ be a directed tree and $u,v \in V$.

$u \preceq v$ if u is an ancestor of v in T.

$u \prec v$ if u is a proper ancestor of v in T.

Definition: The *lowest common ancestor* , LCA(u,v), of two vertices u, v in T is defined as:

LCA(u,v) = $(\max \preceq)\{w \mid w \preceq u \text{ and } w \preceq v\}$.

Definition: Let $v \in V$, the *level* of v in T is defined as:

$\mathit{level}(r)=1$;

$\mathit{level}(v)=\mathit{level}(F(v))+1$, $v \ne r$.

where r is the root of T and $F(v)$ is the father of v in T.

Note that $\mathit{level}(v)>0\ \forall v \in V$.

We denote $\mathit{level}(\mathrm{LCA}(u,v))$ by $l\mathrm{LCA}(u,v)$.

Definition: Let $v \in V$ and T is a directed spanning tree of an undirected graph $G(V,E)$.

$\mathrm{HLCA}(v)=(\min \preceq)(\{\mathrm{LCA}(v,w) \mid (v,w) \text{ is in } G-T\} \cup \{v\})$.

$l\mathrm{HLCA}(v) = \mathit{level}(\mathrm{HLCA}(v))$.

$\alpha(v)=\min\{l\mathrm{HLCA}(w) \mid v \preceq w\}$.

3. Outline of the Algorithms

Algorithm:Bridge-connectivity

```
1. Find a directed spanning tree $T(V,E')$ of $G(V,E)$;
2. Compute $l\mathrm{LCA}(u,v)\ \forall (u,v) \in V \times V$;
3. Compute $l\mathrm{HLCA}(v)\ \forall v \in V$;
4. Compute $\alpha(v)\ \forall v \in V$;
5. Test if $\mathit{level}(v) \le \alpha(v)\ \forall (u,v) \in E'$. (* $(u,v)$ is a bridge iff $\mathit{level}(v) \le \alpha(v)$ *)
6. Delete all the bridges in $G$ and find the connected components of the resulting graph. (* These are the bridge-connected components of $G$ *)
```

Algorithm:Biconnectivity

```
1. Find a directed spanning tree $T(V,E')$ of $G(V,E)$;
2. Compute $l\mathrm{LCA}(u,v)\ \forall (u,v) \in V \times V$;
3. Compute HLCA$(v)$ and $l\mathrm{HLCA}(v)$ $v \in V$;
4. Construct an undirected graph $G''(E',E'')$ such that
       $((F(u),u),(F(v),v))\ E''$ iff   $\mathrm{HLCA}(u) \prec v \preceq u$ or $\mathrm{HLCA}(v) \prec u \preceq v$
                                        or $(u,v)$ is in $G-T$ and $u \not\preceq v$, $v \not\preceq u$.
5. Find the connected components $\{C_i\}$ of $G''$. (* Each of them uniquely determines a biconnected component in $G$ and vice versa. *) Find the roots $\{spt_i\}$ of the induced trees $\{T \cap C_i\}$. (* $\{spt_i\}$ forms the set of separation vertices of $G$. $r$ is excluded unless $r=spt_i=spt_j$ for some $i \ne j$ *)
```

4. Correctness of the Algorithms

The correctness is based on the following theorems.

Lemma 4.1: If e is a bridge in G, then $e \in E'$.

Theorem 4.2: $(F(v),v)\ E'$ is a bridge in G iff $\mathit{level}(v) \le \alpha(v)$.

Proof: If $(F(v),v)$ is a bridge, then there is no fundamental cycle containing $(F(v),v)$ in G. Therefore for all descendants w of v in T, $l\mathrm{HLCA}(w) \ge \mathit{level}(v)$. Hence $\mathit{level}(v) \le \alpha(v)$.

If $(F(v),v)\ E'$ is not a bridge, then there is a fundamental cycle C containing $(F(v),v)$. Let C be determined by (x,y) where (x,y) is in $G-T$. Without loss of generality, we assume $v \preceq x$. Clearly, $\alpha(v) \le l\mathrm{HLCA}(x) \le l\mathrm{LCA}(x,y) < \mathit{level}(v)$. □

Denote uRv iff $((F(u),u),(F(v),v)) \in E''$. It is easy to prove the following lemma:

Lemma 4.3: If $w_1 R w_2$, $w_2 R w_3$, ..., $w_{\ell-1} R w_\ell$, then $(F(w_1),w_1)$ and $(F(w_\ell),w_\ell)$ belong to the same cycle in G.

Theorem 4.4: $(F(u),u)$ and $(F(v),v)$ belong to the same connected component in G'' iff $(F(u),u)$ and $(F(v),v)$ belong to the same biconnected component in G.

Proof: The 'only if' part is obvious due to Lemma 4.3.

Let $(F(u),u)$ and $(F(v),v)$ belong to the same biconnected component in G. There exists a simple cycle C containing $(F(u),u)$ and $(F(v),v)$. Let $C_1, C_2, ..., C_\ell$ be the set of fundamental cycles such that $C=\oplus\{C_i\}_{1 \le i \le \ell}$($\oplus$ is the mod-two sum). Without loss of generality, let $(F(u),u) \in C_1$, $(F(v),v) \in C_\ell$, and $(F(w_i),w_i) \in C_i$ and C_{i+1} $1 \le i < l$. Let (a_i, b_i) be the edge in $G-T$ determining C_i $1 \le i \le l$, then in each C_i , one of the following must hold true: (i) $a_i R b_i$ and ($w_{i-1} R a_i$ or $w_{i-1} R b_i$) and ($w_i R a_i$ or $w_i R b_i$); (ii) $a_i R w_{i-1}$ and $a_i R w_i$; (iii) $b_i R w_{i-1}$ and $b_i R w_i$. In any of the cases, there

0190-3918/83/0000/0180$01.00 © 1983 IEEE

is a path from $(F(w_{i-1}),w_{i-1})$ to $(F(w_i),w_i)$. In particular, there is a path from $(F(u),u)$ to $(F(w_1),w_1)$ in C_1 and a path from $(F(w_\ell),w_\ell)$ to $(F(v),v)$ in C . Joining all these paths together, we have a path from $(F(u),u)$ to $(F(v),v)$ in G''. Hence, $(F(u),u)$ and $(F(v),v)$ belong to the same connected component in G''. □

5. An Implementation Based on Matrix Multipliction

On the parallel computer models we consider, we assume that there exists an ordinary matrix multiplication algorithm and that each processor is capable of carrying out any of the operations $+,-,*,\wedge,\vee,\neg,=,\neq,\leq,\geq$ in constant time. 0 is used to represent the boolean constant 'true' while 1 is used to represent 'false'. Each processor contains a constant number of registers. Communication between interconnected processors and between registers within the same processor takes constant time. The undirected graph is represented by an adjacency matrix M such that entry $M[i,j]$ is stored in $PE[i,j]$. A register, say A, in $PE[i,j]$ is denoted by $A[i,j]$. Without loss of generality, we assume G is connected and $V=\{1,2,3,\dots,n\}$.

Definition: A function f is called an **extended monadic function w.r.t. i, j** if the arguments of f are of the form $OP[i,j]$ where OP is either the name of a register or a function of i, j. we denote it by $f[i,j]$.

Definition: An **ordinary matrix multiplication algorithm** is an algorithm which uses only the associativity of +.

Lemma 5.0: If an ordinary matrix multiplication algorithm for two $n\times n$ matrices exists on a computer model, then the following operation could be carried out on the same model using the same order of magnitude of time and hardware resources.

$$M[i,j]:=f_3(\underset{k}{h}(g(f_1[i,k],f_2[k,j]))) \ \forall\ i,j,\ 1\leq i,j\leq n,$$

where f_1, f_2, f_3 are extended monadic functions w.r.t. i,k; k,j and i,j respectively. g is a composite function of the arithmetic and boolean operations mentioned above and h is an associative operator.
Proof: Trivial. □

Lemma 5.1: The time and hardware resources needed to broadcast the contents of register $M[a,b]$ columnwise (rowwise) is at worst the same as that needed by the ordinary matrix multiplication algorithm.
Proof: To broadcast the contents of $M[a,b]$ columnwise, we perform

$$M[w,b] = \sum_k ((M[w,k]*0)+(M[k,b]*(k=a)))\ \ 1\leq w\leq n.$$

Clearly, $M[w,b]=M[a,b]$, $1\leq w\leq n$. By Lemma 5.0, the lemma follows. Broadcasting rowwise is handled in a similar way. □

Lemma 5.2: Suppose there exists an ordinary matrix multiplication algorithm on a computer model. Then the all-pair shortest path of an undirected graph G with diameter d can be determined in $O(t'(n))$ time with $H'(n)$ processors on that model,
where $t'(n)=t(n)*(\lceil\log d\rceil+1)$ and $H'(n)=H(n)$
Proof: Construct matrix D such that

$$D[i,j]=\begin{cases}1 \text{ if } M[i,j]=1 \text{ and } i\neq j;\\ 0 \text{ if } i=j;\\ +\infty \text{ if } M[i,j]=0.\end{cases}$$

Compute the matrix D^d as follows:

$$D^1 = D$$

$$D^{2^i}[u,v]=\min_w(D^{2^{i-1}}[u,w]+D^{2^{i-1}}[w,v]),\ i\geq 1.$$

Clearly, $D^{2^i}[u,v]$ contains the the shortest distance from u to v containing no more than 2^i edges. Therefore after $\lceil\log d\rceil$ iterations, $D^d[u,v]$ contains the shortest distance from u to v in G. One more iteration is required to verify that D^d has been computed. By Lemma 5.0, $t'(n)=t(n)*(\lceil\log d\rceil+1)$ and $H'(n)=H(n)$. □

Lemma 5.3: Computing the transitive closure M^+ takes $O(t'(n))$ time with $H(n)$ hardware resources.
Proof: $M^+[a,b]=1$ iff $D^d[a,b]\neq+\infty$. □

A detailed implementation of Algorithm Bridge-connectivity based on matrix multiplication.

```
(1). Given the $n\times n$ adjacency matrix M of $G(V,E)$ where $M[i,j]$ is stored in processor $PE[i,j]$ $1\leq i,j\leq n$. (* We shall construct a directed breath-first search spanning tree for $G$ *)
    1.1 Compute the all-pair shortest path matrix $D^d$
    1.2 Choose a vertex $r$ (say 1) as the root.
        $level[r,v] := D^d[r,v]+1$. (* $level(v)=level[r,v]$ *)
        (Broadcast columnwise) $level[k,v] := level[r,v]\ \forall v\in V$.
        (Broadcast rowwise) $level[v,k] := level[v,v]\ \forall v\in V$.
        (* At this point, $level(v)=level[v,v]$ *)
    1.3 Compute matrix $F$:
        $F'[v,j] := \bigvee_k (level[v,k]=((1+level[k,j])*(k=j)))$.
            (* $F'[v,j]=1$ iff $level(v)=level(j)+1$ *)
        $F[v,j] := \max_k((F'[v,k]\wedge M[v,k])*k+M[k,j]*0)$.
        (* $F(v)=F[v,v]$, for $v\neq r$, is chosen as the father of $v$ *)
    1.4 (* Construct the adjacency matrix $T$ for the BFS spanning tree, and $T'$, the transpose of $T$ *)
        (Broadcast columnwise:) $T[k,v] := F[v,v]\ \forall v\in V$.
        (Broadcast rowwise:) $T'[v,k] := F[v,v]\ \forall v\in V$.
            $T[w,v] := (w=T[w,v])\ \forall\ v,w\in V$.
            $T'[v,w] := (w=T'[v,w])\ \forall v,w\in V$.
(2). 2.1 Compute the transitive closure $T^+$ and $(T')^+$ of $T$ and $T'$.
    (* Note that $T^+[v,v]=1$ and $(T')^+[v,v]=1\ \forall v\in V$ *)
    2.2 $lLCA[i,j] := \max_k\{(T')^+[i,k]*(T^+[k,j]*level[k,j])\}$
    (* $lLCA[v,v] = level(v)\ \forall v\in V$ *)
(3). 3.1 Construct the adjacency matrix $M'$ of $G-T$.
        $M'[i,j] := M[i,j]\wedge(\neg(T[i,j]\vee T'[i,j]))$. (* $M'[v,v]=1\ \forall v\in V$ *)
    3.2 $lHLCA[i,j] := \min_k\{lLCA[i,k]*M'[k,j]\}$.
    (* $lHLCA[v,v] = lHLCA(v)$ and $lHLCA[v,v]\leq level(v)\ \forall v\in V$ *)
(4). 4.1 (Broadcast rowwise:) $lHLCA[v,w] := lHLCA[v,v]\ \forall w\in V$.
    4.2 $\alpha[i,j] := \min_k\{T^+[i,k]*lHLCA[k,j]\}$ (* Note: $\alpha[v,v]=\alpha(v)$ *)
(5). 5.1 $Bridge[u,v] := \bigvee_k(T[u,k]\wedge((\alpha[k,v]\geq level[k,v])\wedge(k=v)))$.
(6). 6.1 Compute the matrix $M''$: $M''[i,j] := M[i,j]\wedge(\neg Bridge[i,j])$.
    6.2 Compute $(M'')^+$. □
```

The method used in step 1 to find a directed BFS spanning tree was first 'implicitly' given by Savage[11]. It also appeared in [1,3]. The resource complexities are as follows:

Step	time	hardware	resources
1.1	$O(t'(n))$	$H(n)$	Lemma 5.2
1.2	$O(t(n))$	$H(n)$	Lemma 5.1
1.3	$O(t(n))$	$H(n)$	Lemma 5.0
1.4	$O(t(n))$	$H(n)$	Lemma 5.1
2.1	$O(t'(n))$	$H(n)$	Lemma 5.3
2.2	$O(t(n))$	$H(n)$	Lemma 5.0
3.1	$O(1)$	$H(n)$	trivial
3.2	$O(t(n))$	$H(n)$	Lemma 5.0
4.1	$O(t(n))$	$H(n)$	Lemma 5.1
4.2	$O(t(n))$	$H(n)$	Lemma 5.0
5.1	$O(t(n))$	$H(n)$	Lemma 5.0
6.1	$O(1)$	$H(n)$	trivial
6.2	$O(t'(n))$	$H(n)$	Lemma 5.3

Theorem 5.4: Algorithm Bridge-connectivity takes $O(t(n)*(\log d+1))$time with $H(n)$ hardware resources.

A detailed implementation of Algorithm Biconnectivity based on matrix multiplication.

```
Steps (1)-(3). Same as Steps (1)-(3) of Algorithm Bridge-connectivity. (* After step 3, $level[v,k]=level(v)$, $lHLCA[v,v]=lHLCA(v)$, $\forall v,k\in V$ *)
(4). 4.1 (Broadcast rowwise:) $lHLCA[v,k] := lHLCA[v,v]$, $\forall v\in V$.
        (Broadcast columnwise:) $level'[k,v] := level[v,v]$, $\forall v\in V$.
        (Broadcast columnwise:) $lHLCA'[k,v] := lHLCA[v,v]\ \forall v\in V$.
    4.2 Construct an adjacency matrix $M''$ for $G''(E',E'')$.
        $M''[u,v] := (T')^+[u,v]\wedge(level'[u,v]>lHLCA[u,v])$.
        $M''[u,v] := M''[u,v]\vee(T^+[u,v]\wedge(level[u,v]>lHLCA'[u,v]))$
        $M''[u,v] := M''[u,v]\vee M'[u,v]$
    (* Since each $v$ uniquely determines $F(v)$, we use $v$ to represent $(F(v),v)$ in the vertex set of $G''$ *)
(5). 5.1 Compute $(M'')^+$.
    5.2 (* $F(v)=F[v,v]$, $\forall v\in V$ after step 1.4 *)
        (Broadcast columnwise:) $F[k,v] := F[v,v]$, $\forall v\in V$.
        Compute the matrix $subroot$: $\forall\ v\neq r$ :
        $subroot[v,j] := (\min_k\preceq)\{((M'')^+[v,k]*F[v,k])+(F[k,j]*0)\}$.
        (* $subroot[v,k]$, $\forall k\in V$, contains the root of the subtree (of the BFS spanning tree) containing $v$ *)
    5.3 $\forall v\neq r$: $C[v,w] := \min_k((((M'')^+[v,k]*k)+((M'')^+[k,w]*0))-\{0\})$.
            $BC[v,w] := (M'')^+[v,w]\wedge(w=C[v,w])$.
    5.4 $\forall v\in V$ such that $v=C[v,w]$ and $v\neq r$ :
        (Broadcast columnwise:) $subroot[k,v] := subroot[v,v]$;
        $BC[w,v] := BC[w,v]\vee(w=subroot[w,v])$;
        (* $BC[u,v]=BC[w,v]=1$ iff $u$ and $w$ belong to the same biconnected component represented by $v$ in $G$ *)
        (* $Flag[w,v]:=0\ \forall w,v\in V$ initially *)
        $Flag[w,v] := (w=subroot[w,v])$.
```

```
5.5 $Flagsum[u,v]:= \sum_k (Flag[u,k]+(Flag[k,v]*0)))$
    $Spt[v,v]:= \begin{cases} Flagsum[v,v]>0, \ v \neq r; \\ Flagsum[v,v]>1, \ v=r. \end{cases}$
    (* $Spt[v,v]=1$ iff $v$ is a separation vertex *) □
```

The correctness is easily verified. the resource complexities are as follows:

Step	time	hardware	resources
(1) to (3)	$O(t'(n))$	$H(n)$	
4.1	$O(t(n))$	$H(n)$	Lemma 5.1
4.2	$O(1)$	$H(n)$	trivial
5.1	$O(t(n)*\log d'')$	$H(n)$	Lemma 5.3
5.2	$O(t(n))$	$H(n)$	Lemmas 5.0,5.1
5.3	$O(t(n))$	$H(n)$	Lemma 5.0
5.4	$O(t(n))$	$H(n)$	Lemmas 5.0,5.1
5.5	$O(t(n))$	$H(n)$	Lemma 5.0

(* d'' is the diameter of G'', $1 \leq d'' \leq n$. *)

Theorem 5.5: Algorithm Biconnectivity takes $O(t(n)*(\max(\log d, \log d'')+1))$ time and $H(n)$ hardware resources.

6. Implementation on Existing Models

The models we consider are the following:

MCN(VLSI) : Mesh connected Networks[1,3].

PSN(VLSI) : Perfect Shuffle Networks[3,14].

CCC(VLSI) : Cube Connected Cycles[9].

OTN(VLSI) : Orthogonal Tree Networks[8].

OTC(VLSI) : Orthogonal Tree Cycles[8].

SIMD-CCC : SIMD Cube Connected Computers[3].

SIMD-SM-R : SIMD Shared Memory model with read conflicts permitted[11,12]. This includes the P-RAM[18].

SIMD-SR-RW : SIMD Shared Memory Model with read and write conflicts permitted.[5,13].

From the references cited above and Theorems 5.4, 5.5, it is not difficult to verify the results stated below:

The time and hardware resource complexities of the previously known bridge-connectivity and biconnectivity algorithms on various existing models.

model	time	chip area	AT²	# of processors
MCN(VLSI)	$O(n)$	n^2	$O(n^4)$	--- [1]
PSN(VLSI)	$O(\log^2 n \log\log n)$	$n^6/\log n$	$O(n^6 \log^3 n \log^2 \log n)$	--- [7]$^+$
CCC(VLSI)	$O(\log^2 n)$	$n^8/\log^2 n$	$O(n^8 \log^2 n)$	--- [7]$^+$
OTN(VLSI)	$O(\log^2 n \log\log n)$	$n^3 \log^2 n$	$O(n^3 \log^6 n \log^2 \log n)$	--- [8]$^+$
OTC(VLSI)	$O(\log^3 n)$	n^3	$O(n^3 \log^6 n)$	--- [7]$^+$
SIMD-CCC	$O(\log^2 n)$	---	---	$n\lceil n^3/\log n \rceil$ [3]$^+$
SIMD-SM-R	$O(\log^2 n)$	---	---	$\lceil n^3/\log n \rceil$ [12]
or	$O(\log^2 n)$	---	---	$mn+n^2$ [18]$^+$
SIMD-SM-RW	$O(\log n)$	---	---	n^5 [5]$^+$
or	$O(\log n)$	---	---	n^2+2nm [13]$^+$

(* Note: m is the number of edges in the graph; $^+$ indicates that the simple-minded algorithm based on the graph-connectivity algorithm or the matrix multiplication algorithm applies *)

The time and hardware resource complexities of our algorithms on various existing models.

model	time	chip area	AT²	# of processors
MCN(VLSI)	$O(n)$	n^2	$O(n^4)$	--- [19]
PSN(VLSI)	$O(\log n*L)$	$n^6/\log^3 n$	$O(L^2*n^6/\log n)$	--- [3]
CCC(VLSI)	$O(\log n*L)$	$n^6/\log^2 n$	$O(n^6*L^2)$	--- [9]
OTN(VLSI)	$O(\log n*L)$	$n^4 \log^2 n$	$O(n^4 \log^4 n*L^2)$	--- [7]
OTC(VLSI)	$O(\log n*L)$	n^4	$O(n^4 \log^2 n*L^2)$	--- [7]
SIMD-CCC	$O(\log n*L)$	---	---	$\lceil n^3/\log n \rceil$ [3]
SIMD-SM-R	$O(\log n*L)$	---	---	$\lceil n^3/\log n \rceil$ [11]
SIMD-SM-RW	$O(L)$	---	---	n^4 [5]

(* $L = \log d+1$ for bridge-connectivity;
$=\mathbf{max}(\log d, \log d'')+1$ for biconnectivity. $1 \leq d,d'' \leq n$. *)

The efficiency of our algorithms should be evident.

7. Conclusion

In Section 1, we indicated that the time and hardware resource bounds of our algorithms are *bounded above* by $O(t(n)*L)$ and $H(n)$ respectively. This means that our algorithms have the potential of achieving other good complexity bounds if more elaborate techniques are used. As a matter of fact, it has been shown that: (i) for the SIMD-SM-R model, our algorithms could acheive the $O(\log^2 n)$ time bound using only $n\lceil n/\log^2 n \rceil$ processors. This result is optimal for dense graphs[15]; (ii) for the conventional sequential computers, our algorithms could run in optimal time and space[16]. It could be shown that using Reif's recent result on the minimum spanning forest[10], our algorithms could run in $O(\log n)$ time with probabilistic error ε, $0<\varepsilon<1$, using $|E|n^3 \log n$ processors. Similarly, using Awerbuch and Shiloach's recent result on the minimum spanning forest[20], our algorithms could run in $O(\log n)$ time with $\lceil n^3/\log n \rceil$ processors on SIMD-SM-RW.

The strategy we use in this paper, when applied to Savage and Ja'Ja's' algorithms[12] on any of the computer models other than the SIMD-SM-R, does not result in algorithms better than the simple-minded algorithm. However, we do observe that Atallah and Kosarajus' algorithms[1] could adopt our strategy and achieve the same hardware resource bound and a slightly higher time bound as ours on all the computer models discussed in the preceding section. However, their algorithms are inferior to ours for the following reasons: Besides being more complicated, their algorithms rely on the algorithm for finding the transitive closure of a 'directed' graph and thus has the same time and resource bounds as that algorithm. As the resource bounds of the transitive closure problem are difficult to improve, so are the bounds of their algorithms. In fact, we do not think that their algorithms can be modified to acheive the optimal bounds we have achieved on the SIMD-SM-R and sequential models. Perhaps even more importantly, their algorithms do not lead to $O(\log n)$ time probabilistic algorithms for the P-RAM as ours do.

References

1. Atallah,M., Kosaraju, S., "Graph Problems on a Mesh-connected processor array", 14th ACM SOTOC, San Francisco, CA, May 1982, pp.345-353.
2. Chin,F.Y., J.Lam, I-Ngo Chen, "Efficient Parallel Algorithms for Some Graph Problems", **CACM**, Sept.1982, pp.659-665.
3. Dekel,E., D.Nassimi, S.Sahni, "Parallel matrix and graph algorithms", **SIAM** J.Computing, Nov.1981, pp.657-675.
4. Hirschberg,D.S., F.P.Preparata, D.V.Sarwate, "Computing Connected Components on Parallel Computers", **CACM**, Vol.22,Aug.1979,pp.461-464.
5. Kučera,L., "Parallel computation and conflicts in memory access", Info. Processing Letters, April,1980, pp.93-96.
6. Nassimi,D., Sahni,S., "Finding connected components and connected ones on a Mesh-connected parallel computer", **SIAM** J.Computing, Nov 1980,pp.744-757.
7. Nath,D., S.N.Maheshwari, P.C.P.Bhatt, "Efficient VLSI networks for parallel processing based on orthogonal trees", Technical Report No.EE8113. I.I.T. New Delhi,India.
8. Nath,D., S.N.Maheshwari, "Parallel algorithms for the connected components and minimal spanning problems", Info. Processing Letters, Mar.1982, pp.7-11.
9. Preparata,F.P., J.Vuillemin, "The Cube-connected Cycles : A Versatile Network for Parallel Computation", **CACM**, vol.24,May 1981,pp.300-309.
10. Reif,J.H., "Symmetric Complementation", 14th ACM SOTOC, San Francisco, CA, May 1982, pp.201-214.
11. Savage,C.D., *Parallel Algorithms for Graph Theoretic Problems*, Ph.D. Dissertation, U. of Illinois, Urbana,1977.
12. Savage,C.D., J.Ja'Ja', "Fast, Efficient Parallel Algorithms for Some Graph Problems", **SIAM** J.Computing, Nov.1981, pp.682-691.
13. Shiloach,Y., U.Vishkin, "An $O(\log n)$ Parallel Connectivity Algorithm", *Journal of Algorithms*,3.1982, pp.57-67.
14. Stone,H.S., "Parallel processing with the perfect shuffle", *IEEE* Trans. Comput Feb 1971,pp153-161.
15. Tsin,Y.H., F.Chin, "Efficient parallel algorithms for a class of graph theoretic problems", to appear.
16. Tsin,Y.H., "A generalization of Tarjan's depth first search algorithm for the biconnectivity problem", TR82-2, University of Alberta, April 1982.
17. Vishkin,U., "An optimal parallel connectivity algorithm", Research Rerort, IBM Yorktown Heights, 1982.
18. Wyllie,J., *The Complexity of Parallel Computations*, Ph.D.Thesis,Cornell University,1979.
19. Van Scoy,F.C., "The parallel recognition of classes of graphs",*IEEE* Trans.Comput.July 1980,pp.563.
20. Awerbuch,B., Y.Shiloach, "New Connectivity and MSF Algorithms for Ultracomputer and PRAM", submitted.

Anomalies In Parallel Branch-and-Bound Algorithms*

Ten-Hwang Lai and Sartaj Sahni
The Ohio State University University of Minnesota

Abstract

We consider the effects of parallelizing branch-and-bound algorithms by expanding several live nodes simultaneously. It is shown that it is quite possible for a parallel branch-and-bound algorithm using n_2 processors to take more time than one using n_1 processors even though $n_1 < n_2$. Furthermore, it is alsopossible to achieve speedups that are in excess of the ratio n_2/n_1. Experimental results with the 0/1 Knapsack and Traveling Salesperson problems are also presented.

Key Words and Phrases

Parallel Algorithms, branch-and-bound, anomalous behavior.

1. Introduction

Branch-and-bound is a popular algorithm design technique that has been successfully used in the solution of problems that arise in various fields (e.g., combinatorial optimization, artificial intelligence, etc.) [1, 6 - 12]. We shall briefly describe the branch-and-bound method as used in the solution of combinatorial optimization problems. Our terminology is from Horowiz and Sahni [7].

In a combinatorial optimization problem we are required to find a vector x = $(x_1, x_2, ..., x_n)$ that optimizes some criterion function f(x) subject to a set C of constraints. This constraint set may be partioned into two subsets: explicit and implicit constraints. Implicit constraints specify how the x_is must relate to each other. Two examples are:

1) $\sum_{i=1}^{n} a_i x_i \le b$

2) $a_1 x_1^2 - a_2 x_1 x_2 + a_3 x_3^4 = 6$

Explicit constraints specify the range of values each x_i can take. For example:

1) $x_i \in \{0, 1\}$

2) $x_i \ge 0$

The set of vectors that satisfy the explicit constraints defines the *solution space*. In a branch-and-bound approach this solution space is organized as a graph which is usually a tree. This resulting organization is called a *state space graph (tree)*. All the state space graphs used in this paper are *trees*. So we shall henceforth only refer to state space trees. Figure 1 shows a state space tree for the case n = 3 and $x_i \in \{0, 1\}$. The path from the root to some of the nodes (in this case the leaves) defines an element of the solution space. Nodes with this property are called *solution nodes*. Solution nodes that satisfy the implicit constraints are called *feasible solution nodes* or *answer nodes*. Answer nodes have been drawn as double circles in Figure 1. The *cost* of an answer node is the value of the criterion function at that node. In solving a combinatorial optimization problem we wish to find a *least cost answer node*.

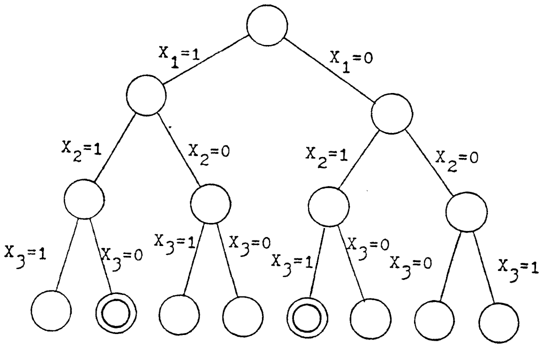


Figure 1 A state space tree

For convenience we assume that we wish to minimize f(x). With every node N in the state space tree, we associate a value $f_{\min}(N) = \min\{f(Q) : Q$ is a feasible solution node in the subtree N$\}$. (If there exists no such Q, then let $f_{\min}(N) = \infty$.)

* This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

0190-3918/83/0000/0183$01.00 © 1983 IEEE

While there are several types of branch-and-bound algorithms, we shall be concerned only with the more popular *least cost branch-and-bound* (lcbb). In this method a heuristic function g( ) with the following properties is used:

(P1) g(N) $\le f_{\min}$ (N) for every node N in the state space tree.

(P2) g(N) = f(N) for solution nodes representing feasible solutions (i.e., answer nodes).

(P3) g(N) = ∞ for solution nodes representing infeasible solutions.

(P4) g(N) $\ge$ g(P) if N is a child of P.

g( ) is called a *bounding function.* lcbb generates the nodes in a state space tree using g( ). A node that has been genereated, can lead to a feasible solution, and whose children haven't yet been generated is called a *live node.* A list of live nodes (generally as a heap) is maintained. In each iteration of the lcbb a live node, N, with least g( ) value is selected. This node is called the current *E-node.* If N is an answer node, it must be a least cost answer node. If N is not an answer node, its children are generated. Children that cannot lead to a least cost answer node (as determined by some heuristic) are discarded. The remaining children are added to the list of live nodes.

The problem of parallelizing lcbb has been studied earlier [2 - 5, 13]. There are essentially three ways to introduce parallelism into lcbb:

(1) Expand more than 1 E-node during each iteration.

(2) Evaluate g( ) and determine feasibility in parallel.

(3) Use parallelism in the selection of the next E-node(s).

Wah and Ma [13] exclusively consider (1) above (though they point out (2) and (3) as possible sources of parallelism). If p processors are available then q = min{p, number of live nodes} live nodes are selected as the next set of E-nodes (these are the q live nodes with smallest g( ) values). Let $g_{\min}$ be the least g value among these q nodes. If any of these E-nodes is an answer node and has g( ) value equal to $g_{\min}$ then a least cost answer node has been found. Otherwise all q E-nodes are expanded and their children added to the list of live nodes. Each such expansion of q E-node counts as one iteration of the parallel lcbb. For any given problem instance and g, let I(p) denote the number of iterations needed when p processors are available. Intuition suggests that the following might be true about I(p):

(I1) I(n_1) $\ge$ I(n_2) whenever $n_1 < n_2$

(I2) $\frac{I(n_1)}{I(n_2)} \le \frac{n_2}{n_1}$

In Section 2, we show that neither of these two relations is in fact valid. Even if the g( )s are restricted beyond (P1) - (P4), these relations do not hold. The experimental results provided in Section 3 do, however, show that (I1) and (I2) can be expected to hold "most" of the time.

Wah and Ma [13] experimented with the vertex cover problem using 2^k, $0 \le k \le 6$ processor. Their results indicate that I(1)/I(p) $\simeq$ p. Our experiments with the 0/1-Knapsack and Traveling Salesperson problems indicate that I(1)/I(p) $\simeq$ p only for "small" values of p (say p $\le$ 16).

2. Somc Thcorcms For Parallcl Branch-and-Bound

As remarked in the introduction, several anomalies occur when one parallelizes branch-and-bound algorithms by using several E-nodes at each iteration. In this section we establish these anomalies under varying constraints for the bounding function g( ). First, it should be recalled that the g( ) functions typically used (eg. for the knapsack problem, traveling salesperson problem, etc. cf. [7]) have the following properties:

(a) g(N) $\ge$ g(M) whenever N is a child of node M. Thus, the g( ) values along any path from the root to a leaf form a nondecreasing sequence.

(b) Several nodes in the state space tree may have the same g( ) value. In fact, many nonsolution nodes may have a g( ) value equal to f*. This is partioularly true of nodes that are near ancestors of solution nodes.

In constructing example state space trees, we shall keep (a) in mind. None of the trees constructed will violate (a) and we shall not explictly make this point in further discussion. The first result we shall establish is that it is quite possible for a parallel branch-and-bound using n_2 processors to perform much worse than one using a fewer number n_1 of processors.

Theorem 1: Let $n_1 < n_2$. For any $k > 0$, there exists a problem instance such that $kI(n_1) < I(n_2)$.

Proof: Consider a problem instance with the state space tree of Figure 2. All nonleaf nodes have the same g( ) value equal to f*, the f value of the least cost answer node (node A). When n_1 processors are available, one processor expands the root and generates its $n_1 + 1$ children. Let us suppose that on iteration 2, the left n_1 nodes on level 2 get expanded. Of the n_1 children generated $n_1 - 1$ get bounded and only one remains live. On iteration 3 the remaining live node on level 2 (B) and the one on level 3 are expanded. The level 3 node leads to the solution node and the algorithm terminates with $I(n_1) = 3$.

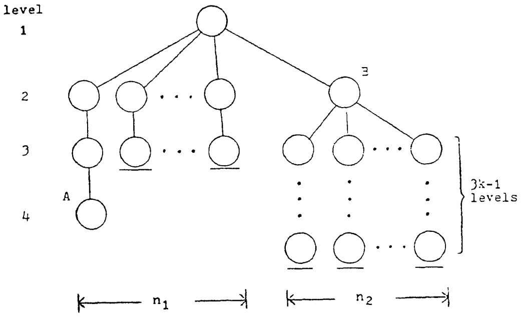


Figure 2: Instance for Theorem 1

When n_2 processors are available, the root is expanded on iteration 1 and all $n_1 + 1$ live nodes from level 2 get expanded on iteration 2. The result is $n_2 + 1$ live nodes on level 3. Of these, only n_2 can be expanded on iteration 3. These n_2 could well be the rightmost n_2 nodes. And iterations 4, 5, ..., 3k could very well be limited to the rightmost subtree of the root. Finally in iteration 3k + 1, the least cost answer node a is generated. Hence, $I(n_2) = 3k + 1$ and $kI(n_1) < I(n_2)$. []

In the above construction, all nodes have the same g( ) value, f*. While this might seem extreme, property (b) above states that it is not unusual for real g-functions to have a value f* at many nodes. The example of Figure 2 does serve to illustrate why the use of additional processors may not always be rewarding. The use of an additional processor can lead to the development of a node N (such as node B of Figure 2) that looks "promising" and eventually diverts all or a significant number of the processors into its subtree. When a fewer number of processors are used, the upper bound U at the time this "promising" node is to get expanded might be such that $U \le g(N)$ and so N is not expanded when a fewer number of processors are available.

The proof of Theorem 1 hinges on the fact that g(N) may equal f* for many nodes (independent of whether these nodes are least cost answer nodes or not). If we require the use of g-functions that can have the value f* only for least cost answer nodes, then Theorem 1 is no longer valid for all combinations of n_1 and n_2, $n_1 < n_2$. In particular, if $n_1 = 1$ then the use of more processors never increases the number of iterations (Theorem 2).

Definition: A node N is *critical* iff $g(N) < f^*$.

Theorem 2: If $g(N) \neq f^*$ whenever N is not a least cost answer node, then $I(1) \ge I(n)$ for $n > 1$.

Proof: When the number of processors is 1, only critical nodes and least cost answer nodes can become E-nodes (as whenever an E-node is to be selected there is at least one node N with $g(N) \le f^*$ in the list of live nodes) Furthermore, every critical node becomes an E-node by the time the branch-and-bound algorithm terminates. Hence, if the number of critical nodes is m, $I(1) = m$.

When n > 1 processors are available, some noncritical nodes may become E-nodes. However, at each iteration, at least one of the E-nodes must be a critical node. So, $I(n) \le m$. Hence, $I(1) \ge I(n)$. []

When $n_1 \neq 1$, a degradation in performance is possible with $n_2 > n_1$ even if we restrict the g( )s as in Theorem 2.

Theorem 3: Assume that $g(N) \neq f^*$ whenever N is not a least cost answer node. Let $1 < n_1 < n_2$ and $k > 0$. There exists a problem instance such that $I(n_1) + k \le I(n_2)$.

Proof: Figures 3(a) and 3(b) show two identical subtrees T. Assume that all nodes have the same g( ) value and are critical. The numbers inside each node give the iteration number in which that node becomes an E-node when n_1 processors are used (Figure 3(a)) and when n_2 processors are used (Figure 3(b)). Other evaluation orders are possible. However, the ones shown in Figures 3(a) and 3(b) will lead to a proof of this theorem.

We can construct a larger state space tree by connecting together k copies of T (Figure 3(c)). The B node of one copy connects to the A node (root) of the next. Each triangle in this figure represents a copy of T. The least cost answer node is the child of the B node of the last copy of T. It is clear that for the state space tree of Figure 3(c), $I(n_1)$ = jk while $I(n_2)$ = (j + 1)k. Hence, $I(n_1) + k = I(n_2)$. []

The assumption that $g(N) \neq f^*$ when N is not a least cost answer node is not too unrealistic as it is often possible to modify typical g( )s so that they satisfy this requirement. The example of Figure 3 has many nodes with the same g( ) value and so we might wonder what would happen if we restricted the g( )s so that only least cost answer nodes can have the same g( ) value. This restriction on g( ) is quite severe and, in practice, it is often not possible to guarantee that the g( ) in use satisfies this restriction. However, despite the severity of the restriction one cannot guarantee that there will be no degradation of performance using n_2 processors when $n_1 < n_2 < 2(n_1 - 1)$. We have unfortunately been unable to extend our result of Theorem 4 to the case when $n_2 \geq 2(n_1 - 1)$. So, it is quite possible that no degradation is possible when the number of processors is (approximately) doubled and g( ) is restricted as above.

Theorem 4: Let $n_1 < n_2 < 2(n_1 - 1)$ and let k > 0. There exists a g( ) and a problem instance that satisfy the following properties:

(a) $g(N_1) \neq g(N_2)$ unless both of N_1 and N_2 are least cost answer nodes.

(b) $I(n_1) + k \leq I(n_2)$.

Proof: Consider the state space tree of Figure 4(a). The number outside each node is its g( ) value while the number inside a node gives the iteration in which that node is the E-node when n_1 processors are used. It takes n_1 processors 4 iterations to get to and evaluate node B. When n_2 processors are available, $n_1 < n_2 < 2(n_1 - 1)$, the iteration numbers are as given in Figure 4(b). This time 5 iterations are needed. Combining k copies of this tree and setting the g( ) values in each copy to be different from those in other copies yields the tree of Figure 4(c). For this tree, we see that $I(n_1) + k = I(n_2)$. []

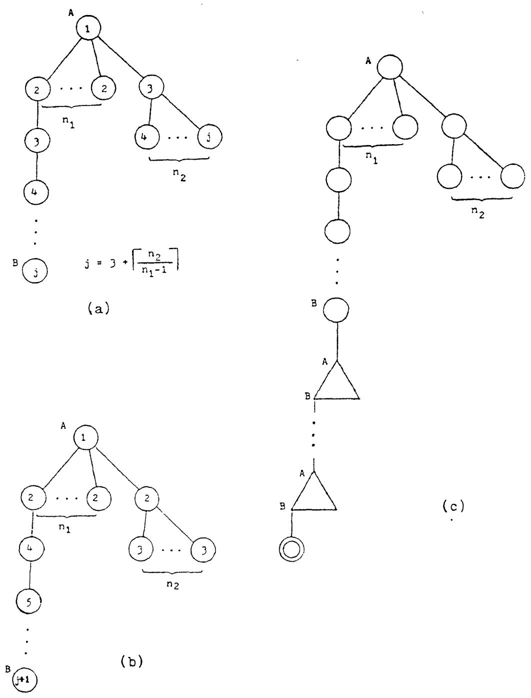


Figure 3: Instance for Theorem 3

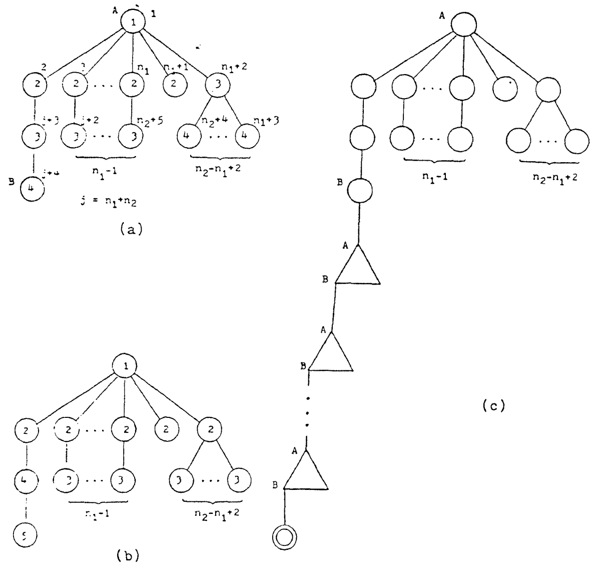


Figure 4: Instance for Theorem 4

The remaining results we shall establish in this section are concerned with the maximum improvement in performance one can get in going from n_1 to n_2 processors, $n_1 < n_2$. Generally, one would expect that the performance can increase by at most n_2 / n_1. This is not true for branch-and-bound. In fact, Theorem 5 shows that using g( )s that satisfy properties (a) and (b), an unbounded improvement in performance is possible. The reason for this is much the same as for the possibility of an unbounded loss in performance. The additional processors might enable us to improve the upper bound quickly thereby curtailing the expansion of some of the nodes that might get expanded without these processors.

Theorem 5: Let $n_1 < n_2$. For any k > n_2/n_1, there exists a problem instance for which $I(n_1)/I(n_2) \geq$ k > n_2/n_1.

Proof: See [14]. ▪

As in the case of Theorem 2, we can show that when g(N) ≠ f* whenever N is not a least cost answer node, $I(1)/I(n) \leq n$.

Theorem 6: Assume that g(N) ≠ f* whenever N is not a least cost answer node. $I(1)/I(n) \leq n$ for n > 1.

Proof: From the proof of Theorem 2, we know that I(1) = m where m is the number of critical nodes. Since all critical nodes must become E-nodes before the branch-and-bound algorithm can terminate, $I(n) \geq m/n$. Hence, $I(1)/I(n) \leq n$. []

When $1 < n_1 < n_2$ and g(N) is restricted as above, $I(n_1)/I(n_2)$ can exceed n_2/n_1 but cannot exceed n_2.

Theorem 7: Assume that g(N) ≠ f* whenever N is not a least cost answer node. Let $1 < n_1 < n_2$. The following are true:

(1) $I(n_1)/I(n_2) \leq n_2$.

(2) There exists a problem instance for which $I(n_1)/I(n_2) > n_2/n_1$.

Proof: (1) From Theorems 2 and 6, we immediately obtain:

$$\frac{I(n_1)}{I(n_2)} = \frac{I(n_1)}{I(1)} * \frac{I(1)}{I(n_2)} \leq n_2.$$

(2) See [14]. ▪

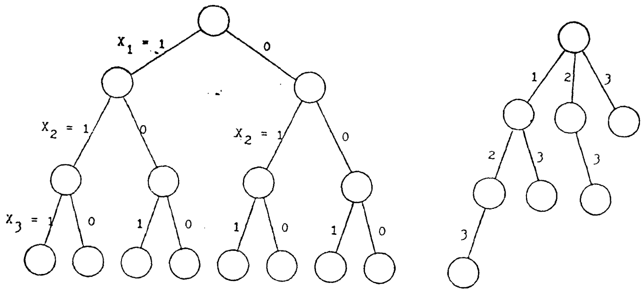
In order to determine the frequency of anomalous behavior described in the previous section, we simulated a parallel branch-and-bound with 2^k processors for k = 0, 1, 2, ..., 9. Two test problems were used: 0/1-Knapsack and Traveling Salesperson. These are described below.

0/1-Knapsack:

In this problem we are given n objects and a knapsack with capacity M. Object i has associated with it a profit p_i and a weight w_i. We wish to place a subset of the n objects into the knapsack such that the knapsack capacity is not exceeded and the sum of the profits of the objects in the knapsack is maximum. Formally, we wish to solve the following problem:

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq M,\ x_i \in \{0, 1\}.$$



(a) binary tree

(b) 3-ary tree

Figure 5

Horowitz and Sahni [7] describe two state space trees that could be used to solve this problem. One results from what they call the fixed tuple size formulation. This is a binary tree such as the one shown in Figure 5(a) for the case n = 3. The other results from the variable tuple size formulation. This is an n-ary tree. When n = 3, the resulting tree is as in Figure 5(b). The bounding function used is the same as the one described in [7]. Since the bounding function requires **that objects be ordered such that $p_i / w_i \geq p_{i+1} / w_{i+1}$, $1 \leq i < n$, we generated our test data by first generating random** w_is. The p_is were then computed from the w_is by using a random nonincreasing sequence f_1, f_2, ..., f_n and the equation $p_i = f_i w_i$. We generated 100 instances with n = 50 and 60 instances with n = 100. **These 160 instances were solved using the binary state** space tree described above. (We also tried the n-ary state space tree but found that it would take several weeks of computer time to complete our simulation. The reason it will take so much time is that when n-ary state space trees are used a great number of nodes will be generated and the queue of live nodes will exceed the capacity of main memory and has to be moved to the secondary storage. In our program, it is time consumming to maintain a queue of live nodes that must be partly stored in secondary storage.)

	n	=	50	n	=	100
p	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{I(2p)}$	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{i(2p)}$
1	363	1.00	1.87	2814	1.00	2.19
2	188	1.87	1.68	1351	2.19	1.85
4	106	3.17	1.42	754	3.69	1.75
8	70	4.66	1.22	402	6.47	1.60
16	56	5.97	1.09	232	10.58	1.35
32	51	6.84	1.03	162	14.94	1.22
64	50	7.23	1.00	126	19.35	1.14
128	50	7.26	1.00	108	23.68	1.05
256	50	7.26	1.00	102	25.84	1.02
512	50	7.26	1.00	100	27.06	1.01
1024	50	7.26		100	27.68	

Table 1: Experimental results (knapsack)

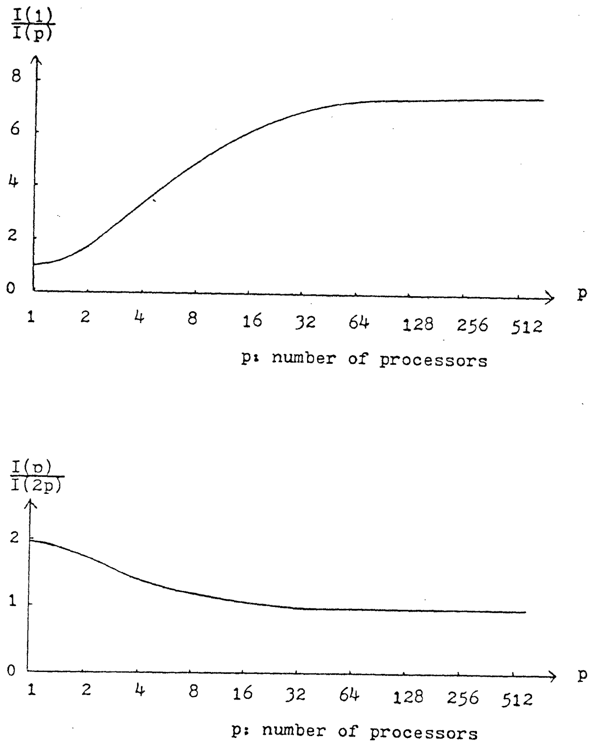


Figure 6 Knapsack with 50 objects

Table 1 gives the average values for I(p), I(1)/I(p) and I(p)/I(2p). From Table 1, we see that when n = 50, I(1)/I(p) is significantly less than p for p > 2. The observed improvement in performance is not as high as one might expect. Similarly, the ratio I(p)/I(2p) drops rapidly to 1 and is acceptable only for p = 1 and 2 (see also Figure 6).

In none of the 100 instances tried for n = 50 did we observe anomalous behavior. I.e., it was never the case that I(p) < I(2p) or that I(p) > 2I(2p).

When n = 100, the ratio I(1)/I(p) is significantly less than p for p > 8 (see also Figure 7). Of the 60 instances run, 6 (or 10%) exhibited anomalous behavior. For all 6 of these there was at least one p for which I(p) > 2I(2p). There was only one case where I(p) < I(2p). The values of I(p), I(1)/I(p), and I(p)/I(2p) for these six instances is given in Table 2. It is striking to note the instance for which I(1)/I(2) = 14.6 and I(2)/I(4) = 0.15.

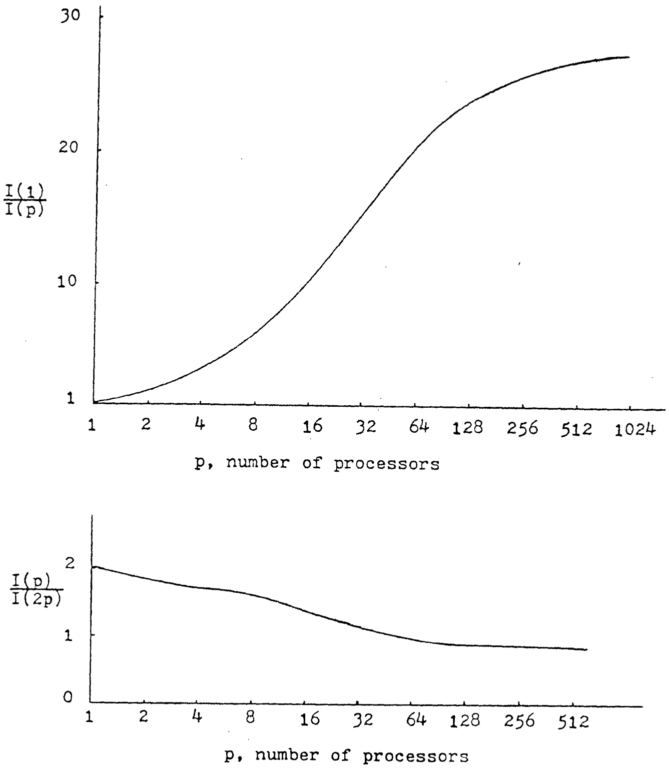


Figure 7 Knapsack with 100 objects

The Traveling Salesperson Problem:

Here we are given an n vertex undirected complete graph. Each edge is assigned a weight. A *tour* is a cycle that includes every vertex (i.e., it is a Hamiltonian cycle). The *cost* of a tour is the sum of the weights of

p	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{I(2p)}$
1	2131	1.00	1.79
2	1191	1.79	**2.23**
4	533	4.00	1.49
8	357	5.97	**2.01**
16	178	11.97	1.09
1	1009	1.00	**2.19**
2	461	**2.19**	1.57
1	21593	1.00	1.99
8	3060	7.06	**2.04**
16	1503	14.37	1.81
1	4119	1.00	**2.03**
2	2034	**2.03**	**2.06**
4	987	**4.17**	1.84
1	5251	1.00	**3.04**
2	1725	**3.04**	1.90
4	909	**5.78**	1.41
8	646	**8.13**	1.74
16	372	14.12	**2.01**
32	185	28.38	1.42
1	7510	1.00	**14.64**
2	513	**14.64**	**0.15**
4	3346	**2.24**	**2.85**
8	1174	6.40	**2.23**
16	527	14.25	**0.95**
32	552	13.61	1.71

Table 2: Data exhibiting anomalous behavior

the edges on the tour. We wish to find a tour of minimum cost.

The branch-and-bound strategy that we used is a simplified version of the one proposed by Held and Karp [6]. Vertex 1 is chosen as the start vertex. There are n - 1 possibilities for the next vertex and n - 2 for the preceding vertex (assume n > 2). This leads to (n - 1)(n - 2) sequences of 3 vertices each. Half of these may be discarded as they are symmetric to other sequences. Any sequence with an edge having infinite weight may also be discarded. Paths are expanded one vertex at a time using the set of vertices adjacent to the end of the path. A lower bound for the path $(i_1, i_2, \ldots, i_k)$ is obtained by computing the cost of the minimum spanning tree for $\{1, 2, \ldots, n\} - \{i_1, i_2, \ldots, i_k\}$ and adding an edge from each of i_1, and i_k to this spanning tree in such a way that these edges connect to the two nearest vertices in the spanning tree.

In our experiment with the traveling salesperson problem we generated 45 instances each having 20 vertices. The weights were assigned randomly. However, each edge had a finite weight with probability 0.35. Use of a much higher probability results in instances that take years of computer time to solve by the branch-and-bound method.

Those 45 instances were solved using $p = 2^k$, $0 \leq k \leq 9$ processors. The average values of I(p), I(1)/I(p), and I(p)/I(2p) are tabulated in Table 3. As can be seen, for p ≤ 32 the average value of I(1)/I(p) is quite close to p and the average value of I(p)/I(2p) is quite close to 2 (see also Figure 8). No anomalies were observed for any of these 45 instances.

p	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{I(2p)}$
1	3974	1.000	1.996
2	1989	1.996	1.990
4	996	3.973	1.976
8	500	7.849	1.943
16	252	15.258	1.873
32	129	28.685	1.753
64	68	51.126	1.609
128	39	85.378	1.417
256	25	129.411	1.252
512	19	177.459	

Table 3: Experimental results (traveling salesperson)

4. Conclusions

We have demonstrated the existence of anomalous behavior in parallel branch-and-bound. Our experimental results indicate that such anomalous behavior will be rarely witnessed in practice. Furthermore, there is little advantage to expanding more than k nodes in parallel. k will in general depend on both the problem and the problem size being solved. If we require I(p)/I(2p) to be at least 1.66, then for the knacksack problem with n = 50, k is between 4 and 8 whereas with n =100 it is between 8 and 16 (based on our experimental results). For the traveling salesperson problem with 20 vertices k is between 8 and 16. If p is larger than k, then more effective use of the processors is made when they are divided into k groups each of size approximately p/k. Each group of processors is used to expand a single E-node in parallel. If s is the speedup obtained by expanding an E-node using q processors, then allocating q processors to each E-node and expanding only p/q E-nodes in parallel is preferable to expanding p E-nodes in parallel provided that sI(1)/I(p/q) > I(1)/I(p).

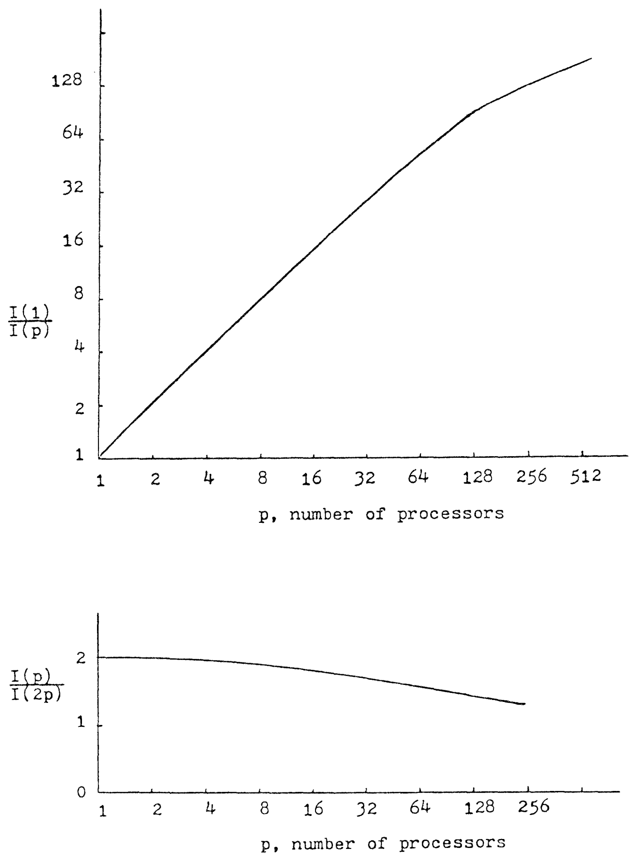


Figure 8: Traveling salesperson

References

1. N. Agin, "Optimum seeking with branch-and bound," *Manage. Sci.*, Vol. 13, pp. B176-B185.

2. B. Desai, "The BPU, a staged parallel processing system to solve the zero-one problem," *Proceedings of ICS '78*, 1978, pp. 802-817.

3. B. Desai, "A parallel microprocessing system," *Proceedings of the 1979 International Conference on Parallel Processing*, 1979.

4. O. El-Dessouki and W. Huen, "Distributed enumeration on network computers," *IEEE Transactions on Computers*, C-29, 1980, pp. 818-825.

5. J. Harris and D. Smith, "Hierarchical multiprocessor organizations," *Proceedings of the 4th Annual Symposium on Computer Architecture*, 1977, pp. 41-48.

6. M. Held and R Karp, "The traveling salesman problem and minimum spanning trees: part II," *Math Prog.*, 1, pp. 6-25, 1971.

7. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Inc., 1978.

8. E. Ignall and L.Schrage, "Application of the branch-and-bound technique to some flow-shop scheduling problems," *Oper. Res., 13, pp. 400-412, 1965.*

9. W. Kohler and K. Steiglitz, "Enumerative and iterative computational approaches," in E. Coffman (ed.) *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, Inc., New York, 1976, pp. 229-287.

10. E. Lawer and D. Wood, "Branch-and bound methods: a survey," *Oper. Res.*, 14, pp. 699-719, 1966.

11. L. Mitten, "Branch-and-bound methods: general formulation and properties," *Oper. Res.*, 18, pp. 24-34, 1970.

12. N. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

13. B. Wah and Y. Ma, "NANIP - a parallel computer system for implementing branch-and-bound algorithm," *Proceedings of The 8th Annual Symposium on Computer Architecture*, 1982, pp. 239-262.

14. T. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms", University of Minnesota, Technical Report, 1982.

Experience with Two Parallel Programs Solving the Traveling Salesman Problem

Joseph Mohan
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract: The traveling salesman problem is solved on Cm*, a multiprocessor system, using two parallel search programs based on the branch and bound algorithm of Little, Murty, Sweeny and Karel. One of these programs is synchronous and has a master-slave process structure, while the other is asynchronous and has an egalitarian structure. The absolute execution times and the speedups of the two programs differ significantly. Their execution times differ because of the difference in their process structure. Their speedups differ because they require different amounts of computation to solve the same problem. This difference in the amount of computation is explained by their different heuristic granularities. The difference between the speedup of the asynchronous second program and linear speedup is attributed to processors idling owing to resource contention.

1. Introduction

The traveling salesman problem (TSP, for short) is to find the round-trip tour that visits each of N cities once, for the minimum cost, given an $N \times N$ matrix of the costs of traveling from one city to another. There are several variations to the problem. I chose to find the exact solution to the asymmetric non-euclidean TSP because it is the most general of all variations.

Many algorithms based on the techniques of dynamic programming and branch and bound solve the TSP. The programs studied here are based on the branch and bound algorithm of Little, Murty, Sweeny and Karel [3]. This algorithm will be called the LMSK algorithm hereafter. The two programs that were implemented lie on significantly different points in the implementation spectrum so that a comparative analysis of their performance will shed some light on the relation between performance and parallel program attributes. An earlier technical report [4] contains more details on the algorithm and the implementations.

The experiments were conducted on the multiprocessor system Cm* [6], running the operating system StarOS [1]. Cm* hardware consists of fifty Digital Equipment LSI-11's. These processor-memory pairs, called *Cm's*, are connected by a hierarchical, distributed switching structure. Cm* is partitioned into five *clusters* of up to 14 Cm's each; the Cm's in an individual cluster are connected via a *map bus* to a mapping processor, called a *Kmap*, through which they communicate with each other. The clusters themselves are connected via *intercluster busses*. Any Cm can reference memory anywhere in the system. However the cluster structure induces an access hierarchy: typically, access to the memory of another Cm in the same cluster costs about three times the access to the Cm's own (local) memory and access to remote clusters costs about twelve times [2]. StarOS is an object-oriented, message-based operating system to support collections of processes that cooperate to solve problems.

2. LMSK Algorithm

The LMSK algorithm works by partitioning the set of all possible tours into progressively smaller subsets, which are represented on the nodes of a state-space tree, and then expanding the state-space tree incrementally toward the goal node using heuristics to guide the search. The algorithm uses two heuristics to guide its search toward the solution node. The *node-selection heuristic* chooses, from among all the leaf nodes of the current tree, that leaf node whose estimated lower-bound tour cost is the least. The *edge-selection heuristic* computes the increments in tour cost when different edges are excluded from the tour, and chooses the edge that causes the maximum increment. The description of the algorithm below explains how the nodes and edges thus chosen are used.

Starting with a tree consisting of just the root node, which represents the set of all tours, the algorithm *repeatedly* executes these steps *in sequence*: it first chooses, from all the leaf nodes of the current tree, the node that is most likely to lead to the solution (using the node-selection heuristic) and designates it to be the next expansion node; it then chooses one or more edges (legs of a tour) using the edge-selection heuristic; it sets up the child nodes with all the different combinations of inclusion and exclusion of the selected edges; and finally it computes for each child node the lower-bound cost of all tours in the subset defined by the child node. The algorithm terminates when a leaf node is found that represents a single complete tour with a cost less than or equal to the lower-bound costs of all possible tours, which are represented by the leaf nodes of the current state-space tree.

Here is a high level representation of the algorithm:

```
repeat
    1: Select node in state-space tree using node-selection heuristic;
    2: Select one or more edges using edge-selection heuristic;
    3: For each child corresponding to one inclusion/exclusion
       combination of the selected edges,
        3.1: Create a node and link it to tree;
        3.2: Derive cost matrix for child node;
        3.3: Reduce matrix and find new lower bound for all tours
             defined by child node;
until a full tour, with a cost less than or equal to the lower bounds on all
possible tours, is obtained.
```

3. Implementations

One technique for adapting an algorithm for parallel execution is unfolding a loop and letting multiple processes work on different iterations of the unfolded loop. This technique is adopted here in two different ways to get two parallel programs, which unfold different loops of the algorithm. One program, TSP1, unfolds the *for* loop that sets up child nodes (step 3 of algorithm in previous section), while the other, TSP2, unfolds the outermost (*repeat*) loop.

The first program, TSP1, is a *synchronous master-slave* program. The master process implements the outer loop of the algorithm, steps 1 and 2, and the loop control of step 3. The slave processes (one for each child) implement the steps within the inner loop, that is, steps 3.1, 3.2, and 3.3. To obtain a parallelism of N during an execution, $\log_2 N$ edges are selected by the master during each iteration, causing N child nodes to be set up by N slaves, one child node by each slave.

The other program, TSP2, is implemented as a collection of *asynchronous* cooperating processes with no master-slave relationship among them; such a process structure is referred to here as *egalitarian*. Each process, during each of its iterations, selects one edge and sets up two child nodes: one node including the edge in the tour, and the other excluding it. Each process executes all the steps in the algorithm and does so repeatedly until it is determined by consensus that a tour has been found.

4. Speedup

In this section the speedups of the two programs will be presented. Speedup is the ratio of serial execution time to parallel execution time. It reflects the effective parallelism achieved at a given nominal parallelism.

4.1. TSP1

Figure 1 plots speedup against parallelism for TSP1. As explained before, the degree of parallelism for this program can assume only values that are powers of 2. However data points in all the figures in this section are shown interpolated to depict trends of smoothened values. Speedup at a given parallelism is computed as

2 * solution time for parallelism of two / solution time for given parallelism.

(This slightly non-standard definition of speedup is adopted here for practical reasons.) The solution time is taken to be the elapsed time for solving the problem, excluding the time spent creating the slave processes. The execution times vary depending on the distribution of processes among the clusters because of the hierarchical memory access structure of Cm*. To factor out this phenomenon, only the best times, among all the different placements for which the experiments were performed, were used when calculating speedups.

The dashed line shows linear speedup (speedup equals parallelism) and the solid

----

This research was supported in part by the National Science Foundation under grant ECS-8120270.

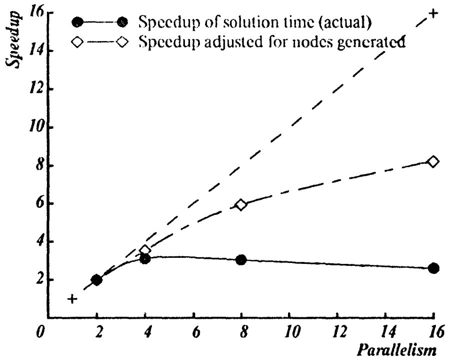


Figure 1: Speedup versus parallelism for TSP1

line plots the actual speedup achieved for this program. The actual speedup is reasonable between parallelisms of 2 and 4 (speedup at a parallelism of 4 is 2.8). However it starts going down after a parallelism of about 6, and remains constant at a speedup of 2.6 after a parallelism of about 8. This droop is explained by the greater total amount of work done at a greater parallelism for this program and by the saturation of system bottlenecks when greater amount of simultaneous computation occurs in the system. The dot-dashed line in the graph factors out the effect of the greater amount of work done with greater parallelism. This curve would have been the speedup curve of this program, if the total computation for solving the problem did not increase with increasing parallelism. The primary work of the algorithm consists of setting up and reducing the cost matrix corresponding to the nodes of the state-space tree. Thus the number of nodes generated is a measure of the total computation pefformed for soiving the TSP. Assuming that total computation for solution was directly proportional to the number of nodes generated,

speedup adjusted for nodes generated corresponding to parallelism N
= 2 * solution time per unit computation with 2 processes /
solution time per unit computation with N processes

= 2 * (solution time with 2 processes / number of nodes generated
with 2 processes) / (solution time with N processes /
number of nodes generated with N processes)

The curve corresponding to the adjusted speedup behaves much better; it is a lot nearer to the linear speedup line, does not show any signs of peaking or saturation and continues to increase reasonably till a parallelism of 16, which was the maximum parallelism of the experiment. The difference between the linear speedup line and this curve corresponds to loss of processor time owing to the synchronous control of the program and to contention for system resources such as the Kmaps, various system busses, SLocals, and the Object Manager.

4.2. TSP2

Figure 2 plots speedup against parallelism for TSP2. Since for this program single process execution is possible, actual speedup at a given parallelism is computed as

solution time for parallelism of one / solution time for given parallelism.

The speedup curve looks much better than that for TSP1. It is close to the linear speedup line until a parallelism of 6 and does not show any signs of peaking or saturating at higher parallelisms. It is almost identical to the speedup adjusted for nodes generated in TSP1. This adjusted speedup was what the system was capable of achieving, if the total computation for solving the problem had remained constant with increasing parallelism. This close correspondence of the two curves suggests that the amount of computation does not increase with parallelism here. This inference is reinforced by the number of nodes generated by TSP2 remaining nearly constant for all parallelisms.

The absolute execution times for TSP2 are lower than for TSP1 by about 25% at a parallelism of 2 and by about 80% at a parallelism of 16. For a parallelism of 2, the nodes generated and the nodes expanded for both programs are the same, suggesting that the total amount of computation done by all processes for solving the problem is about the same for both the programs. Consequently one has to look for some explanation other than increased computation for the 25% decrease in absolute execution time at this parallelism. The following are two possible explanations for this behavior: First, in TSP1 (with its master-slave structure), when the master is busy deciding on which node to expand next, the slaves idle (after creating a look-ahead node for the next iteration) wasting processor time. Because of the egalitarian process structure of TSP2, processes here do not idle waiting for more work. Secondly, in TSP1 (with its synchronous program control), some time is wasted because of a lack of absolute work balance between the slave processes. On the other hand, in TSP2, with its asynchronous control, absolute work balance between processes is not critical, and therefore no time is lost because of imbalance.

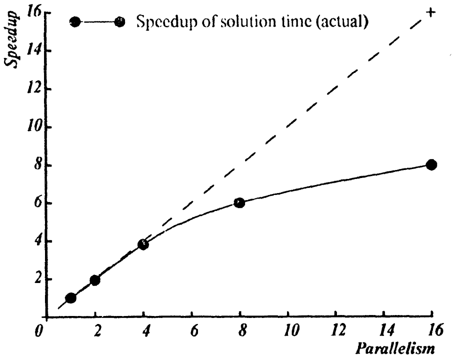


Figure 2: Speedup versus parallelism for TSP2

Figure 3 factors out the effects of cluster level contention on speedup. Cluster level contention here refers to contention for resources that are replicated for each cluster in the run-time system; these resources include the Kmaps, the Map busses, and the Object Manager. Cluster level contention is distinguished from system level contention, that is, contention for system-unique resources of the run-time environment and of the application program itself. Examples of system-unique resources are the intercluster bus and user locks. Speedup for this figure is computed using solution times with the same number of processes in each cluster to ensure that cluster level contention is nearly the same for all data points on a given curve. The curves hug the linear speedup line closely except near a parallelism of 16; this suggests that below a parallelism of 12 there is hardly any system level contention and most of the loss in execution time seen in the previous speedup figure is attributable to cluster level contention. Near a parallelism of 16, system level contention affects performance adversely to a more significant extent.

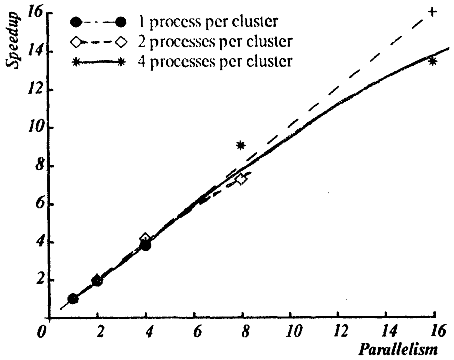


Figure 3: Speedup (normalized for cluster level contention) versus parallelism for TSP2

5. Work and Heuristic Granularity

Work of a program, here, refers to the total amount of basic computation done by all the processes of the program to solve the given problem cooperatively; work excludes processor idling, resource contention, and overhead costs, such as the costs of locking and communication. Though these excluded costs are expected to change with the implementation strategy that is used to map a general algorithm into a parallel program, work itself, as defined above, is usually assumed to remain constant. In practice, however, the mapping decisions can affect the work of a program. Earlier experiments by other researchers [5] showed that work can change with the degree of parallelism of an execution instance of a program and with other program attributes, such as control synchronism. The two implementations of the LMSK algorithm illustrate this phenomenon further.

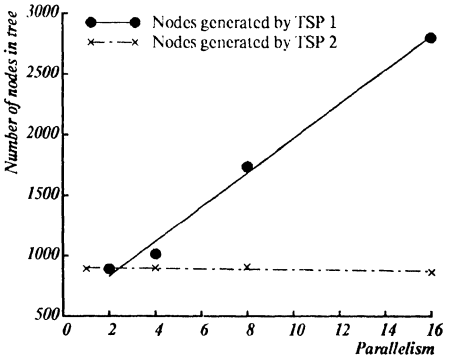


Figure 4: Work variation in TSP programs

As observed before, the primary reason for the better speedup performance of TSP2 is directly attributable to its work remaining constant with parallelism (compare Figures 1 and 2). For the LMSK algorithm, the total number of nodes in the final state-space tree is an indicator of work done by the programs. Figure 4 shows how the work for the two programs varies with parallelism. For TSP1, work increases with parallelism, while for TSP2 it remains nearly constant.

Where does this difference in work between the two programs arise from? The amount of work done by a heuristic search algorithm, such as the LMSK algorithm, is determined by the effectiveness of the heuristics in bounding the search. This difference in work can be explained by the difference in their *heuristic granularities*. For a heuristic search program, the average amount of computation by all its processes between points of heuristic application is the *heuristic granularity* of that program [5]. For the TSP programs, one kind of heuristic granularity can be associated with the node-selection heuristic, and another with the edge-selection heuristic. Figure 5 depicts how the two kinds of heuristic granularities of the TSP programs vary with parallelism. Since the major amount of computation in an iteration goes into setting up new nodes, granularity here is expressed in terms of the amount of computation needed to set up one new node. The heuristic granularity corresponding to the node-selection heuristic and that corresponding to the edge-selection heuristic will be examined separately below.

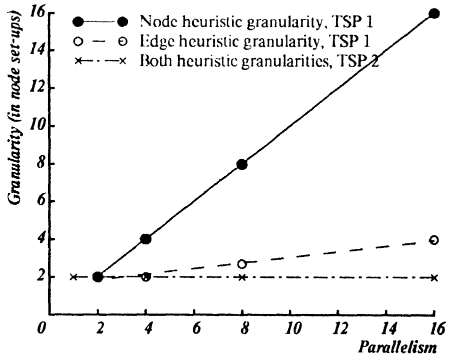


Figure 5: Granularity variation in TSP programs

First, let us consider the node-selection heuristic. Both programs apply this heuristic once every iteration. In TSP1, the number of nodes set up by the master process in one iteration equals the degree of parallelism and hence the heuristic granularity equals the degree of parallelism. The heuristic granularity then increases linearly with parallelism. In TSP2, however, for each iteration two nodes are set up by each process, irrespective of parallelism. Therefore heuristic granularity for this program remains constant at two nodes.

The edge-selection heuristic is applied once for each edge selected in both programs. In TSP1, for a parallelism of N, during each iteration $\log_2 N$ edges are selected by applying the heuristic once for each edge selected and N nodes are set up. The heuristic granularity corresponding to this heuristic, then, is $N/\log_2 N$ when the parallelism is N. In TSP2, irrespective of parallelism, each process applies the edge-selection heuristic once to select an edge and sets up two nodes during each iteration. The heuristic granularity for this program thus remains constant at two with changing parallelism.

A comparison of Figures 4 and 5 will reveal the close parallel between the variation in the heuristic granularities of the two programs and the work done by them. The heuristics in these programs serve to limit the extent of the search needed to solve the problem. An increase in heuristic granularity decreases the effectiveness of the heuristics in limiting the search and consequently leads to an increase in the total amount of computation needed to solve the problem. The relation between work of a heuristic search program and its heuristic granularity (or alternatively the frequency of application of heuristics) should not come as a surprise. However, the possibility of heuristic granularity changing with the degree of parallelism is peculiar to parallel heuristic programs.

6. Conclusion

In general when a programmer sets out to implement a parallel program based on some algorithm, he can design many parallel programs with different attributes. These attributes will depend on the design decisions he makes when mapping the algorithm to a program and, in particular, on the parallelization strategy that he adopts. As this work demonstrates, some attributes of the resulting program will influence its execution time and speedup characteristics to a significant extent. A programmer has to analyze the effects of not only the obvious program attributes such as the control synchronism and the pattern of resource usage, but also the more subtle ones such as the granularity of program events and the amount of computation the program needs to solve the problem at different degrees of parallelisms.

Acknowledgements

I would like to thank many for their help during this work. Anita Jones and Zary Segall were the research guides and spent many hours supervising the work. Gerald Thompson and Robert Whiteside helped me hone many of the ideas by their discussions with me. Cynthia Hibbard gave many suggestions to improve the prose of this paper.

References

[1] A. K. Jones et. al.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM/SIGOPS, December, 1979.

[2] Anita K. Jones and Edward F. Gehringer [eds.].
The Cm multiprocessor project: A research review.*
Technical Report CMU-CS-80-131, Computer Science Department, Carnegie-Mellon University, July, 1980.

[3] Little J. D. C., Murty K. G., Sweeney D. W., and Karel C.
An Algorithm for the Traveling Salesman Problem.
Operations Research 11, 1963.

[4] Mohan J.
A Study in Parallel Computation—the Traveling Salesman Problem.
Technical Report CMU-CS-82-136, Computer Science Department, Carnegie-Mellon University, August, 1982.

[5] Mohan J., Jones A., Gehringer E., and Segall Z.
Granularity of Parallel Computation.
In *Computer Science Research Review,* Carnegie-Mellon University, 1982.

[6] Richard J. Swan.
The switching structure and addressing architecture of an extensible multiprocessor, Cm.*
PhD thesis, Carnegie-Mellon University, August, 1978.

DOT, A DISTRIBUTED OPERATING SYSTEM MODEL OF A TREE-STRUCTURED MULTIPROCESSOR

Scott Danforth
Department of Computer Science
University of North Carolina, Chapel Hill 27514

Abstract -- This paper describes DOT, a model of an architecture implementation specifically designed for direct and maximally parallel execution of FFP (formal functional programming) language programs. The model is represented using tasks and abstract data types in C and is running on UNIX®

DOT is a refinement and extension of the architecture proposed by Mago [1,2]. User programs consisting of FFP language symbols are placed in a linear array of cells (the leaves of a binary tree of processors), and segments of this array that contain innermost FFP applications execute system programs in order to perform the required reductions. The system programs are written in LPL, a low-level concurrent programming language used to implement FFP primitives on the architecture.

INTRODUCTION

This paper deals with a language-driven architecture. Over the last decade there has been increased interest in this area, and a number of machines oriented toward efficient support for high level languages have been designed. This approach has been used for support of sequential languages such as Basic [3], Lisp [4], and Pascal [5].

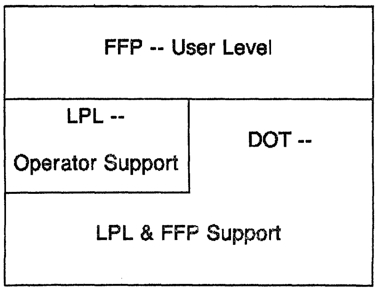
With the increasing potential of VLSI implementation technologies, highly parallel architectures that hold great promise for increased performance have become feasible. Although a number of parallel computer architectures have been proposed, few of these have been directly associated in the above sense with a general purpose programming language. This is due to the low-level sequential transformation of states and reliance on global memory embodied in most languages, which make it difficult to express a direct mapping between a language and its execution on a parallel architecture. Notable exceptions to this are data flow languages [6], and functional languages [7].

Data flow languages arose from the search for a general model of parallel computation, and it is therefore not surprising that a variety of parallel architectures for their support have been proposed and some implemented [8]. Functional languages, on the other hand, arose from the search for an algebra of programs, as described by Backus [9,10]. In this case, sequential transformation of states and global memory were banished because of their unsatisfactory properties with respect to program semantics, and as a serendipitous result, functional languages are promising candidates for parallel support.

Among possible architectures for this purpose are those suggested by Keller et al. [11], Treleaven [12], and Mago [1,2]. This paper is based on the work of Mago, which differs from other proposals in its use of *fine grain parallelism*. This approach removes assumptions of global memory and overall processor state from the language support level as well, and completely realizes the parallelism allowed by functional programs.

We now present a programming system composed of three logical levels. As shown in Figure 1, the top (user) level is that of FFP languages, and the middle (system support) level is that of LPL, the concurrent programming language used to define and implement arbitrary FFP operators on the architecture. DOT is both a design and an implementation model of the desired parallel architecture, and is the lowest level.

Figure 1 -- Three System Levels



FFP LANGUAGES

FFP languages have been formally defined by Backus [13]. Informally, an FFP language program is a linear sequence of symbols, of which four types of symbol are specially distinguished for the purpose of providing structure: opening and closing application-forming symbols for *applications*, and similarly balanced list-forming symbols. An application is composed of an operator and exactly one operand. Both operator and operand may be lists and may contain further (i.e., nested) applications. A non-trivial FFP program is an application, and execution proceeds by successively reducing innermost applications according to the semantics of their respective operators until there are no further applications. The ultimate result is a constant (i.e., non-reducible) expression. This is called ***reduction style*** execution, since the program source is rewritten in a succession of semantically equivalent forms until the final result is achieved.

FFP reductions are completely local in nature and are tightly encapsulated with respect to the rest of the program. This fact allows immediate, completely parallel and non-interfering execution of all innermost applications (hereafter referred to as *reducible applications*, or *RAs)*, and it is this property of FFP languages that makes them so attractive for multiprocessor support.

Figure 2 shows an FFP program which calculates the inner product of two vectors. The application symbol in our representation is a parenthesis "(", and the list-forming symbol is an angle bracket "<". Within DOT, all program symbols have an associated FFP text nesting level, which removes the need for the balancing symbols ")" and ">". Examples of this representation are found in Figure 4 (at end).

This work was supported by the National Science Foundation under grant MCS80-04206, and by a grant from the Harris Corporation.

0190-3918/83/0000/0194$01.00 © 1983 IEEE

FIGURE 2: Inner Product of < 1 2 3 > with < 4 5 6 >

```
    --  The original FFP program is:
( + ( < α * > ( τ < < 1 2 3 > < 4 5 6 > > ) ) )
    --  τ (matrix transpose) is innermost
( + ( < α * > < < 1 4 > < 2 5 > < 3 6 > > ) )
    --  <α * > (apply-to-all multiply) is innermost
( + < ( * < 1 4 > ) ( * < 2 5 > ) ( * < 3 6 > ) > )
    --  three multiplications are innermost
( + < 4 10 18 > )
    --  + (n-ary add) is innermost
32
    --  which is the answer
```

Mapping FFP onto a Linear Array of Cells

The advent of VLSI has encouraged the design of content addressable storage and other types of "intelligent" memory [14], and it is only natural to attempt to envision an intelligent memory in which a program might be loaded and then executed in place. The locality properties of FFP languages indicate they would be good candidates for such treatment, and this would avoid the CPU bottleneck associated with von Neumann processors. The usual idea of memory is a linear address space, so as a first step we can imagine placing the symbols of an FFP program into a linear array of cells (hereafter called *lcells)*, each of which comprises processing power as well as memory.

Reduction of an FFP RA within a group of contiguous lcells will produce a new FFP program segment in place of the original RA. Each symbol of this new segment is a function of the original contents of the lcells comprising the RA, and the semantics of the operator of the reduction. We therefore need a way of specifying the actions to be taken within the lcells of an RA that will result in the correct (operator-defined) transformation, i.e., we need an *lcell programming language*, or LPL. Once an RA has been detected, the lcells of its containing segment will each execute the LPL program appropriate to the operator of the reduction.

The particular FFP primitives chosen and their LPL implementations will be the concern of a system manager, possibly in communication with knowledgeable users. The compiled LPL code modules, which may be thought of as system software support routines, are held in a library available to the IO subsystem, and are broadcast to the lcells when they are required.

A Tree-Structured Architecture

The existence of such FFP operators as ***apply-to-all*** (shown in Figure 2) implies global communication within an RA, and the topology chosen to support this communication is that of a binary tree with dynamically reconfigurable routing. Non-leaf nodes of this tree are hereafter called ***tcells***, and the leaves will be the *lcells*, as already discussed.

Figure 3 -- DOT MODEL STRUCTURE

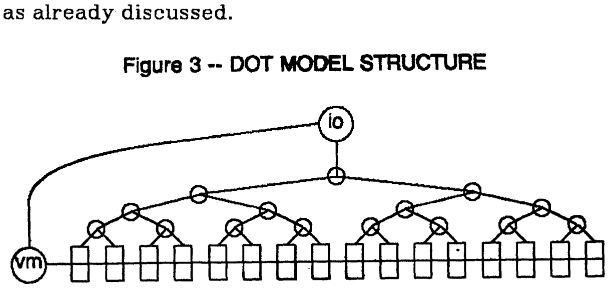


Figure 3 shows the logical structure of DOT. In addition to the usual parent-child links associated with binary tree structures, we make use of lateral connections at the lcell level in order to facilitate shifting the FFP text about to accommodate its expansion and contraction. While these connections are not strictly necessary, they simplify the model and seem feasible within a hardware implementation.

LPL -- LCELL PROGRAMMING LANGUAGE

The aim of an LPL program is to define an FFP primitive. This is done by specifying appropriate actions for each lcell of an application. LPL is therefore designed to manipulate local lcell registers, and possibly invoke simple global operations (e.g., message passing) with which LPL statements in other lcells of the same RA may interact. In practice, various groups of lcells within the RA are given the same instructions (e.g., all elements of a sequence), so an LPL program consists of code segments -- one for each such group.

The most interesting aspects of LPL include the message interactions between the lcells of an RA (send/receive/endfilter), and the way in which LPL contexts may spawn copies of themselves (fork) in order to create additional FFP text symbols within the lcell array. With these capabilities, LPL programs can implement powerful FFP operators such as matrix transpose, and parallelism within the tree structure can be used very effectively. For example, sorting is Θ(n), and max is Θ(log n).

LPL Environment

There are no local (stack-based) variables in LPL. Instead, a fixed number of LPL *environment variables* within local lcell registers may be referred to. Many environment variables are set up by DOT before LPL statements are allowed to execute. Among these are the local FFP symbol (called *symbol)*, its nesting level within the FFP program (called *aln* for absolute level number), its nesting level within the RA (called *rln* for relative level number -- i.e., relative to the aln of the application symbol) and the "directory," which is used to specify the location of the FFP symbol within the RA.

The directory is composed of two parts: the first part is a *symbol_index*, specifying the position of *symbol* within the RA; the second is a 4-tuple that encodes the symbol location based on up to four levels of hierarchical nesting within the RA. Figure 6 shows the directory during execution of an **example** FFP program. In addition to its use by LPL statements, the directory 4-tuple is also used by DOT to choose which segment of an LPL program should be executed within an individual lcell. This will be explained in conjunction with the LPL destination statement.

When the lcells in an RA have all completed execution of their LPL program segments, the reduction is "stepped forward" to its result. This is done by DOT with the aid of the environment variables *nsymbol_cnt*, *nsymbol*, and *naln*. The "n" prefix stands for "next," and these variables are set up in each lcell of an RA by the LPL program. If nsymbol_cnt is zero when the RA is stepped forward, the containing lcell becomes empty (i.e., there is no FFP symbol in the lcell). Otherwise nsymbol is moved to symbol, and naln is moved to aln. Thus, the LPL programmer is primarily concerned with creating code which (for each lcell of the RA) will load nsymbol and naln **with the symbol** and aln values which should next appear within the lcells of the RA in order to implement the required reduction.

Having described the objective and environment of LPL programs, we now briefly consider their structure and the more interesting statements.

LPL Statements

program/endprogram. A program statement is the first statement of an LPL program. Its form is **program x** where "x" is the (numeric) identifier of the FFP operator whose operation this LPL program is intended to implement. The assembler creates a library object file for subsequent use whose name is based on this identifier. The end of an LPL program is signalled with an **endprogram** statement.

destination/endsegment. The same sequence of LPL statements is not executed in each lcell of an RA. Instead, an LPL program defines code segments and specifies their respective lcell destinations through the use of the destination statement. The first segment of an LPL program whose destination matches an lcell's directory 4-tuple is the segment that the lcell will execute, and all following segments are ignored. The form of the destination statement is **destination d1 d2 d3 d4** where "d1" through "d4" are either an integer, or an integer followed by "*". A match, as referred to above, occurs if each of the lcell 4-tuple directory entries is either equal-to (no "*" used) or equal-to-or-greater-than ("*" used) the the respective destination value. The end of a program segment is signalled with an **endsegment** statement.

fork. This statement is the means by which additional lcells are allocated to hold expanding FFP text. The name "fork" is given this statement because each lcell may be thought of as a single process that executes a sequential LPL program segment. A fork spawns copies of its program segment and its execution context to create new processes in the requested number of consecutive lcells. Execution continues after allocation and loading of these lcells by DOT (during storage management). The form of the statement is **fork forksize** where "forksize" is the (non negative) number of lcells desired. The ***fork_id*** environment variable is set by DOT during support for this operation. The "parent" of the fork operation is always given fork_id = 1, while the children are given fork_id = 2 through forksize in left-to-right ordering. This fact can be used in subsequent LPL statements to condition execution. Copying or moving groups of FFP symbols into new locations within the FFP text is done using fork followed by send/receive.

send/receive/endfilter. These statements are the means by which global communication within an RA is carried out. They are supported by DOT processes within both lcells and tcells. Messages are sent and received during globally sequenced activities called ***message waves***, and all the lcells of an RA have the option of participating in any of them. A limited amount of processing can take place within the tcells during transmission of a message wave, and appropriate instructions to the tcells concerning processing requirements are sent up by the lcells to introduce each new message wave. (This information is supplied in the send statement.) The LPL messages within a message wave travel from the lcells up through the tree structure above them until they reach the lowest common ancestor of all lcells within the RA (called the ***top of area*** or ***toa***). At the top of area, all messages that have come this far (messages may be combined, or passed selectively on the way up) turn around and are broadcast to all lcells in the RA. Those lcells doing either a send or a receive for that particular message wave then "see" all returning messages on that wave. Send and receive statements have a filter portion that describes the actions to be taken for each incoming message, and a local DOT message process invokes this filter for each arrival after first moving the message into a reception area within the lcell. The difference between send and receive is that a send sends a message (then filters incoming messages, including its own), while a receive merely filters incoming messages. Their forms are as follows:

```
send mwave order combine-op key1 key2 arg_cnt
      filter-statements
      endfilter
```

```
receive mwave
      filter-statements
      endfilter
```

The mwave argument is the index of the message wave desired. The order argument indicates the order in which two messages of differing key values should be sent up from a tcell where they meet. (Key1 is given precedence over key2.) When two messages arriving at a tcell both have the same key values, this indicates that the respective messages should be combined according to the combine-op argument. "Arg-cnt" is the number of additional message arguments (in addition to the key values) which are to be sent. The additional arguments are taken from lcell registers with names *m_arg1* ... *m_arg5*. When messages are combined arithmetically, it is the m_arg1 values which are actually combined. *r_key1*, *r_key2*, *r_arg1* ... *r_arg5* are the lcell registers into which a message is placed by DOT prior to executing a filter.

LPL Program Example

This section presents the LPL program for n-ary add which supports the FFP program whose execution trace is given later.

```
program 004 /* FFP N-ARY ADD OPERATOR */

destination 0 0 0 0  /* The application symbol
    keep
    receive 1    /* receives the result.
        mov r_arg1 nsymbol
        endfilter
    endsegment
destination 1* 0 0 0   /* Operator, seq symbols
    endsegment          /* go away.
destination 0* 0* 0* 0*     /* Numbers to be added
    mov symbol m_arg1   /* send themselves
    send 1 + + #0 #0 1  /* using addition
        endfilter
    endsegment
endprogram
```

DOT -- SUPPORT FOR LPL and FFP

We have described the functions and controlled environment made available to LPL by DOT. It is now necessary to examine how these are provided while also supporting the innermost reduction semantics of FFP languages.

The Basic Machine Cycle

A DOT machine cycle starts with looking at the lcell array to see what is in it. In the course of this operation, RAs are discovered, and the machine is *partitioned* so as to correctly allocate communication channels and tcell processing power to the discovered RAs. The first time a particular RA is encountered (RAs may exist over a period of many machine cycles), DOT processes within the tcells and lcells build the lcell environment directory, and the LPL program is loaded using the IO subsystem. After these operations, all of which take place in the *partitioning phase* of the machine cycle, the LPL programs in RAs are started (or restarted). This happens seperately within each RA, so RAs that do not require a new directory and LPL code can be restarted earlier.

At this point, the notion of a single machine is misleading; each RA has its own dedicated hardware and is completely independent of the others. Nevertheless, after the RAs are started (or restarted), the machine may be thought of as being in an *execution phase*. The LPL programs run, with the aid of DOT-provided services, until they become blocked or are preempted by DOT for the purpose of storage management.

The *storage management phase* includes stepping forward any RAs that have finished, determining the new storage requirements of the FFP programs within the lcell array (due to LPL fork statements that have been executed), and shifting LPL program segments and their contexts within the lcell array to make room for newly required symbols. The shifting process is performed using the lateral lcell connections (shown in Figure 3) and may result in the overflow of contexts into the overflow subsystem if enough lcells are not available, reentry of previously overflowed lcell contexts back into the physical lcell array if there is room, or entry of new FFP programs if there is room after previous overflow has been taken care of. The prescription for exactly how the lcell contents are to be shifted about is called the *specification for storage management*.

The basic machine cycle is thus partitioning, execution, and storage management. These phases will now be described in more detail.

Partitioning Phase

Partitioning creates *active areas*, each of which is a binary tree dynamically imbedded within the overall tree-structured architecture. Each active area is composed of the dedicated communication channels, and the lcell and tcell hardware required for supporting computation in an individual RA. Partitioning begins in the lcells and is continued in the tcells. Each tcell receives (from its childen) and sends (to its parent) a code containing the information necessary for an *initial partitioning* of the tcells.

The initial partitioning allocates and connects dedicated area communication channels (called *area channels)* and dedicated tcell processing power (called *area nodes)* to each underlying group of lcells that may contain a different RA. While the area channel connections are changed with each partitioning, the information required for the initial partitioning travels upwards on *cell manager channels* whose connections never change.

The initial partitioning is terminated within the IO subsystem, which may be thought of as the parent of the root of the tree. (In addition to its IO-related activities, the IO subsystem offloads special termination processing from the tree root.) RAs are finally located and their corresponding active areas created with the aid of concurrent downsweeps within each of the candidate areas created by the initial partitioning. Figure 4 (at end) shows the area channels and nodes for a partitioned DOT processor.

Area Nodes. As suggested by Figure 4, a tcell need only provide processing power for one active area. This is because even though area channels for more than one active area may pass through a given tcell, it is always possible to directly route through the tcell (via what may be considered a straight wire connection) all but one set of area channels, which, if it exists, is composed of the area channels leading to two children and possibly a parent, all in the same area. During partitioning, such a set of area channels is connected to a tcell *area node* whose primary purpose is to support all subsequent area-related processing within the tcell.

This support begins with completion of the partitioning phase (i.e., discovery of active status, discovery of the FFP operator if the area is active, pruning of channels that are for one reason or another not required for area processing, creation of the *toa* where messages will turn around, and directory creation). Partitioning is then followed in an active area by support for the LPL send and receive operations. This is then followed by correctly shutting down operation prior to the storage management phase of the machine cycle. This shutdown must disconnect area channels (which were created during partitioning), but only after stopping messages in such a way as to guarantee that all lcells in the area will have seen exactly the same messages during the execution phase. This must be done in order to guarantee a consistent restart following storage management and re-partitioning.

Directory Creation. Following the initial partitioning upsweep, each toa returns to its descendent lcells notification of their active status and the LPL program to be used if one is necessary. Given this information, an lcell will decide to create a directory if it is contained in a new RA. This requires an upsweep and a downsweep within the area channels, and the result is to load *symbol_index*, and the 4-tuple directory, *d1 ... d4*, with the correct values. If the RA is not new, the old directory is still valid and execution may begin immediately without this step.

Loading LPL Programs. The LPL programs are delivered from the IO subsystem on *io channels* that follow the hardware tree structure. Within a tcell, each parent io channel splits into two child io channels and data movement is as follows: input to the lcell array comes from "above" and is broadcast to all lcells by successively splitting data so what comes in from a parent input channel is sent down both child input channels; output comes from "below", and is sequenced by handling the child output channels in cyclic left-to-right order. There are two very simple processes in the tcell that perform these functions. At present, the input channels are used to deliver LPL programs from the library, and output channels are used to return execution results and trace information to the outside world.

Execution Phase

The lcell LPL interpreter is a process that receives starting addresses from a queue. It begins execution at the requested address, performs local data movement and manipulation as indicated by the loaded LPL object code, and continues until encountering one of the following DOT service requests which require special handling: **send, receive, endfilter, fork,** and **endsegment.**

These special services are initiated by setting up an LPL context area associated with the particular service required. These areas are checked by the DOT processes whose job it is to provide the services. Having set up the service request area, the interpreter then cycles back for another start address. The reason for this approach will be seen in the following discussion of message support.

Lcell Message Support. Whenever a message arrives which should be filtered (due to a send or receive for the present message wave), the DOT lcell message input process first puts the newly received message into a receive area (accessible to filter statements using named variables such as r_arg1), and then uses information deposited earlier (by the interpreter) in the LPL program context to insert the beginning address of the message filter statements into the interpreter start address queue. The interpreter executes the message filter for the message instance, and then encounters the endfilter statement, which then halts the interpreter as described above. This is done for each message that arrives on the present message wave. When the wave has completed, the lcell message input process places the continue address (i.e., the address of the first statement following the endfilter statement) into the interpreter start address queue, and LPL execution then continues.

Message waves are sequenced activities whose completion requires agreement among all of the lcells of an RA. The basis of this agreement is an end-of-wave message or *eow* that is sent for each message wave by all lcells of an RA, merged into a single message by the time it reaches the toa, and then returned to the lcells in the RA. Lcells keep a counter which contains the present message wave number.

Whenever the message wave counter is incremented in an executing lcell (due to receiving eow), the LPL program context is checked for a send or receive request for the new wave number. If there is such a request, an eow is sent (after message transmission if the request was send). If the request is for a send or receive on a higher numbered wave, eow is also sent. If, however, the last message request is for a lower numbered message wave, a fork has been executed. In this case, the eow is not sent. Instead, we wait for storage management to complete the fork operation. The result is that the new message wave cannot pass through the toa until after storage management (and completion of the fork operation).

Following storage management, everything is restarted correctly so a message wave interrupted by storage management may complete, and the next one begin (all transparent to the LPL program). This allows implicit synchronization of a fork operation with a corresponding send designed to copy information.

Fork Support. A fork statement halts execution within the requesting lcell until the operation can complete during storage management (when LPL program contexts are shifted in the lcell array). Execution then resumes in the child lcells as well as the parent. LPL program contexts begin each execution phase acting as if they had requested a forksize of 1. The fork statement merely modifies the lcell variable (not directly available by name to an LPL program) in which this value is stored so that multiple copies of an LPL program context are shifted rather than just one during the next storage management.

Storage Management Phase

This phase is necessary to accommodate growth and compaction of the FFP text while retaining the necessary ordering of FFP symbols. It is unfortunate that execution of LPL programs should in general require interruption in order to implement this phase of the machine cycle. One alternative is to let all LPL programs complete (or become blocked as in a fork operation) before storage management is performed, but this could put RAs with quickly executing LPL programs at a disadvantage, and would likely result in inferior utilization of the available processing power in a large machine.

Attempts have been made to do storage management in locally restricted segments of the lcell array (as computation proceeds elsewhere) by Tolle [15], but the complexity of the overall solution is considerable, and the resulting performance is not always superior to the preemptive approach that we use.

In the present design, lcells send permission to start storage management upwards on the cell manager channels to the IO subsystem. Lcells which are not active do this following partitioning. Active lcells wait for the LPL program to complete, or fork before sending permission. These *sm_grant messages* are merged on their way up the tree, and upon reaching the IO subsystem, result in generation of a *stop message* which then travels down the tree and shuts down message activity. This approach is attractive, since it places control of the processing cycle explicitly within LPL, and allows a system manager to tailor FFP operators for large operands if this is desired. Another possibility would be to allow the IO subsystem to use heuristics based on lcell contents (discovered during partitioning) to determine an appropriate cycle time.

The Specification for Storage Management. Once the LPL programs are shut down, a specification for storage management must be computed. This is done by sending and merging forksize information up the cell manager channels until it arrives at the IO subsystem, where, as in partitioning, the upsweep is terminated. There, a specification for storage management is computed, and sent back down the tree in such a way as to distribute the necessary information to each lcell. A variation on the scheme suggested by Mago [1] is used, so that total compaction will be performed only when necessary.

Overflow and Program Entry. The virtual memory concept used in the model is based on the work of Siddall [16] and Frank [17], who have examined various ways to accommodate overflow from the lcell array. The approach used is to allow storage management into and out of the left lcell tree boundary. To the left of this boundary is a deque structure (interfaced with a file system), that receives from its right any lcell contents that overflow from the tree, and from its left new programs for execution in the tree. The state of the overflow and program entry subsystem (e.g. if there is presently overflow in virtual memory, if so how much, how large the next FFP program to be entered is, etc.) is used by the IO subsystem in its determination of the actual storage management specification to be used.

REPRESENTING DOT

In order to specify a detailed implementation for the architecture, we required a powerful representation language capable of expressing the parallel activities of the envisioned architecture with precision. We have access to a UNIX®system, so we decided to use the C language augmented with abstract data types (classes and tasks [18]) in order to produce a comprehensive and executable model.

The DOT lcell and tcell classes are shown in Figure 5 (at end). The complete DOT model representation includes 25 classes. When the model is executed, a tree height parameter is given, and the required number of these classes are instantiated and connected to form a machine of the desired size. Operation then begins with partitioning of the (empty) machine. DOT models for machines containing hundreds of lcells may be created and observed within computationally reasonable time periods.

To illustrate operation of the model, Figure 6 (at end) contains trace output from the example FFP program:

(+ (< apply-to-all * > < < 1 3 > < 2 4 > >))

Column headings are for the user program id, lcell symbol, lcell state (0=ground, 1=executing, 2=completed), fork_id, aln, rln, symbol_index, and the directory 4-tuple. Columns to the right of the arrow are used to indicate the result of stepping a completed reduction forward. Empty lcells do not appear, and cells with symbols in them are listed in left-to-right order. Output is at the end of each cycle.

CONCLUSIONS

This paper has described a complete and operational model of a multiprocessor architecture designed for maximally parallel execution of FFP programs. The word "complete" is relative to (among other things) the level of detail chosen, and we have deliberately stopped before the hardware level. Nevertheless, all necessary algorithms (sequential and concurrent) have been concretely represented within DOT, as have all external interfaces, and many other design details. LPL has been defined, and an assembler written. The resulting model is as close to a true machine (at levels above the hardware) as is possible. Major improvements (with respect to both simplicity and performance) on the earlier design by Mago have been realized. These further increase the desirability of the architecture.

Representing the model in a concurrent programming language has allowed testing and verification of DOT. The importance of verification is obvious when the model is to be imbedded in hardware. Other benefits of a complete software model include the possibility of performance simulation, and estimation of hardware complexity.

An analytic model of program execution time on such a tree machine has been proposed by Mago et al. [19]. Using this as a guide, a similar model based on the DOT model has been developed. This work, and other details not included here will be found in a forthcoming dissertation [20].

If performance simulation supports the belief that an architecture based on the model we have described is a good idea, the next hurdle is mapping the DOT design into hardware. DOT has been created with VLSI technology in mind. Binary tree structures map well onto VLSI, the processing cells are self-timed, and active tasks in a model instance should correspond to hardware areas specifically designed for their support. The lcell interpreter is probably the most traditional and straightforward of these, while the tcell IO relay mechanisms and downward message handler will likely be trivial due to their simplicity. All of the data and message movement in the machine has been designed with a bit serial pipelined approach in mind.

ACKNOWLEDGEMENTS

The work described in this paper relied heavily on previous work done in connection with this architecture here at UNC Chapel Hill. Much of this work has been explicitly referred to within the paper, however I would also like to express my indebtedness to George Links Entenman and David Middleton for their active, creative, and invaluable participation in our meetings with Gyula Mago. In addition, previous work on partitioning by Peter Chen was helpful.

REFERENCES

[1] G. Mago, "A Network of Microprocessors to Execute Reduction Languages," International Journal of Computer and Information Science, (October, December, 1979), pp. 349-385 and 435-471.

[2] G. Mago, "A Cellular Computer Architecture for Functional Programming," IEEE Computer Society COMPCON, (Spring, 1980), pp. 179-187.

[3] H. Burkle, A. Frick, and Ch. Schlier, "High Level Language Oriented Hardware and the Post-Von Neumann Era," Fifth Annual Symposium on Computer Architecture, ACM-SIGARCH Newsletter (April, 1978), pp. 60-65.

[4] G. Steele, and G. Sussman, "Design of a LISP-based microprocessor," CACM, (November, 1981), pp. 628-645.

[5] W. Carlson, "The Pascal Microengine", Workshop on High-Level Language Computer Architecture, Los Angeles, Ca., 1981.

[6] W. Ackerman, "Data Flow Languages," Computer, (February, 1982), pp. 15-25.

[7] J. Backus, Programming Language Semantics and Closed Applicative Languages, IBM Research Report RJ1245, Yorktown Heights, New York, (July, 1973).

[8] J. Dennis, "The Varieties of Data Flow Computers," First Intl Conference on Distributed Computing Systems, (October, 1979) pp. 430-439.

[9] J. Backus, "Is Computer Science Based on the Wrong Fundamental Concept of Program? An Extended Concept," in Algorithmic Languages, Bakker/Vliet (eds.), IFIP, North-Holland Publishing Co., (1981), pp. 133-165.

[10] J. Backus, "Function Level Computing," IEEE Spectrum, (Vol. 19, #8, 1982), .pp 22-27.

[11] R. Keller, G. Lindstrom, and S. Patil, "A Loosely-coupled Applicative Multi-processing System," AFIPS Conference Proceedings, (Vol. 48, 1979), pp. 613-622.

[12] P. Treleaven and G. Mole, "A Multi-processor Reduction Machine for User-defined Reduction Languages," Seventh Annual Symposium on Computer Architecture, (May, 1980), pp. 121-130.

[13] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," CACM, (August, 1978), pp. 613-641.

[14] H. Fuchs, J. Poulton, A. Paeth, and A. Bell, "Developing Pixels-Planes, A Smart Memory-Based Raster Graphics System," MIT Conference on Advanced Research in VLSI, (January, 1982), pp. 137-146.

[15] D. Tolle, "Coordination of Computation in a Binary Tree of Processors: An Architectural Proposal," Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, (1981).

16] W. Siddall, "Virtual Memory Algorithms for Tree-Structured Processors," Ph.D. dissertation in preparation, Department of Computer Science, University of North Carolina at Chapel Hill.

[17] G. Frank, "Virtual Memory Systems for Closed Applicative Language Interpreters," Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, (1979).

[18] B. Stroustrup, "Adding Classes to the C Language: An Exercise in Language Evolution," to appear in Software: Practice and Experience.

[19] G. Mago, D. Stanat, and A. Koster, "Program Execution in a Cellular Computer: Some Matrix Algorithms," in preparation.

[20] S. Danforth, Ph.D. dissertation in preparation, Department of Computer Science, University of North Carolina at Chapel Hill.

Figure 4 --

Partitioning for (+ < (* < 1 3 >) (* < 2 4 >) >)

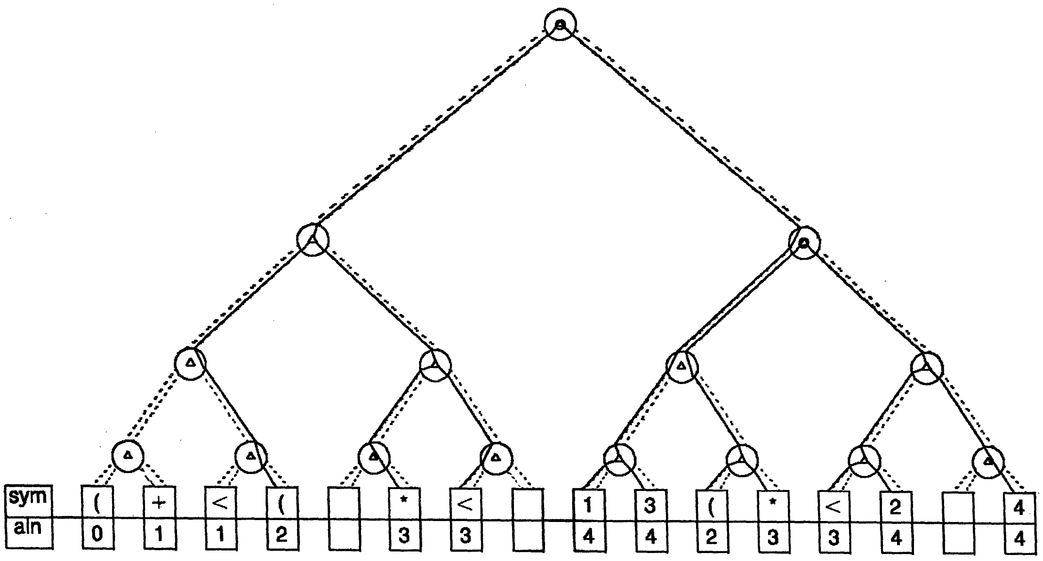


Figure 5a -- The DOT TCELL Class

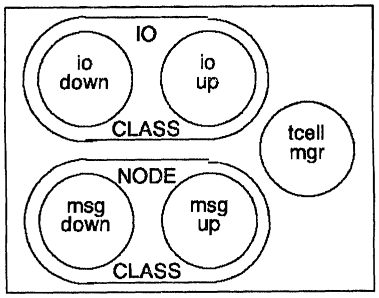


Figure 5b -- The DOT LCELL Class

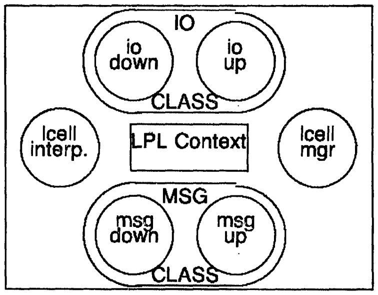


Figure 6 -- TRACE OUTPUT

```
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN   ###### End of Cycle 1 ######
001  (   0 001 000 000 000 0000                This application not innermost
001 #004 0 001 001 000 000 0000                 4 is the op-code for n-ary add
001  (   1 001 001 000 000 0000                This application is innermost
001  <   1 001 002 001 001 1000                so state = executing
001 #008 1 001 003 002 002 1100                 8 is the op_code for apply-to-all
001 #012 1 001 003 002 003 1200                12 is the op_code for multiply
001  <   1 001 002 001 004 2000
001  <   1 001 003 002 005 2100
001 #001 1 001 004 003 006 2110
001 #003 1 001 004 003 007 2120
001  <   1 001 003 002 008 2200                This symbol forks and receives
001 #002 1 001 004 003 009 2210                a copy of the operator (mult)
001 #004 1 001 004 003 010 2220                as required by apply-to-all
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN   ###### End of Cycle 2 ######
001  (   0 001 000 000 000 0000
001 #004 0 001 001 000 000 0000
001  (   2 001 001 000 000 0000      <    001  The reduction has completed
001  <   2 001 002 001 001 1000      (    002  and so is stepped forward.
001 #008 2 001 003 002 002 1100
001 #012 2 001 003 002 003 1200    #012   003  The result is a sequence of
001  <   2 001 002 001 004 2000                parallel multiplications.
001  <   2 001 003 002 005 2100      <    003
001 #001 2 001 004 003 006 2110    #001   004
001 #003 2 001 004 003 007 2120    #003   004
001  <   2 001 003 002 008 2200      (    002  Note that the fork_id tells
001  <   2 002 003 002 008 2200    #012   003  how "<" should step forward
001  <   2 003 003 002 008 2200      <    003  for these three symbols.
001 #002 2 001 004 003 009 2210    #002   004
001 #004 2 001 004 003 010 2220    #004   004
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN  ###### End of Cycle 3 ######
001  (   0 001 000 000 000 0000               Figure 4 gave a partitioning
001 #004 0 001 001 000 000 0000               for exactly this FFP text, with
001  <   0 001 001 000 000 0000               its two parallel RAs and two
001  (   2 001 002 000 000 0000    #003   002 supporting active areas.
001 #012 2 001 003 001 001 1000               Both multiplications complete
001  <   2 001 003 001 002 2000               in one cycle, and are stepped
001 #001 2 001 004 002 003 2100               forward.
001 #003 2 001 004 002 004 2200
001  (   2 001 002 000 000 0000    #008   002
001 #012 2 001 003 001 001 1000
001  <   2 001 003 001 002 2000
001 #002 2 001 004 002 003 2100
001 #004 2 001 004 002 004 2200
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN  ###### End of Cycle 4 ######
001  (   2 001 000 000 000 0000    #011   000 Add is now innermost, and it
001 #004 2 001 001 001 001 1000               also completes in one cycle.
001  <   2 001 001 001 002 2000               11 is the answer.
001 #003 2 001 002 002 003 2100
001 #008 2 001 002 002 004 2200
```

THE TREE MACHINE
AN EVALUATION OF STRATEGIES FOR REDUCING PROGRAM LOADING TIME

Pey-yun Peggy Li and Lennart Johnsson
Computer Science
California Institute of Technology
Pasadena, CA. 91125

Abstract -- The Caltech Tree Machine has an ensemble architecture with processors interconnected as a binary tree. The program code has to be loaded to the Tree Machine through the root processor. By exploiting the regularity of the user defined tree structure, the downloading time can be reduced from $O(N)$ to $O(\log_2 N)$, where N is the total number of nodes in the tree.

This paper presents three algorithms for loading the node types of the tree. All algorithms have a best case loading time of $O(\log_2 N)$. Two of them, which take a binary tree descriptions as input, have a worst case performance of $O(\sqrt{N/f})$ and $O(N^{1/\log_2 f})$ for regular trees of fanout f. The third algorithm has a worst case performance of $O(\log_2 N)$.

Introduction

The Caltech Tree Machine has an ensemble architecture, [1],[2] with processors interconnected as a binary tree. Each node is a complete, although small, von Neumann machine that can execute its own program. Synchronization is made by message passing between adjacent nodes. Hence, it is significantly different from other tree machine projects such as [3], [4]. The Tree Machine is an attached machine. The host compiles and loads a program into the Tree Machine, loads data into it and interacts with it during execution. The root is the only point of communication with the outside world [5].

To program the tree machine, a user has to be aware of the tree structure. It is necessary to devise a tree data structure and algorithm for the problem to be solved. The tree may have arbitrary fanout and arbitrary size. Mapping onto the binary tree configuration of the Tree Machine is performed by the software system. A fanout greater than two is achieved by introducing so called padding nodes as descendants to a Tree Machine node until the specified fanout is realized by the descendants of the last level of padding nodes. Padding nodes are inserted in the left and right subtrees in alternating order, at every level, to minimize the unbalancing. Padding nodes as well as their code is generated by the software system.

In many tree machine algorithms devised so far [1],[6], the user defined logical tree is often designed to have only two or three node types, one for the root, one for the leaf processors, and one for the intermediate processors of the tree. Only different pieces of code, i.e., one copy of the code for each node type need to be supplied to the root. Replication of program code is easily accomplished by having the processor copy the code it receives, if it is of its type, and always pass it on to its two descendants.

Assigning node types in the Tree Machine

All considered algorithms load types into tree nodes by recursively and concurrently expanding a description received by the root. A naive approach is to simply load the types of all nodes in a left to right, root to leaf order. The length of the input of node types is proportional to the size of the tree. This fully explicit description rapidly becomes too costly.

The size of the tree descriptions we propose is at best proportional to the number of different node types, which in many cases is very small, and independent of the tree size. However, it should be recognized that the binary tree that results from mapping a regular, arbitrary fanout tree onto a binary tree, in many cases is much less regular than the original tree, Figure 5.

Two different forms of the tree description supplied to the root of the Tree Machine are analyzed:

-- a binary tree, i.e., the host performs the mapping.

-- a logical tree, i.e., the mapping onto a binary tree is performed by the Tree Machine itself.

The tree description supplied to the root is constructed by the host from the specification given in a Tree Machine program. In creating this description the host is traversing or scanning the tree in a predefined order. Nodes of identical type and fanout appearing consecutively in scan order only need to be represented once, together with a number specifying the number of occurrences. Subtrees can be defined to achieve an extremely compact input for regular trees having few node types.

Two scan orders are studied, normal and bit-reversed. Both orders progress from root to leaves, level by level. Normal scan order implies that nodes within a level are in left to right order, Figure 1.a. In bit-reversed scan order, nodes within a level are visited in bit-reversed order, Figure 1.b. This ordering implies that for any node its left and right subtrees are visited alternately.

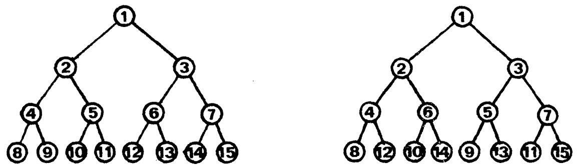


Figure 1. Normal and bit-reversed scan order

The scan order used in generating the description affects both its length and the complexity of the loading algorithm. The bit-reversed ordering results in a simpler loading algorithm because a node only needs to set a flag to indicate that the next input should go to either the left or the right port. With normal scan order, a Tree Machine node needs to monitor when the left half of the level currently being treated has been exhausted, and type assignment should progress to the right subtree, as well as when that is exhausted and the next level of the left subtree shall be assigned types. The decoding of the tree description is more complex in this case.

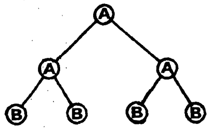
I. Binary tree as input

For this case the host is assumed to perform the mapping onto a binary tree, if required. The padding processors are inserted in bit reversed order to keep the tree balanced.

Identical consecutive nodes in the given scan order are

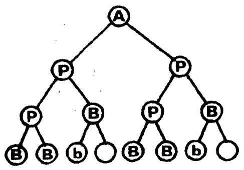
0190-3918/83/0000/0202$01.00 © 1983 IEEE

represented by a pair, (NUM, ID). NUM is the number of consecutive occurrences of nodes of type ID. The same node type may appear several times in the description. Two special delimiters, '(' and ')', are introduced to allow subtrees to be grouped together in the input. Identical subtrees encountered consecutively in scan order only need to be specified once.



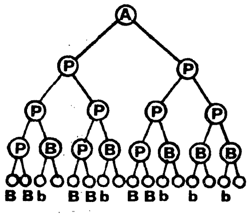
		Length
NO:	3 A, 4 B	4
BRO:	3 A, 4 B	4

Figure 2. A 3-level binary tree with two different node types



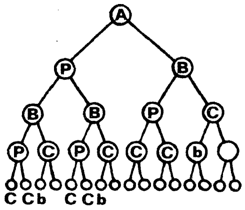
		Length
NO:	1A,2(2P,3B)	9
BRO:	1A,4P,4B,2b,2B	10

Figure 3. A two-level, 6-ary tree with padding nodes



		Length
NO:	1A,2P,3(2P,3B),1(1P,2B)	18
BRO:	1A,9P,8B,5b,3B	10

Figure 4. A two-level, 11-ary tree with padding nodes



		Length
NO:	1A,1P,3(1B,1P,3C)	13
BRO:	1A,1P,2(1B,1P,2C,1b,1C), 1b,1(1B,1P,2C,1b,1C)	32

Figure 5. A three-level, 3-ary tree with padding nodes

The two different scan orders, in the following refered to as NO and BRO, are analyzed on three simple cases:

Case 1. A binary tree with identical nodes at each level: Different levels may have different node types. The two scan orders render the same description. Its length is at best twice the number of different types, at worst twice the tree height since all nodes on a level of the tree is assumed to be of the same type, Figure 2.

Case 2. The binary tree description converted from a two level f-ary logical tree: To simplify the analysis all leaf nodes are assumed to be of the same type.

For normal scan order and $f \leq 10$, a tree description can always such that the leaf node type only appears once in it. Under this condition, all the leaf nodes either appear consecutively in normal order in the binary tree, or there exists one subtree in which the leaf nodes appear consecutively in scan order and repeated use of the subtree covers all the leaf nodes, Figure 3.

For $f > 10$, the number of occurrences of leaf nodes in the tree description increases with f, Figure 4. Case 1 gives a lower bound. A fairly tight upper bound is derived below.

In seeking an upper bound for the number of occurrences of leaf nodes in the binary tree description of a two-level f-ary tree with all leaf nodes being identical, it is first observed that the worst case fanout must be odd. If f is even then the f-ary tree can always be described as two identical subtrees, which are represented only once using the subtree notation. If f is odd then there is one subtree instantiating an odd fanout, and one subtree representing an even fanout. They differ in fanout by one, and appear with the subtree representing the larger fanout to the left. The subtree notation can not be used to reduce the description at this level, but there are common subtrees at lower levels.

The worst case occurs when the f-ary tree is described by one subtree of odd fanout at distance two from the root, and one subtree also of odd fanout at distance three from the root, and both these subtrees have a maximum number of occurrences of leaf nodes. Let $x_k = \max\{\text{leaf occurrences} |\ 2^k \leq f < 2^{k+1}\}$. Then, it follows that $x_k \leq x_{k-2} + x_{k-3}$, with initial conditions $x_0 = x_1 = x_2 = 1$. There exist fanouts for which equality holds. Some of the worst case fanouts d_k can be derived by the recurrence equation $d_k = 2 * d_{k-1} + (-1)^{k-1}$, $d_2 = 5$. An upper bound for the third order recurrence equation $x_k = x_{k-2} + x_{k-3}$ is given by $\sqrt{f/2}$. The length of NO is proportional to x_k, i.e., NO is $O(\sqrt{f})$ in the worst case. However, the number of occurrences of the leaf nodes grows much slower on the average.

In BRO the number of occurrences of a leaf node equals 1 if f is a power of two, otherwise 2. These bounds are a consequence of padding nodes being inserted in bit-reversed order. Padding nodes are always encountered first. However, even though leaf nodes appear last, they form one group only when f is a power of two. The reason is that leaf nodes are attached **pairwise** to the padding nodes of the previous level. With bit-riversed scan order all left descendants of the previous level are treated before any right descendant is assigned a type. Empty nodes are always encountered, except when the last level of the binary tree is fully populated. Therefore, BRO is of fixed length, 6 words if leaf nodes appear once in the tree description, otherwise 10 words, Figures 3 and 4.

Case 3. The binary tree description converted from a k-level f-ary tree: All nodes at one level of the f-ary tree are assumed identical, Figure 5.

As the number of levels in the tree grows, leaf nodes are replaced by subtrees. This replacement is easily accommodated in the description using the subtree notation. If the number of occurrences of the leaf node type in a two-level subtree is x, and the length of the two-level subtree description is s, then adding one level to the tree increases the description by x*s, since each occurrence of a leaf node is replaced by s. Therefore, there are totally x^2 occurrences of the leaf node type in the resulting 3-level tree. Repeating this substitution process, the sequence length for a k-level, f-ary tree is $s(1+x+x^2+x^3+....+x^{k-2}) = s(x^{k-1}-1)/(x-1) = r$.

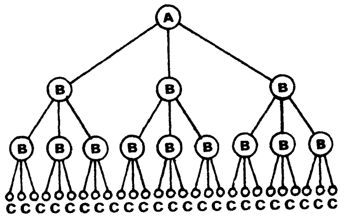
For x=1, i.e., the leaf node type occurs only once in the description of a two-level f-ary tree, the sequence length is $s*(k-1)$. It is proportional to the height of the tree. It is the minimum length possible. For x=2, the worst case for BRO, the sequence length is $s*(2^{k-1}-1)$. For x>2, s is proportional to x, which for NO is bounded by $\sqrt{f}$. The sequence length r is bounded by $\sqrt{N/f}$, where $N=f^k$.

In conclusion, the length of both scanning orders, as well as the processing time at the root processor is at best $O(\log_f N)$. The worst case for f-ary trees with all nodes at one level being identical is $O(\sqrt{N/f})$ for normal ordering, and $O(N^{1/\log_2 f})$ for bit-reversed ordering. In addition it should be noticed that there is always a minimum propagation time of $O(\log_2 N)$ from the root to the leaves

II. Arbitrary fanout tree as input

In this second case, the input consists of a description of an arbitrary fanout tree. The Tree Machine nodes insert padding nodes to create the necessary fanout. Identical nodes are represented only once if they appear consecutively in the scan order. However, for this algorithm we limit the grouping of nodes to one level of the arbitrary fanout tree. The tree description will contain at least one entry per logical tree level.

An entry in the type file is of the form (NUM, ID, FANOUT), where NUM is the number of nodes of the same type and with fanout FANOUT that appears in succession in the scan order in one level of the input tree. Normal scan order is assumed. Subtrees can be defined. They are contained within '(' and ')' and can be nested. Figure 6 shows an example of a tree description for Algorithm II.



Input string:

1 A 3, 3 B 3, 9 B 3, 27 C 0

Figure 6. A 4-level, 3-ary logical tree

The basic characteristics of this algorithm are:

- A node in the Tree Machine defines, as required, one or both of its descendants to become padding nodes. Generated padding nodes may have a fanout greater than two. Every node accepts a description of an arbitrary fanout tree as input. The padding node is no different from other nodes. It generate its own padding nodes. Every node uses the first entry it receives to determine its own type.

- A Tree Machine node has to monitor the structure of the logical tree in order to dispatch the input to proper descendants. The nodes in one level of the input tree may have different fanouts. The total number of nodes at one level of the input tree equals the sum of the fanouts of all the nodes at the preceding level. A Tree Machine node needs to keep and update information of one level of the input tree, including the total number of nodes in the input tree at the level which the current input entry is filling, the accumulated total fanouts of those nodes, the number of nodes at the current level of the input tree that shall be placed in the left and right subtrees of a Tree Machine node, and how many of those that still remains to be placed.

  In the process of dispatching one input entry, two multiplications are required to calculate the total fanouts of the nodes going to the left and to the right subtrees of a processor. The multiplication time is proportional to $\log_2 f$.

The length of the input for this algorithm is at best $O(\log_f N)$ for a regular logical tree of fanout f. At worst it is $O(N)$. The total loading time is at best $O(\log_2 N)$.

Summary and Conclusions

Table 1 contains the loading times in instruction cycles for; Algorithm I.a, binary tree input using normal scan order; Algorithm I.b, bit-reversed order; and Algorithm II, arbitrary fanout tree input. The first figure for the execution time is the time to finish processing the input at the root. The second figure is the total time elapsed until all leaves are assigned a type. The difference between these two figures corresponds to the propagation time of the last input entry. The program code is loaded immediately after downloading the tree description. The propagation delay can be reduced by pipelining the code loading phase with the node typing phase.

Algorithm	I.a		I.b		II	
Loader size	123 words		72 words		90 words	
	input words	time inst. cyc.	input words	time inst. cyc.	input words	time inst. cyc.
Ex 1	4	75/141	4	64/122	12	137/201
Ex 2	18	255/471	18	220/408	27	413/621
Ex 3	27	269/433	164	1420/1661	15	186/370
Ex 4	6	123/339	6	64/252	6	51/235
Ex 5	69	627/836	10	116/306	6	51/235

Ex 1. 4-level binary tree with two different node types

Ex 2. 9-level binary tree, one node type per level

Ex 3. 5-level 3-ary tree, one node type per level

Ex 4. 2-level 2^8-ary tree, best fanout for I.a and I.b

Ex 5. 2-level 171-ary tree (worst case fanout for Algorithm I.a, $2^7 \leq f < 2^8$)

Table 1. Comparison of Algorithms I.a, I.b, and II for some simple trees

The ratios of program sizes of Algorithms Ia,Ib, and II are 1.7:1:1.3. The smaller program size for Algorithm II compared to Ia is a consequence of the decision to require at least one entry per level in the logical tree in Algorithm II. This constraint reduces the program complexity significantly. The program of Algorithm I.b is the shortest among the three because the bit-reversed ordering simplifies the logic of the algorithm.

Algorithm II is the slowest of the three algorithms for binary trees, Examples 1, and 2. Its processing at the root requires about 50% longer time than Algorithm I.b for binary trees with all nodes at a level being equal. The processing time increases linearly with the height of the binary tree for all three algorithms.

Examples 2, 3, 4, and 5, are all mapped onto a 9-level binary tree. The total number of nodes in each logical tree is 511, 121, 257 and 172 respectively. The total number of used nodes in the binary tree, including the padding nodes, is 511, 161, 511 and 341 respectively. Because of the difference of the logical tree structures, the execution time differs significantly for different algorithms. Notice in example 3(5), Algorithm I.b (Algorithm I.a) has a performance much worse than the other two algorithms. On the contrary, Algorithm II always yields a satisfactory performance for a large fanout.

The length of the input is independent of the fanout for Algorithm II, whereas the length of the input for Algorithm I.a and I.b depends on the fanout, Figures 7 and 10.

For fanouts equal to a power of two the corresponding binary tree is always balanced. The input length for Algorithm I.b as well as the total execution time is less than that of Algorithm II, except for a two level tree, Figures 7 and 8. Algorithm II is more efficient than Algorithm I.a for fanouts equal to a power of two.

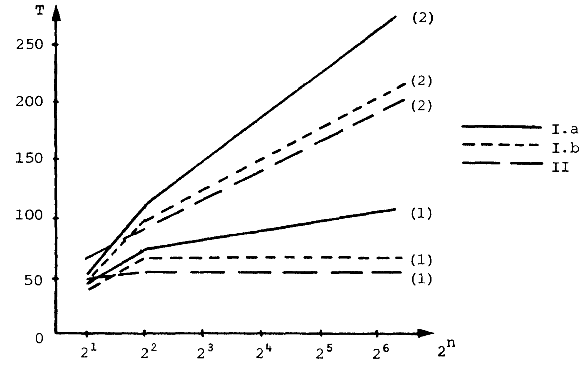


Figure 7. Instruction cycles for the root (1) and for completing the loading phase (2) for a 2-level 2^n-ary tree. Successive levels having different node types.

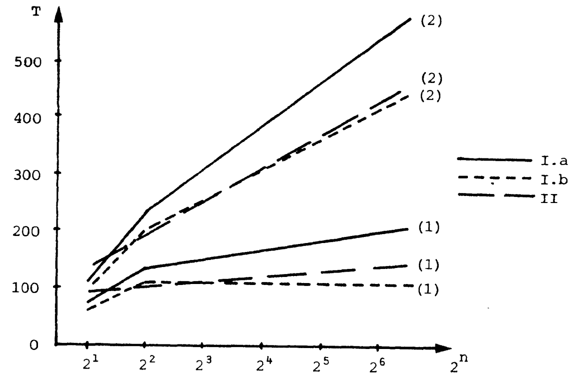


Figure 8. Instruction cycles for the root (1) and for completing the loading phase (2) for a 3-level 2^n-ary tree. Successive levels having different node types.

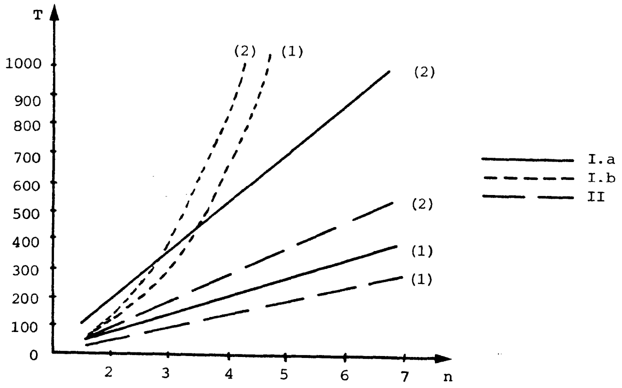


Figure 9. Instruction cycles for the root (1) and for completing the loading phase (2) for a n-level 3-ary tree. Successive levels having different node types.

However, Algorithm II performs in a superior manner for most fanouts. Algorithm I.b has an exponential growth with the height of the tree, Figure 9, except for fanouts equal to a power of two, Figure 8. The exponential growth rate is independent of the fanout. The growth rate for Algorithm I.a varies from linear to exponential depending upon the fanout. It may be better or worse than Algorithm I.b, Figures 9 and 10. The growth rate of Algorithm I.a as a function of the worst case fanout in each interval, $2^n \leq f < 2^{n+1}$ follows a square root function, Figure 10.

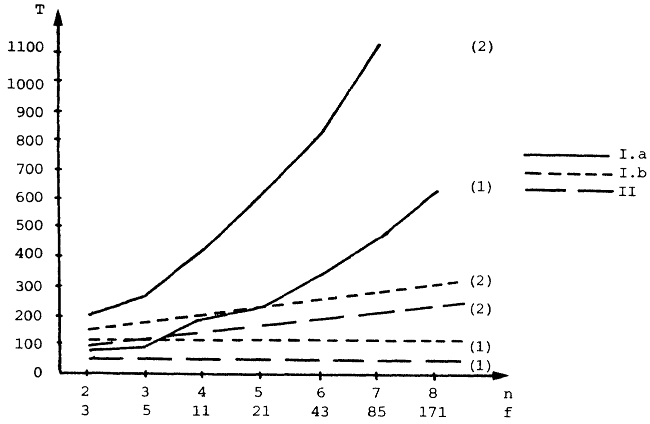


Figure 10. Instruction cycles for the root (1) and for completing the loading phase (2) for a 2-level f-ary tree. Successive levels having different node types.

The asymptotic loading times are summarized in Table 2. These time bounds are also bounds for the length of the input, except for Algorithms I.a and I.b, for which the input can be reduced to O(1) for a binary tree with all nodes identical.

Algorithm	I.a	I.b	II
Best case	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$
Worst case	$O(\sqrt{N/f})$	$O(N^{1/\log_2 f})$	$O(\log_2 N)$

Table 2. Estimated loading times for k-level, f-ary trees, of N nodes. All nodes of a logic level are of the same type

In conclusion, for a non-binary logical tree, Algorithm II has the best asymptotic properties, $O(\log_2 N)$, is also efficient for small trees, and has a compact, simple code. Its performance for binary trees is close to that of the other two algorithms.

References

[1] Browning, S.A., "The Tree Machine: A Highly Concurrent Computing Environment", TR3760, Ph.D. Thesis, Computer Science, Caltech, Jan. 1980

[2] Seitz, C.L., "Ensemble Architectures for VLSI - A Survey and Taxonomy" Proc. of Conf. on Advanced Research in VLSI, pp. 130-135, 1982

[3] Mago, G.A., "A Cellular Computer Architecture for Functional Programming" in "Proc. Compcon Spring, 1980", pp 179-187, IEEE Computer Society, 1980

[4] Shaw, D.E., "The Non-Von Supercomputer", Technical Report, Columbia University, August, 1982

[5] Li, P, "The Tree Machine Operating System", TR4618, Computer Science, Caltech, July 1981

[6] Johnsson, S.L., "Highly Concurrent Algorithms for Solving Linear Systems of Equations" in "Elliptic Problem Solver II", Academic Press, 1983

OPTIMAL ROUTING ALGORITHMS IN MULTICOMPUTER NETWORKS ORGANIZED AS RECONFIGURABLE BINARY TREES

Svetlana P. Kartashev (University of Nebraska, Lincoln)

and

Steven I. Kartashev (Dynamic Computer Architecture, Inc., Lincoln, Nebraska)

Abstract

This paper discusses efficient routing algorithms for multicomputer networks organized as *reconfigurable binary trees*. Communication techniques introduced are optimal from both viewpoints: the total bit size of routing information code, BS(RI), that routes the message among various transit nodes of the communication path and the total time of communication. It is shown that $BS(RI) \leq \log_2 K + 2 \log_2 \log_2 K$, where K is the total number of nodes in a reconfigurable binary tree. Further, since each transit node N selects its either right or left successor in the communication path via simple logical operation that takes time of one gate delay, the total time, CD, of the $N_s \rightarrow N_d$ communication also approaches the theoretical minimum where N_s is a source node, N_d is a destination node.

1. Introduction

Binary tree is a very popular implementation for a distributed computing system in which each tree node is implemented as a separate and autonomous computational node [1,2,3]. The reason for this is that for conventional computations, the tree structure describes a variety of control and computational algorithms.

Especially important is the use of binary trees in distributed data bases, since most of the file accesses algorithms for hierarchical data bases are based on queries organized as a binary tree [4]. Trees have two types of nodes: *leaves* and *non-leaves*, where a leaf is understood as a node of the lowest level, i=0, and a non-leaf node has level $i \geq 1$, where $i \leq n$ for the n-level tree. In this tree the node of the highest (n) level is called *the root*.

A multicomputer network that reconfigures into trees as well as other useful structures (stars and rings) can be organized if its nodes identified with computer elements, CE, are interconnected with the memory-processor bus (or DC-bus) described in [5,6]. To organize a data broadcast among a pair of nodes N and N* interconnected with the DC bus, it is sufficient for network node N to generate the position code or address of N*.

Activation of data path between N and N* will be denoted as *transition* N → N* meaning that N will generate the position code or address of N*; and N will establish a data path between N and N*.

It is assumed that during reconfiguration, the maintained direction of succession is from leaves to the root, whereby each node N generates only one position code N* of its immediate successor in the binary tree. This organization will minimize the total time of reconfiguration.

During regular node to node communications, both transitions N → N* and N* → N will be maintained.

The following procedure will be used to generate various tree structures that can be assumed by a distributed computing system. (This procedure is a particular case of more general procedure described in [6,7] that can generate not only trees, but also rings and stars.)

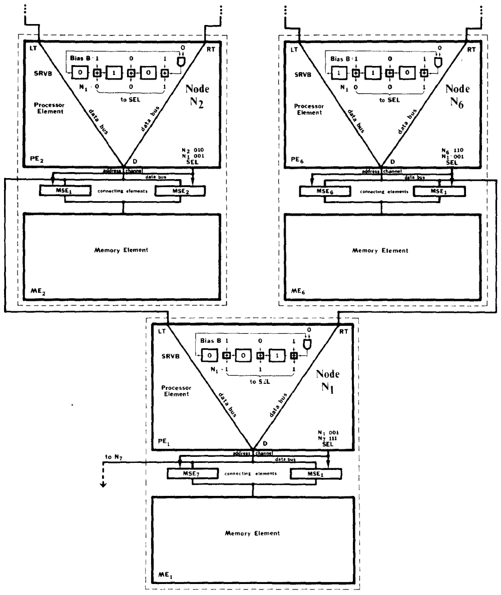


Fig. 1

Fig. 1 Generation of reconfigurable binary tree with the use of the SRVB-register

Assume that each tree node N is provided with a special shift-register of length n which stores its position code N, where n is the size of the code (Fig. 1). Then for each type of communication between N and N* (PE-PE*, PE-ME*, ME-ME*), node N uses the following *reconfiguration equation* to generate the position code N* of its single successor in the binary tree:

$$N^* = 1[N]_0 + B \qquad (1)$$

where $1[N]_0$ is one bit non-circular shift of N to the left and B is an n-bit reconfiguration constant called *bias* and brought with the reconfiguration instruction to *all network nodes that are requested for reconfiguration*. The shift-register of Fig. 1 is called a *shift-register with the variable bias* (SRVB). Thereafter, subscript 0 will be omitted in this paper because it is always 0 for reconfigurable binary trees. For other network structures, the SRVB may perform a circular shift whereby the gate that follows the least significant bit is fed with signal 1[6].

In Fig. 2, the network of thirty two nodes receiving bias

0190-3918/83/0000/0206$01.00 © 1983 IEEE

B=11010 reconfigures into the 5-level with the root R=10110. Since there are 2^n different biases, it is possible to generate 2^n different trees with these techniques.

In any reconfigurable binary tree, organization of efficient communication among tree nodes presents a major problem because position codes of tree nodes in the hierarchy of tree levels may change. Therefore, to route a communication message from a source node, N_s, to a destination node, N_d, requires to store the position codes of all transit nodes of the $N_s \rightarrow N_d$ communication path. This routing information RI can be either stored in a communication message CM or distributed among transit nodes so that each transit node stores the position code of its successor in the $N_s \rightarrow N_d$ communication path.

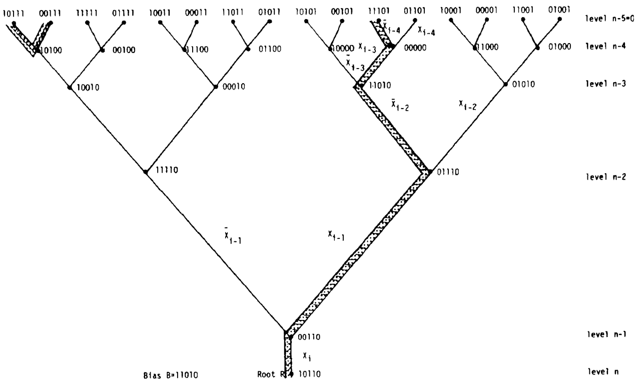


Fig. 2 Reconfigurable binary tree having bias B=11010 and root R=10110.

Both these alternatives are infeasible since they lead to a large bit size of the RI-code and a long time of internode communication associated with the large size of RI.

This paper is dedicated to solution of communication problem in a reconfigurable binary tree that allows a dramatic reduction in both the total bit size of the RI code and the overall time of communication. Presented routing techniques allow one to obtain the following upper bound on bit size of RI:

$$BS(RI) \leq \log_2 K + 2 \log_2 \log_2 K. \tag{3}$$

Also, each transit node delays a CM-message by not more than $3t_d$. Therefore, the maximal communication delay CD for the longest $N_s \rightarrow N_d$ communication path that includes 2 $\log_2$ K-1 tree nodes is upperbounded as follows:

$$CD \leq 3t_d (2 \log_2 K-1) \tag{4}$$

Therefore, introduced communication algorithms are optimal from both viewpoints--bit size of the routing information and total communication delay.

The relationship of this paper with other work in the area is as follows:

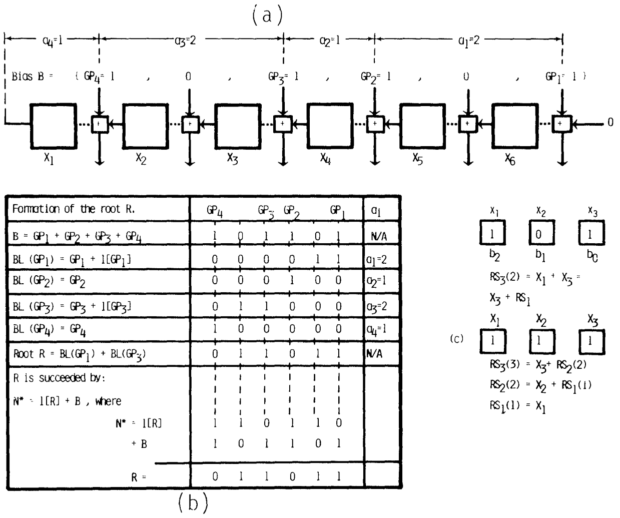
First, by introducing original routing algorithms for distributed reconfigurable computing systems organized as binary trees, it is connected with other works on reorganizations and reconfigurations discussed in [8-13].

Second, by using the shift-register theory to develop reconfigurations of binary trees, it is connected with the literature on shift-register sequences [14-19].

Third, by discussing distributed computing systems organized as binary trees, it is associated with [1-4].

2. Synthesis of a Tree Node of Level n-i

To form efficient routing algorithms, it is necessary to develop a simple synthesis procedure aimed at finding a tree node of level n-i, i<n. If i=0, the node found is the root, R, of level n; if i>1, it is any other node of the tree; if i=n, it is the node of level 0, i.e., a leaf. To find the root, R, we define the bias structure for a non-circular SRVB as follows: Given bias B = GP_1 + ... + GP_t where GP_i (i=1, ..., t) is its ones position otherwise called *generating position*. Let a_1, a_2, ..., a_t be *bias distances* defined as follows: $GP_{i+1} = a_i[GP_i]$, i.e., a_i shows the number of left-hand shifts between GP_i and GP_{i+1}, where i changes from 1 to t-1. For GP_t, $a_t[GP_t]$ = 0, since we are having a non-circular shift (Fig. 3a).



Formation of the root R.	GP_4		GP_3	GP_2		GP_1	a_i
B = GP_1 + GP_2 + GP_3 + GP_4	1	0	1	1	0	1	N/A
BL (GP_1) = GP_1 + 1[GP_1]	0	0	0	0	1	1	a_1=2
BL (GP_2) = GP_2	0	0	0	1	0	0	a_2=1
BL (GP_3) = GP_3 + 1[GP_3]	0	1	1	0	0	0	a_3=2
BL (GP_4) = GP_4	1	0	0	0	0	0	a_4=1
Root R = BL(GP_1) + BL(GP_3)	0	1	1	0	1	1	N/A
R is succeeded by: N* = 1[R] + B , where							
N* = 1[R]	1	1	0	1	1	0	
+ B	1	0	1	1	0	1	
R =	0	1	1	0	1	1	

Fig. 3 (a) Bias structure.
(b) Formation of BL-sums and the root.
(c) Recursive sums.

For each generating position, GP_i, let us form a mod 2 sum of all left j-bit shifts of GP_i ranging from j = 0 to j = a_i-1, where a_i is the bias distance such that $a_i[GP_i] = GP_{i+1}$. Denote this sum as $BL(GP_i) = GP_i + 1[GP_i] + ... + (a_{i-1})[GP_i]$. By construction, GP_i is an addend of $BL(GP_i)$ and GP_{i+1} is not an addend of $BL(GP_i)$ (Fig. 3b). Form BL sums for odd generating positions of the bias as $BL(GP_1)$, $BL(GP_3)$, ..., $BL(GP_k)$ where k=t if t is odd and k=t-1 if t is even. Then the root R is:

$$R = BL(GP_1) + BL(GP_3) + \ldots + BL(GP_k) \tag{4-1}$$

The significance of this formula is illustrated by Fig. 3b.

As we have seen, the technique for constructing the root is very simple and can be easily performed by the programmer who can find the root for the given bias B and then store it in the instruction that reconfigures the network into the given single binary tree or the root R can be formed inside tree node via a dedicated logical circuit that includes two shift-registers interconnected with the mod 2 counter.

Once a tree node N finds the root, R, it can form the position code of any other node N' of level n-i, using a so called *recursive sum* code $RS_i(k)$ defined as follows via a simple inductive procedure:

Basis: $RS_0\ (0) = 0,\ i = 0,\ k = 0$

Inductive Step: $RS_i\ (k) = X_i + RS_m\ (k-1)$, where $k \leq i$, $X_i = 2^{n-i}$, and $m < i$.

Therefore, if SRVB stores $RS_i\ (k)$, its position b_{n-i} is always 1, and (k-1) is the total number of other more significant ones positions (Fig. 3c).

For instance, $RS_3\ (3) = X_1 + X_2 + X_3$ can be represented as follows via this definition: $RS_3\ (3) = X_3 + RS_2\ (2)$;

$RS_2\ (2) = X_2 + RS_1\ (1)\ RS_1\ (1) = X_1$

For a recursive sum, $RS_i\ (k) = X_i + RS_m\ (k-1)$, X_i will be called a *level variable*.

As shown below   level variable $X_i = 1$ uniquely specifies a tree node N(n-i) of level n-i as follows: $N(n-i) = R + RS_i(k)$ where $k \leq i$, R is the root. (4-2)

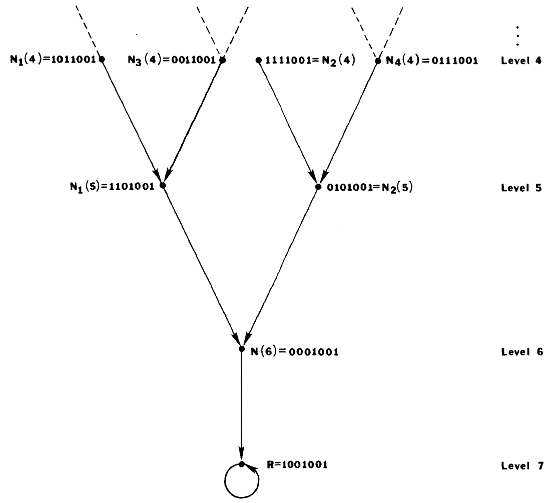


Fig. 4 Construction of tree nodes of arbitrary levels.

Example. Given bias B = 1011011. Let us construct all the nodes of levels n-i = 7, 6, 5, and 4 (Fig. 4). Here, n=7. Thus the root is of level 7: $R = BL(GP_1) + BL(GP_3) + BL(GP_5)$ and $GP_1 = 1$, $GP_3 = 8$, and $GP_5 = 64$; $BL(GP_1) = GP_1$; $BL(GP_3) = GP_3$, and $BL(GP_5) = GP_5$, thus, R = 1001001. Indeed, the successor N* of R is N* = 1[R] + B = 1[1001001] + 1011011 = 0010010 + 1011011 = 1001001 = $R = X_1 + X_4 + X_7$.

The node of level 6 is: $N(n-1) = R + RS_1\ (1) = R + X_1$ = 0001001 = $X_4 + X_7$. This node is succeeded by the root because N* = 1[N(6)] + B = 0010010 + 1011011 = 1001001 = R. There are two nodes of level n-2 = 5: $N_1(5) = R + RS_2(1) = R + X_2$ = 1101001 = 105 and $N_2(5) = R + RS_2(2) = R + X_1 + X_2 = X_2 + X_4 + X_7$ = 0101001. Indeed, $N_1(5)$ is succeeded by $N^* = 1[N_2(5)] + B$ = 1010010 + 1011011 = 0001001 = N(6) = 9. Likewise, $N_2(5)$ is succeeded by the same $N^* = 1[N_2(5)] + B$ = 1010010 + 1011011 = N(6), etc.

Similarly, one can construct all the other nodes of this tree.

3. Generation Inside a Transit Node, N of its Immediate Neighbors in a Reconfigurable Binary Tree

During reconfiguration into a new tree, each *transit* node N is the source of the N → N* transition and the destination of the LN → N and RN → N transitions (Fig. 5a). Since LN and RN are on the same level, they are specified with the same level variable X_i, i.e., $LN = R+RS_i$ and $RN = R+RS_i'$. Assume that the left node LN is always greater than the right node, RN, i.e., LN>RN. To order LN and RN, we have to order recursive sums. Assume that they are ordered as follows:

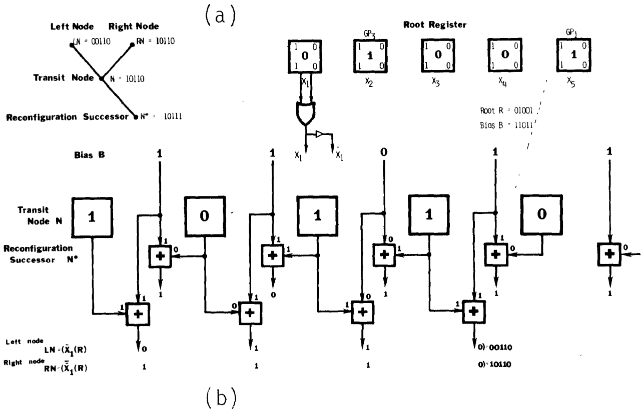


Fig. 5 (a) Transit node N and its closest neighbors in the reconfigurable binary tree (Root R=01001, bias B=11011).

(b) Generation inside node N of the position codes for its immediate neighbors in the reconfigurable binary tree having bias B=11011 and root R=01001.

P1. $RS_i > RS_j$ if $X_i > X_j$ where $X_i = 2^{n-i}$, $X_j = 2^{n-j}$ and

P2. $RS_i > RS_i'$ if $RS_i + X_i > RS_i' + X_i$

Application of P1 and P2 rules gives us $LN = R + RS_i$ and $RN = R + RS_i + X_1$, i.e., $LN + RN = X_1$. Furthermore, since the root, $R = R + RS_0$ and RS_0 is greater than any other recursive sum, inasmuch as it is specified by the level variable $X_0 = 2^{n-0} = 2^n$, we obtain that the left node LN necessarily has the same meaning of $\tilde{X}_1$ variable as X_1 of the root R. On the other hand, the right node RN always has its X_1 variable reversed, i.e., opposite the one of the root.

Therefore, we can use the following techniques for generating LN and RN codes inside node N: Using reconfiguration equation (1), we find that 1[LN] = N + B. Thus,

$$LN = \tilde{X}_1\ (R) + 1^{-1}[N+B] \tag{5}$$

where 1^{-1} [...] is non-circular shift to the right and $\tilde{X}_1$ (R) is the meaning of the most significant variable in the node LN.

Similarly, since 1[RN] = N+B,

$$RN = \bar{\tilde{X}}_1\ (R) + 1^{-1}[N+B] \tag{6}$$

Thus, LN and RN are obtained inside node N via the *right-shifted non-circular* SRVB receiving bias B, whereas N* is obtained conventionally inside node N using the left-shifted non-circular SRVB receiving the same bias B (Fig. 5b).

4. Communication Circuits Inside Transit Node

To perform fast internode communications, each tree node mustbe provided with the communication circuits that allow its effective functioning as a transit node, i.e., the one which is provided with three immediate neighbors LN, RN and N* (Fig. 5) or which merely passes communication messages to one of its neighbors LN, RN or N*. Since messages may flow in the two directions, *TOR* and *TOL*, where TOR means to the root and TOL means to the leaves, each transit node has T and D terminals respectively designated for TOR broadcast if message passes from T to D, or for TOL broadcast otherwise.

To maintain concurrent communications whereby a message received by the D-terminal is allowed to flow concurrently to the left and right neighbors of the given node N, each transit node is provided with the two T-terminals, LT and RT, that connect given node N with LN node and RN node, respectively (Fig. 6).

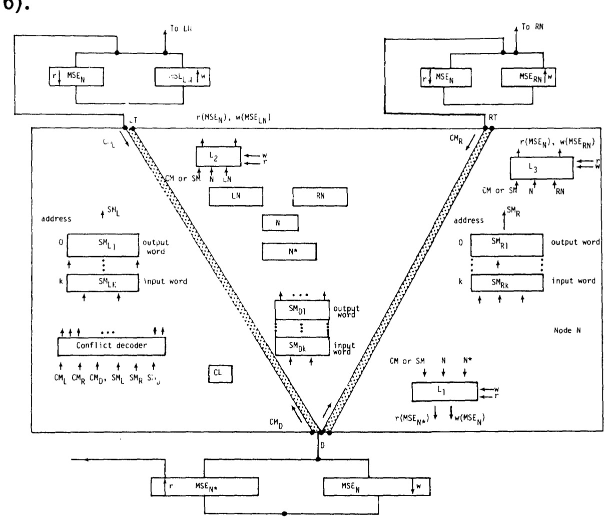


Fig. 6 Communication circuits inside each tree node.

Each connection of the transit node N with its successor N* and the two predecessors LN and RN in the tree is performed via a pair of connecting elements, MSE, selected during reconfiguration.

4.1. Conflict Resolution Among Concurrent Messages

Since there are three independent terminals in each transit node, it may receive up to three concurrent messages at a time: CM_L, CM_R and CM_D. Since it may pass in transit only one message at a time, the node performs the conflict resolution via special logic called *conflict decoder*, CD (Fig. 6). Conflict decoder is a logical circuit that has a separate output for each conflict situation.

The decisions made by the conflict decoder are based on the following rules:

1. Of several messages that must use node N in transit, only one message is allowed to pass at each clock period. The remaining messages are saved. Since a transit node has three terminals (LT, RT, D), each terminal is provided with the individual push-down stack of save registers used to store saved messages, SM.

2. If a saved message, SM, is concurrent with a current message, CM, a decision to pass is given to a SM message and a CM message is saved.

3. If a message received via D-terminal (saved, SM_D, or current, CM_D), is concurrent with left or right messages, a decision to pass is in favor of a message received via D-terminal. The remaining messages are saved.

4. Of the two concurrent left and right messages, the decision to pass favors left message at even clock periods and right message at odd clock periods.

Example: Suppose that a transit node N has six concurrent messages competing for transit: CM_L, CM_R, CM_D, SM^0_L, SM^0_R and SM^0_D, where upper subscript for saved messages shows their addresses in the register stacks. We assume that each stack has only one saved message and no current message arrives until all messages are passed. At moment T_0, conflict decoder allows SM^0_D to pass; CM_D is saved as new SM^0_D. CM_L is saved as SM^1_L; CM_R is saved as SM^1_R; at moment T_1, SM^0_D is allowed to pass; down stack has no saved messages; left and right stacks store two messages each (Fig. 7). At even moment T_2, SM^0_L passes; left stack retains one saved message; right stack retains two messages; at odd moment T_3, SM^0_R passes; left and right stacks retain one message each, etc. The entire message transit ends at T_5.

Time	Passed Messages	Saved Messages			Current Messages
		Left Stack	Right Stack	Down Stack	
Initial Condition	-	SM^0_L	SM^0_R	SM^0_D	CM_L, CM_R, CM_D
T_0	SM^0_D	SM^0_L, SM^1_L	SM^0_R, SM^1_R	SM^0_D	-
T_1	SM^0_D	SM^0_L, SM^1_L	SM^0_R, SM^1_R	-	-
T_2	SM^0_L	SM^0_L	SM^0_R, SM^1_R	-	-
T_3	SM^0_R	SM^0_L	SM^0_R	-	-
T_4	SM^0_L	-	SM^0_R	-	-
T_5	SM^0_R	-	-	-	-

Fig. 7 Conflict resolution among communication messages.

4.2. Dynamic Activation of Transfer Modes in Connecting Elements

As was indicated above, each transit node N is connected with its predecessor (LN or RN) or successor N* via a dedicated pair of connecting elements selected during reconfiguration (Fig. 6). For the N → N* broadcast, (MSE_N-MSE_{N*})-pair belongs to N; for N → LN broadcast, (MSE_N-MSE_{LN})-pair belongs to LN; for N →RN broadcast, (MSE_N-MSE_{RN})-pair belongs to RN (Fig. 1). The mode of transfer of each pair of connecting elements depends on the direction of broadcast. For the TOR communication N → N*, from N to its successor, N*, connecting element MSE_N should be activated in the *write* direction ($w(MSE_N)$) and MSE_{N*} should be activated in the *read* direction ($r(MSE_{N*})$). For the TOL communication N → LN, the direction of transfer in MSE_N and MSE_{LN} reverse to read for MSE_N and write for MSE_{LN} (Fig. 6). Similar directions are true for the N → RN broadcast.

Since at each clock period, a transit node N may change the direction of broadcast, we assume that the modes of transfer in connecting elements will be activated dynamically by the messages that are allowed to pass through a given output terminal.

For the TOR broadcast via D-terminal, such activation will be performed by the L_1-logic which receives several *static inputs* and one *dynamic* input from the allowed message (saved or current). Static inputs are: code N for selecting MSE_N; code N* for selecting MSE_{N*}; r-signal for $r(MSE_{N*})$ and w-signal for $w(MSE_N)$. The L_1 logic is activated concurrently during TOR message transit via N; thus, it introduces no additional delay into a message transit. For the TOL left broadcast via LT-terminal, this dynamic activation is performed by the L_2 logic which issues $r(MSE_N)$ and $w(MSE_{LN})$.

Similarly, for the TOL right broadcast via RT-terminal, L_3 logic performs dynamic activation of two connecting elements MSE_N and MSE_{RN}, as $r(MSE_N)$ and $w(MSE_{RN})$.

Since the same pair of connecting elements MSE_N and MSE_{N*} that connect N and N* may be activated by N and N* in the opposite directions, MSE_N and MSE_{N*} are provided with a simple logic to resolve these conflicts. The organization of this logic is similar to the one discussed above for resolving conflicts among concurrent messages. The only difference is that it is much simpler, since there is only one type of conflict that may arise. Thus, the conflict logic will be reduced to a single 2-input gate.

5. Individual Communications

Below, we will introduce communication algorithms which allow efficient internode communications. Two types of communication will be considered:

1. Node-Root communication, and

2. Node-Node communication.

5.1. Node-Root Communications

There are two types of node-root communication: (a) node → root and (b) root → node.

5.1.1. *Node to Root Communication, $N_S \rightarrow R$.* For the $N_S \rightarrow$ R communication, N_S is the source, R is the destination; N_S generates communication message CM that stores the position code of the root, R, defined in Eq.(4-1). In passing this message, each transit node, N, compares R with its own position code, N. If N = R, the message reaches the destination. If N ≠ R, it is passed top down from one of the top terminal (LT or RT) to the only one D-terminal.

5.1.2. *Root-Node Communication, $R \rightarrow N_d$.* For this case, the message is generated by the root, i.e., R is the source, N_d is the destination. Message, CM, stores a so-called address code of N_d defined as follows:

By the *address code*, AC_N of the node N, we define such a binary word which contains all the routing information that allows a communication message to reach node N from the root via the minimal communication path (i.e., the one which traverses each node of the tree only once).

Below, we define a procedure which allows one to obtain the address code, AC, of node N_d via simple logical operation performed over its recursive sum, RS_i introduced in Eq. (4-2)

As was specified by Eq. (4-2)node N_d of level (n-i) is defined as follows:

$N_d = R+RS_i$, where RS_i is the recursive sum.

The immediate more significant successor of this node in the tree otherwise called reconfiguration 1-successor, can be found via the reconfiguration equation (Equation 1) as:

$$N^* = 1[N_d] + B = 1[R] + 1[RS_i] + B.$$

Since for the root, R,

$$R = 1[R] + B,$$
$$N^* = 1[R] + 1[RS_i] + B = R+1[RS_i] = R+RS_{i-1}$$

Therefore, N* (the reconfiguration 1-successor of N_d) is the node of the next more significant level, n-(i-1), defined via recursive sum RS_{i-1} which is a one-bit shift to the left of the original recursive sum RS_i that specifies node N_d. Similarly, it is easy to show that the reconfiguration 2-successor of node N_d in the communication path with the root is specified by recursive sum RS_{i-2} which is 2-bit shift of RS_i: $RS_{i-2} = 2[RS_i]$, where RS_i specifies node N.

Since by definition, level variable X_i=1 specifies the least significant position of RS_i of the node N_d it will be the last one which will be shifted out during consecutive shifts of RS_i. Further, since $X_i \equiv 1$ in RS_i, it will always specify the only routing alternative from the root R to the next less significant node $R+X_1$ of level n-1 (Fig. 2).

The value of X_{i-1} in RS_i which is next to X_i specifies the route from level n-1 to level n-2.

If X_{i-1} = 1, the only node $R+X_1$ of level n-1 is followed by the right node, RN = $R+X_1+X_2$ of level n-2. If X_{i-1} = 0, node $R+X_1$ is followed by the left node LN = $R+X_2$ of level n-2. Similarly, it is easy to show that variable X_{i-2} is responsible for selecting the route from level n-2 to level n-3, etc. Therefore, as follows from this analysis, consecutive variables X_i, X_{i-1}, X_{i-2}, ..., X_1 that are present in RS_i where X_i = 1 and X_j=0 or 1 are responsible for the entire route selection from one level to the next in the minimal communication path from the root R to the destination node N_d.

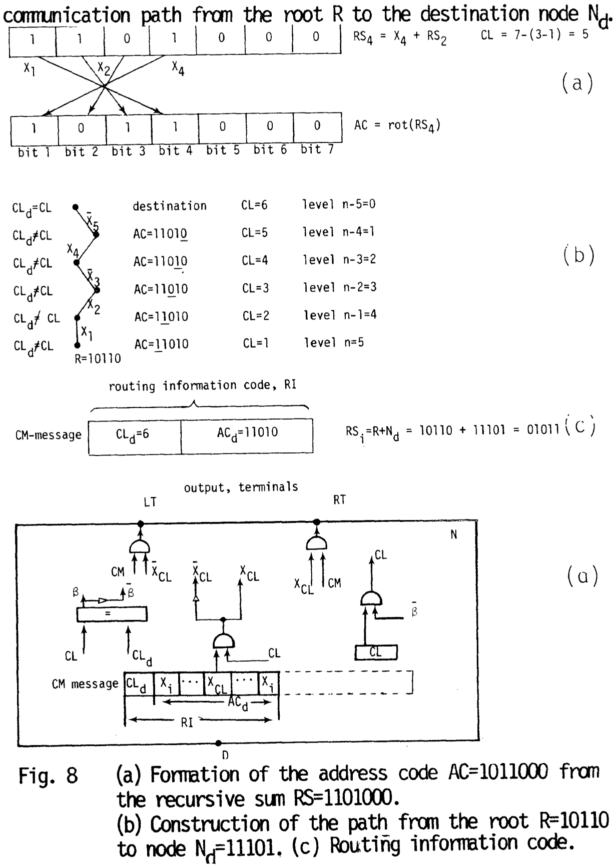


Fig. 8 (a) Formation of the address code AC=1011000 from the recursive sum RS=1101000.
(b) Construction of the path from the root R=10110 to node N_d=11101. (c) Routing information code.

Also,the recursive sum RS_i = R+N_d contains all the routing information necessary to forward the message issued by the root to the destination node N_d. Namely, the address code AC_d may be obtained from $RS_i = N_d+R$ by moving its bit X_i to the bit 1 of AC; X_{i-1} has to be moved to bit 2, ..., X_1 to bit i of AC (Fig. 8a). This simple logical operation will be called *rotate* or symbolically, AC = rot(RS_i) (Fig. 8a).

Therefore, as was shown, the address code AC_d of the destination node N_d obtained from its recursive sum $RS_i = N_d+R$ via simple rotation, as AC_d = rot (RS_i) contains all the routing information necessary to route the message from R to N_d. However, to reduce the routing algorithm to execution of a simple logical operation in each transit node, we will stipulate that each node of level n-j will store a so-called *complemented level code*, CL (Fig. 8b) of the size $\log_2 n = \log_2 \log_2 K$ that shows which bit of the address code must be selected for finding the route from the given node to the next node in the path. To find CL, each node finds its individual recursive sum as RS_i = R+N, then shifts RS_i to the LSB until LSB = 1 and counts the number of shifts k; CL = n-(k-1), where n is the number of levels in the tree. Each node finds its CL when a new tree configuration is established. Thereafter, it is stored in a special register CL, of size $\log_2 n$, since it will select the routing bit in all communications that will pass through a given transit node (Fig. 6).

Routing Algorithm for R → N_d *Communication.* Given: root R, destination node N_d; CL value stored in each node of the tree. The algorithm routes the message issued by R to a destination node N_d. The algorithm contains two steps: *root step* and *transit step*. The root step is aimed at finding the address code, AC_d of the destination node, forming the RI code for the communication message, CM and passing CM to the only node of level n-1. Transit step is executed iteratively in each transit node of communication path. It is aimed at finding the next transit node of the communication path and moving the CM message to this node.

The entire execution of this algorithm is shown in Fig. 8(a,b,c), for finding the route from the root R = 10110 to node N_d = 11101 (Fig. 2). Fig. 8a forms the address code AC_d of the node N_d. Fig. 8b shows how to find CL code in each transit node of the path R → N_d. Fig. 8c forms the RI code and shows what actions are performed in each transit node N that receives CM message from its predecessor in the communication path.

5.2. Node-Node Communication, $N_s \to N_d$

The node-to-node communication is a more general case of N_s → R and R → N_d communications considered where N_s is the source node, N_d is the destination node. Indeed, for any N_s → N_d communication, the task of an optimal routing technique is to find a so-called *the closest intermediate root*, CIR, which is understood as the tree node that connects N_s and N_d with the minimal communication path N_s T CIR, CIR → N_d (Fig. 9), i.e., for N_s → R, R → N_d, CIR = R. Thus, by developing efficient node-to-node communication techniques that bypass the absolute root, R, we will reduce the amount of traffic that goes to and from the root, R, and minimize the total length and the total delay of the communication path that connects N_s and N_d.

A particular case of the N_s → N_d communication is when

(1) N_s is a transit node on the path that connects the root, R, with N_d, or

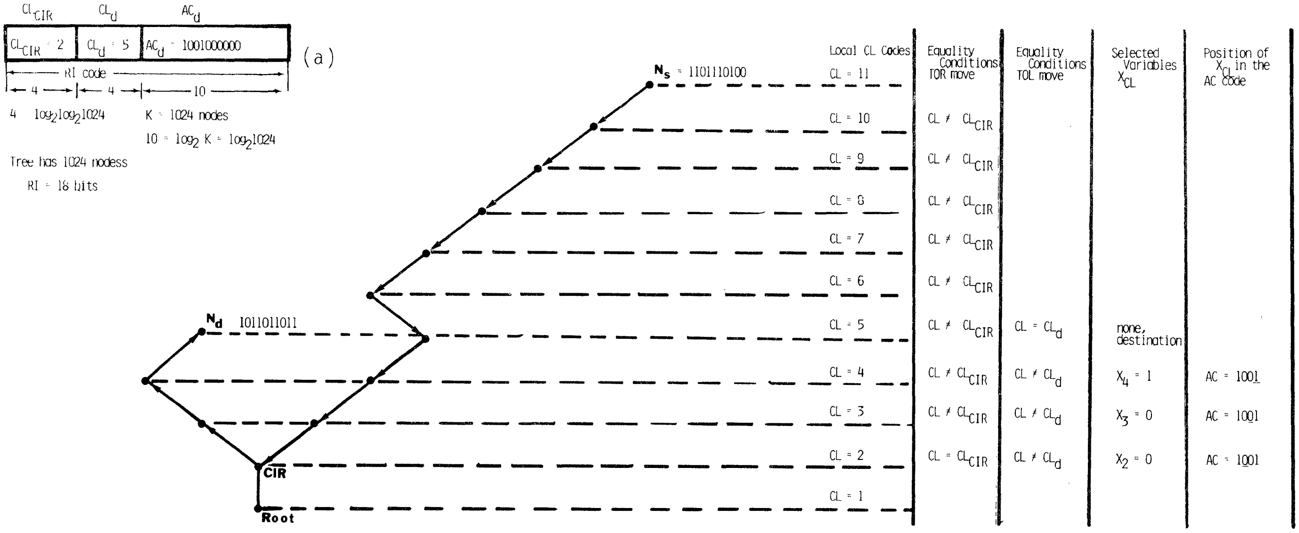
(2) N_d is a transit node on the path that connects the N_s with the root, R.

The first case is reduced to the R → N_d communication, whereas the second case is reduced to the N_s → R communication considered above in Section 5.1. Therefore, our discussion will concentrate on finding the minimal N_s → N_d communication path in which neither N_s nor N_d are on the subpath that connects the root with another node (N_d or N_s).

5.2.1. *Closest Intermediate Root.*

To find the minimal N_s → N_d communication path, we have to find first, the closest intermediate root, CIR (Fig. 9).

Since CIR is on the subpath that connects the root, R, with both N_s and N_d, the address code, AC_{CIR} of the CIR is a left prefix of the two address codes, AC_s and AC_d, because if root, R, generates communication message to N_s or N_d, then the CIR is the transit node on both paths, R → N_s, R → N_d. Thus, every message issued by the R root passes CIR before it reaches N_s and N_d. Therefore, the address code of CIR, AC_{CIR},



Local CL Codes	Equality Conditions TOR move	Equality Conditions TOL move	Selected Variables X_{CL}	Position of X_{CL} in the AC code
CL = 11				
CL = 10	CL ≠ CL_{CIR}			
CL = 9	CL ≠ CL_{CIR}			
CL = 8	CL ≠ CL_{CIR}			
CL = 7	CL ≠ CL_{CIR}			
CL = 6	CL ≠ CL_{CIR}			
CL = 5	CL ≠ CL_{CIR}	CL = CL_d	none, destination	
CL = 4	CL ≠ CL_{CIR}	CL ≠ CL_d	X_4 = 1	AC = 1001
CL = 3	CL ≠ CL_{CIR}	CL ≠ CL_d	X_3 = 0	AC = 1001
CL = 2	CL = CL_{CIR}	CL ≠ CL_d	X_2 = 0	AC = 1001
CL = 1				

Fig. 9 (a) The RI-code for the Node Ns → Node N_d communication path.

(b) Formation of the N_s → N_d communication path.

is the common most significant portion or the common *left prefix* of AC_s and AC_d.

The number of bits in AC_{CIR} can be determined as follows. First, we find MS = $AC_s + AC_d$, and the number of *consecutive most significant zeros* in MS. This will give us the number of bits in the common left prefix of both AC_s and AC_d or the bit size of AC_{CIR}. Thus, the following technique can be used for finding the address code of the closest intermediate root, AC_{CIR}, and its complemented value, CL_{CIR}.

Algorithm for Finding AC_{CIR} and CL_{CIR}

Given: N_s, N_d and R

Result: AC_{CIR} and CL_{CIR}

Step 1: Find AC_s and AC_d.

Step 2: Find $AC_s + AC_d$ = MS.

Step 3: Find the number of consecutive and most significant zeros, LO, in MS; AC_{CIR} is the left prefix of either AC_s or AC_d that contains LO binary digits. Complemented level value of CIR, CL_{CIR} = LO+1.

Example: Let us find AC_{CIR} and CL_{CIR} for N_s and N_d given by the following position codes:

N_s = 1101110100, N_d = 1011011011. Assume that these are tree nodes in a binary tree having root R = 0010011011 and bias B = 0110101101 (Fig. 9). First, we find address codes AC_s and AC_d. To do so, we have to find recursive sums RS(N_s) and RS(N_d):

RS(N_s) = N_s+R = 1101110100 + 0010011011 = 1111101111;

RS(N_d) = N_d+R = 1011011011 + 0010011011 = 1001000000.

Address code of N_s is found by rotation of RS(N_s):

AC_s = rot (RS(N_s)) = 1111011111. Similarly, AC_d = rot (RS(N_d)) = 1001000000. Let us find MS = AC_s+AC_d = 1111011111 + 1001000000 = 0110011111. In the MS, the number of consecutive most significant zeros, LO = 1.

Therefore, AC_{CIR} = 1000000000 and CL_{CIR} = LO+1 = 1+1 = 2.

Position code of CIR can be found by rotating its address code to obtain its recursive sum: RS(CIR) = rot(AC_{CIR}) = 1000000000. Thereafter, the CIR position code is CIR = R+RS(CIR) = 0010011011 + 1000000000 = 1010011011.

5.2.2. *$N_s \to N_d$ Communication Path.* To reach destination node N_d, the source node N_s has to form communication message that contains the following routing information code: RI = CL_{CIR} CL_d AC_d (Fig. 9) where CL_{CIR} is the complemented level value for CIR-node; CL_d is the complemented level value for destination node, N_d, and AC_d is the address code for N_d. All these values can be found using the techniques presented above.

Each communication path from N_s to N_d includes two subpaths $N_s \to$ CIR, and CIR $\to N_d$, where $N_s \to$ CIR is the direction of TOR (towards the root R) and CIR $\to N_d$ is in the direction of TOL (towards the leaves) (Fig. 9b). In moving in the TOR direction from N_s to CIR, a communication message uses the code CL_{CIR} to reach CIR. Namely, in each transit node N of the $N_s \to$ CIR subpath, CL_{CIR} stored in the message is compared with the local CL_N. If $CL_{CIR} \neq CL_N$, communication message is forwarded to the next node in the path. If $CL_{CIR} = CL_N$, the destination node, CIR, of the $N_s \to$ CIR path is reached. Having reached CIR, the message begins the TOL movement (towards the leaves) using AC_d and CL_d portions of the RI code (Fig. 9b). Namely, in each transit node, N, of the CIR $\to N_d$ path, CL_d is compared with the local CL_N. If $CL_d \neq CL_N$, the node N finds the value X_{CL} of the address code, AC_d brought with the message, where CL is the local level value code. If X_{CL} = 1, the message is forwarded to the next right node. If X_{CL} = 0, the message is forwarded to the next left node. If $CL_d = CL_N$, N = N_d, i.e., destination is reached.

Example: For the $N_s \to N_d$ communication path of Fig. 9a, the routing information code, RI, of communication message is shown in Fig. 9a. As seen, CL_{CIR} = 2, CL_d = 5 and AC_d = 1001000000. For the TOR broadcast $N_d \to$ CIR, the message reaches CIR having CL = 2. Thereafter, it starts the TOL transfer, CIR $\to N_d$. For the CIR-node, CL_{CIR} = 2. Therefore, the 2nd bit of the AC_d = 1001 shows what node of the tree follows the CIR-node. Since X_2 = 0, CIR is followed by the left node LN. In the LN node, CL_{LN} = 3 and X_3 = 0 in AC_d = 1001. Thus, LN is followed by the next left node, LN, of the path, etc. The destination is achieved in the node N having CL_N = CL_d.

As follows for the $N_s \to N_d$ communication, the bit size of the routing information code, RI, is upperbounded as follows:

$$BS(RI) \leq n+2\log_2 n = \log_2 K+2\log_2\log_2 K$$

where n is the number of tree levels and K is the number of tree nodes.

Conclusions

In this paper, we discussed efficient communication algorithms for multicomputer networks organized as reconfigurable binary trees.

Organization of optimal node to node communications, $N_s \to N_d$ was considered where N_s is an arbitrary source node N_d is an arbitrary destination node.

Communication between N_s and N_d is organized via the minimal communication path, i.e., the destination node N_d is achieved in such a way that each transit node of the communication path is included only once in the path $N_s \to N_d$. Communication techniques introduced are optimal from both viewpoints: the total bit size of the routing information code, BS(RI), that routes the message among various transit nodes of the communication path and the total time of communication, CD. It is shown that BS(RI) $\leq \log_2 K + 2\log_2\log_2 K$, where K is the total number of nodes in a binary tree. Also, since only one bit of the RI code is responsible for selecting the next node in the path of the two possible alternatives LN or RN, the BS(RI) achieves absolute theoretical minimum, because it is impossible to have 0 bits assigned for such a selection. Further, since each transit node N selects its successor RN or LN in the communication path via logical operation that takes time of one t_d, the total time of $N_s \to N_d$ communication also approaches the theoretical minimum. It should be noted that the importance of these binary trees for computation cannot be underestimated, because of the following unique attributes they possess: (a) reconfiguration into a new binary tree takes the time of one clock period and requires only one control code (bias, B) to be received by all tree nodes requested for reconfiguration; (b) these trees are extremely fault-tolerant, since during one clock period, one can purge out all faulty nodes and form a gracefully degraded tree out of fault-free nodes [21]; (c) as was shown in this paper, communications in such trees are also extremely efficient from both time viewpoint and the bit size of the routing information code.

Therefore, one can utilize broadly unique properties of these reconfigurable binary trees for various computational and control algorithms without paying the price of considerable reconfiguration overhead when no computation can be performed and large size of the address portion of communication message which are the attributes of conventional reconfigurable binary trees.

References

1. A. Despain and D. Patterson, "X-Tree: A Tree Structured Multi-Processor Computer Architecture", Proceedings Fifth Annual Symposium on Computer Architecture, 1978, pp. 144-150.

2. Y. Paxer and M. Bozygit, "Variable Topology Multicomputer", Proceedings of the Second Euromicro Symposium on Microprocessing and Microprogramming, Venice, 1976, pp. 141-149.

3. L. D. Wittie and A. M. van Tilborg, "MICROS, A Distribbuted Operating System for MICRONET, A Reconfigurable Network Computer", IEEE Transactions on Computers, Vol. C-29, December, 1980, pp. 2233-1144.

4. David J. Dewitt and Dina Friedland, "Exploiting Parallelism for the Performance Enhancement of Non-numeric Applications", The AFIPS Conference Proceedings, Vol. 51, 1982, National Computer Conference, pp. 207-216.

5. S. I. Kartashev and S. P. Kartashev, "Problems of Designing Supersystems with Dynamic Architectures", IEEE Transactions on Computers, Vol. C-29, December, 1980, pp. 1114-1132.

6. C. Davis, S. P. Kartashev, and S. I. Kartashev, "Reconfigurable Multicomputer Networks for Very Fast Real-time Applications", 1982 AFIPS Conference Proceedings, Vol. 51, AFIPS Press, pp. 167-185.

7. S. P. Kartashev and S. I. Kartashev, "Reconfiguration of Dynamic Architecture into Multicomputer Networks", Proceedings of the 1981 International Conference on Parallel Processing, August 25-28, 1981, Belleaire, Michigan, pp. 133-141.

8. H. Garcia-Molina, "Elections in a Distributed Computing System", IEEE Transactions on Computers, January, 1982, Vol. C-31, No. 1, pp. 48-60.

9. H. Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database", Department of Computer Science, Stanford University, Stanford, Ca., Rep. STAN-CS-79-744, June, 1979.

10. L. Lamport, "The Implementation of Reliable Distributed Systems", Computer Networks, Vol. 2, pp. 95-114, 1978.

11. B. W. Lampson and H. E. Sturgis, "Crash Recovery in a Distributed Data Storage System", Xerox, PARC, Rep., 1979.

12. D. A. Menasce, G. J. Popeck and R. R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases", ACM Transactions on Database Systems, Vol. 5, pp. 103-138, June, 1980.

13. R. J. McMillen and H. J. Siegel, "Routing Schemes for the Augmented Data Manipulator Network in an MIMD System", IEEE Transactions on Computers, Vol. C-31, December, 1982, pp. 1202-1214.

14. B. Elspas, "The Theory of Autonomous Linear Sequential Networks", IRE Transactions on Circuit Theory, January, 1959, pp. 45-60.

15. N. Zierler, "Linear Recurring Sequencer", J. Siam, 7(1), 1965, pp. 31-48.

16. W. H. Kautz (Ed.), Linear Sequential Switching Circuits, Holden-Day, 1965.

17. S. W. Golomb, Shift Register Sequences, Holden-Day, 1967.

18. T. L. Booth, Sequential Machines and Automata Theory, 1967.

19. S. P. Kartashev, "Theory and Implementation of p-Multiple Sequential Machines", IEEE Transactions on Computers, May, 1974, pp. 500-523.

20. S. P. Kartashev, "State Assignment for Realizing Modular Input-free Sequential Logical Networks Without Inverters", Journal of Computer and System Sciences, Vol. 7, No. 5, October, 1973, 522-542.

21. S. P. Kartashev and S. I. Kartashev, "Reconfigurable Fault-Tolerant Multicomputer Networks", AFIPS Conference Proceedings, Vol. 52, 1983 National Computer Conference, AFIPS Press, pp. 595-610.

SORTING, MERGING, SELECTING, AND FILTERING ON TREE AND PYRAMID MACHINES

(Preliminary version)

Quentin F. Stout
Mathematical Sciences
State University of New York
Binghamton, NY 13901 USA

ABSTRACT: We develop some fundamental algorithms for d-dimensional pyramid computers, where a 1-dimensional pyramid is typically called a tree and a 2-dimensional pyramid is what is commonly meant by a pyramid computer. We give optimal $\theta(n/lg(n))$ algorithms for sorting and merging n items stored in a tree, and show that for higher dimensional pyramids well-known algorithms are optimal. We give a selection algorithm, suitable for pyramids of all dimensions, which runs in less than $n^{**}\varepsilon$ time for any $\varepsilon > 0$. We also consider median filtering of a noisy digitized picture, giving an optimal algorithm whose running time is $\theta(D)$ when using a $D \times D$ window.

1. INTRODUCTION

Sorting, merging, and selection (i.e., finding the k'th item) are fundamental problems, and for almost any machine architecture efficient algorithms have been devised for them. However, for certain tree and pyramid computers we show that when the data is already in the base of the computer the "obvious" algorithms are not always optimal, and we present algorithms superior to any previously published. Some authors [2,3,8,11] have presented optimal sorting algorithms for tree machines in which the data passes through the root, but we are interested in situations where the data is already present and must be rearranged. Such situations arise when a different key is now being used to determine the ordering, or when data can be entered directly into the base. This latter possibility has been raised for pyramid machines devoted to image processing, for which the speed-ups introduced here may be of use. If the data must pass through the apex then all algorithms are at best linear in the number of items, and [2,3,8,11] showed that linear algorithms exist. However, in our situation one can obtain sublinear algorithms for sorting and merging, where the speed-up depends upon the dimension of the pyramid (defined below). For selection we show that the possible speed-up is even more impressive, giving an algorithm which finds, on a pyramid of any dimension, the k'th item among n in $o(n^{**}\varepsilon)$ time for any $\varepsilon > 0$.

Our interest in pyramid machines stems from our belief, and that of many others, that their geometric basis makes them a natural parallel architecture to consider for various database [2,3,11] and image processing [4,5,6,9,12,13,15,16,17,19] applications, applications in which significant parallelism is possible. Pyramids are more attractive than meshes because they have the potential of logarithmic algorithms. Furthermore, the regularity of a pyramid structure makes it feasible to construct such machines with a large number of processing elements. Several tree and pyramid machines are in various stages of design [2,3,8,11,15], and we expect that more advanced machines, with a very large number of processing elements, will be constructed.

Our results are primarily of theoretical use, in that the algorithms are somewhat complicated, but they do at least show that certain problems have solutions which are asymptotically better than was previously thought. Furthermore, our algorithms tend to use the pyramid structure in ways that are different from previously published algorithms, and perhaps such techniques will prove useful in other problems.

Throughout we will use lg to denote log base 2, and θ, Ω, O, and o to denote "order exactly", "order at least", "order at most", and "order strictly smaller than", respectively. Optimal will always mean optimal to within a multiplicative constant.

Partially supported by NSF Grant Number MCS-8301019.

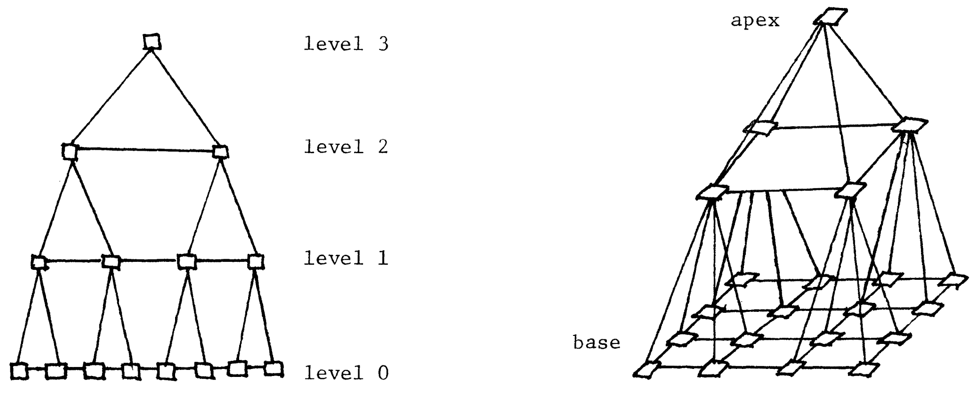
0190-3918/83/0000/0214$01.00 © 1983 IEEE

2. MACHINE MODELS

Several different models have been proposed which might naturally be called "tree" or "pyramid" machines [2,3,4,5,6,8,9,11,15,17,19]. The machines considered here can be viewed as layers of mesh-connected machines, with connections between the layers. To define a pyramid of arbitrary dimension, we first define a mesh-connected computer of arbitrary dimension. Throughout we will use d to indicate the dimension. A d-dimensional mesh-connected computer of size n**d (a d-MCC of size n**d) consists of n**d copies of a single processing element (PE), arranged in a d-dimensional cubic lattice. PEs have indices of the form (I1,...Id), where 0 ≤ Ii ≤ n-1 for 1 ≤ i ≤ d. PEs (I1,...,Id) and (J1,...,Jd) are neighbors if and only if $\sum_{k=1}^{d} | Ik-Jk | = 1$. Each PE (except those on sides) has 2*d neighbors and has a unit-time communication link to each of them.

A d-dimensional pyramid computer of size n**d (a d-PC of size n**d) consists of a d-MCC of size n**d, a d-MCC of size (n/2)**d,..., and a d-MCC of size 1, along with additional communication links specified below. (Note that a d-MCC of size n**d has exactly n**d PEs while a d-PC of size n**d has between n**d and 2*n**d PEs.) The d-MCC of size n**d is called the base of the pyramid, and the d-MCC of size 1 is the apex. The d-MCC of size (n/2**k)**d is at level k, e.g., the base is level 0 and the apex is level lg(n). A PE at position (I1,...,Id) on level k is connected to all 2**d of its sons, which are those processors on level k-1 which are at a position of the form (J1,...,Jd), where each Ji is either 2*Ii or 1+2*Ii. Thus, except for processors along the sides, each PE is connected to 2**d + 1 + 2*d others, namely its 2**d sons on the level below, its parent at the level above, and 2*d neighbors at the same level. Figure 1a shows a 1-PC, typically called a tree machine, and Figure 1b shows a 2-PC, which is often what is meant by a pyramid machine. To date we know of no serious interest in pyramids of higher dimensions, but since our algorithms naturally extend to higher dimensions we have done so. We also note that there is beginning to be some interest in 3-dimensional digitizations [7], and 3-PCs are a natural architecture for such problems.

We assume that the PEs have a fixed number of registers, each of which holds a word of length θ(lg(n)), and all operations take unit time. All communication links are bidirectional, and any PE can send and receive a word along any or all links simultaneously, all taking unit time. We also assume each processor holds its level and its coordinates within that level. If this is not the case, it can easily be performed in θ(lg(n)) time. (Note: all analyses of time assume d is fixed and consider time as a function of n. While some authors [10] also consider d as a parameter, this is made difficult by the fact that a PE in a d-PC is fundamentally different from one in a (d+1)-PC). Earlier pyramid models [4,12,13,19] assumed that the PEs were copies of a finite state automation which was designed independent of n, in which case a single PE could not store the level it was on nor its position within that level. Using finite state automata also drastically changes the nature of



A 1-PC of size 8
a)

A 2-PC of size 16
b)

Figure 1

sorting, merging, and selection since there can only be a fixed number of keys, independent of n .

3. SORTING AND MERGING

[2,3,8,11] considered the problem of using machines similar to a 1-PC to perform selection, insertion, deletion, and retrieval operations. Requests arrive at the apex at unit time intervals and are performed on-line, although there is a logarithmic delay between the time a request arrives and its answer appears. This approach sorts n items in $\Theta(n)$ time, and is optimal if all items are required to pass through the apex. However, we are interested in situations where the data is already in the pyramid, stored one item per PE in the base. For example, in a d-PC of size n**d we can sort n**d items in $\Theta(n)$ time by ignoring everything except the base and using the d-MCC sorting algorithm in [18]. This compares quite favorably with the Ω(n**d) time required if all items pass through the apex, and it is natural to ask if we can do even better by utilizing the rest of the pyramid. We prove that further speed-up is possible if and only if d=1 .

Suppose we have a 2-PC of size n**2 sitting "naturally" in 3-dimensional space, by which we mean the layers are all square grids parallel to each other, with each parent above the middle of its four offspring. If we consider a plane cutting perpendicular to both the base and one of its edges, and passing just to one side of the apex, we see that the plane will cut n wires at level 0 , n/2 wires at level 1,...,2 wires at level lg(n)-1 , and 2 wires connecting the apex to level lg(n)-1 , cutting a total of 2n wires. To sort n**2 items stored one per PE in the base it may be that all of them must pass through the plane, giving a worst case time of $\Omega(n)$. Furthermore, on average half of the items must pass through the plane, giving an expected time of $\Omega(n)$ also. Since one can sort in $\Theta(n)$ time using only the base, we see that the rest of the pyramid is of no significant use. This argument holds whenever $d \geq 2$.

However, if we consider a 1-PC of size n and cut it by a line slightly off-center, the line will cut lg(n)+1 wires. This gives a Ω(n/lg(n)) lower bound for the worst case and expected case sorting times, a bound which is unattainable using the base alone.

Theorem 1 The algorithm outlined below for sorting on a 1-PC of size n has a worst case and expected case time of Θ(n/lg(n)) , and is thus within a constant multiple of optimal.

Our algorithm is a simple merge sort, utilizing the fact that the two sons of the apex can be viewed as the apexes of disjoint subpyramids.

> To sort on a 1-PC of size n , $n \geq 1$, sort each half (separately and in parallel), and merge.

If S(n) denotes the worst case sorting time, we have

S(1) = 0
S(n) = S(n/2) + M(n)   n > 1

where M(n) is the worst case time to merge two runs in a 1-PC of size n . We see that

$$S(n) = \sum_{k=0}^{lg(n)} M(n/2**k)$$

In Theorem 2 below we show that M(n) = Θ(n/lg(n)) , which proves Theorem 1.

MERGING

Suppose we have a run R1 of items in processors 0..k of the base, and a run R2 of items in processors (k+1)..(n-1) . We merge these into a simple run as follows:

1. Find the median of all the items (since n is even we will just use the n/2'th item).
2. There are 4 subruns to consider: those items in R1 less than or equal to the median (called subrun S1), those items in R1 greater than the median (S2) , those items in R2 less than or equal to the median (S3) , and those items in R2 greater than the median (S4) . S1 and S4 stay in place, S3 moves behind S1 , and S2 moves in front of S4 . Notice S1 and S3 together hold half of the items, as do S2 and S4 .
3. Merge within each half (separately and in parallel).

Theorem 2 The above algorithm for merging on a 1-PC of size n has a worst case time of Θ(n/lg(n)) , and hence is within a constant multiple of optimal.

Proof: As for sorting, the worst case time must be Ω(n/lg(n)) . To show that this is attained we analyze the time spent in each step. Step 1 can be done by a simple binary search. First the median of R1 is sent up to the apex and back down to the base, at which time each processor whose item is less than or equal to this sends up a 1 . These are summed, after which it is known if the value is too high or low. Then the median of R2 is sent up, then an appropriate quartile point of R1 , then a quartile point of R2 , etc. There are at most $\theta(lg(n))$ probes, each taking $\theta(lg(n))$ time, for a total of $\theta(lg(n)**2)$ time for step 1.

It is step 2 which takes most of the time. It can be reduced to two applications of the following problem: how fast can a run be moved to its destination? To do this rapidly we divide the base into $M = \lfloor lg(n)/2 \rfloor$ blocks. PEs $0..\lfloor n/M \rfloor-1$ are in block 1, $\lfloor n/M \rfloor..2*\lfloor n/M \rfloor-1$ are in block 2, etc. We define a sequence of processors G1,...,GM by taking G1 to be the apex and G(i+1) as Gi's right son. For each i we construct a path Pi connecting Gi to the leftmost processor in block i , with the property that all paths are disjoint. The paths are constructed recursively, with P1 being the leftmost edge of the pyramid. For i > 1, Pi is construced from the base upwards, at each processor going upwards if it does not run into P(i-1) , and otherwise going right. (See Figure 2.) It can be shown that the longest path has length o(n/lg(n)) , and in o(n/lg(n)) time each processor can decide which, if any, path it is on and which of the processors it is connected to are on the same path.

We notice that when a run needs to move, for any block i containing some items in the run there is a j such that either all of those items are to be moved to block j , or else some go to block j and the others go to block j+1 . We call block j the goal of block i , and move all the run's items from block i to block j . If some of these belong to block j+1 then we move them along the base from block j to block j+1 . We construct a path Qi from the leftmost PE in block i to the leftmost PE in block j as follows: let k=min(i,j) . Qi follows Pi up to level k , then moves sideways to meet Pj , and then goes down Pj . Since any block is the goal of at most two blocks, it is easy to see that any procesor is in at most 3 of the Q paths. An item starting in block i moves left until it reaches the start of Qi , then follows Qi , and then moves right in block j and also in block j+1 if that is its destination. The total distance is O(n/lg(n)) , and since no procesor is in more than 3 paths the data movement can be arranged so that the worst case time for step 2 is $\theta(n/lg(n))$.

Therefore M(n) , the worst case time to merge two runs in a 1-PC of size n , satisfies the equations:

$$M(1) = 0$$
$$M(n) \le A*lg(n)**2 + B*n/lg(n) + M(n/2) \qquad n > 1$$

which gives $M(n) = \theta(n/lg(n))$. This completes the proof of Theorem 2.

4. SELECTION

In this section we consider the problem of finding the k'th smallest item. To simplify notation we will discuss only 1-dimensional pyramids (i.e., trees), but all of our results hold for pyramids of higher dimensions. We will also simplify discussion by assuming that all the items have distinct keys.

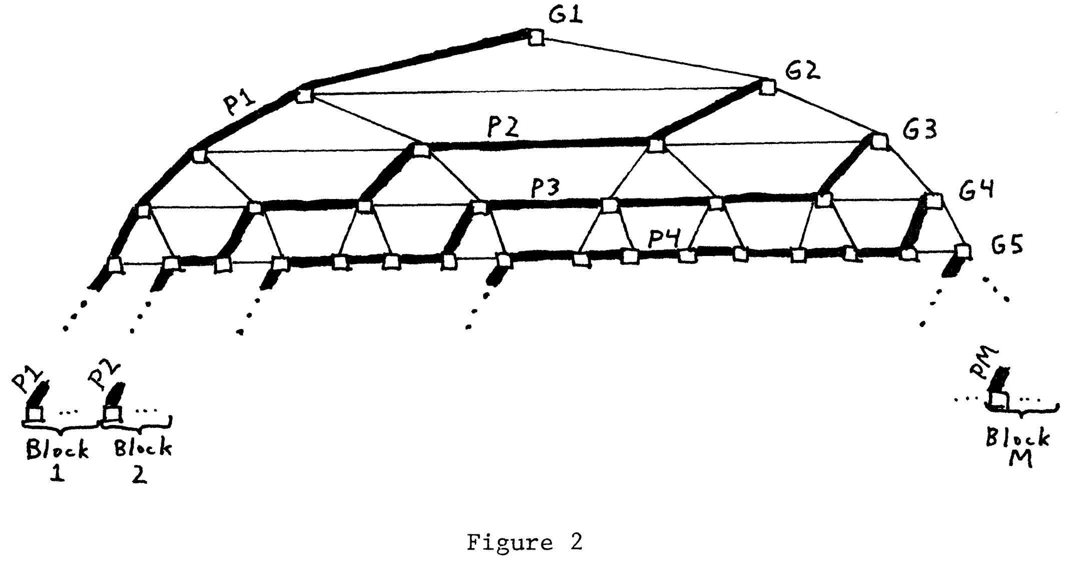


Figure 2

The easiest selection problem is $k=1$, which can be solved by having each base processor pass up its item, and each higher processor pass up the smallest item received. This will take $\theta(lg(n))$ time, and Tanimoto [16] observed that if the apex repeatedly discards the smallest item and asks the son which passed up that item to pass up another then one can find the k'th smallest item in $\theta(lg(n)+k)$ time, which in the worst case requires $\theta(n)$ time to find the median. We can do better by sorting the base and passing up the middle item, taking $\theta(n/lg(n))$ time. For $d > 1$ the differences are more dramatic, going from $\theta(n**d)$ down to $\theta(n)$. (Tanimoto had noted this improvement for $d > 1$.) By abandoning sorting we are able to do far better, finding the k'th item in $o(n**\varepsilon)$ time for any $\varepsilon > 0$.

To solve the selection problem we need to solve the weighted selection problem, in which we are given k and N pairs (vi,wi) , where vi is the value of the pair and wi is its positive integral weight. The vi are all distinct, and we want to find the vI such that

$$\sum\{wi:vi\le vI\} \ge k \text{ and } \sum\{wi:vi< vI\} < k .$$

Each of the original items in the base has a weight of 1, and intermediate calculations produce items with greater weights.

Initially each item is "active", and may later become inactive when it is known that it cannot be the answer. The algorithm is:

```
if n=1 then the item is the answer
else if n=2 then both items are
                 sent to the apex,
                 which determines
                 the answer
else repeat
  Stage 1: Each processor at level
    lg(n)/2 takes as its value the
    median of the active items beneath
    it, and as its weight the number
    of active items beneath it.

  Stage 2: The apex finds the
    weighted median of the items found
    in the previous stage, call this
    W , and transmits it to all pro-
    cessors in the base.

  Verification: Each base processor
    sends up a 1 if its item is less
    than or equal to W . These 1's
    are summed on their way up to the
    apex, which determines if W is
    the k'th item or is too large or
    too small. In the last two cases
    it sends down a message which
    deactivates all items as large as
    W , if W is too large, or all
    items as small as W if it is too
    small.
until the k'th item is found.
```

(This algorithm is similar to one in [14] , which was for a mesh-connected computer with broadcasting.) An important feature of this algorithm is the fact that on each iteration at least 1/4 of all active items become inactive, and hence at most $\log_{4/3}(n)$ iterations are required. To see why the 1/4 appears, suppose on some iteration W was too high. The weights guarantee that at least 1/2 of all active items are beneath a processor of height $lg(n)/2$ which, during stage 1, picked an item at least as large as W . For each such processor on the middle level, at least half of the items beneath it are as large as W .

To see how much time this algorithm takes it is easiest to work with the height of the tree (ie, $lg(n)$) instead of its width. If $T(h)$ denotes the worst case time on a 1-PC of height h , then

$$T(0) = 0$$

$$T(h) \le h*\log_{4/3}(2) * [2*T(h/2) + B*h] \qquad h > 0$$

The solution of this is $O([C*h]**[lg(h/2])$, where $C = 2*[\log_{4/3}(2)]**2$, which is $o(n**\varepsilon)$ for any $\varepsilon > 0$.

<u>Theorem 3</u> In $o(n**\varepsilon)$ time, for any $\varepsilon > 0$, the above algorithm finds the k'th item among n items stored in the base of a 1-PC of size n .

We make no claims about the optimality of our algorithm. In fact, if one uses $lg(lg(n))/2$ stages, instead of the 2 used above, each determining the weighted median of items determined below in the previous stage, then one can do selection in $o(lg(n)**[lg(lg(n))/lg(lg(lg(n)))])$ time. Further fine-tuning of this approach does not seem very interesting. We conjecture that $\theta(lg(n))$ is unattainablę as a worst case time for selection, but have been unable to prove this. There is also the question of expected case time, and we do not even know the expected time of our algorithm.

5. MEDIAN FILTERING

In this section we consider the problem of median filtering of a noisy digitized picture stored one pixel per PE in the base of a 2-PC . In median filtering the idea is to replace each pixel with the median of the pixel values in a $D \times D$ square, D odd, whose center is the original pixel. We call this square a window. Median filtering is a well-known technique (see, for example, [20] and the references therein) and is applicable to data of any dimension. Our reason for concentrating on the 2-dimensional problem is a paper of Tanimoto [16] which gives a simple algorithm. Our goal is to minimize the running time as a function of D , where we show below how to eliminate any dependence on the size of the 2-PC . We give an optimal algorithm, but we must mention that our asymptotic result is misleading since in practice D is quite small.

Tanimoto [16] noted that any dependence on the size of the pyramid could be eliminated by partitioning the image into a set of nonoverlapping regions of area $\theta(D**2)$. Processors at height $\lceil lg(D) \rceil$ are viewed as the apex of a subpyramid in which filtering is performed, with each subpyramid responsible for computing the new value for each pixel within it. Windows around pixels in one subpyramid can include pixels from an adjacent subpyramid, so first all necessary information is exchanged between adjacent subpyramids. This exchange can easily be done in $\theta(D)$ time.

Within each subpyramid Tanimoto computed the new pixel values one at a time. The total time for this method is $\theta(D**2 * T(D))$, where $T(D)$ is the time needed to compute the median in a 2-PC of size $D**2$. The procedure he first mentions finds the median in $\theta(D**2)$ time, giving a total time of $\theta(D**4)$. He also notes that by just sorting in the base the median can be found in $\theta(D)$ time, giving $\theta(D**3)$ total time. By instead using the selection algorithm of the preceeding section one can perform median filtering in $o(D**[2*\varepsilon])$ time for all $\varepsilon > 0$.

However, Tanimoto's approach is not very efficient as it ignores the fact that calculating the median at one point is closely related to calculating the median at any adjacent point. By exploiting this we are able to give an algorithm taking $\theta(D)$ time. A straightforward data movement analysis, as in section 3, shows that this is an optimal worst-case time. It seems that this is also an optimal expected-case time.

Theorem 4 In a 2-PC or a 2-MCC , using a $D \times D$ window, median filtering can be accomplished in $\theta(D)$ time.

The theorem is proved by the following algorithm, which is only briefly sketched. We use only the base of a 2-PC , partitioned into $D \times D$ blocks . Within each block each processor will determine its new pixel value. Because windows of pixels in the block can fall outside the block, the block needs to know all pixel values in a $(2D-1) \times (2D-1)$ square called a neighborhood. Also, for reasons that will be explained later, the block actually must simulate the actions of processors lying in a $(3D-2) \times (3D-2)$ square called a region. (See Figure 3.) Each processor in the block simulates the actions of a square of 9 processors from the region, with the algorithm being described as if all of the processors in the region are assisting the block.

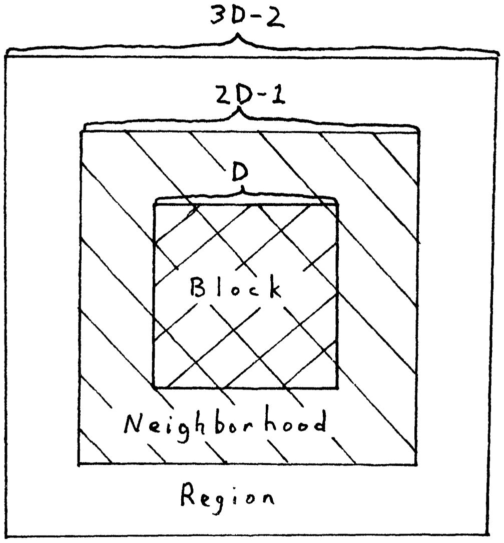


Figure 3

Via sorting, each neighborhood processor determines the order position of its pixel value, e.g., a processor may determine that its pixel value is the fifth smallest in the neighborhood. Each neighborhood processor x sends this order position to the form processors c1,c2,c3,c4 indicated in Figure 4. These four processors are the corners of the square of all processors whose windows include x . Notice that the region has been chosen so that if x is in the neighborhood then c1,c2,c3, and c4 all lie within the region.

Using these corners, given any order interval there is a "spreading wave" process taking $\theta(D)$ time, after which each processor in the block will know how many pixels values in its window fall in the order interval. There are $4D**2 - 2D + 1$

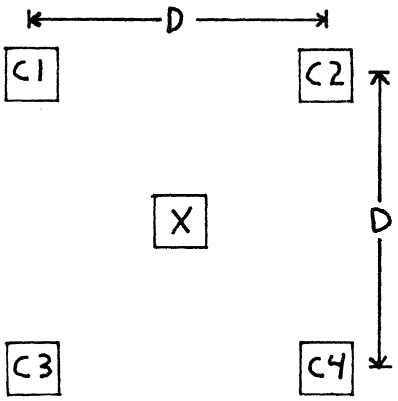


Figure 4

pixels in the neighborhood, so by using 4D order intervals each has at most D values. We repeat the spreading wave process 4D times, pipelining the waves so that it takes only θ(D) time. When finished, each processor in the block knows which order interval contains the median of its window. The processor also knows a relative description of which pixel value in the order interval is the median of its window, e.g., the processor may know that in the appropriate order interval it is looking for the third smallest pixel value belonging to its window. A final combination of sorting and searching, also taking θ(D) time, finishes the algorithm.

The algorithm sketched above can be modified to provide an optimal θ(D/log(D)) median filtering algorithms using a window of size D , for 1-dimensional data, stored in the base of a 1-PC . It can also be adapted to d-PCs and d-MCCs , d ≥ 2 , using windows of size D**d , but the adaptions give θ(D**(d/2)) algorithms. The only lower bound known to the author is θ(D) , so it seems that optimal algorithms for higher dimensions will require different techniques.

6. CONCLUSION

Our results can be rearranged to allow a better comparison between pyramids of different dimensions. If we have a d-PC of size n , with one item per base processor, then these n items can be sorted in θ(n/lg(n)) time if d=1 , and θ(n**(1/d)) time if d > 1 . (This holds for both worst case and expected case times.) The 1-PC algorithm is new, while the others are well-known, and all are optimal to within a constant multiple. Our 1-PC sorting algorithm depended on a new optimal merging algorithm. While there was no time to explore further here, the 1-PC sorting algorithm gives several other optimal, θ(n/log(n)) algorithms.

We also considered selection, providing an algorithm much faster than previous ones. Previous algorithms utilized sorting, which does not fit the pyramid structure very well. Pyramids provide logarithmic paths between any two processors, but if there is too much data movement, such as occurs with sorting, then the apex becomes a bottleneck. By reducing the amount of data being moved we reduced the time to o(n**ε) for any ε > 0 . Thus seletion is faster than sorting on serial computers [1], mesh-connected computers with broadcasting [14] , and pyramids.

Our selection algorithm also gives a faster median filtering algorithm which, when using a D × D filtering window, takes o(D**[2+ε]) time for any ε > 0 . However, by making better use of the fact that windows of adjacent processors overlap extensively we are able to give an optimal θ(D) algorithm. In a future paper the author will consider a variety of windowing operations on data stored in mesh-connected computers and in pyramid computers.

REFERENCES

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Algorithms, Addison-Wesley, 1974.

2. M.J. Atallah and S.R. Kosaraju, A generalized dictionary machine for VLSI, Dept. E.E. Comp. Sci., Johns Hopkins Univ.

3. J.L. Bentley and H.T. Kung, Two papers on a tree-structured parallel computer, Dept. Comp. Sci., Carnegie-Mellon Univ., Rep. CMU-CS-79-142, 1979.

4. C.R. Dyer, A fast parallel algorithm for the closest pair problem, Info. Proc. Let. 11 (1980), 49-52.

5. C.R. Dyer, A quadtree machine for parallel image processing, Dept. Comp. Sci. Univ. Illinois at Chicago Circle, Tech. Rep. KSL 51, 1981.

6. A.R. Hanson and E.M. Riseman, Preprocessing cones: a computational structure for scene analysis, Univ. Mass., COINS Tech. Rep. 74C-7, 1974.

7. C.E. Kim and A. Rosenfeld, Convex digital solids, IEEE Trans. P.A.M.I. 4 (1982), 612-618.

8. C.E. Leiserson, Systolic priority queues, Dept. Comp. Sci., Carnegie-Mellon Univ., Rep. CMU-CS-79-115, 1979.

9. R. Miller, An efficient data movement technique for the pyramid computer, to appear.

10. D. Nassimi and S. Sahni, Finding connected components and connected ones on a mesh-connected parallel computer, SIAM J. Comput. 9 (1980), 744-757.

11. T.A. Ottmann, A.L. Rosenberg, and L.J. Stockmeyer, A dictionary machine (for VLSI), IEEE Trans. Comput. 31 (1982), 892-897.

12. B. Sakoda, Parallel construction of polygonal boundaries from given vertices on a raster, Dept. Comp. Sci., Penn. State Univ., Tech. Rep. CS81 1-21, 1981.

13. Q.F. Stout, Drawing straight lines with a pyramid cellular automation, Info. Proc. Let. 15 (1982), 233-237.

14. Q.F. Stout, Mesh-connected computers with broadcasting, IEEE Trans. Comp., to appear.

15. S.L. Tanimoto, Towards hierarchical cellular logic: design considerations for pyramid machines, Dept. Comp. Sci., Univ. Washington, Tech. Rep. 81-02-01, 1981.

16. S.L. Tanimoto, Sorting, histogramming, and other statistical operations on a pyramid machine, Dept. Comp. Sci., Univ. Washington, Tech. Rep. 82-08-02, 1982.

17. S.L. Tanimoto and A. Klinger (eds.), Structured Computer Vision: Machine Perception Through Hierarchical Computation Structures, Academic Press, 1980.

18. C.D. Thompson and H.T. Kung, Sorting on a mesh-connected parallel computer, Comm. A.C.M. 20 (1977), 263-270.

19. L. Uhr, Layered "recognition cone" networks that reprocess, classify, and describe, IEEE Trans. Comp. 21 (1972), 758-768.

20. Two Dimensional Digital Signal Processing II, Transforms and Median Filters, Springer-Verlag, 1981, particularly the articles by B.I. Justusson and S.G. Tyan.

Omni-sort: A Versatile Data Processing Operation for VLSI[a]

Ching C. Hsiao
Western Electric
Engineering Research Center
P. O. Box 900
Princeton, New Jersey

Lawrence Snyder
Purdue University
Dept. of Computer Sciences
West Lafayette, Indiana

ABSTRACT -- Omni-sort is a versatile operation proposed to perform basic database operations such as union, intersection, difference, duplicate-removal, and sorting. We present adaptation schemes to implement the omni-sort by modifying parallel sorting algorithms. While the modification cost is small, the initial expenses of design and test development is offset by the increased usage. In the algorithmic point of view, the omni-sort also provides potentially much faster algorithms for those data processing operations than before.

1. INTRODUCTION

Since sorting is one of the most important operations in data processing, high-speed parallel sorting methods are very desirable. In the last fifteen years, many parallel algorithms have been proposed and developed for sorting. Because of the advances in VLSI technologies, highly parallel computing systems consisting of hundreds of thousands of processing cells are feasible. With the technologies, those sorting algorithms may be realized as algorithmically specialized processors [7] or as parallel programs in general-purpose computing systems. In both cases, initial cost of design and test development is so high that it had better be offset by mass production or heavy usage.

In this paper we propose a powerful operation called omni-sort[b] which can selectively perform multiple data processing functions. The functions performed by the omni-sort are five basic data processing operations: union, intersection, difference, duplicate-removal, and sorting [10]. Omni-sort can be implemented by slightly modifying the already developed parallel sorting algorithms. We do not distinguish sorting algorithms to be implemented as specialized processors from those to be programmed in general-purpose computing systems, since the results shown are applicable to both.

Here we also present schemes to extend sorting algorithms to implement the versatile omni-sort. The key point is to compose elegantly the sorting capability and some minimal, extra computation capabilities. Using the schemes, sorting algorithms can be modified to perform the omni-sort such that the modification overhead is small and the performance overhead is negligible. Based on some sorting algorithms, the omni-sort offers faster or even optimal algorithms for duplicate-removal, union, intersection, and difference.

2. THE OMNI-SORT OPERATION

In this section we first give definitions of the five data processing operations and the omni-sort. We then show that any sorting algorithm can be extended to perform the omni-sort. A step-by-step synthesis is given to demonstrate what and how extra capabilities are attached to a bare sorting algorithm. The added-on capabilities are endeavored to be easy for implementation as well as minimal in introducing overhead.

2.1 DEFINITION

Sorting, duplicate-removal, union, intersection, and difference are basic operations in relational database processing [10]. One can find their conventional definitions in many database textbooks. However, among the five operations, four are defined differently here from their sequential counterparts except sorting. Omni-sort is a new operation defined to perform all of the five operations selectively.

Let $X = \{x_1, x_2, .., x_n\}$ and $Y = \{y_1, y_2, .., y_m\}$ be *multisets*. By multiset it is meant that there may be duplicate items. This is in contrast to the mathematical definition of a *set* in which all the elements are distinguishable. From k equivalent data items in a multiset, $k-1$ items are arbitrarily chosen to be duplicates. Throughout this paper we will assume that a sorted sequence is always in ascending order. The formal definitions of these operations are given as below.

$sort(X)$ to re-order the sequence $x_1, x_2, \cdots, x_n$ such that $x_{p(1)} \leq x_{p(2)} \leq \cdots \leq x_{p(i)} \leq \cdots \leq x_{p(n)}$, where $p(1)p(2) \cdots p(n)$ is called a permutation.

$dup\_rm(X)$ to determine the bit stream $q(1)q(2) \cdots q(i) \cdots q(n)$ such that if $q(i) = 1$ then x_i is a duplicate.

$union(X,Y)$ to compute the union set of X and Y, i.e. $dup\_rm(X) \bigcup dup\_rm(Y)$.

$inter(X,Y)$ to compute the intersection set of X and Y, i.e. $dup\_rm(X) \bigcap dup\_rm(Y)$.

$differ(X,Y)$ to compute the difference set of X and Y, i.e. $dup\_rm(X) - dup\_rm(Y)$.

$omni\text{-}sort(f,X,Y)$ to perform one of the above five operations which is selected and controlled by the argument f.

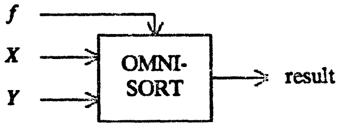


Figure 1. Controlling the functions performed by the omni-sort.

The duplicate-removal operation defined above simply identifies the duplicate items, or marks certain items as duplicates. This definition is reasonable for parallel processing [4]. If actually eliminating the duplicates is required, an additional operation of packing and segregation [6] can be used to separate marked and unmarked data items. Alternatively, marked data items can be filtered off while outputting the sequence.

The operations union, intersection, and difference defined above are more general than standard set operations. They are relaxed from the restriction that operands must be sets of distinguishable elements. This generalization has its practical merit in relational database processing. Multisets are artifacts of relational operations like *projection* and *concatenation*. Evidently

[a] The research described herein was conducted as part of the Blue CHiP project at Purdue University, West Lafayette, Indiana while the first author was a graduate student there. Funding was provided in part by the Office of Naval Research under Contract N00014-80-K-0816 and Contract N00014-81-K-0360, Special Research Opportunities Program Task SR0-100.

[b] Omni-sort was previously called POP-SORT (Primitive-OPeration SORT) in [2].

0190-3918/83/0000/0222$01.00 © 1983 IEEE

many query languages (SEQUAL, QUEL, and QBE) provide operations for working with multisets [10]. It is not unusual that duplicate-removal and a set operation need to be executed subsequently. With the more powerful multiset operations, this can be done by invoking only one single operation. Integrating two operations together gives opportunity to eliminate the communication needed between operations.

2.2 A GENERAL ADAPTATION SCHEME

Assume that a multiset X is sorted in ascending order by some highly parallel sorting method such that $x_1 \leq x_2 \leq .. \leq x_n$.

Definition 1. *Marking-minus* is a parallel process that marks certain elements in the sorted multiset X as "minus". It compares all the pairs of x_i and x_{i+1} for $1 \leq i < n$; if equal, it marks x_i as x_i^-.

A process called *marking-plus* can be defined similarly on a sorted multiset Y. Either marking-minus or marking-plus can detect all the duplicate elements in a sorted multiset. For the union operation, the distinction between its two operands is not important and it can be implemented as duplicate-removal.

To unify all the five operations, however, there is a problem of incompatibility on the number of operands among them; distinction of operands is necessary to perform intersection and difference. To resolve the incompatibility, we propose that tags be attached to data items so that X-elements can always be distinguishable from Y-elements. To perform sorting on two multiset operands, the operands are simply put together and sorted as if the tags were transparent. The use of tags is also useful to extend the sorting algorithm to have a feature defined below. This feature is crucial to the easy adaptation of sorting algorithms to perform the omni-sort.

Definition 2. Given two multisets X and Y and a sorting algorithm to sort them together, the sorting algorithm is called *quasi-stable* if x precedes (follows) y in the final sorted sequence for all $x = y$, $x \in X$ and $y \in Y$.

We summarize the general procedure to extend any sorting algorithm to implement the omni-sort as follows. First, two different tags are attached to X- and Y-elements and a quasi-stable sorting algorithm is executed on $X+Y$. The marking-plus process then detects all the duplicates in Y and the marking-minus detects duplicates in X. Notice that the result sequence now has the following pattern:

$$.. x^- \; . \; x^- x \; y \; y^+ \; . \; y^+ \; ..$$
$$\text{where } x = y \text{ and } x \in X,\; y \in Y.$$

All the equivalent data items, from both X and Y, must be marked off except possibly pairs of x and y as shown above. The unmarked data x and y must be "neighbors" and only x is considered a candidate of valid data in the final result. Then some more neighbor-communication and local processing is enough to implement the operations intersection and difference.

2.3 IMPLEMENTATION

According to Section 2.2, a general omni-sort algorithm may consist of four processing stages:

(1) Initialization - Each data item is attached a tag to identify which operand it belongs to.

(2) Sorting - A quasi-stable sorting algorithm is executed.

(3) Marking - The two marking processes are performed on separate operands.

(4) Completion - Data tags may be further modified to reflect the final result of the intended operation.

To implement the omni-sort, we use a two-bit flag for each data item. The flag bits are appended to each data item as the least significant bits. Initial values of X-tag and Y-tag are carefully designated to be (0,1) and (1,0) respectively. With the meticulous initialization of the flag bits, any sorting algorithm is quasi-stable! For sorting or duplicate-removal, we arbitrarily choose X-tag for its single multiset operand. Similarly, the two multisets of the union operation can be put together and treated as X operand.

Let $a_k..a_0a_{-1}a_{-2}$ be the bit representation of data item a where $a_{-1}a_{-2}$ are the flag bits. The bit representation $b_k..b_0b_{-1}b_{-2}$ is defined similarly for the data item b. Suppose that a and b are the neighboring data items, with b toward the larger end of the sorted sequence. To implement the marking-minus or marking-plus, all the pairs of a and b are compared and the flag bits are modified as following:

Marking-minus: if $a = b$ and $a_{-1}a_{-2} = (0,1)$
then $a_{-1}a_{-2} = (0,0)$;

Marking-plus: if $a = b$ and $b_{-1}b_{-2} = (1,0)$
then $b_{-1}b_{-2} = (1,1)$;

Now all the data with the flag bits (0,1) constitute the final result of the operation sorting, duplicate-removal, or union. However data items with the flag bits (0,1) are just candidates for the result of the operation intersection or difference. To complete the operation intersection or difference, the program segments shown below further screens those data with the flag bits (0,1) to produce the correct result.

```
/* intersection */
if $a_k..a_0 \neq b_k..b_0$ and $a_{-1}a_{-2} = (0,1)$ and $b_{-1}b_{-2} = (1,0)$
    then $a_{-1}a_{-2} = (0,0)$;

/* difference */
if $a_k..a_0 = b_k..b_0$ and $a_{-1}a_{-2} = (0,1)$ and $b_{-1}b_{-2} = (1,0)$
    then $a_{-1}a_{-2} = (0,0)$;
```

2.4 OVERHEAD

The modification overhead of the general adaptation scheme to extend a sorting algorithm to perform the omni-sort is the following:

- two flag bits for each data item,
- some bit manipulation capability of the flag bits,
- maybe some additional capability for performing the local pair-wise comparison, and
- maybe the linear interconnection between processing cells for the neighbor-communication.

The last two items are not necessarily overhead. Comparison function is usually needed to implement sorting anyway and linear interconnection of processing cells is available for many sorting systems. The first two items affect only local circuit; they never create significant overhead since it is the interconnection circuit for communication that occupies most of the chip area.

The time overhead is even more exciting. The initialization stage takes only constant time. The marking stage and the completion stage also take only constant time if the interconnection is available for the neighbor-communication. This time overhead is negligible as compared with the minimum requirement of $O(\log_2(n+m))$ time for sorting $X+Y$ [11].

3. EXAMPLES

In this section we discuss more specific adaptation schemes to turn sorting algorithms into an omni-sorter. Two families of sorting methods (enumeration sort and merge sort) and one type of linear-time sorting systems are considered. From the two sorting methods, we can derive faster-than-ever omni-sort algorithms for duplicate-removal and the three multiset operations. The linear-time sorting systems, which emphasize overlapping sequential I/O time and sorting time, provide a most feasible and practical solution to high-speed hardware sorter.

The general adaptation scheme shown in the previous section implies four processing stages, with the marking stage distinctly separated from the sorting stage. We call this the open-form scheme. For the two families of sorting methods, closed-form schemes requiring less overhead than the open-form scheme are presented. The close-form schemes completely integrate the marking stage into the sorting stage by primarily using a new comparison function in the sorting stage.

3.1 ENUMERATION SORT

In the enumeration sorting methods, each data is compared with all the others [3]. In other words, they perform at least the $\frac{1}{2}n(n-1)$ comparisons to sort a sequence of n data items. Usually enumeration sorting methods consist of two computation phases.

- Rank computation - From the result of the $\frac{1}{2}n(n-1)$ comparisons the ranks of data items are determined.
- Data rearrangement - Each data is moved to its final position according to its rank.

Of course the open-form scheme can be applied to adapt any enumeration sorting algorithm to become omni-sort algorithm. It might be easier for enumeration sorting systems to incorporate the marking capabilities into the sorting stage. The following program describes how easily the marking capabilities can be embedded in the rank computation phase.

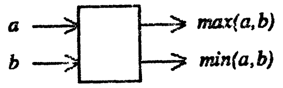
```
1.  for $i = 1$ to $n$ do $rank_i := 1$;      /* the smallest */
2.  for $i = 1$ to $n$ do
      for $j = i+1$ to $n$ do
3.      if $x_i > x_j$ then $rank_i := rank_i + 1$
          else $rank_j := rank_j + 1$;
4.      if $x_i = x_j$ then
          if $a_{-1}a_{-2} = (0,1)$          /* $a = x_i$ */
            then $a_{-1}a_{-2} = (0,0)$;   /* marking-minus */
          if $a_{-1}a_{-2} = (1,0)$          /* $b = x_j$ */
            then $b_{-1}b_{-2} = (1,1)$;   /* marking-plus */
```

Assume that all the data are tagged with the flag bits (0,1) or (1,0) appropriately in the initialization stage. Step 4 of the above program essentially performs the two marking functions. In step 4 the modification of flag bits is carefully programmed to be consistent with the rank computation in step 3.

In [5] a sorter based on the enumeration method was shown. The sorter is optimal in the sense of reaching the lower bound time complexity $O(\log_2(n+m))$ [11]. Using this kind of sorter as the base for omni-sort, all the five operations would be executed in optimal time.

3.2 MERGE SORT

In the merge sorting methods, smaller sorted sub-sequences are merged into larger ones and the merging process is repeated until there is only one final sorted sequence left. To merge sub-sequences, pair-wise comparisons are performed in parallel. A standard pair-wise comparison is usually defined by two operators, *max* and *min*, as shown in the following [3].



$$\text{if } a \geq b \text{ then } \{\ max(a,b) := a;\ min(a,b) := b\ \}$$
$$\text{else } \{\ max(a,b) := b;\ min(a,b) := a\ \}$$

In Definition 3, we define a new pair-wise comparison function, compare-and-mark, which combines the two marking mechanisms with the above comparison function. By using the the compare-and-mark function in a merge-oriented sorting algorithm, we can prove the result described in Theorem 1 by induction [2].

Definition 3. *Compare-and-mark* is a comparison operation that manipulates the flag bits as below and also performs the standard comparison function.

$$\text{if } a = b \text{ and } a_{-1}a_{-2} = (0,1) \text{ then } a_{-1}a_{-2} := (0,0);$$
$$\text{if } a = b \text{ and } a_{-1}a_{-2} = (1,0) \text{ then } a_{-1}a_{-2} := (1,1);$$

Theorem 1. Using the compare-and-mark operation, any merge sorting algorithm can sort in the quasi-stable manner and perform the marking-plus and marking-minus functions correctly.

The marking processes performed by the compare-and-mark function in a merge-oriented sorting algorithm are idempotent; they are depicted as a state diagram in Figure 2. Once data items reach the state (0,0) or (1,1), they remain in that state.

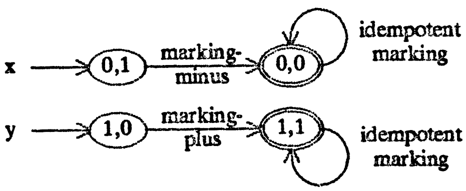


Figure 2. State diagram for the marking processes.

Without using sorting, algorithms to perform the operations such as duplicate-removal, union, intersection, and difference were proposed on systolic arrays [4] and a tree machine [8]. Those algorithms all require linear time $O(n+m)$. Batcher's bitonic merge sort [1] were adapted to several computing systems with different interconnection patterns. Even with the mesh interconnection [9], omni-sort promises sublinear time performance $O(\sqrt{n+m})$ for those operations. If a large bandwidth of I/O is provided then omni-sort algorithms are definitely preferred to those in [4,8].

3.3 LINEAR-TIME SORTER

Here the linear-time sorter refers to those which inputs data sequentially and produces a sorted sequence while outputting it. The open-form scheme can be applied to adapt the linear-time sorter to the omni-sorter. First, data items are properly tagged and fed into the linear-time sorter. Then the computation in the marking and the completion stages is performed while outputting the sequence. This can be easily done by adding to the sorter a special post-processor.

4. OTHER APPLICATIONS

In addition to its power of performing multiple functions, the omni-sort has other useful applications. We describe briefly in this section its two important applications in relational database processing. One is to pre-condition data items for carrying out the equi-join operation efficiently. The other is to process a class of relational queries involving only the five operations in a novel way. Readers interested may refer to [2] for more details.

Sorting has been very useful in improving the performance of the join operations (especially the equi-join operation.) A common approach is to presort database relations A and B on attributes T if they are to be joined on the attributes T. Then a join algorithm can take advantage of the well ordered sequences in A and B. Now, since omni-sort has the nice feature of being quasi-stable, the relations A and B can be sorted together on the attributes T. All the database tuples that should be joined therefore are grouped together as aggregates. Besides, all the A tuples in an aggregate precede their to-be-joined B tuples. Then a simple process to shift and join tuples is enough to implement the equi-join operation.

We demonstrate the application of the omni-sort to query processing by an example. Suppose that a database query involves four relations A, B, C, and D, and requires the processing of $(\Pi(A) \bigcup B) - (\Pi(C) \bigcap \Pi(D))$, where Π is the projection operation with the requirement to remove duplicates. The processing of this query can be implemented by a merge-oriented omni-sort as follows:

(1) $R_1 = omni\text{-}sort(f_1, \Pi(A), B)$ and
$R_2 = omni\text{-}sort(f_2, \Pi(C), \Pi(D))$;
(2) $R_3 = omni\text{-}merge(f_3, R_1, R_2)$.

In the first step two runs of omni-sort on different operands can be executed in parallel, where Π is the projection operation without the requirement to remove duplicates. The control argument f_1 should be set to select X-tag and the union operation, and f_2 to select Y-tag and the intersection operation (Section 2.3). Then in the second step a merge stage is sufficient to perform the difference operation since the intermediate results from the first step are already ordered sub-sequences and appropriately tagged.

5. SUMMARY

Omni-sort is a new operation defined to selectively perform the five important database operations: union, intersection, difference, duplicate-removal, and sorting. Adaptation schemes are presented to modify sorting algorithms to implement the omni-sort operation. We show a general adaptation scheme that is independent of the sorting algorithm and discuss its implementation. We also present specific schemes for two families of sorting methods and a type of linear-time sorter.

A highly parallel sorting algorithm may be proposed to be implemented as a specialized processor or to be programmed in a general-purpose computing system. In the former case, the adaptation schemes presented in this paper can be applied to modify the sorting circuit to implement the omni-sort. In the latter case, the parallel program for sorting can be extended to perform the omni-sort in accordance with the adaptation schemes. The benefit is to offset the initial expenses of design, development, and testing of a sorting system by increasing its usage.

Sorting is one of the most important data processing operations. Many sorting algorithms have been invented and developed. The modification cost is small to extend the sorting algorithms to implement the omni-sort using the adaptation schemes proposed. Besides, the performance overhead is negligible; an omni-sort algorithm should share the same time complexity as its underlying sorting algorithm. Based on some parallel sorting algorithms, omni-sort also provides faster-than-ever algorithms for performing the other four data processing operations.

6. ACKNOWLEDGEMENTS

We would like to thank Tientien Li for his careful reading of a draft of this paper and his valuable suggestions to improve its organization.

REFERENCES

1. K. E. Batcher, "Sorting networks and their applications," *Proc. 1968 National Computer Conference*, AFIPS, 307-313.
2. C. C. Hsiao, "Highly parallel processing of relational databases," Ph. D. thesis, Department of Computer Sciences, Purdue University, Dec. 1982.
3. D. E. Knuth, *The art of computer programming,* Addison Wesley, Vol. 1 & 3.
4. H. T. Kung and P. L. Lehman, "Systolic (VLSI) arrays for relational database. operations," *ACM SIGMOD Intl. Conference*, 1980.
5. D. E. Muller and F. P. Preparata, "Bounds to complexities of networks for sorting and for switching," *J. ACM*, 22:2, April 1975, 195-201.
6. J. T. Schwartz, "Ultracomputer," *ACM Trans. on Programming Languages*, 2:4, October 1980, 484-521.
7. L. Snyder, "Introduction to the Configurable, Highly Parallel computers," *IEEE Computer*, 15:1, 47-56.
8. S. W. Song, "A highly concurrent tree machine for database applications," *IEEE Intl. Conf. on Parallel Processing*, 1980.
9. C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Comm. ACM*, 20:4, 1977.
10. J. D. Ullman, *Principle of Database Systems*, Computer Science Press, 1980, Chapters 4 & 6.
11. Leslie G. Valiant, "Parallelism in comparison problems," *SIAM Journal of Computing*, 4:3, September 1975, 348-355.

PSEUDO ASSOCIATIVE LINKING: A HIGH SPEED SEARCHING ALGORITHM FOR PARALLEL PROCESSORS

F.P. Hiner III
Data Systems Division
Litton Systems, Inc.
8000 Woodley Ave.
Van Nuys, Calif. 91406

Abstract -- The task of correlating a large number of radar reports with a large number of stored radar tracks has long been identified as a natural application of Single Instruction Multiple Data (SIMD) computers. As practical/deployable parallel processors are often limited in power and number of processing elements, searching techniques have been sought that would provide faster execution times than that obtained by brute force searches in an SIMD machine. Specifically we sought some means of utilizing the power of the parallel processor in the creation and searching of a data structure that would give us a species of associativity such as enjoyed by a machine with a far larger number of processing elements (PEs).

A data structure and set of searching algorithms have been developed which we name Pseudo Associative Linking (PAL). With this technique one can, in effect, multiply the power of the SIMD machine on itself. The developed algorithms have been successfully used in the implementation of the Litton Advanced Tracking System (ATS) developed for Rome Air Development Center which uses the 16 element Tracking Array Processor (TAP) as the parallel processor. The Litton ATS has already demonstrated a track processing capability of over 10,000 tracks.

1.0 Background

A need has arisen for radar trackers which will not only track hundreds and even thousands of radar targets but perform this task without producing an untenable number of false alarms. Such a system has never truly been implemented as a standalone system.

The Tracking Array Processor (TAP) [1] was originally designed with the object of providing a physically small computer which would be able to perform tracking in the 'track while scan' sense on up to 2000 radar tracks. It soon became apparant that, despite possessing a parallel processor, achieving these speeds might be possible if and only if we used some searching technique analogous to the double linked lists employed in sequential computers and sort on both azimuth sectors and cartesian sort boxes. Such techniques can be implemented in SIMD machines and can be effective but they suffer from the bane of susceptibility to soft memory failures and they produce messy/complex programs. This latter flaw is not trivial. We thus early on began seeking an alternate to the well known sequential computer searching techniques.

The answer that we found was the Pseudo Associative Link (PAL) memory system. The name derived from the clearly 'pseudo associative' nature of any searching algorithm that requires programming coupled with the notion of linked lists. The technique to be described has been implemented in a number of different fashions within the operational Advanced Tracking System (ATS) built by Litton and tested at Rome Air Development Center. The PAL technique has in fact allowed us to not only reach the 2000 track number but to implement a system capable of dealing with well over 10,000 tracks!

2.0 The Tracking Array Processor (TAP)

The Tracking Array Processor is a classic SIMD machine designed expressly for the radar tracking problem. During its inception there was no interest in applying this machine to general purpose problems. Rather, it's design followed studies which demonstrated that an array of processors might provide an optimum method of doing radar tracking. Thus, the TAP consists of a maximum of 16 PEs, each possessing 64 K of dynamic RAM memory and each processing 16 bit data words at a speed of from 3 to 5 MIPs.

Since the TAP's function is the parallel processing of disparate data sets an extremely simple interconnect network (a linear bus) was deemed and later found to be acceptable. Programming is performed in a high level, english-like assembly language [2]. The language and the architecture allow, at each step, for selective enabling of any combination of the PEs and the execution of conditional instructions based on polled responses from the enabled PEs. A WAIT statement allows any given PE to bypass batches of code based on local tested conditions. Included in the instruction set are Top Down programming constructs such as REPEAT-UNTIL, WHILE statements and IF THEN ELSE.

3.0 A Type I PAL

Consider a system which consists of a 2-dimensional array of memory elements, N rows by m columns. An SIMD processor of m PEs with each PE containing N addresses of memory forms such a 2-dimensional memory (See Figure 1). When characterizing the SIMD machine as 2-D memory, the Arithemetic and control logic of each PE functions only to feed and care for the memory elements in a given column.

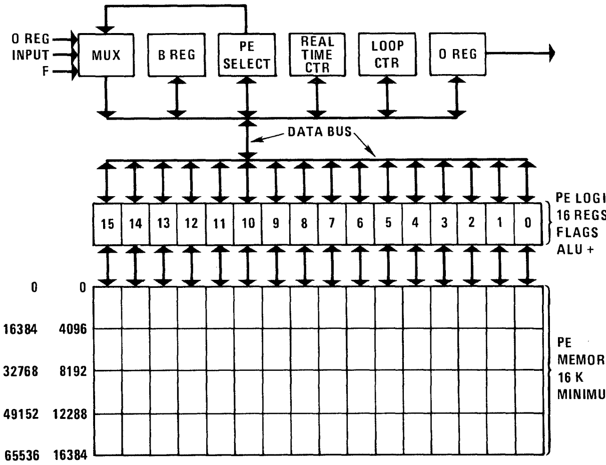


Figure 1. Tracking Array Processor 2-D Memory Structure

A section of the parallel processor memory will be designated a PAL memory. Nominally this section of memory will consist of N1 rows, each one memory cell deep by m columns wide where m is the number of PEs. The PAL memory will be used to point at rows in parallel processor memory defined as the object memory (See Figure 2). This memory is used to store the data being sought. Thus each row in this memory is N2 rows deep. If N3 data items are to be maintained in this memory then the number of actual rows in memory is (N2) (N3) /m. The PAL memory is required however to point at only N3/m rows.

0190-3918/83/0000/0226$01.00 © 1983 IEEE

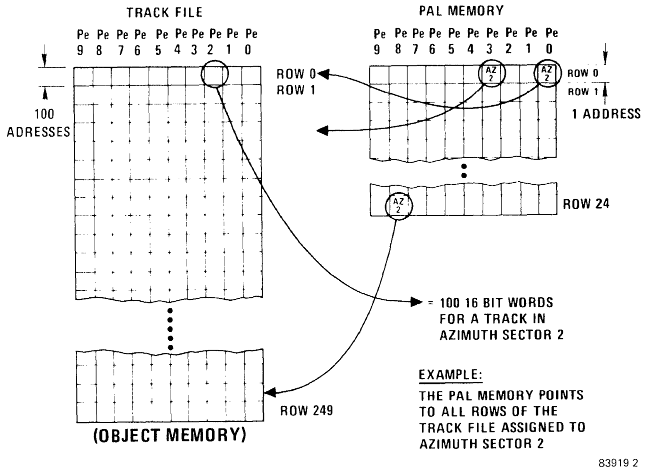

TRACK FILE
Pe Pe Pe Pe Pe Pe Pe Pe Pe Pe
9 8 7 6 5 4 3 2 1 0
100
ADRESSES
(OBJECT MEMORY)
ROW 0
ROW 1
ROW 249
PAL MEMORY
Pe Pe Pe Pe Pe Pe Pe Pe Pe Pe
9 8 7 6 5 4 3 2 1 0
AZ 2
ROW 0
ROW 1
1 ADDRESS
ROW 24
= 100 16 BIT WORDS FOR A TRACK IN AZIMUTH SECTOR 2
EXAMPLE:
THE PAL MEMORY POINTS TO ALL ROWS OF THE TRACK FILE ASSIGNED TO AZIMUTH SECTOR 2
83919 2

Figure 2. PAL Memory to Object Memory Link Structure

3.1 Description of the Type I PAL

We now make the following definitions:

1. We here assume that association will be performed for one field of the data only.
2. All data to be stored in a row of the object memory will be loosely associated. (e.g. if the data to be associated is, say, radar azimuth, then all data stored in the same row might be contained within a 10 degree azimuth wedge).

Upon initializing the system, each i, j memory location of the PAL memory is placed into permanent 1:1 correspondence with a row of the object memory. This is performed by placing the actual address of the row of the object memory into the PAL memory's location. Additional data stored in this memory location are (See Figure 3):

a. A bit which indicates if this PAL location (= row in object memory) is in use.
b. A bit which indicates if the row in object memory is full.
c. A field which defines the value of the association data being stored in the row in the object memory.
(example: if the data is radar tracks and we are storing them in a 5.25 degree azimuth sector then the value of the association data will be a 6 bit word defining one of the 64 azimuth sectors.)

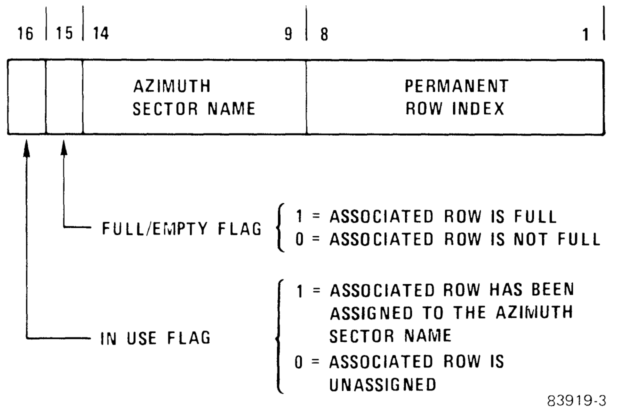

16
15
14
9
8
1
AZIMUTH SECTOR NAME
PERMANENT ROW INDEX
FULL/EMPTY FLAG
1 = ASSOCIATED ROW IS FULL
0 = ASSOCIATED ROW IS NOT FULL
IN USE FLAG
1 = ASSOCIATED ROW HAS BEEN ASSIGNED TO THE AZIMUTH SECTOR NAME
0 = ASSOCIATED ROW IS UNASSIGNED
83919-3

Figure 3. PAL Memory Word Format

Following system initialization the object memory is empty and this of course is indicated by the flag bits in the PAL memory. When the first datum arrives for storage, the program first looks in the PAL memory to see if there is any data with the same 'name' (i.e. association field). This look is done by reading each row of the PAL memory in parallel and looking for a data match. If an "unused" bit is found set during this search, the search can stop. Thus if the memory is empty, the search stops immediately as inspection of the first row of the PAL memory indicates no match on the association field value and at least one empty row flag set.

If there is no association field match, the next available i, j memory location in the PAL memory is assigned. This is carried out by,

a. setting the "in use" flag in the PAL i, j location, and
b. inserting the association value into the PAL location's association field.

The pointer into the object memory is now extracted from the PAL memory location, and the indicated row in the object memory is examined to determine where the next empty location exists (there must be at least one). The new data is now written into the next empty location in the indicated row in the object memory. If this location fills the row the "full" bit in the PAL flag word is set true.

In this fashion each piece of data which lies in the same azimuth sector will be stored in rows in the object memory such that all data items in this row are loosely associated. The number of rows assigned to any particular azimuth will of course be a random function of the input data. It is to be hoped that the association variable has been chosen such that the likelihood of input data being concentrated on one or just a few values is small. In other words, this value must be picked with the advanced intelligent expectation that the data has a high probability of spreading over a number of values.

The principal virtue of the PAL technique comes not from the ease of finding data but in the speed of rejecting data! Consider the act of finding a particular track lying within azimuth sector S. We first look at row 1 of the PAL memory and ask if any of the locations are pointing to a row in the object memory containing azimuth sector S data. If the answer is no, we look at row 2 of the PAL and ask the question again. If we get the answer that there is at least one, we may check the first candidate row that comes to view.

Checking the candidates means going into the object memory at the indicated row and checking m candidates simultaneously. If the sought for datum is not found, we go back to the row of the PAL memory and see if the row we are in points to any more candiate rows in the object memory. If so, we check them. The important thing is that;

a. If the PAL memory does not point to any candidate rows in the object memory, then we have in fact checked and discarded m x m memory locations in the object memory!

b. If a row contains just one pointer to a candidate row in the object memory then we have excluded m x (m-1) memory locations and now go to perform a detailed check of the single candidate row at the normal speed up factor of m (the number of PEs).

Thus if we can guarantee the association parameter produces values which spread over some range we can search the parallel processor at a speed somewhat less than m squared times an equivalent sequential computer and considerably greater than m times the equivalent sequential computer.

3.2 Basic PAL Operation

Each time data is to be accessed from a PAL governed memory one goes into the PAL itself and reads each row of the PAL until either the end of the PAL is reached or a PAL location is detected which contains an unused object row flag.

When data is to be stuffed into a PAL governed memory one must read every row of the PAL memory until a location is found which points to a row in object memory which is not full and possesses the same association value. If no such row is found, one must be created.

If the data stored in the object memory has an association value that changes due to periodic access and recalculation, then upon each recalculation this data may have to be moved within the PAL memory.

3.3 Garbage Collection in a PAL System

In order to use a parallel processor effectively it is necessary that virtually all PEs are utilized a very high percentage of the time. Thus, during searches we wish to guarantee that every PE, at every step, contains data that is maximally likely to be associated with neighboring PE's data. A similar consideration applies to the act of computing on data in object memory.

The PAL memory structure has as its purpose the job of guaranteeing that data will be optimally stored in memory so as to allow parallel access. If data were only loaded into memory, the PAL system so far described would suffice. In the real world, however, data must be randomly deleted from memory. If data is deleted from a row in the object memory that was formerly full then the system following the deletion, is left with two rows in the object memory containing data with the same association value which are not full. Now clearly if there are m PEs and the data is effectively random over the association value then there is a probability of (m-1)/m that on loading the PAL/object memory that one row of PAL object memory or any association value will not be full. This is unavoidable. If there is a great deal of data we are simply stuck with one of many rows which is not full and hence processor effectiveness is only slightly impaired.

If however, there is more than one row not full per each association value our efficiency could fall off dramatically; in fact, we have defeated the purpose of the PAL system. As data continues to be randomly deleted from the system we could reach a state where a great many rows contained but a single datum all with the same association value.

All of this merely goes to say that a form of garbage collection must be implemented. In this case we perform the following algorithm on any data deletion:

3.4 Garbage Collection Algorithm

1. Upon finding and deleting the data, test this row in the object memory to determine if it was already less than full. If so, test if this deletion empties the row. If emptied, return to the PAL and reset the 'in use' bit and exit.

2. If the object memory row was full before the deletion then search the PAL memory to find if there exists a row in object memory which is less then full.

Remove a datum from the row which is already less then full and move this datum so as to fill up the hole just made by the deleted datum and check if this deletion now makes the row from which it was removed empty; if so, clear the association value from the linked PAL location and set the full/empty flag to empty.

3.5 Computation Times For A Type I PAL

We shall continue to draw our examples from the radar case which after all was the heuristic behind this work. Let us consider the case where a type I PAL is storing radar data and is accessed by azimuth wedges. Then

$$Trt = (Nr)\ [Nt/(Aw/Ac)]\ (Os)\ (1/m)\ (T1) \qquad (1)$$

where,

Trt = the plot to track correlation time
Nr = the no. of reports
Nt = the no. of tracks
Aw = the azimuth wedge containing all data.
Ac = the azimuth cell size
m = the number of PEs
Os = overlap scalar
T1 = the time to correlate one report against one track

Obviously computation time is a function of how the targets tend to cluster in azimuth cells. (NOTE: As targets may lie on the edge of azimuth sectors it is often necessary to search more than one sector. The overlap scalar (Os) indicates the number of sectors that must be searched.)

To illustrate the timing we shall assume Nr = Nt = 2000, Ac = 3 degrees, Os = 2.25, and T1 = 10 microseconds. Figure 4 was generated by assuming that all of the tracks were distributed randomly in a fixed azimuth wedge as indicated on the abcissa. Thus if the data is randomly distributed over the entire 360 degrees we have the point(s) on the far right of the figure. If all the tracks are contained within a 120 degree wedge, we get the indicated point.

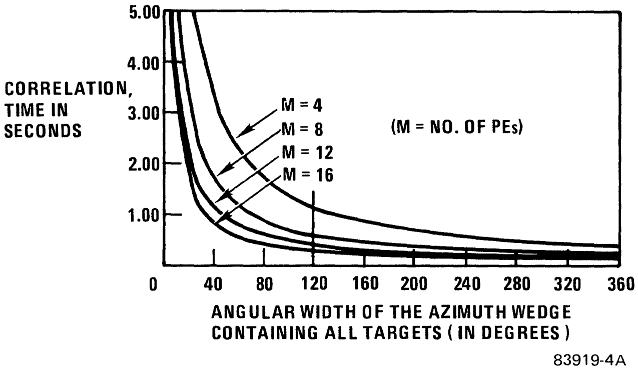


Figure 4. Report to Track Correlation Times for a Type I PAL Organized with Report/Track Azimuth as the Association Field. (m = the no. of PEs)

4.0 A Type II PAL

Suppose we have quite a large number of tracks and try to perform our searching with a type I PAL. Certainly there is a point, depending upon our system constraints at which we will run into problems. Assume, for example, that we must process 20,000 trial tracks and there are 3000 radar reports which must be correlated with these tracks. As a method of comparison let us compare the time to search the entire track memory for correlation if there exists no matching association field. If we use a type I PAL, then;

$$Tnc1 = (Nr)\ [Nt/m^2]\, tc \qquad (2)$$

where:

Tnc1 = the time to search a type I PAL and get NO correlations.
Nr = the number of reports
Nt = the number of tracks
m = the number of PEs
tc = the time to check and reject one row of the PAL

If Nr = 3000 reports and N = 20000 tracks and m = 16 and tc = 3 microseconds then,

$$tnc1 = (3000)\ [20000/(256)]\ (.000003) = 0.7 \text{ seconds} \qquad (3)$$

Can a PAL operate on another PAL system so as to produce another order of magnitude of improvement? The answer is yes and this system we arbitrarily refer to as a type II PAL. Besides being of intellectual interest, the type II PAL became of practical necessity when the number of tracks that we had to process rose beyond the several thousand level. At this number of tracks we had but 16 PEs to play with and we found that the overhead operation of searching for empty rows in the object memory and the basic search was consuming far too much time.

Consider the system illustrated in Figure 5. Track data is stored in cartesian boxes in the object memory defined by an association field which specifies 16 mile by 16 mile squares in the x-y plane. There are two PAL memories, a level 0 PAL and a level 1 PAL. Each memory location of the level 0 PAL points to a row in the level 1 PAL. And every memory location of the level 1 PAL points to a row in the object memory.

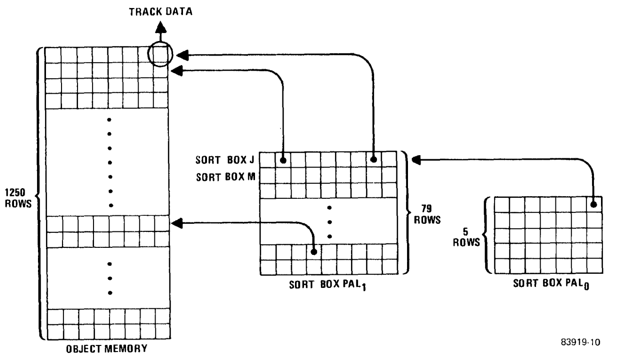


Figure 5. A Type II PAL Structure

As with the type 1 PAL each row in the object memory contains data which is loosely associated by the assocation field. Each such row is pointed to by a memory location in the level I PAL. By our definition, each row of the level I PAL is allowed to point to the rows in the object memory containing the same sort box (i.e. 16 mi. x 16 mi. area). This defines a maximum of m x m tracks that may be contained in any given sort box.

Each row of the level 0 PAL is addressed by a truncated version of the sort box address. Thus if there are 16 PEs, each row of the level 0 PAL points to a 4096 sq. mi. region.

In summary, we provide in excess of 20,000 memory locations in the object memory arranged as 16 columns (PEs) by at least 1250 rows. Each row in the object memory is pointed to by a location in the level 1 PAL. Each row in the level 1 PAL is dedicated to a single 16 mi. x 16 mi. sort box in the correlation space.

Each location of the level 0 PAL points to a single row of the level 1 PAL and consequently is pointing to 256 tracks. If there are 20,000 memory locations then there are least 20,000/m rows in the object memory. If m, the number of PEs, is equal to 16, then the number of rows is 79. Assume the maximum range is 256 miles. Thus as each element of the level 1 PAL points to 16 16 mi. x 16 mi. sort boxes there must be 1250/16 = 78.125 rows, implying 79 rows, in the level 1 PAL. As each row of the level 0 PAL points to 16 16 mi. x 16 mi. sort boxes there must be 79/16 = 4.94 rows, implying 5 rows, in the level 0 PAL.

Let us consider now the same timing problem that brought us into this solution. The timing formula to search the entire track file is now

$$\text{Tnc1} = \quad (\text{Nr})\ [\text{Nt}/(\text{m}^3)]\ (\text{tc}) \qquad (4)$$

If Nr=3000, Nt = 20,000, m= 16 and tc = 3 x 10 -6 sec., then,

$$\text{Tnc1} = \quad (3000)\ [(20000)/4096]\ (.000003) = 0.029 \text{ seconds} \qquad (5)$$

Let us now directly compare the PAL I and the PAL II systems utilizing the task of performing radar report to stored track correlation. As a means of making our comparison, we for purposes of analysis, shall assume that all reports and tracks are constrained to lie within a defined azimuth wedge whose size, we, in our analysis can vary. The targets/tracks shall be assumed to be uniformly and randomly distributed throughout the entire defined wedge. To facilitate our analysis we have calculated the area contained in each such wedge and divided the area of the cartesian sort box into this wedge area to obtain the number of sort boxes, containing the track/report data. To further simplify matters the number of tracks per sort box is assumed to be the total number of tracks divided by the number of sort boxes. This calculation is expressed by the formula:

$$\text{Ntsb} = \text{Nt}/[2(\pi)\ (\text{R}^2)\ (\text{Aw}/360 \text{ deg})/(\text{sort box area})] \qquad (6)$$

where,

Ntsb = the No. of sort boxes
R = the maximum range of the radar
Aw = the angular width of the azimuth wedge in degrees
Nt = the total number of tracks currently active in the system

Any formula for the computation time expended in correlating radar reports with radar tracks must address two principal tasks;

a) Every report must scan the entire PAL system to find any possible links. In the PAL I system, given that there are a maximum of 20,000 tracks, each report must look at 20,000/ (m^2) rows in the PAL memory. If m = 16, then 79 rows in the PAL memory must be examined by every report.

b) Every report that finds a link into the track memory must be correlated against at least one row of m tracks, with m = the number of PEs.

The correlation time for the PAL I system is expressed by:

$$\text{Tr1} = \text{Nr}\ [(\text{Ntsb}/\text{m})\ (\text{Os})\ (\text{T1} \times \text{T2}) + (\text{Nmx}/(\text{m}^2))\ \text{T2}] \qquad (7)$$

For the PAL II system,

$$\text{Tr2} = \text{Nr}\ [(\text{Ntsb}/\text{m})\ (\text{Os})\ (\text{T1}) + (\text{Nmx}/(\text{m}^3))\ \text{T2}] \qquad (8)$$

Where;

Nr = The no. of reports
Ntsb = the no. of tracks per sort box
T1 = the time to correlate one report against one row of tracks
T2 = the time to check one row of a PAL memory
Nmx = the maximum no. of tracks the system can hold
m = the no. of PEs
Os = The overlap scalar

Figures 6a and 6b plot the correlation time for both of these PALs given Nr = 3000 reports, Nt = 10,000 tracks, Nmx = 20,000 tracks, Os = 2.25, T1 = 20 microseconds and t2 = 4 microseconds. Figure 7 provides a direct comparison of the results for the two PAL systems with the number of PEs set to 16.

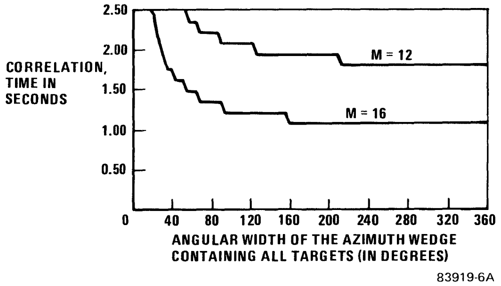


Figure 6A. Type I PAL Results

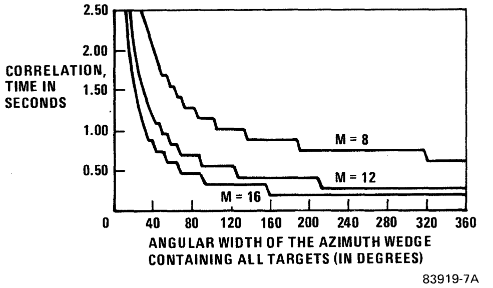


Figure 6B. Type II PAL Results

Figure 6. Plots of Report to Track Correlation Time for Type I and Type II PALs Using Target x, y as the Association Variables.

5.0 A Type III PAL

The data structure described in the previous section is clearly faster than the Type I PAL yet in the tracking application it contains a not insignificant defect. By only allowing 79 rows (in our example) in the PAL I memory we in fact are allowing only 79 sort boxes. There are, in fact, nearly 1000 possible sort boxes. Once our allowed 79 have been assigned, any new track falling outside of this set is not accommodated in the just stated system. One could have a very sparse track load yet not be tracking all the aircraft.

The solution is our third and last PAL system; the Type III PAL structure. In this system we provide a PAL structure wherein there exists a row in the PAL I memory (here called the PAL S1 memory) for every possible sort box. Each row in this PAL S1 memory may now be addressed directly by the data field as the data field is the address of a row. The individual locations within the PAL S1 memory row are not linked permanently to rows in the object memory as was the case in the Type I and Type II systems. Instead, such a linking is made if and only if a datum occurs within the indicated sort box. This means that upon each new track entering the system, we must search every row of the object memory and find the first unassigned row in the object memory and assign it to the appropriate location in the PAL S1 memory. Since there are at least 1200 rows in the object memory this search itself becomes the major time sink. Thus we complete this system by incorporating a Type II PAL structure into this system whose sole purpose is to point to unused rows in the object memory. Figure 7 illustrates the complete Type III PAL system. The included Type II PAL structure (PAL U0 and PAL U1) upon system initialization permanently link themselves to the object memory such that each location of PAL U1 points to a single row of the object memory and each location of PAL U0 points to a single row of the PAL U1 memory.

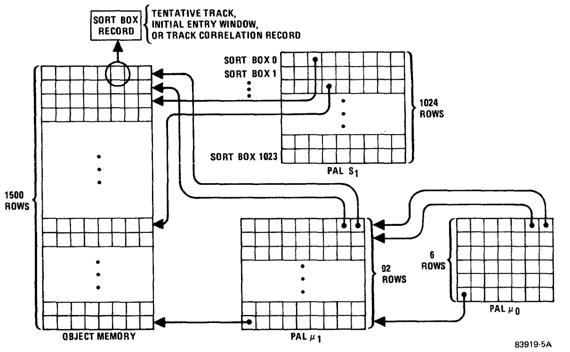


Figure 7. A Type III PAL Structure – The Sort Box PAL and PAL^0 are Used to Correlate a Report Against a Track. The PAL$_{uo}$ Structure is Used to Locate Empty Rows in the Object Memory.

When searching for unused object memory rows the alogorithm first looks in the PAL U0 memory. If there are any unused rows in the first 256 rows of the object memory at least one location in this first row will have a bit set indicating the fact. The algorithm then draws out the first indicated word from the PAL U1 memory and at least one of its locations (PEs) will point to an unused row in the object memory. Upon using the row, the algorithm now resets the 'unused row' bit in the PAL U1 location indicating the linked row is now in use. If this was the last location of that row in the PAL U1 which had been empty, the appropriate flag bit must be reset in the PAL U0 row/column location.

The actual report-to-track correlation time is now very rapid as the algorithm is able to address the PAL S1 memory directly which in turn is pointing directly to all rows in the object memory containing tracks in the referenced sort box.

The correlation time for the Type III PAL system is

$$Tr3 = Nr\ (Ntsb/m)\ (Os)\ (T1) \qquad (9)$$

Using the U0 and U1 structure to find an unused row in the object memory occurs only upon a new track being loaded into the system or when an existing track moves into a previously unused sort box. The computation time to find a unused row in the object memory is,

$$Tns = \frac{Nmx}{m^3}\ (T2) \qquad (10)$$

For Nmx = 20000, m= 16 and T2 = 4 microseconds, we have Tns = 19.5 microseconds.

Figure 8 indicates the performance of the Type III PAL and Figure 9 contrasts the performance of the three PAL types. Both of these plots were generated using the same input conditions as the plots of Figure 6a and 6b.

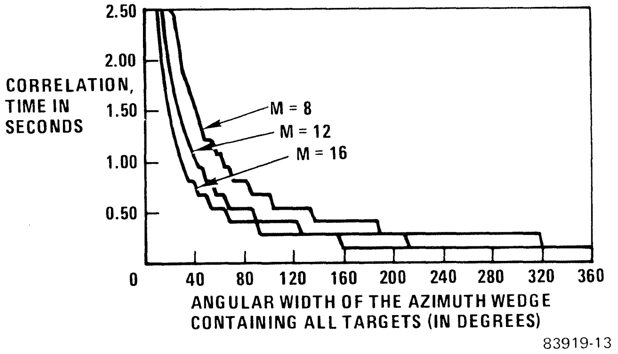


Figure 8. Plots of report to track correlation time for a Type III PAL using target x,y as the association variable.

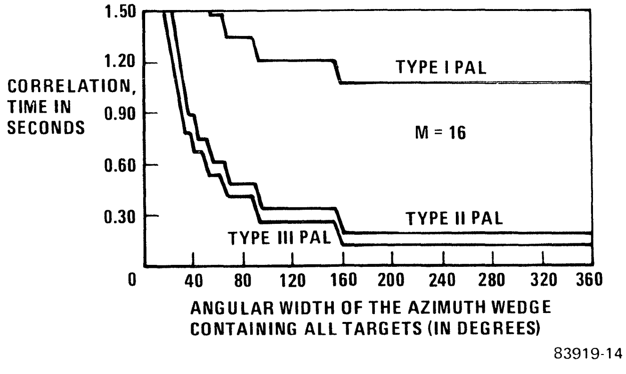


Figure 9. Performance contrasts for the three types of PAL structures described.

6.0 Summary

A natural application of an SIMD machine is in parallel searches in algorithmic operations such as correlation, where a large amount of input data must be associated with a mass of stored data where there is ideally only one match for every stored datum. If one is blessed with an enormous number of processing elements and memory then there is no problem; one may attack the problem in a truly parallel sense. Real world cost and finite size limitations usually restrict the number of processing elements to far less than a desired number.

The PAL technique was devised as a practical method where the inherant power of the parallel processor might be multiplied by itself one or two times producing very great effective speedups. This is essentially accomplished by trading off memory which is now cheap against processing elements which are still not cheap. The PAL system allocates a section of two dimensional memory (The PAL memory) wherin each element points permanently at a row elsewhere in the 2-D memory (called the object memory). The contents of each row in the object memory are loosely associated under the association variable. By testing a row in the PAL memory and not finding a match, m rows of the object memory may be discarded and consequently m x m potential candidates may be discarded. The key element of the trick is obviously choosing the association variable so that at any time a correlation candidate only finds a few rows against which to be correlated.

The PAL technique has been implemented in a now operational radar tracker (The Advanced Tracking System) utilizing a 16 PE parallel processor, the TAP. The result has been a machine which is able to maintain over 10,000 radar trial tracks and is consequently able to tolerate considerably higher input false alarm levels than here-to-fore possible. The resulting tracker is consequently able to maintain over 1000 actual tracks in very noisy environments and has substantially altered the expectations of tracking systems and changed the notion of their limitations.

7.0 Acknowledgements

The first programmed implementation of the PAL technique was performed by Mr. Dale Frederick who also helped design the TAP programming language. The language and compiler design were completed by Mr. Leigh Murphy who, even more significantly, was the first person to utilize the PAL type I and type II data structures and algorithms in the construction of the operational Litton Advanced Tracking System.

8.0 References and Bibliography

(1) F.P. Hiner III, F. Erickson, D. Frederick, D. Johnson, "The Tracking Array Processor", Second Tactical Air Surveillance and Control Conference, RADC, Griffiss Air Force Base, New York, 1980.

(2) F. Erickson. A description of the Building Block Signal Processor Assembly Language (BUBAL). Litton internal Report.

(3) F. Hiner III, "The Tracking Array Processor", Proceedings of the 1978 International Conference on Parallel Processing".

(4) F.P. Hiner III, 'The L2000, A Remote Radar Tracking Station," International Conference "Radar-77" London, Oct 1977.

(5) H. Gregory Schmitz and Cheng-chi Huang, "An Efficient Implementation of conflict Prediction in a Parallel Processor", Lecture Notes in Computer Science, pg. 383-399, 1974, Springer-Verlag.

IMPLEMENTATION OF AN ARRAY AND VECTOR PROCESSING LANGUAGE

R.H. Perrott, D. Crookes,
P. Milligan and W.R.M. Purdy

Department of Computer Science
The Queen´s University of Belfast
Belfast, BT7 1NN, N. Ireland

ABSTRACT -- A compiler for a Pascal based language Actus is described. The language is suitable for the expression of the type of parallelism offered by both array and vector processors. The implementation described is for the Cray-1 computer. An objective of the implementation has been to construct an optimizing compiler which can be readily adapted for a range of array and vector processors. As a result the machine dependent sections of the compiler have been clearly identified.

INDEX TERMS -- array processors, vector processors, graph transformations, abstract representation, optimization.

INTRODUCTION

The programming languages which have been designed and implemented for array and vector processors rely on strategies which can be divided into two categories:

(i) detection of parallelism, in which a programmer constructs a problem solution in a sequential programming language (usually FORTRAN), and a compiler tries to detect any inherent parallelism [3, 6];
(ii) expression of machine parallelism, in which the syntax of the language reflects the underlying parallelism of the hardware, either directly, or by means of subroutine calls [1, 7].

A third category has recently been proposed, which aims to exploit the parallelism inherent in the problem; the program and data structures of the language enable a programmer to express directly the parallel nature of a problem, without reference to specific architectural features [4].

The implementation of these languages present the compiler writer with problems of varying degrees of difficulty. In the detection category, the compiler must determine which statements can be grouped together and executed in parallel. In the second category, the syntax assists the task of compiler construction, since it is designed to map directly onto the instruction set of the machine. The third category, which neither relies on there being a direct hardware correspondence, nor a detection mechanism, requires a new approach for the construction of a compiler for these parallel machines. In this paper, we consider a compiler which is being constructed for the Pascal-based language Actus, on the Cray-1 [6]. The aim has been to construct an optimizing compiler which can be used as the basis for a range of Actus implementations. As this aim ultimately conflicts with the task of exploitation of the hardware, the machine-dependent sections of the compiler must be clearly identified.

LANGUAGE DETAILS

Actus provides the scalar structured programming concepts of Pascal. In addition, the declaration part of a block is used to indicate the maximum extent of parallel processing which can be applied to a structure. For example,in the declarations

```
var a, b : array [1:100] of real;
     cc : array [1..3, 10:90] of integer;
```

the extent of parallelism (eop) is indicated by parallel dots : and is 1:100 for the arrays a and b, 10:90 for the array cc. The elements of an array such as a may be referenced in parallel by means of an expression such as

```
a[i:j]
```

where i <= j and both variables are in the range 1..100. A non contiguous set of elements may be referenced by establishing (static) index sets such as

```
index
   odds = 1:[2]99; (* 1,3,5,- - -,99 *)
   primes = 2:2+3:3+5:5+7:7+11:11+13:13+17:17;
```

and using the index set identifiers as follows:

```
a[odds]
b[primes].
```

Other more complex index sets can be established during execution of a program by combining index sets using the operators + (union), * (intersection) and - (difference).

0190-3918/83/0000/0232$01.00 © 1983 IEEE

During the course of program execution, the extent of parallel processing can be manipulated dynamically by the language statements. Actus thus provides the concept of a dynamic eop, which can be set and adjusted by the following statement types:

```
assignment,
if,
case,
while,
for,
with,
within.
```

Three examples which are relevant to other sections of the paper are:

(i) **if** statement

In a parallel **if** statement, the eop within the **then** clause refers to the indices of only those elements of the parallel condition which were true. If there is an **else** clause, the eop refers to the complement set of indices i.e. those indices which were false.

For example:

```
if a[2:50] > 0.0 then  cc[i,#] := 0
```

The anonymous sharp symbol # is used to represent the current eop. In this example it is determined by the Boolean expression

```
a[2:50] > 0.0
```

i.e. if a[2] > 0.0 then 2 is in the set # ,
if a[3] > 0.0 then 3 is in the set #, etc.

(ii) **while** statement

During a parallel **while** statement the eop is a non-increasing set which is re-evaluated on each iteration until it is empty. For example,

```
while a[1:100] > 0.0 do
   begin
     b[#] := b[#] + a[#] * a[#];
     a[#] := a[#] - 1.0
   end
```

The statements are applied to only those elements of a which are currently in the eop and which are positive; this determines the set as represented by the # on each iteration.

(iii) **within** statement

The **within** statement can be used to indicate the eop for a series of statements in which the eop will not change. It sets the eop as follows:

```
within 50:70 do
  begin
    a[#] := 0.0;
    cc[2,#] := 1
  end
```

If any of these extent setting constructs is nested, the eop is stacked as each new construct is entered, and unstacked upon exit. In addition, the scalar equivalents of these statements (except for the **within** statement) are available.

Data alignment

There are two data alignment operators which can be used to move data within a parallel structure namely,

(i) the **shift** operator which enables the alignment of data within the range of the declared extent of parallelism.
For example

```
var a, b: array [1:10] of real;
- - -
a[1:3] := b[1:3 shift 1]
```

assigns

```
a[1] := b[2], a[2] := b[3], a[3] := b[4].
```

(ii) the **rotate** operator which enables the data to be shifted circularly with respect to the extent of parallelism. For example

```
a[1:3] := b[1:3 rotate 1]
```

assigns

```
a[1] := b[2], a[2] := b[3], a[3] := b[1].
```

The general form of an index expressions using these operators is

```
eop   alignment operator   distance
```

where eop is either an explicit definition of the extent of parallelism or an index set, and distance is an integer expression whose value can be positive or negative.

This brief summary of Actus has described only some of the features of the language which are relevant to the discussion which follows. Further details can be found in [4, 5].

STRUCTURE OF THE COMPILER

The Actus compiler is implemented in Pascal and based on a multipass scheme. On the first pass, syntactic and semantic analysis is carried out by the analyser and a pseudo-analyser constructs an abstract representation of the program. On the second pass, the pseudo-analyser transforms this representation to include both machine-dependent and machine-independent optimizations. Although the machine-dependent transformations will be described before the machine-independent ones, this is not necessarily the order of application. Indeed, the machine-dependent transformations are generally applied first, because they can often introduce additional code structures which require optimization. Once the graph has been transformed, code synthesis follows. Certain aspects of the code generation process have been rendered machine-independent by the construction of an abstract description of the target machine (described in a later section).

Program representation

The abstract representation of a source program is a graph, which is constructed by the pseudo-analyser. However, the graph is not designed to reflect exactly the constructs of the source program as is usually the case for graph-based compilers [2,9]; rather, the node-types are chosen so as to facilitate the transformations to be effected during code synthesis. For example, in Actus several structures set the extent of parallelism and there are several forms of loops; on the graph only one node-type may set the extent of parallelism and there is a reduced number of loop structures.

(i) Control flow structures

There are five semantically distinguishable loop structures in Actus, namely the scalar and parallel **for** statement, scalar and parallel **while** statement and the scalar **repeat** statement. One of the loop structures on the graph is equivalent to the language construct (not actually available in Actus)

```
repeat
   statement
while  boolean expression
```

which is chosen for the consequential simplicity in removing loop invariants. To express a **while** statement in terms of this structure requires duplication of the terminating condition in an enclosing **if** statement:

```
while test do statement
```

becomes

```
if test then
  repeat
    statement
  while test
```

This transformation would be applied when loop invariant removal is applicable. Although it will increase the size of the object program, it does not impair run time speed. Similar remarks apply to representation of **for** statements.

(ii) The extent of parallelism

The pseudo-analyser simplifies references to the extent of parallelism on the first pass. The following statements illustrate the diversity of references to the eop.

```
(1) a[1:10] := b[1:10];

(2) within 1:10 do a[#] := b[#];

(3) if a[1:10] > 0.0 then
         a[#] := b[#];
```

In (1), the eop is set explicitly and explicitly referenced on every use. In (2), it is set explicitly but referenced by implication. In (3), its production is based on calculation and its reference is again by implication.

Actus index sets are static and therefore are unsuitable for representing the dynamic eop on the graph. For this purpose the graph uses a "run-time-set" construct, which contains information on the base value, stepping value, length and regularity of a parallel index. A regular parallel index is one which has a constant increment between all ordered adjacent values.

In addition to the usual set arithmetic operations, the pseudo-analyser uses four operations in constructing and handling run-time-sets. These are

(1) setfrom (index set expression) - converts an index set expression to a run-time-set;

(2) truevalues (vector expression) - returns a run-time-set corresponding to those members of the current eop which yield "true" when substituted for the parallel index in the vector expression;

(3) anymembers (run-time-set) - returns a boolean value determined by whether the "run-time-set" is empty or not.

(4) setcomplement (run-time-set) - complements a run-time-set within the current eop.

"Setcomplement" and "truevalues" are always subsets of the run-time-set of the current eop. Set calculations are performed by "setassign" structures on the graph.

Given this facility for calculating and storing the extent of parallelism, we require a unique means for setting the eop and a unique means for accessing it. The pseudo-analyser provides a "new-eop-scope" structure for setting the eop and uses a "sharp" node to refer to it thereafter. The new-eop-scope structure has the form

```
eop  run-time-set  do  statement
```

Using these structures, a parallel form of the **if** statement reduces as shown in the following example.

Example

Illustrating the graph for a parallel **if** statement.

("#rts-n#" denotes a run-time-set identified as n.)

```
if a[1:10] > 0 then
   a[#] := a[#] - 1
```

becomes

```
begin
#rts-0# := setfrom (1:[1]10);
eop #rts-0# do
  begin
  #rts-1#  := truevalues (a[#] > 0) ;
  if anymembers (#rts-1#) then
     eop  #rts-1# do
        begin
        a[#] := a[#] - 1 ;
        #rts-1# := truevalues (a[#] > 0)
        end
  end
end
```

The structures described above are built on the first pass of the compiler. Calls to the pseudo-analyser are embedded in the corresponding analyser recognition procedures. To reduce storage requirements, the unit of compilation is currently one block. When the graph for a block has been constructed, it is passed on for further transformation and code synthesis.

Machine-independent transformations

The purposes of these transformations can be classified into two main areas:

(1) allocating storage units to variables;
(2) program optimization.

Scalar variable allocation

The time consuming task of fetching variables from memory is one which the compiler must eliminate whenever possible. Thus an efficient register maintenance scheme is essential. In the Actus compiler, storage for scalars is not allocated statically. Each time a variable assumes a new value it is allocated fresh storage which becomes free immediately the last access to that value has been made. When there is redundancy it is possible to allocate more scalar variables to registers than there are registers on the machine. Resulting housekeeping operations (such as dumping values to temporary store) are kept to a minimum.

A variable may have many "lifetimes" in its scope and thereby be allocated storage many times. Consequentially, one major difficulty in the implementation of this scheme is to ensure consistency of the allocation, particularly through the conditional statements sequence of the program. To fulfil this, all nodes corresponding to scalar variables on the graph refer the pseudo-analyser to a version number for this variable by means of a pointer. Each time a new value is assigned to the variable, a new reference area containing a new version number is created. Until the last reference to this value has been made, all nodes for the variable will refer to this area. The pair (variable, version number) is referred to as a key. Storage is allocated to the keys, rather than to variables.

The consistency problem is now to ensure that each variable produces the same key (nodes refer to the same version number) whether or not a conditional statement sequence is executed. Variables which are changed in this statement sequence will have been allocated new version numbers and are compelled to conform. The reference area set up by the last assignment to the variable in the statement sequence is rewritten with the version number of the variable before the condition was evaluated; all nodes referring to this value will therefore produce the original key and hence be allocated the same storage.

Particular care has to be taken when allocating version numbers to variables in the alternative control paths through parallel conditional statements (**if..then..else**, or **case**). This is because each alternative path is executed in turn (though each with its own eop), but each path assumes that the variables initially have their ´start-of-statement´ values. For instance, in the statement

```
if  a[#] < 0 then
   begin
   ....
   i := ...
   end
else
   begin
   ....
   j := ..expression using ´i´..
   end
```

the second assignment must use the original value of i, so the first assignment must not update the original location. In general, one solution to this problem is to stipulate that alternative paths through such a statement must delay updating those assigned variables which are referenced in later paths until the end of the statement.

Access of structured variables, such as arrays or records, is controlled by address calculations represented explicitly in the graph. Such schemes are commonplace and will not be discussed here.

Program optimization

Most of the optimizations traditionally associated with sequential languages are applicable (eg., removal of loop invariants, constant folding, common sub-expression elimination, etc.). Since such optimizations can be thought of as a manipulation of the program text, their implementation is machine-independent. Techniques for these optimizations are well documented elsewhere.

Machine-dependent transformations

The fixed length of the vector registers on the Cray-1 necessitates splitting long vectors into 64-word slices (and usually a remainder), referred to as vector slicing. Each "new-eop-scope" construct will be governed by some loop structure; so each nested structure is encountered more than once. This creates difficulty when any part of the enclosed structure has a different extent of parallelism.

(i) Creating slicing loops.

The run-time-set describing the eop will come from one of two sources:

(a) A "setfrom" operation, which defines a base, step and length of an eop from an index set expression and determines its regularity;

(b) A "truevalues" operation which defines an irregular subset of an enclosing regular or irregular eop.

The "truevalues" operator does not affect the choice of elements to be processed on the Cray-1; rather, it defines which elements of the result vectors are meaningful and,thereby , which elements of the parallel variables are to be updated after the calculation is completed, using a vector merge operation. These elements are always a subset of the current eop. On the other hand, "setfrom" defines a vector length, which may be arbitrarily large. The target machine restricts the vector length to a maximum of 64 elements, so a "setfrom" operation must always define a slicing, which will remain unaffected by any nested "truevalues" operations. Therefore, "new-eop-scope" constructs setting new extents which were defined by "setfrom" operations may not be nested. These extents must be stacked explicitly and recovered explicitly, as the language scope rules dictate. Extents set by "truevalues" operations are held and nested as before; we note the necessity of merging the original and result vectors according to the extent at the close of every "truevalues" scope.

Finally, we emphasize that these transformations are machine-specific, whereas the selection of the graph operations "setfrom" and "truevalues" is not. Although they bear an inexact analogy with hardware features specific to the target machine (respectively the VL- and VM-registers), their properties were dictated by the requirements of the language, as described in the previous section, not for the convenience of the target architecture.

Problems associated with the implementation of this plan fall into two categories. We must decide how to deal with structures which cannot be successfully sliced, such as certain scalar assignments, procedure calls, gotos and nested slicing loops; and we must be able to make good all implications of execution order expressed in the language.

(ii) Statements with non-conforming extents of parallelism.

When the compiler discovers a structure equivalent to

```
within 1:100 do
   begin
   a[#] := b[#] + i ;
   i := i + 1;   (* a scalar statement *)
   b[#] := j * a[#]
   end
```

the scalar statement cannot be included in any slicing loops which are created (otherwise i would be incremented a number of times). To try to reduce the slicing overheads, the compiler will attempt to perform a (machine-specific) optimization equivalent to

```
within 1:100 do
   begin
   a[#] := b[#] + i ;
   b[#] := j * a[#]
   end;
i := i + 1
```

where the integrity of the implied loop is preserved. Problems would arise if the scalar factor in the second assignment were i, not j; the loop would have to be broken to produce

```
within 1:100 do a[#] := b[#] + i ;
i := i + 1;
within 1:100 do b[#] := i * a[#]
```

Although this example could be resolved by precalculation, in general we cannot restructure to maintain a single slicing loop. We note that the scalar statement must be moved outside all the slicing loops. This is non-trivial if several nested conditional ("truevalues") scopes are to be interrupted; all must be stored to maximize the effectiveness of the slicing when resumed and to ensure that result vectors are composed correctly.

(iii) Shift operator

Two problems are considered here. The first occurs in statement sequences such as

```
within 1:100 do
   begin
   a[#] := b[#] + 1 ;
   c[#] := a[# shift 1]
   end
```

where the difficulty derives from the statement order. Contending that the Cray-1 performs the body of the construct in two slices using eops of 1:36 and 37:100, we find on the first pass the second vector statement attempts the assignment

```
c[36] := a[37]
```

thereby assuming that a[37] has been set up previously. In fact a[37] is part of the second slice of the vector and would normally be set up on the second pass. Again, splitting the slicing loop is necessary to accommodate the general case, as below.

```
within 1:100 do a[#] := b[#] + 1 ;
within 1:100 do c[#] := a[# shift 1]
```

Second, problems arise when performing a statement such as

```
within 1:100 do
   a[#] := a [# shift -1]
```

To avoid overwriting a[36] (with a[35]) before it is copied into a[37], it is necessary to introduce a temporary storage array (a´), and implement the statement as

```
within 1:100 do
   a´[#] := a [# shift -1];

within 1:100 do
   a[#] := a´[#]
```

(iv) Set operators

The last problem described here involves the use of the "setfrom" operator. Consider the statement

```
within (i:[2]j + p:[3]q) do
   a[#] := a[#] + 1
```

where the variables i, j, p and q have, for example, the values -1, 17, 1 and 25 respectively.
The problem arises in that the run-time-set is difficult to construct for efficient use. We require a run-time-set representation of the set

{ -1, 1, 3, 4, 5, 7, 9, 10, 11, 13, 15, 16, 17, 19, 22, 25 }

which is far from regular. If the index set segments were non-intersecting, they might be processed separately. Otherwise, as in this case, a set must be built in steps (at best) of the highest common factor of the two supplied steps, and using the maximal bounds. This may be an expensive operation, and could produce a run-time-set with many more slices than the individual components required.

THE ABSTRACT MACHINE

The compiler generates code for a target machine via an abstract description of the machine.

The incentive for building an abstract description of the target machine was not only the desire for machine independence. It arose also from the nature of the Cray-1 architecture, and from the fact that its instruction set contains a large number of special cases. For instance, the CAL instruction

$$A_i \quad A_j + A_k$$

stores the integer sum of registers A_j and A_k in A_i, unless either j or k is zero. If j is zero, the instruction uses the number 0 instead of A_0; if k is zero, 1 is used instead of A_0. These special cases are common to much of the instruction set, and it would be inefficient for the compiler to look for every such case explicitly. Instead, the compiler sets up an abstract description of each instruction, and treats special cases as different instructions. An instruction is defined in terms of the register classes it accesses and an Aoperation.

A register class is a set of target machine registers which are addressed identically by a field in a target machine instruction. Immediate constants are conceptually held in registers and are classified in exactly the same way. The description of a register class is supplemented by the types of values it can hold and by the number of target registers of that class.

For the Cray-1, an examination of the instruction set produces the register type description

```
Aregclass = (    A0, A1to7, A0to7, S0, S1to7,
                 S0to7, B0to63, T0to63, V0to7,
                 VL, VM, Amemory, Aconst0,
                 Aconst1, --- );
```

A0 and S0 are on occasion separated from their respective superclasses as their appearance in some instructions implies use of a special value (as illustrated above); also they are the only registers which may be used in determining conditional branches. From the information already on the graph, it is possible to determine the set of register classes which could hold the value expressed by any expressional node.

The Aoperations implemented on the abstract machine are derived from the operators present in the language. They are described by the type

```
Aoperation = (      (* Arithmetic operations *)
                    Aintplus , Aintminus , -- ,
                    Arealplus, Arealminus, -- ,
                    Aor      , Aand      , -- ,
                    Asetplus , Asetminus , -- ,

                    (* Relational operators *)
                    Arealgt  , Arealge   , -- ,
                    Aintgt   , Aintge    , -- ,

                    (* Standard functions *)
                    Asin     , -- , Afloat, -- ,
                    Asumreal , -- , Afirst, -- ,

                    (* Eop manipulation *)
                    Asetfrom, Atruevalues,
                    Asetcomplement, Aanymembers );
```

An instruction description includes information on the mnemonic, etc., and also the (two) source and (one) destination register classes. Instructions are classified by the Aoperation which they implement. For instance, the above CAL instruction would appear in the abstract machine as two instruction descriptions, with the destination and source classes:

```
   (A0to7   A1to7 + A1to7  )
   (A0to7   A1to7 + Aconst1)
```

With the instruction set described thus in an abstract way, the choosing of instructions for each expression-evaluating operation on the graph can be performed by the procedure

```
   procedure chooseinstructions;
```

This associates an instruction with each Aoperation in an expression, and assigns a unique register class to each valued node, taking into account future uses of the result. Its operation is machine-independent.

An Aoperation for which there is no single instruction is implemented by an abstract code sequence. Included in the abstract description of a code sequence is a set of registers, called Pregisters, which refer the description to registers used by the code sequence (for passing values to and from the code sequence, or for working space needed by the code sequence). The purpose of these Pregisters is to ´parameterize´ the code sequence, so that the registers which they use do not need to be pre-allocated, and do not need to be fixed for all invocations of the code sequence in a program.

The Pregisters are described by the type

```
Pregister = (Ans, Arg1, Arg2, Dummy, Local1,
             Local2, Local3, -- );
```

Each code sequence is held in a record of type "codedescription" which describes the register classes of each Pregister used by the code sequence and timing information, in addition to the code sequence itself. An instruction description is set up to look like a code sequence of length one.

The code sequences for Aoperations are held by a variable

```
Acodesfor : array [ Aoperation ] of
                Acodelist ;
```

A list of sequences is held to enable the compiler to choose the implementation most appropriate to the source and destination register classes. To generate machine code for an expression, an actual register of the required class is found for each Pregister and this replaces the Pregister in the abstract sequence, leaving target machine code.

Abstract code sequences are also used to represent data transfer between register classes using the array

```
Path : array [ Aregclass, Aregclass ] of
          Acodesequence ;
```

as some moves are non-trivial. The same means may be exploited in representing run time support code, such as subrange or array subscript checks.

The allocation of actual registers to expression values treats program variables and temporary storage alike; the allocation of registers to keys for scalar variables allows a variable to be allocated different store from one use to another. While this produces efficient register use, it leads to considerable overheads in the production of meaningful diagnostics.

The most critical regions of a block are the bodies of the innermost loops as these are the sections which will be executed most frequently. Register allocation is therefore performed from the innermost level of loop nesting, working outwards.

CONCLUSIONS

This paper has described the approach taken in the construction of a compiler for an array and vector processing language. The compiler has the task of compiling machine-independent programs, while at the same time exploiting special-purpose architectural facilities.

It has been found that a large proportion of the compiler is machine independent, partly because of the abstract representation which have been used to represent both the source program and the target machine.

ACKNOWLEDGEMENTS

The work described in this paper was supported by the U.K. Science and Engineering Research Council under grant number GR/B/23601.

REFERENCES

1. Star Programming manual, Control Data Corporation, 1976.

2. J.J. Donovan, "Systems Programming", McGraw-Hill, New York, 1972. Chapter 8.

3. R.F. Millstein, "Control Structures in Illiac IV Fortran", Commun. Ass. Comput. Mach., Vol. 16, pp 622-625, Oct. 1973.

4. R.H. Perrott, "A language for array and vector processors", ACM Trans. on Programming Langs. and Systems, Vol 1, pp 177-195, Oct.1979.

5. R.H. Perrott, "Actus user manual", Dept. of Computer Science, Queen´s University, Belfast. 1982

6. R.M. Russell, "The CRAY-1 Computer System", Commun. Ass. Comput. Mach., Vol 21, pp 63-72, Jan. 1978.

7. K. Stevens, "CFD - a Fortran-like language for the Illiac-IV", ACM SIGPLAN Notices, pp 72-80, Mar. 1978.

8. W.M. Waite and L.R. Carter, "An analysis/synthesis interface for Pascal compilers", Software - Practice and Experience, Vol 11, pp 769-787, Aug. 1981.

9. W.A. Wulf, "PQCC : a machine-relative compiler technology", Carnegie-Mellon University, Pittsburgh, 1980.

A PARALLEL P-CODE FOR PARALLEL PASCAL AND OTHER HIGH LEVEL LANGUAGES

John D. Bruner
Lawrence Livermore National Laboratory
Livermore, CA 94550

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, NY 14853

Abstract -- Parallel P-code is an intermediate compiler language for parallel processors. It was originally designed as part of a Parallel Pascal compiler for NASA's Massively Parallel Processor (MPP). However, it should also be suitable for a wide variety of high level languages and parallel architectures. Parallel P-code is based on a P-code language for serial processors; this paper describes the extensions which were necessary for the parallel environment.

Introduction

Parallel Pascal was designed to be a high-level programming language for the MPP [1] and other parallel processors. Parallel Pascal is characterized by having very few extensions to the basic Pascal language in an attempt to provide the programmer with good primitive tools rather than canned solutions.

Since its initial specification [3], the design of Parallel Pascal has been refined and simplified through experience with a Parallel Pascal to standard Pascal translator [2] and the development of a compiler. The final language specification includes array expressions, conditional array assignment with a where statement and the manipulation of subarrays with consecutive elements. Portability of the language to other parallel architectures is enhanced by implementing permutation operations with standard functions. A complete specification of the language is given in references 4 and 5.

The Parallel Pascal compiler consists of a syntax analysis "front end" and a code generation "back end." These two phases of the compiler communicate through an intermediate language called Parallel P-code. The compiler and its intermediate language are based upon the P-4 Pascal compiler [6].

The most significant difference between standard P-code and Parallel P-code is the way in which they treat data types. In standard P-code only a few data types are supported - integer, real, Boolean, character, set, and pointer. Parallel Pascal, however, permits (and in fact encourages) the manipulation of arrays as aggregates, requiring additional data types. Parallel P-code therefore defines a set of base types and provides facilities for constructing more complex data types.

The base types are very similar to those in standard P-code: integer, real, character, and Boolean. With the appropriate definition statements, these base types are used to define all structured types.

The definition of subrange types, set types, file types, and pointer types is fairly straightforward. Objects of these types can be directly manipulated because their behavior is known at compile time. They are defined by statements of the form:

```
.RANGE  rng,1,5         ;define subrange 1..5
.SET    sst,5,8         ;define set of 5..8
.FILE   ftype,char      ;define file of char
```

Array and record types cannot be so easily manipulated, because they are not always manipulated as entire entities.

Arrays

When arrays are manipulated in standard P-code, almost all operations are performed on scalar elements. (The exception to this rule is a provision for moving blocks of data from one place to another.) Parallel Pascal, however, requires operations to be performed upon arrays as aggregates. It was necessary to provide a formalism for specifying these parallel operations in the intermediate language.

In order to process array operations, the code generator must know at least the size of the array and the type of elements. For more sophisticated operations (e.g. operations which involve only a subset of the array) it must also know the layout of the array - the number and range of array dimensions. This information can be divided into two portions, static and dynamic.

The static portion represents information that is known at compile time. It consists of such things as the base type (i.e. the type of the array elements), the number of dimensions, and the low and high bounds of each dimension. This portion can be considered the logical specification of the data.

The dynamic portion of an array type consists of the address of the array and the specification of which elements are to participate in an operation. This portion therefore represents the physical specification of the data - where it is stored and what portions of it (e.g. which array elements) are to be affected.

0190-3918/83/0000/0240$01.00 © 1983 IEEE

The static and dynamic information is collectively referred to as an array descriptor. These descriptors are essentially generalized dope vectors which specify not only the type and size of the data but also the subset upon which operations are to be performed. An important consideration is that while Parallel P-code performs transformations upon descriptors, it does not specify their exact format. In fact, a typical code generator will probably treat the descriptors as conceptual entities which have no physical counterparts at run time. (A possible exception to this would be the treatment of array arguments to user-defined functions and procedures.)

The static portion of an array descriptor is specified in Parallel P-code via the ".ARRAY" pseudo-operator. The base type (i.e. array element type), number of dimensions, and range of all dimensions are specified. For instance, the array type defined by

```
arr = array [1..5,2..6] of integer;
```

would be defined in Parallel P-code with the statement:

```
.ARRAY  arr,integer,2,1,5,2,6
```

Parallel Pascal provides the parallel reserved word for declaring that an array should be allocated in the parallel array memory rather than the sequential control unit memory. If an array is declared parallel, this fact is reflected in Parallel P-code by a negative rank.

Records

In order for arrays of records to be intelligently processed, it is necessary for the intermediate language to define descriptors for records as well as arrays. Like array descriptors, record descriptors consist of a static and a dynamic portion. The static portion specifies the record: the fields and their types. The dynamic portion specifies the address of the record and the field which has been selected for a particular operation.

Because the structure of a record is not as regular as the structure of an array, a single type definition statement for the static portion of a record would be cumbersome. For that reason, Parallel P-code defines records according to the fields which they contain. The pseudo-operator used to define record components is ".RECORD". One ".RECORD" is generated for each field.

Parallel Pascal, like standard Pascal, permits variant records. When a record has variants, several components will share the same memory allocation. (Only one is in use at any given time.) Parallel P-code permits the specification of an offset with each field declaration. A record definition in Parallel P-code consists of a sequence of ".RECORD" statements. Normally, each successive field in the same record is assigned a sequential location in memory. However, this behavior can be overridden so that a field is aligned at the same offset as a previous field.

The general syntax of the ".RECORD" pseudo-op is

```
.RECORD rname,fname,offset,ftype
```

where "rname" is the name of the record being defined, "fname" is the name of the field being defined, "ftype" is the type of the field, and "offset" is either "nil" or the name of a previously-defined offset. If "offset" is the literal string "nil" the next sequential memory location is assigned; otherwise, the new field "fname" is aligned with the existing field "offset". As an example, the record defined by:

```
rec = record
        x: integer;
        y: real;
        case Boolean of
          false: (zf: integer);
          true:  (zt: real)
      end;
```

would be translated to

```
.RECORD rec,x,nil,integer
.RECORD rec,y,nil,real
.RECORD rec,zf,nil,integer
.RECORD rec,zt,zf,real
```

Variable Allocation

In order to examine the specification of the dynamic portion of array and record descriptors it is necessary to first consider the way in which Parallel P-code allocates and references local variables.

Corresponding with each called function or procedure is an area on the runtime stack called the stack frame (or activation record). In addition to the arguments to the function or procedure, the local variables, and space for temporary results, the stack frame includes some linkage information. In standard P-code this includes the return address, space for a returned function result (this field is unused for procedures), and two locations for the static and dynamic links. The static and dynamic links point to the appropriate previous stack frames. The hypothetical machine which implements P-code contains a non-user-accessible register called the "frame pointer" which hold the address of the current stack frame.

Parallel P-code, like standard P-code, addresses operands according to the lexical level at which they are defined. However, in order to deal with objects on an abstract basis (and thereby avoid specifying memory allocation in the intermediate language), physical address offsets are not used. Rather, variables are referred to by a logical index, so that the lexical address is

(level,index)

Hence, Parallel P-code does not define the exact format of a stack frame; this is left to the implementation.

An index is assigned to all local variables, procedure or function arguments, and the function return value (if the routine is a function). All of these share the same set of indices. The index zero is reserved for the result of a function. Arguments are assigned indices starting at 1, and local variables are assigned indices beginning immediately after the last argument.

Although function (or procedure) arguments and local variables share the same set of indices, they require somewhat different treatment when they are defined. Thus, two statements are used for arguments and local variables. The definition statements are:

```
.ARG     index,type,rv
.LOCAL   index,type,overlay
```

where "index" is the index number, "type" is the argument type, and "rv" (for ".ARG") is zero if the argument was passed by value or one if it was passed by reference. The "overlay" field (for ".LOCAL") is similar to the "align" field for the ".RECORD" pseudo-operator. It is normally zero, indicating that the local variable should be allocated the next available memory location (or locations). If it is non-zero, it specifies a previously-defined local variable (at the same lexical level); the new variable is to be overlaid on the memory allocated for the specified old variable.

Parallel P-code provides two statements, ".ENTRY" and ".EXIT" to define the lexical level of the procedure they enclose.

Runtime Operation

In Parallel P-code, as in standard P-code, all operations are performed by means of a run-time stack. Data is loaded onto the top of the stack, manipulated on the stack, and stored from the top of the stack. In standard P-code, data is manipulated in one of two ways. The first way is to load the data onto the stack and manipulate it directly. This is the most common method (in standard P-code) and it works well because Pascal usually deals only with one item at a time. An alternate way is to perform a data transfer of a compile-time specified number of elements between two addresses which are computed at runtime. In this second case (used in assignment statements where both sides are identical arrays or records) the addresses, not the data, reside on the stack. They could be called very simple descriptors because they describe where the referenced data is (or is to go).

Parallel P-code also makes use of these two mechanisms. When an operation is performed on scalar data, the data itself is loaded onto the runtime stack, manipulated, and stored from the stack. When an operation involves an array or record, or some combination thereof, the second method is called for. However, because Parallel Pascal provides more flexibility in aggregate operations, an address alone is not sufficient; rather, information must be provided about the shape and type of the data. The type information is supplied by the static descriptor (i.e. by an ".ARRAY" or ".RECORD" definition). The runtime-dependent shape information is provided by the dynamic descriptors on the runtime stack.

The runtime nature of an array is determined by two dynamic attributes: the address of the array and the index ranges of its dimensions. The dynamic (physical) portion of the array descriptor which resides upon the runtime stack specifies these attributes. This information is constructed by loading a "blank" descriptor (one which specifies the array address but does not specify index ranges) and then "filling in" the index ranges using one of three operators corresponding to Parallel Pascal's array indexing modes: "IX0" (select entire index range), "IX1" (index by a scalar), or "IX2" (index by a subrange). Each successive index instruction is applied to the next unspecified array index range. The intermediate language does not specify the format of the dynamic array descriptor; this is solely the domain of the code generator.

In contrast with arrays, only one component of a record may be specified at a time. However, unlike arrays, the fields in a record are non-homogeneous. The manner in which the target machine stores the fields of the records will affect how a record field is specified; the compiler cannot simply calculate a constant offset (as is done in standard P-code). All record field selection in Parallel P-code is performed with symbolic names. The names correspond to the field names defined in ".RECORD" statements.

The exact format of a record descriptor is not known to the compiler "front end." Instead, the record descriptor is constructed with the aid of the "select" ("SEL") instruction. A descriptor that specifies the entire record is loaded onto the stack; this is similar to the "blank" descriptor described above for arrays but may be used without further modification to access the entire record. The "SEL" operator is used to select a field from the record. This replaces the record descriptor on top of the stack with a modified descriptor that indicates the address of the record and the selected field. If that field is itself a record, another "SEL" is then used to select a field within that sub-record.

Descriptors for more complex structures (e.g. arrays of records, arrays within records) are constructed by repeated application of the techniques described above.

When an operation is performed on scalars, the address where the result is to be stored is loaded onto the stack, the scalar expression is calculated, and a "store indirect" is performed to store the result of the expression (on top of the runtime stack) at the specified address (the second item on the runtime stack).

When an operation is performed on a structured type, the result must be stored in a temporary area and a descriptor for that temporary placed upon the runtime stack. The automatic allocation of the temporary storage to which the descriptors refer is the responsibility of the implementation.

Parallel Control

Parallel Pascal provides the standard Pascal control statements if, case, for, while, and repeat-until. The implementation of these control-flow constructs in Parallel P-code is identical to the implementation in standard P-code. Parallel Pascal also provides a construct to allow masked assignment of arrays - the where statement - which cannot be (efficiently) implemented with the scalar-oriented control mechanisms of standard P-code.

Since the where statement controls array assignments, the implementation in Parallel P-code will only affect stores. In general, SIMD-class parallel processors associate with each processing element a flag known as the "mask bit" or "activity bit." This bit controls whether or not the processor is enabled or disabled. The collection of mask bits for each processor can be considered to be a Boolean "mask array." The controlling expression of a where statement in Parallel Pascal is a Boolean array; hence, it is natural to implement the where statement by using this array as a mask array.

The current conditional status of a set of N nested conditionals can be determined by using a stack of length N bits. If the current conditional state is A and a where statement is encountered which evaluates to B, the new conditional state is AB (the Boolean product of A and B). At some later point, if an otherwise is encountered, the desired conditional state is A(~B). This can be computed by

$$A(\sim B) = AB \oplus A$$

The stack implementation is defined as follows. Initially the stack is empty and all processors are enabled. When a where conditional is encountered, a Boolean "and" is performed with the current top of the stack (if the stack is non-empty) and the result is pushed onto the stack. When an otherwise is encountered, a Boolean "exclusive-or" is computed between the top two elements of the stack and the result replaces the top of the stack. (If the stack contains only one item, it is complemented.) When the end of the conditional is encountered, the stack is popped.

These three operations - pushing a new mask, complementing the current mask, and popping the mask - are provided in Parallel P-code by the "WHR", "OTW", and "ENW" operators.

Conclusion

Parallel P-code has the following extensions relative to standard P-code for parallel languages and processors. First, it provides a mechanism by which non-primitive types may be specified. Second, it provides an abstract addressing scheme for allocating and referencing automatically-allocated variables. Third, it provides mechanisms for operating upon arrays, array subsets, and individual array elements. Fourth, it provides a symbolic mechanism for defining and referencing the fields of a record structure. Finally, it facilitates conditional assignment by providing mechanisms for establishing, altering and removing a Boolean mask array.

Acknowledgement

This work was funded in part by NASA grant NAG 5-3.

References

1. K. E. Batcher, "Design of a Massively Parallel Processor," IEEE Transactions on Computers, vol. C-29 (9), pp. 836-840.

2. A. P. Reeves, J. D. Bruner and T. M. Brewer, "High Level Languages for the Massively Parallel Processor," TR-EE 81-45, School of Electrical Engineering, Purdue University, W. Lafayette, IN (November 1981).

3. A. P. Reeves, J. D. Bruner, M. S. Poret, "The Programming Language Parallel PASCAL," Proceedings, International Conference on Parallel Processing, pp. 5-6, (1980).

4. A. P. Reeves and J. D. Bruner, "The Language Parallel Pascal and Other Aspects of the Massively Parallel Processor," Cornell University, Electrical Engineering technical report (1982).

5. J. D. Bruner, "Efficient Implementation of a High-Level Language on a Bit-Serial Parallel Matrix Processor," Ph.D. thesis, Purdue University (1982).

6. K. V. Nori, U. Ammann, K. Jensen, and H. Naegeli, The Pascal (P) Compiler - Implementation Notes, Institut fur Informatik, Eidgenoessische Technische Hochschule, Zurich (1975).

This paper was written under the auspices
of the U.S. Department of Energy at the
Lawrence Livermore National Laboratory,
under Contract W-7405-ENG-48.

The DC1 Flow Schema with the Data/Control-driven Evaluation

Nam Sung Woo
Ashok A. Agrawala

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

This paper introduces the DC1 flow schema, a pragmatic asynchronous parallel computation model. The disadvantages of the currently existing flow schemas are discussed. The representation and the evaluation of computation in the DC1 flow schema are described. The operations of the primitive operators are described. Two applications of the DC1 flow schema are shown. These are the representation and evaluation of the nondeterministic programs, and of the programs processing the infinite data structures. The DC1 flow schema is compared with the data flow schema and the demand-driven computation in those applications.

Keywords

flow schema, computation model,
computation representation,
computation evaluation,
DC1 flow schema, data flow schema,
demand-driven computation

1. Introduction

The von Neumann architecture has been the basis of most of the computers built to date, even though this architecture imposes several unnecessary restrictions [Back78] [Myer78]. For example, the thinking and programming are done in the ´primitive word-at-a-time´ style. Further, the evaluation of the programs has the ´von Neumann bottleneck´ between the CPU (central processing unit) and the store [Back78].

Based on the recognition, there has been some new approaches for computation. The new approaches usually use the side-effect free (functional) languages as programming languages. They also use asynchronous parallel computation models to represent and evaluate programs. One of the objectives of these computation models is to exploit the implicit concurrencies in programs. We concentrate on the asynchronous parallel computation models in this paper.

A computation model may be called a ´flow schema´. The flow schema is defined as follows: [1]

> The flow schema is an operational model of computation. It consists of a representation of computation and an interpretation which operates on the representation.

The interpretation in this definition may be called the "evaluation".

The currently existing asynchronous parallel computation models include the data flow schema [DEMi75] [AgAr82] and the demand-driven computation [KLP 79] [2]. The data flow schema can be classified as the static data flow schema and the dynamic data flow schema on the basis of enabling rule. In this paper the demand driven computation, the delayed evaluation and the lazy evaluation [Hend80] [FrWi76] are used interchangeably.

There are some drawbacks in the above existing flow schemas. The demand-driven computation is usually inefficient compared to the data flow schema. It is because the ´demand´ signal should propagate [DaKe82]. As a result, the propagation delay time is included in the execution time. Also, the amount of communication among the nodes is doubled, which results in the inefficiency of evaluation. The static data flow schema usually exploits less concurrency than the dynamic data flow schema. It is because the enabling rule of the static data flow schema is stronger than that of the dynamic data flow schema. While the dynamic data flow schema is the most efficient, it may not be safe when it evaluates the programs that process the infinite structure [ArPi82].

In this paper we propose a new flow schema, called the DC1 flow schema [3]. The representation and evaluation of computation in the DC1 flow schema are described. Some applications of the DC1 flow schema are given. They are the representation and evaluation of the nondeterministic programs, and of the programs which process infinite data structures. Through these examples it is shown that the DC1 flow schema is as efficient as the dynamic data flow schema [4].

[1] This definition is a modification of that in [Weng79]. In that paper, this term was used to define the data flow schema.

[2] Since the demand-driven computation is a standard terminology, we use it rather than the the demand flow schema.

[3] The DC1 is an acronym of Data Control flow schema version 1.

[4] For some cases such as the nondeterministic program evaluation, the DC1 flow schema is more efficient than the dynamic data flow schema as shown later in this paper.

0190-3918/83/0000/0244$01.00 © 1983 IEEE

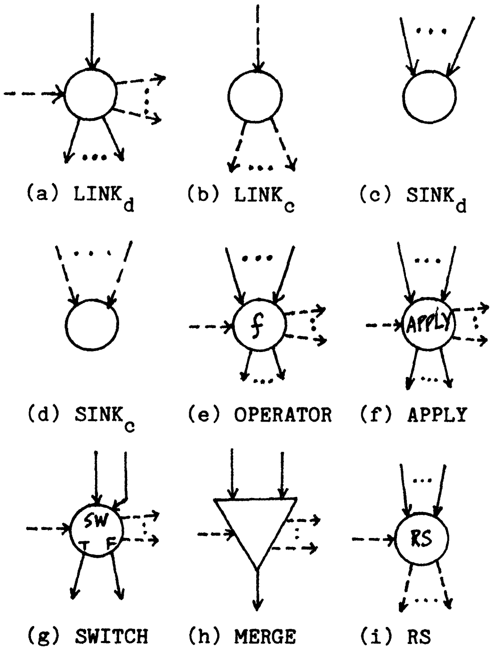
Also, it is shown that the DC1 flow schema is safe in evaluating the programs that process the infinite structures.

2. The DC1 flow schema

The computation is represented as a directed graph in the DC1 flow schema. This graph is called the "DC1 graph" in the sequel. Evaluation of the computation is done by following the two rules: enabling rule and execution rule. These are described later in this section.

2.1. The DC1 Graph : Structure and Characteristics

The DC1 graph is a directed graph consisting of a set of nodes and a set of directed arcs. The nodes in the graph are instances of the node schemas of the DC1 flow schema. These node schemas are shown in figure 2-1. The node schemas shown in figure 2-1(a),(c),(e),(f),(g) and (h) are similar to the actors in the data flow schema in [Weng79]. One difference is that every node in the DC1 flow schema may have input/output control signal arc(s) besides the data arcs. Note that the LINK node and the SINK node can handle both control signals and data in the DC1 flow schema. (See figure 2-1(b) and (d)). The RS (Random Selector) node schema, shown in figure 2-1(i), is introduced in the DC1 flow schema. It is a nondeterministic node. The operation of each type of node is described in section 2.2.3.



(a) LINK_d (b) LINK_c (c) SINK_d

(d) SINK_c (e) OPERATOR (f) APPLY

(g) SWITCH (h) MERGE (i) RS

Figure 2-1 The Node Schemas in the DC1 Flow Schema

There are two types of information representable in the DC1 graph. The first type is the data type. The data type may be integer, real, boolean, string, error, structure, etc. [5]. The value in one of the first five types is called the simple value. The structure may be represented as the stream value internally. All the variables in the DC1 flow graph is either the simple value or the stream value. The second type is the control signal type. There are two control signals of the control signal type in the DC1 graph : DELAY-ENABLE and FORCE-ENABLE. The DELAY-ENABLE control signal is used to delay the enabling of a node whose inputs are available. The DELAY-ENABLE control signal may have one of the two values : EN and NEN. The FORCE-ENABLE control signal is used to force a node to be enabled. These two control signals provide the DC1 graph with additional means for sequencing in addition to the data dependent sequencing.

In the DC1 flow graph we restrict the usage of the two control signals as follows:
(i) a node cannot have both types of control signal inputs at the same time,
(ii) a node cannot have both FORCE-ENABLE control signal input arc and data input arcs at the same time.

There are two types of arcs in the DC1 flow schema. The first type is the data arc. It transmits data tokens which contain values. It is represented as an arrow with a solid line. The second type is the control signal arc. It transmits control signal tokens which contain control signals. It is represented as an arrow with dashed line.

2.2. Interpretation in the DC1 Flow Schema

The interpretation of the DC1 graph consists of the enabling rule and the execution rule. The enabling rule governs when a node becomes enabled. The execution rule governs how an enabled node is executed. The interpretation in the DC1 flow schema is called the "DC1 evaluation".

2.2.1. The Enabling Rule of the DC1 Evaluation

An actor in the DC1 flow schema is enabled based on the following three rules.

(E1) If a node has no input control signal arc, it is enabled when all its necessary input data are available.

(E2) If a node has a DELAY-ENABLE input control signal arc, it is enabled when all its necessary input data are available and the control signal is available (regardless of the value of the delay-enable control signal).

[5] As in the Id [AGP 78], it may include procedure definition, manager definition, and manager object in the data type.

(E3) If a node has a FORCE-ENABLE input control signal arc, it is enabled if the control is available.

Note that the enabling rule for nodes without any input control signal arc is purely data dependent (by rule E1). The DELAY-ENABLE input control signal arc for a node may be used to delay the enabling of the node when its input data are available (by rule E2). The FORCE-ENABLE control signal forces a node to be enabled (by rule E3).

2.2.2. The Execution Rule of the DC1 Evaluation

A node in the DC1 flow schema is executed on the basis of the following execution rules in the DC1 evaluation:

(X1) Enabled nodes are executed concurrently or in random order.

(X2) Tokens on the input arcs are consumed before execution. After execution, tokens may be generated on the output arcs of the node.

(X3) The execution of the operation of the node is performed as follows:

(X3.a) A node without any input control signal arc : performs the operation of the node on the input data,

(X3.b) A node with the FORCE-ENABLE input control signal arc : performs the operation of the node,

(X3.c) A node with the DELAY-ENABLE input control signal arc : if the control signal value is EN, it performs the operation of the node on the input data; otherwise (i.e., control signal value is NEN) do nothing.

A node with the input FORCE-ENABLE control signal arc performs its operation which does not require any input data (see rule X3.b). Examples of this type of node include a constant generation function, which generates a constant, and a RS node, which is described later in the next section. Following the execution rule (X3.c), a node with the DELAY-ENABLE input control signal consumes token(s) but not generate one if the control signal value is NEN.

2.2.3. Operation of Nodes in the DC1 flow schema

The nodes in the DC1 flow schema can be classified as the deterministic and the nondeterministic nodes. There are two types of the nondeterministic nodes: MERGE and RS(Random Selection). The other nodes are the deterministic nodes.

The operations of the deterministic nodes are similar to those of the corresponding actors in the data flow schema in [Weng79], if we neglect the control signal arc(s) of the nodes. The operation of any deterministic nodes shall be described when it is necessary in this paper. The operations of the two nondeterministic nodes are described below.

The nondeterministic MERGE nodes in the DC1 flow schema may operate on the simple values as well as the stream values [6]. The operation of the MERGE node is to reproduce an input data token (in an input data arc) onto the output data arc as soon as one is available. This operation implies that the output of a MERGE node depends not only on the input data values but also on the time when the input data tokens arrive to the node. Thus the operation of the nondeterministic MERGE node is not a function.

A RS node which has the FORCE-ENABLE control signal arc as its input arc is called the RS_{cs} node. A RS node which has the DELAY-ENABLE control signal and/or the data arc(s) as its input arc(s) is called the RS_{dc} node. For the purpose of explanation in this paper only the operation of the RS_{cs} node is described below.

<Operation of the RS_{cs} node>

(a) selects one output control signal arc randomly among the output control signal arcs,

(b) sends a EN control signal on the selected output arc; sends a NEN control signal on each unselected output arc.

Only one output arc of the RS_{cs} node transmits the EN control signal. All the other output arcs transmit the NEN control signals.

3. The DC1 Flow Schema for the Nondeterministic Programs

A program is said to be nondeterministic if a given input state can lead to more than one possible terminal state [Gold82]. It may happen if the program is permitted to make a random choice of its next action from a number of possibilities. In this section we will describe the DC1 graph and the DC1 evaluation for the nondeterministic programs [7].

The <u>or</u> operator in [Hend80] is used to represent the random choice operation in this paper. Consider a program as follows:

[6] This is a difference between the MERGE node in the DC1 flow schema and that in the data flow schema in [ArBr82]. In [ArBr82] the MERGE actor operates only on the stream values. But it is necessary for the MERGE node to operate on the simple values in order to collect values generated mutually exclusively.

[7] In fact, these programs may be called the indeterminate programs

```
f(x) = (g_1(x) or g_2(x) or . . . or g_n(x)) <1>
```

The value of the function f may be any one of the values of the functions $g_1, \ldots, g_n$.

A DC1 graph for the program <1> is shown in figure 3-1. Each function g_i, $1 \leq i \leq n$, is represented as a node in the figure. (This representation is sufficient for the description below). There are two initial tokens; one is a data token which contains the value of x and another is a control signal token of the FORCE-ENABLE control signal to the RS_{cs} node.

This DC1 graph is evaluated as follows. Initially, only the RS_{cs} node is enabled. (by the FORCE-ENABLE control signal). Being executed this RS_{cs} node generates an EN control signal and (n-1) NEN control signals. A function, say g_i, whose input DELAY-ENABLE control signal value is EN is enabled next. Other functions absorb both the data token and the control signal token. After the function g_i completes its operation its result is transmitted as a data token from the function g_i to the MERGE node. Then, the MERGE node produces a token, whose value is the same as that of an input data token, which is the result of the graph. And the evaluation is completed.

As a more practical example, we show a DC1 graph and DC1 evaluation for the following nondeterministic program [8].

```
choice(n) = if n=1 then 1
                    else ( choice(n-1) or n )
```

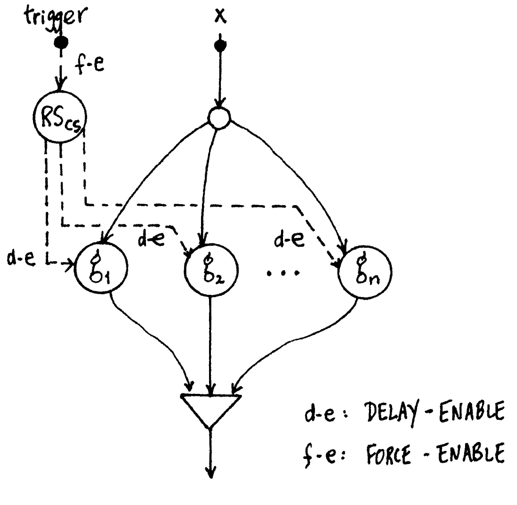


Figure 3-1 A DC1 graph for the program <1>

Given a positive integer number, the program renders an arbitrary integer ranging from 1 to the given input value.

A DC1 graph representation of the above program is shown in figure 3-2. The node EQFFE in the graph is a predicate operator which generates a data token and a control signal token. The control signal generated at the EQFFE operator is used as an input FORCE-ENABLE control signal to the RS_{cs} node. A small square box with a constant value in it in the DC1 graph represents a constant generating function. A subgraph in the inner box represents the nondeterministic selection of a value. As shown in the figure, there are two initial tokens. One is a data token which contains the value of n. Another is a data token whose value is the function definition of the ´choice´ function, which is represented as {choice}.

The operation of the APPLY node is invoking the recursion. The operation of the $LINK_d$ node is to reproduce its input data token on the output data arc(s). The operation of the EQFFE operator is described as follows:

```
compare the two data inputs;
if equal
   then generate ´true´ value
              on the output data arc;
   else generate ´false´ value
              on the output data arc;
        generate a token
              on the output control signal arc;
endif
```

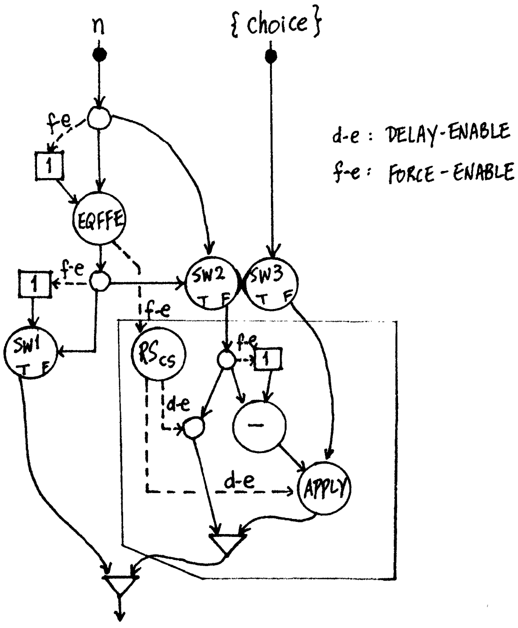


Figure 3-2 A DC1 graph for the ´choice´ program

[8] This program is from [Hend80].

The evaluation of the DC1 graph in figure 3-2 is done as follows. The first node enabled and executed is the EQFFE operator.

(1) Case 1: If the two inputs to the EQFFE operator are the same (i.e., n=1) then a 'true' value is generated as an output data token of the operator. Then, the SWITCH node SW1 passes its input value onto its output arc. And the output value of the DC1 graph becomes 1.

(2) Case 2: If the two inputs to the EQFFE operator are not the same then a 'false' value is generated as an output data value of the operator. At the same time, the EQFFE operator generates a control signal token, which forces the RS_{cs} node to be enabled. Since the SWITCH nodes SW2 and SW3 pass their input values, a value which would be chosen by the RS_{cs} node (and be passed by the MERGE node) is either n or choice(n-1) [9]. Therefore, an output of the graph is either n or choice(n-1).

From the above evaluation it is clear that the graph performs the operation specified by the program [10].

In this section, we described the application of the DC1 flow schema for the nondeterministic programs. We considered two examples of the nondeterministic program.

4. The DC1 Flow Schema for the Programs processing the Infinite Data Structures

It has been observed that the incorporation of infinite data structures into programming languages provides the programmer with a powerful tool for writing structured and elegant programs [ArPi82]. One example of the programs using the infinite data structure is the 'sieve' function which generates the prime numbers. Details of the function can be found in [Weng79], [Hend80].

However, it has been noticed that the data-driven evaluation of infinite data structures tends to usurp system resources unnecessarily [DaKe82]. Furthermore, without a mechanism of enforcing the 'fair scheduling' policy among the enabled nodes the data-driven evaluation may not be safe for programs which process infinite data structures [ArPi82] [11].

On the other hand, the demand-driven evaluation is safe in evaluation of the programs which process the infinite data structures [12]. However, in addition to the run-time overhead mentioned in section 1, there is additional run-time overhead of time and memory space to evaluate the infinite data structures processing programs. This run-time overhead results from generating the suspensions for structure construction operators and coercing them for the operators which use the structures [FrWi76].

The basic mechanism of dealing with the infinite structures in the DC1 flow schema is to control the infinite structures generating operators. An infinite structure may be generated either by iteration (i.e., infinite loop) or by recursion (i.e., infinite invocation of the recursion). Therefore, the iteration operator or the recursion invocation operator is controlled (by the DELAY-ENABLE control signal) in the DC1 flow schema to evaluate the infinite structure processing program.

In order to illustrate the DC1 flow schema for the programs processing the infinite data structures, we use the following program [13]

```
-------------------------------------------

integersfrom (m) = cons (m, integersfrom(m+1))

getfirst (k, x)
   = if k = 0
          then NIL
          else cons (car(x),
                    getfirst(k-1, cdr(x))

-------------------------------------------
```

Note that the function

getfirst (k, integersfrom(m))   <2>

yields a finite list with k elements, $k \geq 0$, although the integersfrom(m) function renders an infinite list.

A DC1 graph for the function <2> is shown in figure 4-1. It consists of a DC1 graph for the 'integersfrom' function and one for the

[9] If the choice(n-1) is selected by the RS_{cs} node, the recursion does occur.

[10] Of course, there may be a formal correctness proof for the graph. However, we will not pursue this topic in this paper.

[11] One way of enforcing the fair scheduling policy is to limit the number of tokens on each arc. This approach is adopted in the static data flow architecture in the Dennis' research group in the MIT [Denn80], [DeMi75]. In the MIT static data flow architecture each arc can contain at most one token. But enforcing the limitation requires much overhead in the execution time. It is because, in addition to the (data) tokens, the status signal (,which is called the Acknowledge signal in Dennis' group,) should also be transmitted between operators and processed. As a result, a data-driven evaluation which enforces the fair scheduling may be inefficient.

[12] In the demand-driven evaluation, the evaluation and construction of the infinite data structures are delayed and part of the infinite structures are evaluated if they are required by some other expression(s). Thus, the evaluation of the programs processing the infinite data structures is possible in the demand-driven evaluation as long as the users use a finite part of them.

[13] This program is from [Hend80].

´getfirst´ function. The FIRST and REST operators are stream operators ; i.e., operators whose input/output data are streams. The EQTNE operator generates control signal output as well as data output. Note that the output control signal arc of the EQTNE node is connected to the DELAY-ENABLE control signal input of the APPLY1 node.

The list generated by the ´integersfrom´ function is represented as a stream internally [14]. The functions of the stream operators FIRST and REST are similar to the LISP functions CAR and CDR, respectively. (Details of the operations of the FIRST and REST operators can be found in [ArTh80], [AGP 78].) The CONS operator is a stream operator, too. It generates an output data if its simple value input data is available. The operation of the EQTNE node is summarized below.

```
compare the two data inputs;
if equal
   then generate ´true´ as data output;
        generate ´NEN´ as control signal output
   else generate ´false´ as data output;
        generate ´EN´ as control signal output
endif
```

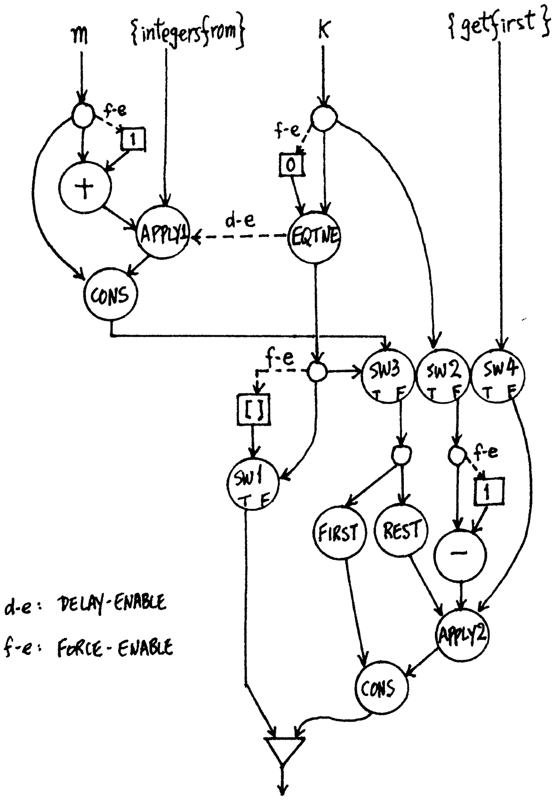


Figure 4-1. A DC1 Graph for the function <2>

The initial token distribution is shown as dots in figure 4-1(c).

The evaluation for the DC1 graph in figure 4-1(c) is as follows. Suppose that we consider the jth instance (j = 0,1,...) of the recursion of the ´getfirst´ function. The nodes enabled by the initial tokens are CONS1 and EQTNE nodes.

(1) Case 1: Assume the two data inputs to the EQTNE are equal. Then,
   (a) The SWITCH node SW1 passes its input data onto its output data arc. Thus output of the DC1 graph becomes [] (or est), which is a null stream. It is the last element of the output stream of the DC1 graph.
   (b) The APPLY1 node absorbs its control signal token as well as data tokens because the DELAY-ENABLE control signal is NEN. It prevents the APPLY1 node from being enabled further. As a consequence the elements of the infinite list are not evaluated any more.
   (c) The three SWITCH nodes SW2, SW3, and SW4 absorb input data tokens.

(2) Case 2: Assume the two data inputs to the EQTNE node are not equal. Then,
   (a) The SWITCH nodes SW2, SW3 and SW4 pass their input data onto their output data arcs.
   (b) Thus a value (m+j), which is to be the (j+1)th element in the output stream, goes through the FIRST and CONS nodes. It then becomes the output of the DC1 graph at this jth instance.
   (c) The APPLY2 node is enabled and executed so that the (j+1)th recursion of the ´getfirst´ function can occur.
   (d) Also, an EN of the DELAY-ENABLE control signal allows the APPLY1 node to cause recursion of the ´integersfrom´ function. As a consequence, the next element of the infinite list generated by the ´integersfrom´ function will be available in the (j+1) th recursion instance of the ´getfirst´ function.

Note that the infinite list is not evaluated and constructed at one time by the ´integersfrom´ function. Rather, part of the infinite list is evaluated incrementally. In fact, each element of the infinite list is evaluated at each instance of the recursion of the ´getfirst´ function. Also, the generating and consuming operations of an element of the list (i.e., stream) is interleaved so that more concurrency can be achieved. As is clear from the above description, the DC1 evaluation evaluates (k+1) elements of the infinite list of integers which starts from m. The first k elements are passed as outputs of the DC1 graph. The last one element is

[14] This stream representation allows the concurrent operations in manipulating the list [Weng79].

absorbed by the SWITCH node SW3 at the kth instance of recursion of the ′getfirst′ function.

5. Concluding Remarks

In this paper we proposed the DC1 flow schema, a pragmatic parallel asynchronous computation model. It contains two kinds of control signals as well as the data values. It is an asynchronous evaluation scheme and it can exploit the implicit concurrency of the computation. Its sequencing is based on both the control signal and the data availability.

The properties of the DC1 flow schema are studied elsewhere [Woo 83]. There it has been proved that the DC1 flow schema is determinate.

The application of the DC1 flow schema to the nondeterministic programs is shown in this paper. If the (pure) data-driven approach is used to evaluate the program <1>, all of the n functions would be performed. It is because the input, which is x, to every function is available. One of the results of all functions is randomly chosen as the result of the program. Now, let us compare these two flow schemas for the general nondeterministic program in <1>. In the DC1 flow schema only one function, which is chosen by the EN value of the DELAY-ENABLE control signal, out of the n functions is evaluated. On the other hand, in the data flow schema, all of the n functions are evaluated. As a consequence, in general, the DC1 flow schema performs the nondeterministic programs more efficiently (using less time and hardware resources) than the data flow schema. In addition, the DC1 flow schema has better convergence property in the evaluation of the nondeterministic program. The reason is as follows. The function f in the program <1> diverges in the data-driven evaluation if any one of the n functions diverges. However, the function f converges in the DC1 evaluation even though there are some diverging functions among the n functions, if those diverging functions are not selected by the RS_{cs} node.

The application of the DC1 flow schema for the infinite structure processing program is discussed in this paper. It is shown that the DC1 flow schema is safe in evaluation of the programs which process the infinite data structures. Therefore, the DC1 evaluation has advantage over a data-driven evaluation. Furthermore, the DC1 evaluation need not any run-time overhead to evaluate those programs. Thus the DC1 evaluation has advantage over the demand-driven evaluation, since the latter requires the run-time overhead as discussed above.

The technique of controlling the infinite structure constructing operator can also be used to make the for each - while loop self-cleaning [AGP78]. Detailed description about this can be found in [Woo 83].

It is noted that the computer architecture for the DC1 flow schema may be similar to the data flow architecture except several (minor) modifications [Woo 83]. Building a simulator for the DC1 flow schema is planned.

Acknowledgement

This research was supported in part by the National Science Foundation Grant MCS 81-12896 at the University of Maryland, College Park.

References

[AgAr82]
Agerwala T., Arvind, "Data Flow Systems," Computer, Feb. 1982, pp. 10-13

[AGP 78]
Arvind, Gostelow K., Plouffe W., An Asynchronous Programming Language and Computing Machine, Dept. of Information and Computer Science, U.C. Irvine, Dec. 1978. Revised June 1980.

[ArBr82]
Arvind, Brock J., Streams and Managers, Computation Structures Group Memo 217, MIT, June 1982.

[ArPr82]
Arvind, Pingali K., Safe Data driven Evaluation, Laboratory for Computer Science, MIT, April 1982

[ArTh80]
Arvind, Thomas R., I-Structures : An Efficient Data Structure for Functional Languages, MIT/LCS/TM-78, MIT, September 1980, revised in October 1981.

[Back78]
Backus J., "Can Programming Be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs," CACM, Vol.21 No.8, August 1978, pp.613-641

[Dake82]
Davis A., Keller R., "Data Flow Program Graphs," Computer, Feb. 1982, pp. 26-41

[DeMi75]
Dennis J., Misunas D., "A Preliminary Architecture for a Basic Data-Flow Processor," The 2nd Annual Symposium on Computer Architecture, 1975, pp. 126-132

[Denn80]
Dennis J., "Data Flow Supercomputers," Computer, November 1980, pp. 48-56

[FrWi76]
Friedman D., Wise D., "CONS Should Not Evaluate Its Arguments," Michaelson S., Milner R., eds. Automata, Languages and Programming, Edinburg Univ. Press 1976, pp.257-284

[Gold82]
Axiomatising the Logic of Computer Programming, Goldblatt R., Lecture Notes in Computer Science 130, Springler-Verlag, 1982

[Hend80]
Functional Programming : Application and Implementation, Henderson P., Prentice-Hall, 1980

[KLP 79]
Keller R., Lindstrom G., Patil S., "A loosely-coupled applicative multiprocessing system," AFIPS Conference Proceedings, Vol.48,, 1979, pp. 613-621

[Myer78]
Myers G., Advances in Computer Architecture, John Wiley & Sons, 1978

[Weng79]
Weng K., An Abstract Implementation for a Generalized Data Flow Language, Ph.D. thesis, MIT/LCS/TR-228, MIT, 1979

[Woo 83]
Woo N., Ph.D. dissertation, in preparation.

TOP-DOWN DATA FLOW PROGRAMMING

Yury Litvin

GTE Laboratories, Inc.
Waltham, MA 02254

Abstract -- A simple control mechanism for non-functional data flow languages is described. It is based on hierarchical structuring of data flow programs. The hierarchy is described metaphorically using the notion of "execution time" for program statements and blocks. It is postulated that statements within a block are executed "infinitely" faster than hierarchically superior statements. Resulting program structure corresponds to the intuitive top-down structure of the program. A parallel language based on this principle would bridge the gap between correct and "correctly" structured programs.

1. Introduction

The advantages of the top-down structuring of programs and the style of programming by stepwise refinement hardly need to be reiterated. In sequential von Neumann programs, top-down structure is more a matter of style: generally speaking it may not be required by the semantics of the programming language. In this brief paper I attempt to define simple semantics for a non-functional data flow language. The analysis of this problem immediately shows that one or another form of hierarchical program structure is necessary. Thus, top-down programming is optional for sequential programs, but is essential for high-level data flow programs. In Section 2 I discuss briefly why this is so, and in Section 3 I describe a data flow control mechanism based on the hierarchical program structure. The hierarchy is implemented using the metaphor of "execution time." It is postulated that statements within a program block are executed "infinitely" faster than hierarchically superior statements. A parallel language based on this principle would bridge the gap between correct and "correctly" structured programs. The problems of a plausible syntax for such a language are not addressed in this paper. A (rather naive) attempt at language definition including syntax is described in [7].

2. Two Components of Determinism

For abstract analysis it is convenient to view computations as relaxation in a transition system. I use the term "relaxation" to refer to a sequence of transitions leading from a given state to a terminal state. For our purposes I use the following definition of determinacy: the system is called deterministic if for any initial state all relaxations from this state lead to the same terminal state. Asynchronous data flow computations are possible due to the determinacy of computation graphs [10]. In general, in data-driven systems determinacy depends on two conditions:

1. Availability of arguments: all designated inputs for a function or an activity must be available before the computation can proceed.
2. Preservation of results: all results must be made available to their "users" before they are overwritten or erased.

The first condition is the essence of data-driven computations. There is some freedom, however, in the choice of a formalism to satisfy the second "preservation-of-results" condition. In data flow graphs, for example, it can be done by
(a) providing FIFO queues on all arcs;
(b) allowing a node to fire only when all output arcs are vacant;
(c) explicit scheduling or partial scheduling of activities and their durations;
(d) setting synchronized "locks" at the entrances of computation blocks (iterative loops, in particular) so that each arc is enabled (carries a value) only once for each cycle through the "locks".
Theoretically any one of these mechanisms could be used as a basis for defining the semantics of a high-level data flow language. But (a) and (b) would result in a totally incomprehensible programming logic, and (c) would contradict the spirit of asynchronous computations.

Option (d) is successfully implemented in the ID [3] and other languages [6]. This approach is based on hierarchical structuring of programs. A chunk of a program designated for repetitive use (e.g. an iterative loop) must be defined formally as a block with certain inputs and outputs. The inputs are synchronized through a chain of synchronized locks, so that only one datum can enter through each input arc before all output arcs receive the results.

3. Designing a Procedural Data Flow Language

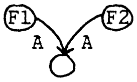
Painful experiences with software development and maintenance stimulated efforts to impose severe restrictions on languages. The freedom of programmers was curtailed by the elimination of global variables, GOTO statements and other drastic measures. The most radical approach advocates pure functional languages without side effects [4]. These efforts have resulted in dramatic improvements in programming productivity. The impetus of restrictive measures was to force some order and structure into conventional von Neumann programs. It is not obvious, however, that the same strict measures must be applied to data flow programs. First, as discussed earlier, proper definition of semantics of a data flow language may automatically require proper structuring of programs. Second, attempts at functional or "value-oriented" programming languages for data flow computers, such as VAL [2,9], are not free from difficulties and compromises, especially with regard to handling iterations and history-sensitive computations [8]. Third, object-oriented programming offers a viable alternative to value-oriented programming: its semantics may be more comprehensible for a mathematically-naive programmer.

0190-3918/83/0000/0252$01.00 © 1983 IEEE

A detailed comparison of functional vs "procedural" data flow languages is beyond the scope of this paper. Our objective is merely to show that a "procedural" data flow language is possible. Note that pure functional programs are top-down structured in a natural way -- [5] provides an example. A comprehensive review of properties and constraints for data flow languages can be found in [1].

My intention is to define simple semantics for a language with as few restrictions as possible. The language should allow global variables, partial execution of blocks under certain conditions, loosening of the single assignment rule and same variable on both sides of the assignment statement, etc. More restrictive semantics or programming styles can be superimposed on this basic language. The price paid for this freedom is the necessity to introduce the metaphor of stepwise computation with external timing pulses signalling the beginning of each cycle.

The task of defining language semantics includes two subtasks. The first is to represent data flow graphs in the textual form, the second is to define a control mechanism. The first subtask is straightforward. Each node of the data flow graph is more than simply a node: it corresponds to a function with some ordered set of arguments and one or several results. We can assign names to the arcs of the graph (several arcs may get the same name). The graph then can be represented as a set of nodes with labelled "receptacles" and "transmitters", and then translated into a list of statements (Figure 1). That is, of course, where the data flow graphs originally came from.

The single assignment rule requires that arcs with the same name must come out of the same node. This makes connections of the type 

illegal. It may be convenient in some cases to relax this constraint and permit such connections with the understanding that nodes F1 and F2 can fire only under mutually exclusive conditions, but not simultaneously. This, of course, would place additional burden of avoiding conflicts on the programmer.

The second subtask is to define the control mechanism. It would be unwise to use conventional data flow control, which requires that a node can fire only when all output arcs are free. This would violate the principle of locality, since intermediate results can be "consumed" in remote places and times. Another alternative, as mentioned earlier, is explicit scheduling. The control mechanism proposed here is a compromise. It is based on the metaphor of "execution time" for statements. It is postulated that each statement takes "the same time" for execution. Operations proceed in a stepwise manner and are synchronized with an imaginary external timing pulse. All enabled (possessing all arguments) statements are executed at each cycle, and all the arguments of executed statements are consumed by the end of the cycle. Suppose we have an iterative loop, and computations in the body of the loop take more than one cycle:

```
I:=I+1
X:=F1(I)
Y:=F2(X)
```

How can we synchronize the increment operator I:=I+1 (which fires at each cycle) with the body of the loop? To accomplish this we combine the statements of the body into a program block and postulate that the "execution time" for the block is the same as for peer elemental statements:

```
I:=I+1
[ X:=F1(I)
  Y:=F2(X) ]
```

The statements within the block then must be executed faster than outside statements. In fact, they must operate "as fast as necessary" or "infinitely" faster to keep pace with hierarchically superior statements. The computation within the block is a relaxation from the initial state -- variables defined at the beginning of the (external) cycle -- to a terminal state where all statements in the block are disabled. The whole process takes exactly one cycle when viewed from the next superior level of the hierarchy. Figure 2 illustrates a data flow program with this control mechanism. For simplicity, only one level of nested blocks is used in this example. At the upper level, statements X:=X+DX, PRINT(X,Y) and the block are executed at each cycle. At the lower level, computations within the block take 6 "subcycles".

4. Conclusions

Simple semantics for a data flow language have been described. The objective was to define semantics of the data flow language with as few constraints as possible: to allow global variables, partial execution of blocks and relaxation of the single assignment rule. The control mechanism is based on the "execution time" hierarchy. It turns out that for a properly programmed algorithm the hierarchical "execution-time" structure closely corresponds to the intuitive top-down structure. Therefore the proposed control mechanism bridges the gap between correct and "correctly" structured programming. Hierarchical structuring also permits structured debugging. The programmer can monitor program execution step by step at any given level in the hierarchy.

References

[1] Ackerman, W.B. "Data Flow Languages." MIT Laboratory for Computer Science, Computation Structures Group, Memo 177-1, May 1979.

[2] Ackerman, W.B., J.B.Dennis. "VAL - A Value-Oriented Algorithmic Language: Preliminary Reference Manual." MIT Laboratory for Computer Science, LCS/TR-278, June 1979.

[3] Arvind, K.P.Gostelow, W.Plouffe. "An Asynchronous Programming Language and Computing Machine." Dept. of Information and Computer Science, Univ of California, Irvine, California, TR 114-a, Dec 1978.

[4] Backus, J. "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs." Comm. ACM, 21, No.8, Aug 1978, pp. 613-641

[5] De Jong M.D, C.L.Hankin. "Structured Data Flow Programming," ACM SIGPLAN Notices, 17, No. 8, Aug 1982, pp. 18-27

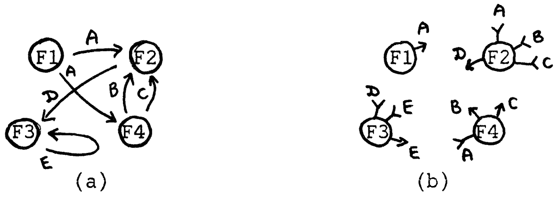
[6] Kotov, V.E. "On Basic Parallel Language." Information Processing 1980. Proceedings of the IFIP Congress 80, Oct 1980, Tokyo, Japan, pp. 229-240

[7] Litvin, Y. "Parallel Evolution Programming Language for Data Flow Machines." ACM SIGPLAN Notices, 17, No.11, Nov 1982, pp. 50-58

[8] McGraw, J.R. "Data Flow Computing: Software Development." IEEE Trans. Computers, C-29, No.12, Dec 1980, pp. 1095-1103

[9] McGraw, J.R. "The VAL Language: Description and Analysis." ACM Trans. on Programming Languages and Systems, 4, No.1, 1982, pp. 44-82

[10] Miller, R.E. "A Comparison of Some Theoretical Models of Parallel Computations." IEEE Trans. Computers, C-22, No.8 Aug 1973, pp. 710-717



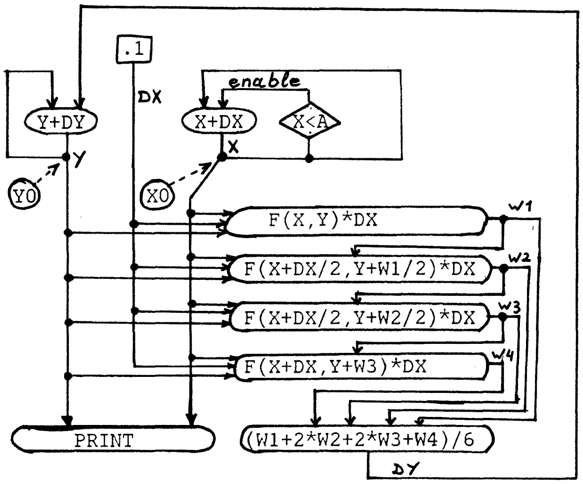
(a)

(b)

```
    A:=F1
    D:=F2(A,B,C)
    E:=F3(D,E)
(B,C):=F4(A)
```

(c)

Figure 1
"Receptor/transmitter" and textual representations of directed graphs: (a) arc labelling; (b) "receptor/transmitter" representation; (c) textual representation.



```
          PROGRAM RUNGE_KUTTA(X0,Y0)

CONSTANT DX=.1, A=1.    % Initialization section
X:=X0
Y:=Y0
=============================================
X:=X+DX | X<A           % Execution section
[ Y:=Y+DY
  DY:=(W1+2*W2+2*W3+W4)/6.
  W1:=F(X,Y)*DX
  W2:=F(X+DX/2.,Y+W1/2.)*DX
  W3:=F(X+DX/2.,Y+W2/2.)*DX
  W4:=F(X+DX,Y+W3)*DX  ]
PRINT X,Y
=============================================
```

% Comments:

This program implements one of the Runge-Kutta methods for numerical solution of a differential equation. The initialization section serves to define constants and initialize variables. The program is executed according to the "execution time" hierarchy: all statements within the block enclosed in brackets are executed "infinitely" faster than statements outside the block. This hierarchy is used to synchronize X:=X+DX and Y:=Y+DY.

===

Figure 2
A data flow graph and a corresponding program with the "execution-time" hierarchical structure.

A PIPELINE MACHINE FOR IMAGE PROCESSING APPLICATIONS

Ikram E. Abdou
Aydin Computer Systems
Fort Washington, PA 19034

Abstract—The design of a general purpose image processing machine is currently one of the most active research areas. This paper introduces a pipeline machine designed for such applications. The architecture of the machine can be described as a simplified form of a data flow machine. This new design combines the advantages of the fast throughput needed for image processing applications with a simple and easy-to-implement instruction set. To achieve these normally mutually exclusive features, the machine is based on a pipeline architecture which is configurable at execution time. This allows the user to include only the hardware components which are needed for a given application. The instruction fields are selected accordingly, so the user is only concerned with the fields of the components he has included. The design is optimized for array or vector manipulation. The paper begins with a description of some of the image processing algorithms. The hardware components needed to implement each algorithm are combined to form a basic data processing module. A brief description of the microcode used in the machine is given. The paper concludes with a summary of the important features of this new design.

Introduction

Recently, there has been an intensified effort toward the design of general purpose image processing machines. A survey of these efforts is given in reference [1]. Most of the existing machines are similar in the fact that they are based on fast multipliers and rely on pipeline processing to achieve high throughput. A major disadvantage of these machines is the complicated programming procedures that are normally needed to execute the desired functions. The following sections introduce a signal processing machine that combines the advantages of the fast throughput needed for image processing applications, with a simple and easy-to-implement instruction set.

Since the machine is tailored specifically for image processing applications, the data is represented using a block floating point format. This format provides a sufficient accuracy for most image applications, while allowing a fast and simple data manipulation. The paper begins with a survey of the various image processing functions. Based on these functions, the necessary hardware elements are defined. The microcode used in the machine is introduced, and some of the basic features of the machine are discussed briefly.

Survey of Image Processing Functions

This section surveys the various functions needed in any image processing machine. For a better understanding of these functions, the reader should consult references [2], [3], and [4].

Digital images are large arrays of data. Processing these images requires the implementation of functions which can be classified into one of the following groups:
1) Simple functions of a single variable.
2) Simple functions of two variables.
3) Compound functions of a single variable.
4) Compound functions of multiple variables.
In the following, a discussion of each group is given.

Group 1, Simple Functions of a Single Variable

These functions are defined as follows: Given an input array F, the output array G is given by

$$G = T(F). \qquad (1)$$

Where T is any linear or nonlinear mapping function. The mapping function is implemented through a Look-Up Table (LUT), stored in a Programmable Read-Only Memory (PROM), or down loaded into a Random Access Memory (RAM).

Some of the single variable functions used in image processing are intensity mapping, contrast enhancement, and pseudo coloring.

Group 2, Simple Functions of Two Variables

The functions of interest in this group are defined as follows: Given two input arrays F1 and F2, the output array G is given by

$$G = O(F1,F2), \qquad (2)$$

where O is any dyadic operator. The most important of these operators are the logical and arithmetic functions. An Arithmetic Logic Unit (ALU) is needed to implement the functions: Add, Subtract, OR, AND, and Exclusive-OR. Multiplication is implemented using a special hardware such as the TRW Multiplier to achieve the required speed performance. Division is implemented using lookup tables.

The functions of two variables are used to compare two images, to modify one image according to a spatial function described by another image, or in general to combine two input images into one output image.

Group 3, Compound Functions of a Single Variable

One important function that belongs to this group is the histogram calculation. The histogram is a function showing for each possible value of an input array F, the number of elements that have this value. To calculate the histogram, the array F is stored in a RAM and used to address the

0190-3918/83/0000/0255$01.00 © 1983 IEEE

cumulative count in another RAM. Every time an element is addressed it is incremented by one, and the incremented value replaces the original value.

The histogram is used to determine the proper mapping function for contrast enhancement. It is also used in pattern recognition.

Group 4, Compound Functions of Multiple Variables

Image processing functions that belong to this group are normally compute bound. An efficient implementation of these algorithms is a measure of the performance of any image processing machine. Some of the important functions that belong to this group are:

Sum of an Array. Given an input array, f1,..., fn, the output g is defined as

$$g = f1 + f2 + ... + fn. \qquad (3)$$

This function is implemented using an ALU with Accumulator. It is used to minify an input image by averaging. Also, it is used to average more than one image for noise cleaning.

Sum of Products. Given a set of weighting coefficients, a1,..., an, the output g is defined as

$$g = a1*f1 + a2*f2 + ... + an*fn. \qquad (4)$$

This function is implemented using a Multiplier/ Accumulator Unit. The sum of products is used in convolution, recursive filter, and interpolation.

Data Processing Module

Combining the hardware components needed for the various image processing functions, produces the data processing module shown in Figure 1. The machine consists of the following elements:

1) Two large Random Access Memory units RAMA and RAMB, used to store the processed data and the coefficients.
2) A small register file RAMF, for temporary storage.
3) A Read Only Memory PLUT, that stores the inverse, sine, cosine, and square root tables.
4) Two Arithmetic Logic Units ALUA and ALUB. In addition to the normal ALU functions, these two units have the capability to accumulate the sum of an array.
5) A Multiplier/ Accumulator Unit, MAC.
6) A Scaler Unit that reduces a 35-bit multiplier output into a 16-bit number.
7) A third Arithmetic Logic Unit, ALUC. This unit has a divide-by-two capability.

The connections between the various units in this machine are determined by the 4-way multiplexers at the inputs of RAMA, RAMB, RAMF, Multiplier, and ALUC. As an example, if this machine is used to multiply two images, the data path is configured such that the outputs of RAMA and RAMB are connected to the multiplier input. The output of the Scaler is connected to either RAMA or RAMB. In this application, RAMF, PLUT, and the three ALUs are not included in the data path. If in another application the machine is used to add two images, the multiplexers are controlled such that the outputs of RAMA and RAMB are connected to ALUC and the output of ALUC is written back to either RAMA or RAMB. Other applications may require more than one configuration of the machine. In these cases, all the processing that belongs to one machine configuration is finished before the machine is reconfigured.

Introduction to Microcode

The instruction used in this machine, is divided into the following eight fields:

Field	Function	Time
1	Sequence controller.	0 t
2	Address generator control.	1 t
3	Read RAMA, RAMB, RAMF, and PLUT.	3 t
4	ALUA and ALUB operations.	4 t
5	Multiplier operation.	5 t
6	Scaler operation.	7 t
7	ALUC operation.	9 t
8	Write RAMA, RAMB, and RAMF.	10 t

Table 1. Machine Instruction Fields.

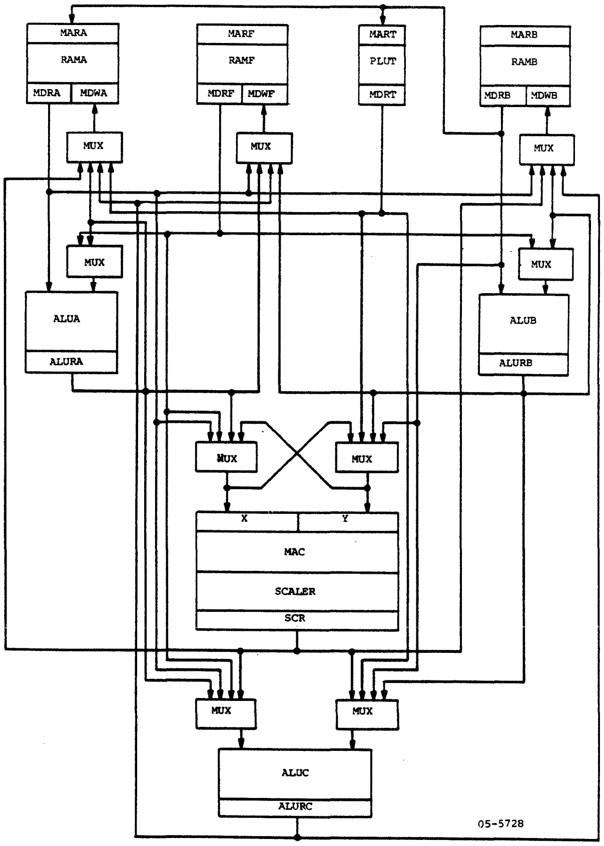


Figure 1. Data Processing Module.

Field 1 determines the sequence of instruction being executed. It controls the increment, repeat, or branching of the instruction sequence. Fields 2 through 8 control the address generation and processing of data. The elements of the instruction resemble those of any microcoded machine. The most important diference between this design and many existing machines is the time delays shown in the right column of the table. Executing a single instruction is not done in one machine cycle. It is spread over many cycles with the various fields being executed at the relative delays given in the above table. These delays allow the instruction to follow the data it has originally created, through the read command, till this data is written into RAM. The ability of ALUA, ALUB, and the MAC unit to accumulate data, allows the user to store the intermediate results of a summation in the machine registers. Thus, reducing the need to access the RAM units during processing. This feature is important in many image processing applications such as convolution and edge detection. It should be noted that the machine is a special case of the data flow machine, in which the data is always synchronized with the instruction, and all the data processing elements operate at a fixed rate. The general case of the data flow machine is discussed in reference [5].

Also, because of the pipelined nature of the machine, a new instruction can be started every machine cycle. After the initial time needed to fill the data pipe , all the included components of the machine will be simultaneously processing different data elements. The overhead needed to fill the pipe is usually a very small percentage of the total processing time. At execution time, the instruction is modified to resemble the hardware configuration being implemented. This modification is achieved through a set-up procedure that is executed once at the beginning of every new configuration. As an example, if the machine is configured for convolution, the set-up procedure controls the instruction flow such that only the fields related to convolution are included. These fields are the following:

Field	Function	Time
1	Sequence controller.	0 t
2	Address generator control.	1 t
3	Read RAMA and RAMB.	3 t
5	Multiplier operation.	4 t
6	Scaler operation.	6 t
8	Write RAMA or RAMB.	8 t

Table 2. Instruction Fields for Convolution.

In this case, the relative delay is determined only by the components being included in the configuration.

The microcode needed for convolution cosists of three instructions: The first is Read and Multiply instruction; the second is Read, Multiply, and Accumulate instruction; the third is Read, Multiply, Accumulate, and Write instruction. The most important feature of this code is its compactness. This feature simplifies both the coding and the debugging effort. The compactness of the code is a result of the data flow nature of the machine which minimizes the interaction between various instructions.

Conclusion

In this paper, a general purpose image processing machine was introduced. The machine is based on a pipelined architecture with the various machine elements connected through multiplexers. This structure offers independent data paths, thus preventing the bus contention problem. In addition these multiplexers are used to configure the machine at execution time. Therefore only the hardware needed for a given application is included, and the data follow the shortest possible path. The instruction fields are modified accordingly. These fields control the processing sequence starting with reading the data; then modifying it using the ALUs, the Multiplier, and the Scaler units; and finally writing it back. The structure can be described as a synchronous data flow machine.

These features produce a machine that is as fast as any specialized image processing machine, while maintaining the flexibility of a general purpose array processor. In addition, the machine is capable of handling many image processing applications in both the spatial and the frequency domains. Another advantage of the machine is its simple and compact code, that makes the machine easy to program.

Acknowledgements

I wish to thank the management at Aydin. Also, I would like to thank the dedicated team of hardware and software engineers who are currently working on this project.

References

[1] P. Danielsson and S. Levialdi, "Computer Architectures for Pictorial Information Systems," Computer (November, 1981), pp 53-67.

[2] W.K. Pratt, Digital Image Processing, Wiley Interscience, (1978), 750 pp.

[3] E.L. Hall, Coputer Image Processing and Recognition, Academic Press, (1979), 584 pp.

[4] K.R. Castleman, Digital Image Processing, Prentice-Hall, (1979), 429 pp.

[5] Special Issue on Data Flow Systems, Computer (February, 1982),

AN EVALUATION STUDY OF SIX TOPOLOGIES OF PARALLEL COMPUTER ARCHITECTURES FOR SCENE MATCHING

Yee-Hong Yang
Mellon Institute
Computer Engineering Center
4616 Henry Street
Pittsburgh, PA 15213

Tsung-Wei Sze
Department of Electrical Engineering
University of Pittsburgh
Pittsburgh, PA 15261

ABSTRACT

Six topologies of parallel computer architectures were evaluated for implementation of scene matching. These topologies are: (1) cluster-connected, (2) mesh-connected, (3) cluster-cluster-connected, (4) cluster-mesh-connected, (5) mesh-cluster-connected, and (6) mesh-mesh-connected. Analytic results indicate that the mesh-connected architecture is the best candidate. However, under the constraint of the physical pin-out number, the mesh-mesh-connected is a good alternative.

1. Parallel Computer Architectures

During the past twenty years, a significant number of high speed computer architectures have been proposed. Some of these architectures that were thought to be impractical to build are now considered not only viable but also necessary for speeding up computation. This is made possible by the advancement of solid state technology. Signal/Image processing requires a large number of computations. This is particularly true for scene matching algorithms. Scene matching is the process of locating a subimage in a sensed image using a template [Hall,79]. Scene matching algorithms have potential applications in cruise guidance missles [Berry,80]. A limitation to their use is in the intense computational requirements. One of the advantages that all scene matching algorithms have is that independence of computations exist in the algorithms. This independency offers opportunities for concurrency and thus favors the use of innovative parallel computer architectures for speeding up the processing. A tradeoff study which will be discussed in the following can show the advantages and the disadvantages of six promising computer architectures with respect to the implementation of a scene matching algorithm. The analysis parallels the technique used by Feather et al.[Feather,80].

2. Six Promising Computer Architectures

Six types of architectures were analysed, namely, (i) a cluster-connected architecture (CCA), (ii) a mesh-connected architecture (MCA), (iii) a cluster-cluster-connected architecture (CCCA), (iv) a cluster-mesh-connected architecture (CMCA), (v) a mesh-cluster-connected architecture (MCCA), and (vi) a mesh-mesh-connected architecture (MMCA). Schematic diagrams of these six architectures can be seen in Figure 1(a)-1(f).

2.1. Analysis of the CCA

A CCA (Cluster-Connected Architecture) is an architecture having a common bus. Each processing element (PE) has its own local memory. The PEs are controlled by a central controller. In the following analysis, we will demonstrate the limitation to this approach.

Let the sensed image be of size $2^L x 2^L$ while the template be of size $2^m x 2^m$. Let the number of processing elements (PEs) be $2^k x 2^k$. Let the image be partitioned such that each PE occupies a subimage of size $2^{(L-k)} x 2^{(L-k)}$. In the following analysis, it is assumed that the sensed image is surrounded by a boundary of width 2^{m-1}. Hence no special processing is required by PEs located on the boundary partitions of a sensed image. It is assumed that the template information has already been stored in the local memory of each PE. Therefore, no time is required to transmit template information to individual PEs. In order for each PE to perform a scene matching algorithm, it has to acquire a subimage of size $(2^{(L-k)}+2^m)^2$ of which $2^{2(L-k)}$ is already in its local memory.

It can be shown that the total time required to obtain the best match is:

$$T_{CCA}(L,k,m)=t_{xfr}(2^{2k}(2^{(L-k+m+1)}+2^{2m})$$
$$+t_c(2^{2(L-k+m)}+2^{2k})$$

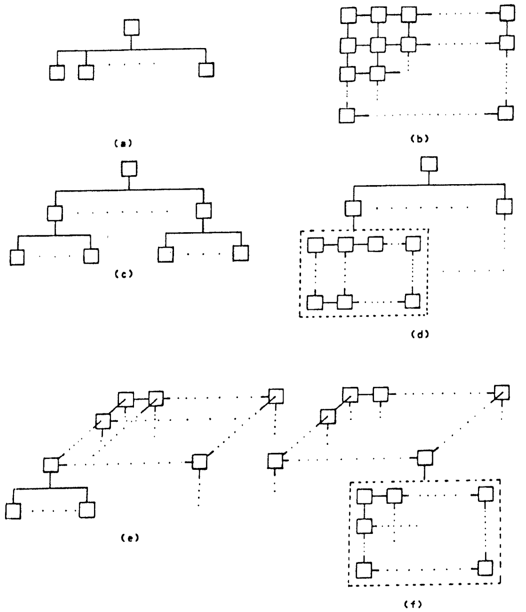


Figure 1 Vaious architectures for analysis.
(a) Cluster-Connected Architecture,
(b) Mesh-Connected Architecture,
(c) Cluster-Cluster-Connected Architecture,
(d) Cluster-Mesh-Connected Architecture,
(e) Mesh-Cluster-Connected Architecture and
(f) Mesh-Mesh-Connected Architecture.

Suppose the system is implemented using current technolgy, typical values for t_{xfr} and t_c are respectively, 0.4 microseconds and 2 microseconds.

Let us define the speed up factor S as the ratio of the time to perform the computation sequentially (T_s) over the time to perform it in parallel. Then:

$$S_{CCA}(L,k,m)=\frac{T_s}{T_{CCA}}.$$

Using the above values of t_{xfr} and t_c, the variation of $\log_{10}S_{CCA}$ is plotted versus k with L=10 and m=5 (see Figure 2). On the same figure, the speed up S per PE, which is defined as the PE utilization factor F, is also plotted.

The results can be interpreted physically as follows. When k is small, that is, when the number of PEs is small, the time spent is mostly on computation. When k increases, S will increase because of the decrease in computation required by each PE. However, when k reaches a certain value, the time required to align the data becomes dominant and hence causes S to decrease. The implication of this observation is important. Suppose the size of a template is large, one might want to increase the number of PEs hoping to decrease the computation time. The computation time decreases. However, the overhead is so dominating that the net effect will degrade the performance of the whole system. This is very crucial because the system is not extendible to meet unpredictable needs. Furthermore, on the same figure, one sees that the PE utilization factor F decreases significantly after k=6. This indicates that the PEs are not utilized efficiently.

2.2. Analysis of the MCA

The proposal for this kind of architecture was first given by Unger [Unger,54]. Over the past twenty nine years, surprisingly, only a few computer systems having this architecture were actually built. The main reason for such

This work was supported by the United States Department of Defense, contract number N00039-80-C-0640, and monitored by the Naval Air Development Center.

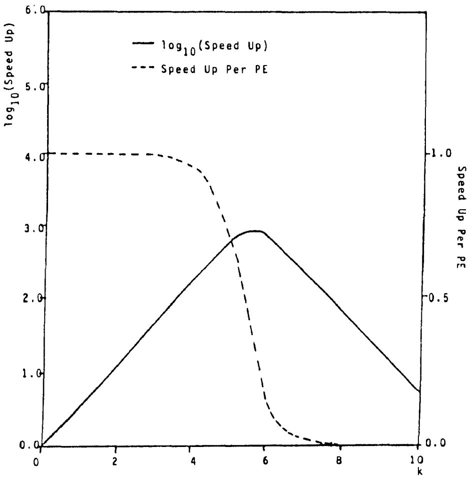


Figure 2 Performance analysis of Cluster-Connected Architecture.

a delay can be attributed to the complexity and cost involved in building any mesh-connected computer system. Representatives are ILLIAC IV [Barnes,78] DAP by ICL [Hunt,79], CLIPP 4 [Preston,79], and MPP [Batcher,80]. Among them, the DAP the CLIPP 4 and the MPP are recently built machines.

A similar analysis shows that the time required to locate the best match in this architecture is:

$$T_{MCA}(L,k,m)=(2^{L-k+m-1}+2^{2m}+2k)t_{xfr}$$
$$+(2^{2(L-k+m)}+2k)t_{c}$$

$$S_{MCA}=\frac{T_s}{T_{MCA}}$$

Using the same values of t_{xfr} and t_c, the variation of S_{MCA} and F_{MCA} with k can be seen in Figure 3. It can be seen that $\log_{10}S_{MCA}$ increases linearly with k while the PE utilization factor F remains relatively constant from k=0 to k=8. When k>8, F_{MCA} decreases because of the overhead in aligning the data.

In the following, we will analyze four derivations of the above architectures.

2.3. Analysis of the CCCA

A schematic diagram of the CCCA (Cluster-Cluster-Connected Architecture) can be seen in Figure 1(c). In this architecture, it is assumed that there are two levels of clusters. Each cluster at the first level has $2^p x 2^p$ PEs while each cluster at the second level has $2^q x 2^q$ PEs. The advantage of this interconnection is that the communication load at the first level is reduced. A representative of this architecture is the CM* [Swan,77].

Using similar analyses, it can be shown that the time to complete the scene matching algorithm is:

$$T_{CCCA}(L,p,q,m)=\left|\begin{array}{c} t_{xfr}(2^{L+m+p+1}+2^{2m+2p} \\ +2^{L+m-p+q+1}+2^{2m+2q}+2^{2p}+2^{2q}) \\ +t_c(2^{2(L+m-p-q)}+2^{2p}+2^{2q}) \end{array}\right|$$

And hence the speed up S:

$$S_{CCCA}(L,p,q,m)=\frac{T_s}{T_{CCCA}}$$

Since S_{CCCA} varies with both p and q. We assume that for a given p, q can be chosen to maximize S_{CCCA}. With this assumption, the variation of S_{CCCA} and the PE utilization factor F_{CCCA} with respect to p can be seen in Figure 4. It can be seen that S_{CCCA} has a similar trend as the S_{CCA}.

The variation of F_{CCCA} with respect to p is more difficult to interpret physically. However, one can conclude that when p>2, there is no good way to utilize the PEs efficiently using this architecture. In the above figure, the analysis is valid up to p=9 because when p=10, it is not meaningful to have a second level of PEs.

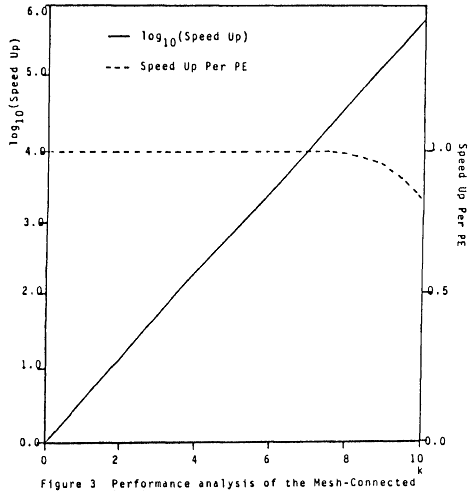


Figure 3 Performance analysis of the Mesh-Connected Architecture.

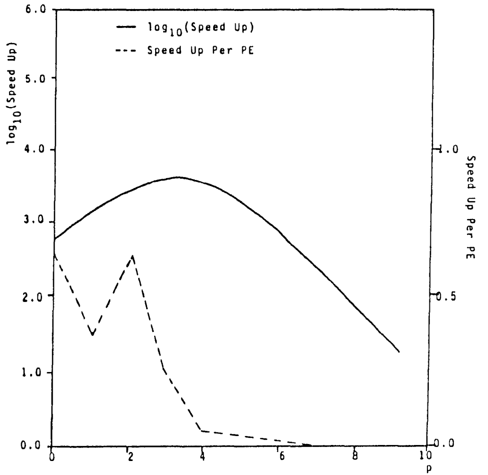


Figure 4 Performance analysis of the Cluster-Cluster-Connected Architecture.

2.4. Analysis of the CMCA

One may think that by changing the second level of architecture to a MCA then a better performance may be attained. The following analysis shows that this expectation is completely incorrect. By repeating the above analysis, one obtains:

$$T_{CMCA}(L,p,q,m)=\left|\begin{array}{l} t_{xfr}(2^{L+m+p+1}+2^{2m+2p} \\ 2^{L+m-p+q+1}+2^{2m}+2^{2p}+2q) \\ +t_c(2^{2(L+m-p-q)}+2^{2p}+2q) \end{array}\right|$$

Then:

$$S_{CMCA}=\frac{T_s}{T_{CMCA}}$$

The variation of S_{CMCA} with respect to p can be seen in Figure 5. It can be seen that S_{CMCA} decreases monotonically with increasing p and the PE utilization factor F_{CMCA} is very low. Therefore, this rearrangement is not a suitable one.

2.5. Analysis of the MCCA

It is interesting to see the effect of interchanging the architectures in the two levels, i.e. having the MCA at the first level and a CCA at the second level. A schematic diagram of this topology can be seen in Figure 1(e). With similar analysis, one can deduce that:

$$S_{MCCA}=\frac{T_s}{\left|\begin{array}{c} t_{xfr}(2^{L+m-p+1}+2^{2m}+2^{L+m-p+q+1} \\ +2^{2m+2q}+2^{2q}+2p)+t_c(2^{2(L-p-q+m)}+2^{2q}) \end{array}\right|}$$

We again assume that for a given p, the best q is selected

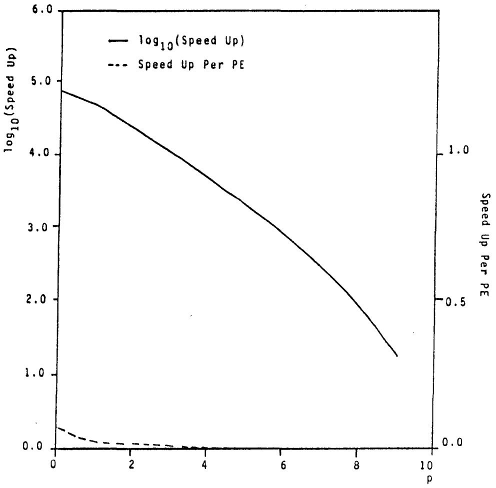


Figure 5 Performance analysis of the Cluster-Mesh-Connected Architecture.

to maximize the speed up S. Then the variation of S versus p can be seen in Figure 6.

It is surprising to see that by interchanging the architectures of the two levels, a significant change can be observed in performance. It can be seen that S_{MCCA} varies linearly with p. It has a different slope than the S_{MCA}. For some combinations of p and q, the F_{MCCA} is higher than that of other combinations of p and q. This fact can be used in the actual design of a MCCA machine to fine tune its performance.

2.6. Analysis of the MMCA

The final architecture that will be analyzed is the MMCA (Mesh-Mesh-Connected Architecture). A schematic diagram of this architecture is shown in Figure 1(f). The S_{MMCA} can be shown to be:

$$S_{MMCA}=\frac{T_s}{t_{xfr}(2^{L+m-p+1}+2^{2m}+2^{L+m-p-q+1}) +2^{2m}+2p+2q)+t_c(2^{2(L+m-p-q)}+2p+2q)}$$

The variation of $\log_{10}S_{MMCA}$ and F_{MMCA} with p can be seen in Figure 7. It can be seen that the speed up is relatively constant. This can be attributed to the fact that at the second level, a lot of concurrency is already made use of. In order to maximize S at the second level, each PE at the second level has only one pixel to perform the scene matching algorithm. The peak of F can be used to fine tune the system for maximum utilization. It can be seen from the figure that the peak occurs when p=7. In other words, q has to be equal to 3. This result has an interesting implication and application which will be discussed in the following.

3. Discussion

It is known that with current VLSI technology one of the major problems is the packaging problem. The pin-out-number problem restricts the designer to limit the amount of input information to a chip at any specific time. Therefore, one of the solutions is to partition a design such that a maximum amount of functionality can be implemented on the chip without increasing inter-chip communication. Suppose an MCA machine is to be built; it is obvious that one cannot put 1024x1024 PEs onto a single chip. Even if IC technology permits it, the packaging technology will impose severe limitations on the way communications are to be made with the outside world. However, by breaking a MCA machine into two levels, either a MCCA or a MMCA topology, the implementation problem will then be more feasible. For instance, using the MMCA, with q=3, i.e. by putting 8x8 PEs on one chip and by interconnecting 128x128 such chips to form a MMCA machine, the overall performance is comparable with a strict MCA machine.

From the above analysis, we can conclude that the MCA is the best candidate for implementing scene matching. The second choice will then be either a MMCA or a MCCA topology.

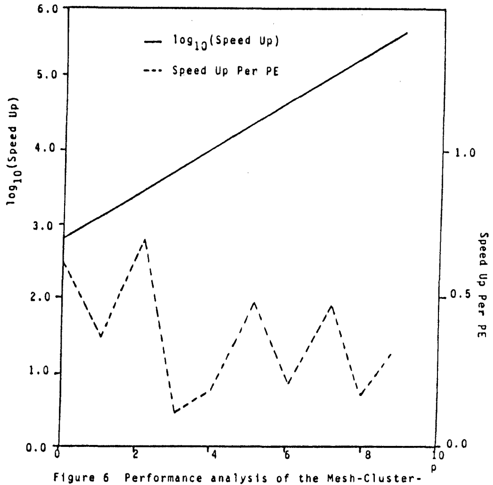


Figure 6 Performance analysis of the Mesh-Cluster-Connected Architecture.

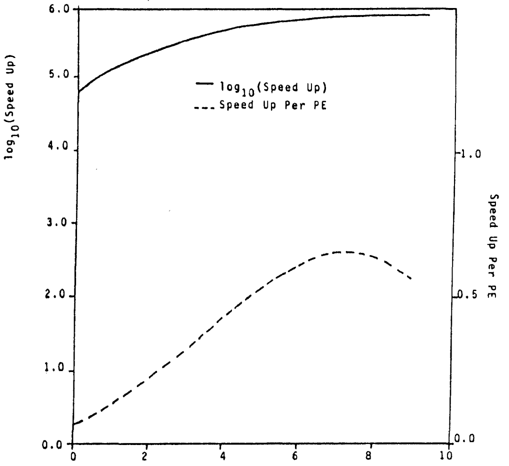


Figure 7 Performance analysis of the Mesh-Mesh-Connected Architecture.

ACKNOWLEDGEMENT

The authors would like to thank Dr. Ronald L. Krutz and Mrs. Lorna Cheng of the Computer Engineering Center, and Mr. John Lawler of the Naval Air Development Center for their support and encouragement throughout the project.

BIBLIOGRAPHY

Barnes, G.H. et al., "The ILLIAC IV Computer," IEEE Trans. C-17, 1968, pp. 746-757.

Batcher, K.E., "Design of a Massively Parallel Processor," IEEE Trans. C-29, 1980, pp. 836-840.

Berry, J.E., "Three-Dimensional Autonomous Scene Matching," in Proc. SPIE, Vol. 238, 1980, pp.138-145.

Feather, A.E., L.J. Siegel and H.J. Siegel, "Image Correlation Using Parallel Processing," 5th Inter. Conf. on Pattern Recognition, 1980, pp. 503-509.

Hall, E.L., Computer Pattern Recognition and Image Processing, Academic Press, 1979.

Hunt, D.J., " Application Techniques for Parallel Hardware," Infotech State of the Art Report on Supercomputer, Vol. 2 1979.

Preston, K., et al., "Basics of Cellular Logic with Some Application in Medical Image Processing," Proc. IEEE, Vol. 67, 1979, pp. 826-856.

Swan, R.J., S.H. Fuller and D. Siewiorek, "Cm* - A Modular Multimicroprocessor," AFIPS Conf. Proc., Vol. 46, 1977, NCC, pp. 637-644.

Unger, S.H., "A Computer Oriented Toward Spatial Problems," Proc. of the IRE, October 1958, pp. 1744-1750.

AN ARCHITECTURE FOR
EFFICIENT GENERATION OF FRACTAL SURFACES

Stephen L. Stepoway
David L. Wells
Gerald R. Kane
Department of Computer Science and Engineering
Southern Methodist University
Dallas, TEXAS 75275

Abstract -- Fractal surfaces have been shown to be a useful model for generating images of terrain in computer graphics. Unfortunately, the generation of fractal images is very costly in CPU time. A multi-processor architecture is described which takes advantage of the parallelism inherant in fractals to speed the generation of images. The performance of the processing array is analyzed along with the suitability of implementation in VLSI.

Introduction

Modeling Objects in Computer Graphics

One challenging problem in computer graphics is the generation of images which closely resemble objects in the real world. Images are acceptable if they are readily recognizable and are not obviously synthesized, that is if they might be objects in the real world. Many approaches, ranging from describing objects by a collection of planar polygons to describing them by Bezier [Bezi74] or B-spline [Gord74] surface patches, have been used. Although these techniques have proved adequate for modeling artificial objects, most natural objects (such as clouds, terrain, mountains, and the like) have few regular features and no simple detail. The results of using these techniques are usually "obviously synthetic," and are not sufficiently realistic to be indistinguishable from a scene in nature.

Other techniques have been proposed for modeling natural objects which partially avoid this problem. Objects are sometimes modeled by a collection of polygons with some texture mapped upon them. This technique suffers from the fact that the texture on the polygons tends to have a great deal of regularity, and thus objects appear artificial.

Another approach is to use sufficient detail in the model to make the scene appear realistic with no added texture needed. Although this will result in realistic scenes, it requires very large databases containing sufficient data to display a realistic scene from any viewpoint at the maximum desired level of resolution.

All of these techniques for modeling natural objects have a similar drawback: they define an object at a single, pre-determined level of detail. Little computational advantage is gained from displaying the scene as seen from afar, as the detail needed to display a close-in view must be stored regardless of the detail desired in the displayed view. In addition, these techniques do not allow going arbitrarily close to an object, since views which require more detail than the model contains will present the same problems as with the previous techniques in which the model had little detail.

A common feature of most computer models of objects, and one of the significant drawbacks of the techniques previously described, is that the models are defined in terms of deterministic functions. There have been a few exceptions, however. Mezei et al. [Meze74] generated textures and shapes by using random techniques, and Blinn [Blin77] has improved shading methods by using a model based upon probabilistic assumptions.

Stochastic Models

A more general technique for the definition of natural objects is to use a random, or stochastic, process to generate detail on surfaces [Four80, Four82]. This idea is inherent in the concept of a fractal surface [Mand77, Mand82]. The concept of a fractal surface has been shown to be an accurate model for many processes in nature [Four80, Four82]. In particular, terrain and weather phenomena may be represented by a stochastic, fractal process. Images synthesized using these techniques are impressive for their resemblance to the real world they purport to represent.

In this method of modeling surfaces, a basic model is constructed using traditional techniques, such as defining a surface in terms of polygons. The stochastic modeling method adds detail to this basic surface definition by recursively breaking the polygons into smaller, but slightly non-coplanar, polygons. This method of modeling

0190-3918/83/0000/0261$01.00 © 1983 IEEE

surfaces has several advantages over traditional techniques.

First, the definition of an object is simple. Complex terrain may often be modeled by only a few dozen polygons. The realism is added to this definition by the stochastic process.

Second, the object is not defined at a pre-determined level of resolution. If additional resolution is needed for a particular scene, it may easily be generated by continuing the recursive texture generation algorithm to a greater depth. Thus, moving the viewpoint closer to an object or further away from it is easily handled. Because of this ability to generate texture to whatever level of resolution desired, this technique has the significant advantage of requiring computational effort proportional to the complexity of the image on the screen. It is not necessary to compute or store texture which is not needed for the resolution desired.

Unfortunately, even though the effort required is proportional to the on-screen image complexity, the generation of a fractal surface is still very expensive, with scenes of moderate complexity requiring between 30 minutes and several hours of CPU time for a single frame.

Because of the complexity of generating images, fractal surfaces cannot now be used in applications which require real-time generation of images. Although their use in simulators would make the trainee's view of his "world" more realistic, the computational effort required would be prohibitive.

Even in non-real-time applications, such as the generation of motion pictures, the use of fractal surfaces is limited. To generate motion pictures using these techniques is, for the most part, infeasible, due to the cost of generating thousands of such frames. A few organizations do have the necessary computational power to generate movies using these techniques, and the results have been impressive, for example the movie "Star Trek II", but the use of these techniques remains severely limited.

Architectures for Graphics Computation

To make possible the real-time use of fractal surfaces, or the large scale generation of motion pictures using fractal generated terrain, it clearly is necessary to find a method of generating fractal surfaces more quickly than is now possible. One possible approach to this problem is to find multi-processor architectures which are suited to the efficient generation of fractal surfaces.

We describe a new multi-processor architecture, suitable for implementation in VLSI, which is capable of generating fractal surfaces much more rapidly than existing software fractal systems. Preliminary estimates indicate a reduction in the time required to generate a frame by at least two orders of magnitude, and perhaps a sufficient reduction to allow the generation of fractal images in real-time.

Although a number of experiments with multi-processor based systems for computer graphics have been conducted, none have investigated their use in generating fractal textures for surfaces. Moreover, the architectures which have been proposed have been based upon rectangular grids. This is reasonable for most applications in graphics, since objects tend to be defined in rectangular coordinate systems, and since the eventual display is, in almost all cases, a rectangular screen.

The generation of fractal surfaces is most suited, however, not to rectangular grids, but to hexagonal ones. This is primarily due to the way in which a fractalization algorithm decomposes a surface (which we will define in terms of triangles) to generate the fractal surface. The proposed architecture arranges processing elements on a hexagonal array, with interprocess communication between adjacent processors and across certain diagonals. The interconnection network, although complex, does describe a planar graph. Thus the architecture may be implemented easily in VLSI.

The regularity of the processor connections also gives the advantage of easy expandability. A fractal processor may be increased in power and speed by interconnecting several processing units in the appropriate manner. Thus the computational power of a fractal processor may easily be tailored to the application.

In the next section, we will describe fractal surfaces and the techniques used for generating them. We then will show how a multi-processor architecture can be developed which takes advantage of the regularity of fractal surfaces to generate fractal surfaces efficiently. Finally, we will examine some of the performance issues associated with this multi-processor architecture, and discuss some of the preliminary results of our performance studies.

Fractal Surfaces

Generation of Fractal Surfaces

The generation of a fractal surface usually starts with a coarse description of the object in terms of triangular polygons*. Only the general shape of an object usually needs to be specified, since all detail is added by the texturing algorithm.

The triangular definition of the object is broken into smaller triangles which are slightly non-co-planar. These smaller triangles are then broken into even smaller triangles, with the subdivision process continuing until the triangles generated are no larger than a pixel in size. At that point, the pixel-size triangles may be painted onto the screen. The color selection for each pixel is based upon the color of the original triangle and its orientation using conventional techniques.

The processing of an object definition is most easily described in terms of a recursive procedure:

* Although surfaces intended for fractalization may be described by quadrilaterals as well, we will restrict our inquiry to triangular definitions.

```
procedure fractalize( triangle );
begin
  if the triangle is less than a pixel in size
    then paint the triangle onto the screen
    else
      begin
        for each edge of the triangle do
          begin
            find the mid-point of the edge
            pick a point a random distance from the
                edge midpoint in the direction of
                the normal of the triangle*
          end
        connect the three displaced midpoints
        connect each displaced midpoint with the
            vertices of the original triangle
            adjacent to it
        fractalize each of the resulting triangles
      end
  end
begin
  for every triangle t in the object definition do
      fractalize( t )
end
```

This procedure recursively subdivides each triangle in the object definition into smaller triangles, all approximately similar, but slightly non-coplanar. Thus the procedure is replacing the object definition with one which is progressively more textured.

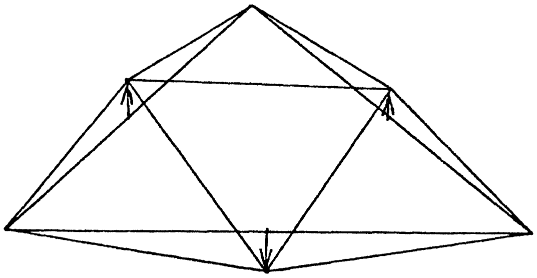


Figure 1
Subdivision of a triangle

Anomalies in Fractal Surfaces

This algorithm, although simple, has an unfortunate property: it generates fractal surfaces with anomalies. If the fractalization process is carried out on two adjacent triangles independently, there is no guarantee that the fractal surfaces will meet along the common edge that the triangles share. In fact, using this technique, it is guaranteed that the surfaces will not meet, since the displacements for each have been made parallel to their respective normals. This kind of anomaly may be seen in Figure 2.

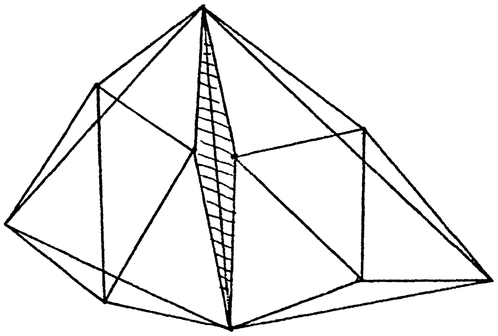


Figure 2
Anomalies in triangular fractals

A solution to this problem is to require that the displacements for the new mid-point of each edge be in the direction of the average of the normals of the two triangles which share that edge. If, in addition, it can be guaranteed that the distance of the displacement will be the same for each of the two triangles, the fractal surfaces resulting from the two triangles will match, and no gaps or other anomalies will result. For an example of how this technique works, see Figure 3. The vector indicated from the midpoint of the common edge is in the direction of the average of the surface normals of the two triangles.

These requirements, although sufficient to guarantee that fractal surfaces are free from anomalies, do pose difficult problems for the implementation of a software fractal system. These conditions must be made to hold without imposing prohibitive memory or computational requirements. These difficulties are avoided in our fractal architecture by allowing the processors to communicate and to come to a mutual agreement concerning the size and direction of the displacements.

Using the techniques described here, fractal surfaces may be generated for arbitrary surfaces defined using triangles. The surfaces are guaranteed to have no gaps or other anomalies.

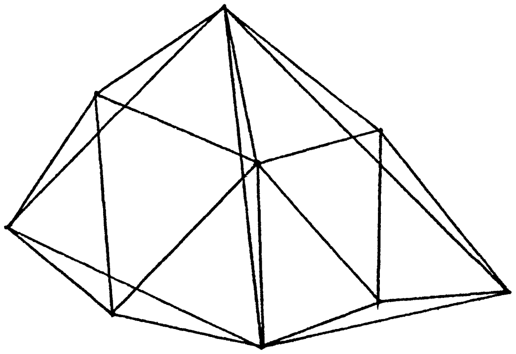


Figure 3
Subdivision of two triangles

* Or in a direction related to the normal - see below.

An Architecture for Generating Fractal Surfaces

Introduction

A study of the generation of fractal surfaces will reveal that a great deal of the work may be done in parallel if an appropriate multi-processor architecture is used. The multi-processor architecture we propose will gain its speed from assigning one processor to each triangle to be subdivided. Once a triangle has been split, the processor responsible for that triangle activates three new processors, assigns one of the generated triangles to each of them, and processes the remaining triangle itself. Each of the four processors then subdivides its triangle in the same manner used by the initial processor to subdivide the first triangle. Thus the number of processors active is equal to the number of triangles created by the division process, and the number of processors required to completely fractalize one triangle in an object definition is equal to the number of pixels that triangle will cover on the screen.

Although the number of processors required is large, the individual processors are very simple. Using the increasing densities of VLSI technologies, it may be possible to place several hundred processors on a single custom chip. Thus a fractal generator of several thousand processors may be constructed using a small number of chips.

Geometry of a Fractal Processor

Triangles to be processes come in many shapes, but for the purpose of subdividing them into smaller triangles their shape is immaterial. Since the fractalization process automatically stops subdivision when pixel-sized triangles are generated, the fact that a triangle is not equilateral will not be of concern. Only the topology of the triangles matters, and that is regular. Thus the geometry of a multi-processor fractal generator is regular. All processors reside at the intersections of a regular hexagonal grid, with communication paths along the grid and across certain diagonals of the grid.

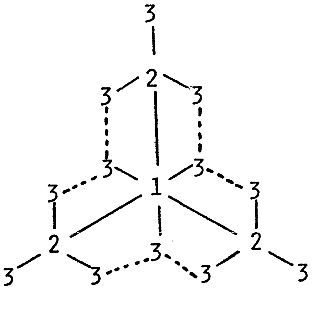


Figure 4
Processor arrangement

The arrangement of processors for the first three levels of the fractalization process is shown in Figure 4. The central processor, numbered 1, initiates the processing of the triangle. It subdivides the triangle, and spawns the three processors numbered 2. These processors and the original processor in turn subdivide their triangles and spawn the processors numbered 3.

In addition to the parent-child communication paths shown by solid arcs, it is necessary for certain "cousins" to communicate to ensure that the triangles they generate will meet along their common edge. Thus, in addition, there must be the communication paths shown by dashed arcs. Although communication paths may form the diagonals of hexagons, it is never necessary for them to cross; the graph described by the processor interconnections is planar.

If we examine the pattern of processors to a greater depth, we see a potential problem. For example, consider the pattern of the children of processor 1 to a depth of 4, as shown in Figure 5. Some processors at level 4 lie on the communication path from 1 to its level 2 children. If it was necessary for 1 to communicate with both a level 2 child and a level 4 child at the same time, communication could become a serious problem, since these communication paths run in the same directions.

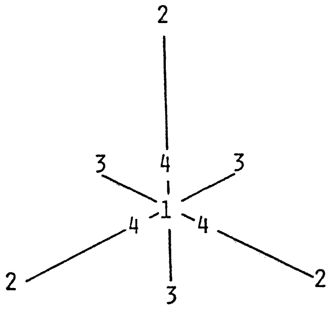


Figure 5
Children of processor 1

If the inter-processor communication needs at the various levels are examined, is seen that this case does not arise. If multiple communications are required along a single ray, the paths are completely disjoint. We will require that the communication paths be between adjacent processors only, and that messages which must travel a greater distance, such as 1 to 2, be relayed by each processor along the path.

Processor Operation

Since the operations performed by all processors are the same at any point in time, the processor array within a single chip may be run synchronously, with all processors running in lock-step. If an application requires a processing array of more than one chip, the speed of the communication lines between chips may require that the different chips run asynchronously.

We will consider the processors to be in either of two phases of computation*: subdivision and broadcast. During the subdivision phase, each

* Three phases, if we include reading out the final data; see below.

processor subdivides the triangle it is currently working on. There is some interprocessor communication involved since processors sharing an edge must ensure that they pick the same point as the new mid-point. This communication is only between processors which have no active processors between them.

To ensure that no anomalies are generated, it must be possible to guarantee that the two processors subdividing adjacent triangles will choose a displacement in the direction of the average of the two normals. This may be accomplished by having each compute the normal to its triangle and send its computations to its neighbor, and then having both processors average the results of its computations and its neighbor's. At the end of this sequence, both processors have computed the average of the normals.

If only single frames are to be generated, this exchange of normals is sufficient to remove all anomalies. If motion pictures are to be generated, however, it must be possible to guarantee that, in successive frames of the film, the direction and distance of the displacements will be the same. If not, the features introduced by the fractal process will change from frame to frame, and the surface will move as a result. This problem may be removed by having the seed of the random number generator be the coordinates in object space of the midpoint of the line to be divided. Thus the random numbers are independent of eye position. It can in this way be guaranteed that the same surface will result every time the object is processed.

The second phase of computation is the broadcast phase, when newly created triangles are sent to the processors which will subdivide them. This communication may be done by observing that the distance that a triangle definition must be sent to reach the processor which will be resposible for it is uniform at a given level. We may then use a broadcast phase which consists of the appropriate number of "relay-triangle" pulses to all processors. If each processor relays the messages in the appropriate direction, the triangle definition will reach the correct processor when the last "relay" pulse arrives. The array then returns to the subdivision phase, during which the new allocation of triangles is broken up.

This sequence of subdivision followed by reassignment continues until all of the created triangles are less than a pixel in size. This may be detected by having each processing element send to its parent a message when it finds that the triangle it is to subdivide is pixel-sized. Once the parent has received such a message from each of its children, and its own triangle is pixel sized, it sends a similar message to its parent. Thus the central processor (level 1 processor) can be notified that no more subdivision is required, and that the data is ready to be extracted. At this point, the data describing the fractal triangles may be used to determine shading for the corresponding point on the screen. To accomplish this, some method must be available to extract the data describing the position and orientation of the fractal triangles from the processors of the fractal processor.

A solution to the problem of extracting data from the array is to notice that communication paths already exist between the processing elements. If the existing paths can be used, the complexity of the processor can be substantially reduced.

We will therefore add a data extraction phase to the processing. All data computed can be extracted by simply reversing the direction of flow of the existing data paths. Details of this technique will be described later.

In use, the host will load a single triangle definition into the central element (level **1**) of the fractal processor. The host then issues a command to the array that processing should commence. The fractal processor then runs independently of the host until processing of that triangle is complete. The fractal processor then issues a signal to the host indicating that, and the data extraction commences.

Performance

Restrictions and Assumptions

Since the goal of this work is to develop a processor architecture capable of generating fractal surfaces at high speed, we must pay considerable attention to the question of how fast this proposed architecture actually can run. Although exact numbers cannot be predicted without a great deal of experimentation, some estimates of performance can be made.

Before making these estimates, we must first make some assumptions which will simplify the task of estimation. These concern whether the fractal processor must be capable of performing floating point arithmetic or only integer, and the number of processing elements in the array and its relationship to the size of triangles to be processed.

The goals of the arithmetic system used are two-fold, and somewhat contradictory: speed and accuracy of computation. That speed is required is obvious if pictures are to be generated quickly. Accuracy is also required if errors induced by the number system are not to appear in the final pictures.

Although floating point numbers have good accuracy and can represent a wide range of numbers simultaneously, they suffer from severe performance problems. Because of this, floating point numbers are not appropriate for this system.

Integer number systems and arithmetic units, on the other hand, are much faster and simpler than floating point units, and are thus much more suited to use in this system. An integer number system of 24 to 32 bit numbers can handle the computations required.

If the triangles to be processed are too large, the fractal processor will be unable to completely decompose them before running out of processors. If this occurs, the fractal processor must stop the subdivision, unload all triangle definitions from all processing elements, and restart the array on those triangles, one-by-one. Not only does this eliminate the parallelism of processing triangles simultaneously, it also imposes the considerable overhead of unloading the processor. If the requirements for processors are examined, we find that this can be guaranteed not to arise if the triangles sent to the processor meet the requirement that:

log(longest side of the triangle)
    <= depth of the processor array

where the size of the triangle is measured in pixels, and the log is taken base 2.

Prior to sending any triangle to the processing array, the host machine may apply this test and subdivide the original triangle into several smaller ones which meet this requirement. Since this is being done in the host, it may be done in parallel with the decomposition of other triangles by the fractal processor. Thus this requirement will not impose a serious burden upon the system.

Given this restriction that guarantees that the fractal processor will be capable of subdividing a triangle into pixel-sized triangles without interruption, the time required to completely subdivide one triangle from the object definition is bounded by

depth * (subdivision + broadcast) + readout

where depth is the depth of the processor array, subdivide is the time required for one triangle subdivision, broadcast is the time for one triangle relay, and readout is the time to extract the final information from the processor. We now must examine each of these time requirements in detail.

Subdivision Phase

The subdivision of a triangle requires that each processing element compute the normal to its triangle, the average of the normal of its triangle with the normal of adjacent triangles, the midpoints of each edge, and three random numbers.

The cost of the random numbers we will ignore, but should not be significant compared to the other costs of subdividing the triangle. Since the random numbers must be a function of the position of the midpoint (to ensure repeatability, as mentioned earlier), they may be computed by a combinatorial circuit using the <x, y, z> object space coordinates of the midpoint as inputs. Although such numbers may not meet strict mathematical tests for randomness, they are sufficient for our purposes.

The processor then exchanges the normal and random number information with each of its three neighbors. The processor averages its normal with the normals received from each of its neighbors, and averages the random numbers to determine the distance of displacement for each of the new midpoints. The new midpoints then are computed. Once this is accomplished, the processor only needs to send the computed information to the appropriate processors for further subdivision.

Many of the operations required to subdivide a triangle may be performed in parallel. If this is done, and the hardware has the needed parallelism to perform computations on edges simultaneously, the number of serial operations required to subdivide a triangle is 6 multiplications, 22 additions and subtractions, and 10 shifts. The time required to perform these operations depends heavily upon the technology used to implement them, but reasonable estimates can be made based upon commercially available micro-processors and other devices, and are shown in Table 1.

	MOS	Bipolar	ECL
Add	1 micro-sec.	0.3 micro-sec.	0.1 micro-sec.
Shift	1 micro-sec.	0.3 micro-sec.	0.1 micro-sec.
Mult.	9 micro-sec.	3.5 micro-sec.	1.0 micro-sec.

Table 1
Performance of Integrated Circuits

Using these estimates, it can be seen that the task of subdividing a triangle can be performed in less than 86 micro-seconds. To be pessimistic in the computation of an order-of-magnitude figure, we will assume this step requires 100 micro-seconds using MOS technology (35 micro-seconds using Bipolar, or 10 micro-seconds using ECL).

Broadcast Phase

The time required to broadcast the triangle definitions to the processors which are to subdivide them is not easily computed. The primary difficulty is that the time required depends upon the inter-node communication technique. If processing nodes must communicate via a serial data path, the time requirements will be substantial; if the communication is via a parallel, or even via mixed serial-parallel, the overhead of broadcast may be greatly reduced. To get estimates of this requirement, we will examine several possible communication techniques. Only after we have some estimates of the node-to-node commmunication times can we examine the total time required to send triangle definitions to their destinations, which in general will not be the adjacent processing element.

Serial Links. If serial links are used, all data defining the triangle to be subdivided must be sent in serial fashion over a single line between processing elements. Each triangle to be sent requires the sending of three points, each of which consists of three values. If the values are defined by 32 bit integers, the transmission of a single triangle will require 288 bits transmitted. Although each processing element must send triangle definitions to three children, this can be done in parallel, if maximum hardware parallelism is used. If we assume that on-chip serial communication links can transmit information at a rate of 10M baud, the transmission of a triangle definition from one processor to the adjacent one will take 28.8 micro-seconds.

Mixed Serial-Parallel Links. Using this technique, several lines run in parallel between adjacent processing elements. The data to be transmitted is sent in several stages, with pieces, for example 8 bit wide sections, sent in parallel. Although the same number of bits must be transmitted, this technique will reduce the number of transfer cycles from 288, with pure serial, to 36. Thus this method will reduce the node-to-node transfer time by a factor of 8 to 3.6 micro-seconds. Since speed is critical, this may be worth the extra communication lines involved.

Parallel Links. Due to the number of bits to be transmitted a pure parallel transmission system is impractical. This would require 288 lines between each pair of adjacent processing elements. It may be possible, however, to have integer-wide communication paths, capable of transfering 32 bits in a single transfer cycle. This technique would require only 9 cycles to transfer one triangle definition, and thus would take only 0.9 micro-seconds, but would be extremely expensive in terms of communication lines (and thus in chip area).

Multi-Chip Implementations. Although the mixed serial-parallel scheme may be possible for communication between processing elements on a single chip, attempting to transfer data between elements on separate chips using it would be impossible. Such chip-to-chip transfers require the transmission of data between many pairs of processing elements (see Figure 6); the pin-outs required would be tremendous. For that matter, the mixed serial-parallel technique requires too many lines for transmitting data between chips. Thus the pure serial technique is probably the only one feasibible, and that at a substantially reduced bandwidth.

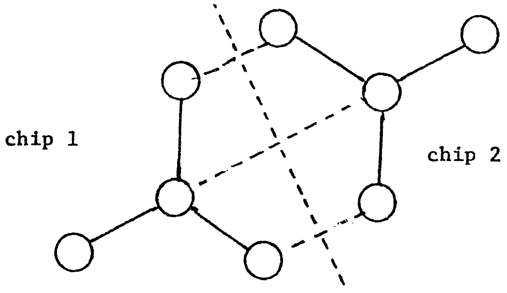


Figure 6
Chip-to-chip Communication

Distance of Transmission

The appropriate processor to subdivide a triangle may be several nodes away along the communication path. It is necessary for triangle definitions to be relayed by elements along the path from the generator of a triangle to the processor which is to subdivide it. The time required to broadcast triangle definitions to the appropriate elements is the product:

number of relays * time per relay.

The number of relays required decreases as a processing proceeds since the processor which should subdivide a triangle is closer to the processor which generated it. For the purposes of a coarse estimate, however, the depth of the processing array is a bound for the number of relays.

Extraction Phase

The final crucial stage in generating a fractal surface is to extract the computed data from the fractal processor array. This may be accomplished by reversing the direction of data flow from that used during the computation stage of processing.

Instead of triangle definition information being sent from a level i processor to a level i+1 processor, the computed data is sent from level i+1 elements to level i processors. The level i node then sends the data on to its parent, until the triangle definition reaches the level 1 element, which sends the data on to the host cpu for display. The time required for the readout phase is bounded by:

#elements in the array * time for one transfer.

As an estimate, we may assume that the time required for one transfer is the same as that required for a relay of a triangle definition. Although this technique is slow, it has the virtue of simplicity. All the needed communication paths exist, and the added complexity to each processor is slight.

Overall Performance

Using the estimates developed in the previous several sections, and noting that the number of elements in an array is 4**(depth - 1), the time required to generate and extract data for a single triangle from the definition is:

```
depth * (subdivision + depth * relay) +
  4**(depth - 1) * relay.
```

Using our estimate of the time to relay a triangle definition using the mixed serial-parallel interface, the total time required to process one triangle from the object definition to pixel sized triangles may be computed. Some of these times are summarized in Table 2 for processing arrays of various sizes and for various implementation technologies. It should be noted that a speed-up by a factor of three has been assumed for relay times for Bipolar, and a factor of three speed-up over that for ECL.

Depth =	4	6	8
Pixels Covered =	64	1024	16,384
MOS	0.688 msec.	4.416 msec.	60.012 msec.
Bipolar	0.236 msec.	1.482 msec.	20.017 msec.
ECL	0.072 msec.	0.484 msec.	6.659 msec.

Table 2
Object Triangle Fractalization Times

It may be noted that, for increased depths, the time required for transfer of data out of the array becomes the primary delay. Although this may be reduced by using a more complex data extraction technique, the limiting factor for the speed at which the data may be extracted is the speed of the host, not of the fractal processor. The fractal processor is capable of supplying data at rates exceeding that at which the host can compute shading and place the data into the frame-buffer.

Conclusions

Great interest is being shown in generating images of terrain using fractal surfaces. Such images are often indistinguishable from natural terrain, if seen at comparable resolution. This technique shows great promise for the generation of realistic scenes of great complexity.

Fractal surfaces have several advantages over other techniques for modeling in computer graphics. Object definitions may be very simple and contain little detail; all detail in the final image is generated by the fractal process. In addition, images may be generated to any level of resolution desired; no limit is imposed by the object definition.

Unfortunately, fractal algorithms suffer from the major drawback of being computationally very expensive. Until this barrier is overcome, the use of fractal images will be limited to still images, or to motion pictures generated only by those with the large computing resources needed. Their use in real-time applications, such as simulators, will be impossible.

Much of this problem with fractal surfaces stems from the fact that the computer must generate the pixel-sized triangles of the image one at a time. If the technique of generating them is examined, however, it may be seen that much of the work may be done in parallel, if the appropriate multi-processor architecture is used.

This paper proposes an architecture designed for the generation of fractal surfaces. The processor array consists of a large number, perhaps several thousand, processing elements, each of which is designed to subdivide one triangle. These processing elements are connected via a hexagonal grid, with communication lines along the lines of the grid and across certain diagonals.

This grid architecture forms a planar graph, so its implementation in VLSI is easy. With such an implementation, several hundred, perhaps a thousand, elements could be placed on one chip. Thus a complete fractal processor could be implemented in a handful of chips.

The advantages of this arrangement are many. This architecture would make possible the generation of fractal images of moderate complexity in a few seconds. This is at least 2 orders of magnitude faster than current software-based systems.

This architecture is simple. Each node in the processing array is functionally identical to every other node. The only differences lie in their communication links to other nodes. Moreover, the nodes only need to be able to perform a simple function - the subdivision of a single triangle.

The architecture is easily expandable. The fractal processor may be increased in power and performance by replicating the basic unit four times, and placing appropriate communication links between the edges of the units. The fractal processor so obtained appears to the host computer to be identical to the original, smaller unit, except that it can proces larger triangles.

The architecture described here makes possible the generation of fractal surfaces with reduced computational expense, and will thus make fractals much more useful in a range of application from motion pictures to high-performance aircraft training simulators.

References

Bézi74 Bezier, P., "Mathematical and practical possibilities of UNISURF," Computer Aided Geometric Design, Barnhill, R. E. and Riesenfeld, R. F. (eds.), Academic Press (1974).

Blin77 Blinn, J. F., "Models of light reflection for computer synthesized pictures," Proceedings of SIGGRAPH '77, also published as Computer Graphics, vol. 11, no. 2 (Aug. 1977), pp.192-198.

Four80 Fournier, A., "Stochastic modeling in computer graphics," Ph.D. Dissertation, University of Texas at Dallas (1980).

Four82 Fournier, A., Fussell, D., and Carpenter, L., "Computer rendering of stochastic models," CACM, vol. 25, no. 6 (June 1982), pp. 371-384.

Gord74 Gordon, W. J. and Riesenfeld, R. F., "B-spline curves and surfaces," Computer Aided Geometric Design, Barnhill, R. E. and Riesenfeld, R. F. (eds.), Academic Press (1974).

Mand77 Mandelbrot, B., Fractals: Form, Chance, and Dimension, W. H. Freeman (1977).

Mand82 Manbelbrot, B., The Fractal Geometry of Nature, W. H. Freeman (1982).

Meze74 Mezei, L., Puzin, M., and Conroy, P., "Simulation of patterns of nature by computer graphics," Information Processing 74, pp. 52-56.

AN ARCHITECTURE FOR THE REAL-TIME DISPLAY AND MANIPULATION OF THREE-DIMENSIONAL OBJECTS

S.M. Goldwasser

Department of Computer and Information Science
The Moore School of Electrical Engineering

R.A. Reynolds

Medical Image Processing Group
Department of Radiology

University of Pennsylvania, Philadelphia, PA 19104

Abstract

A special purpose multiprocessor architecture has been developed which facilitates the high speed display and manipulation of shaded three dimensional objects or object surfaces on a conventional raster scan CRT. The reconstruction algorithms exploit the capability to divide object space into totally disjoint cubical regions permitting multiple display processors to access independent memory banks concurrently without conflict. All of the geometric parameters describing rotation, translation, and scaling are incorporated into one short table facilitating very rapid manipulation of the image display presentation.

Introduction

The desire to generate realistic presentations of three dimensional objects has stimulated a great deal of interest in special purpose hardware and software systems. Applications for such technology include industrial simulation, three dimensional modelling, and medical imaging for clinical diagnosis. Currently, systems that operate in real time (i.e., 15-30 frames/second or more) fall primarily into two classes: The first are based on random scan display generation and thus provide only wireframe images with little realism. Other techniques based on polygons or other geometric primitives are being developed for use in such systems as high performance aircraft simulators. Systems of this type are not entirely suitable for images derived from experimental data but are being increasingly utilized for synthesized computer graphics.

Software based systems which generate realistic images of natural structures are extremely slow. These include the DISPLAY software package developed by Herman et al [1]-[4] for medical applications; other systems have been proposed independently [5]-[7]. The Lexidata SOLIDVIEW system [8] is a commercial product which combines hardware and firmware to offload hidden surface removal and shading algorithms from user software. Even with such hardware assist, a single view may take several minutes to be generated.

One important application of such a system is in the area of medical image processing using CAT, PET, or NMR scanning and reconstruction techniques. A system permitting a physician to visualize and interact with a shaded 3-D representation of an organ would greatly facilitate the examination of anatomical structures in conjunction with medical research, clinical diagnosis, and the planning of surgical procedures.

In this paper we present one possible architecture which will permit the high speed display and manipulation of solid objects represented as a voxel (volume element) database with grayscale. Our objective is to provide certain capabilities at or near video rates facilitating extensive real-time interaction. The architecture is highly modular permitting a cost tradeoff to be made to achieve a given level of performance. It also includes a great deal of regularity in its structure making it directly suitable for VLSI implementation. A key feature is that no computational operations more complex than adds, shifts, and comparisons are required in real time.

The DISPLAY Algorithm

The overall display processor architecture is based on the DISPLAY software package described in [1] and utilizes modified versions of those algorithms. The modified software generates surface views by mapping 3-D object space into 2-D image space using either a Z-buffer or equivalent time-ordered display procedure for hidden surface removal. In the latter version, for a given orientation of the object, pixels are written into the 2-D image (display) buffer in time-order corresponding to reading out voxels from the back to the front of the object. This "painter's algorithm" guarantees that any point that should be obscured by something in front of it will in fact be invisible in the final reconstruction.

Figure 1 illustrates a simple two dimensional analog of the back-to-front readout technique. It can be seen that for any orientation (rotation) of the object, there exists a readout sequence (and hence a processing time sequence) such that voxels early in the sequence (which should be hidden) will be overwritten by voxels later in the sequence. This architecture addresses the recursive decomposition of the sequence in such a

0190-3918/83/0000/0269$01.00 © 1983 IEEE

way that (1) a near real-time update rate is possible, (2) common geometric modifications are instantaneous, and (3) a modular structure is created.

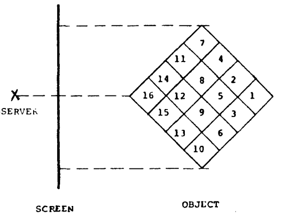


Figure 1 - 2-D Hidden Surface Removal. Voxel readout in order shown

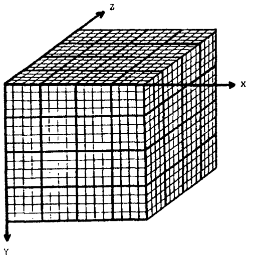


Figure 2 - Object Space Partitioning. Samll divisions represent 16-subcubes

In order to reduce the problem of real-time display of 3-D objects to manageable proportions, it is necessary to partition either the input (object) space or output (image) space - or some combination of these. In a multiprocessor implementation, partitioning input space and assigning each partition to a separate processor will avoid object memory access conflicts, whereas partitioning output space will avoid image memory access conflicts. The former technique is clearly superior and will minimize conflict since a substantial amount of data reduction occurs in the projection from 3-D to 2-D space.

Representation of 3-D Objects

We assume the input to the system (object space) is a 3-D scene subdivided by three sets of parallel planes into cube shaped volume elements or voxels as shown in Figure 2. While the algorithms which will be described can be generalized to any uniform parallelepipeds, throughout this paper we will generally assume a cubical object space. Furthermore, note that the voxel dissection is a special case of a general representation based on convex polyhedrons [9].

Associated with each voxel is a numeric quantity called the density which may correspond to color or brightness, or some other point property of the object. One use of density is to distinguish between different and distinct objects or parts of objects making up the input scene. A natural data structure for such a scene is a 3-D array, indexed by X, Y, and Z where the value of each element is the density of the corresponding voxel. A binary (two level) object would require one bit per point. Typically, however, the storage format is one byte per point supporting up to 256 density levels.

Such a 3-D array is spatially presorted in the sense that for any viewpoint, voxels can be read out and displayed in a sequence which guarantees that voxels retrieved early in the sequence cannot obscure voxels retrieved later in the sequence [10]. This property leads to the simple method of hidden surface removal described above. However, no data compression is achieved.

Two other possible data structures that can be used are octrees [7] and unsorted lists of voxels. The octree representation has the same spatial presorted property as the 3-D array. Octrees achieve excellent data compression when large regions of the scene contain the same density as, for example, in a binary scene. However, in our experience, the advantages for real world (particularly medical) objects are more than offset by the computational overhead associated with traversing the tree structure.

The second method is to store the voxels in random order, using 4 locations for each voxel (X, Y, Z, and density). This method is advantageous when a single small object has already been separated from the surroundings by means of thresholding or segmentation. However, a true Z-buffer is required for hidden surface removal. We have not found it appropriate for a real-time system capable of displaying entire scenes.

Display Processor Organization

The basic hardware realization of the DISPLAY algorithm consists of five components as illustrated by the block diagram in Figure 3:

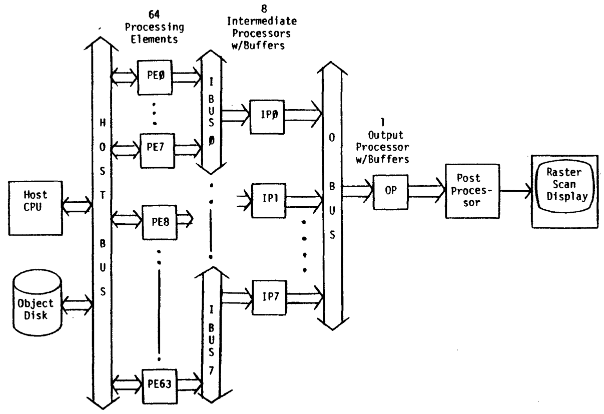


Figure 3 - Overall Display System Architecture

* Display processor array (64 PEs), each with associated object subcube memory module, and double 128x128 image buffer.

* Intermediate processors (for groups of 8 PEs) feeding double 256x256 intermediate image buffers.

* Output processor (for group of 8 intermediate buffers) feeding double 512x512 image buffer.

* Video postprocessor and video interface.

* Host computer interface and microprocessor based system controller.

Briefly, the processing strategy consists of the following. The processing elements (PEs) compute the 2-D subimages from each 64-subcube (64x64x64 voxels) of the overall 256-cube input object. Each PE contains a double buffer, each half of which is sufficient to hold the largest image that can be created from its associated 64x64x64 cube.

The reconstructed image will consist of two components. The first of these is the density of each active point in the object - those which have not been removed through thresholding, for example. Depth or Z coordinate - the distance from the point to the front end of object space - is buffered also for use by the shading postprocessor.

Each of the eight intermediate processors merges the 2-D subimages generated by its set of 8 PEs into the appropriate position in the eight intermediate double buffers following priority rules determined by the sequence control table (see below). Finally, the contents of the intermediate buffers are merged into the double 512x512 frame buffer, following the same priority rules. The two halves of the double frame buffer are filled alternately - one is computed while the other is displayed. Postprocessing consists of a global tone scale lookup table, shading algorithm implementation, and final brightness and/or pseudo-color lookup table.

A high speed interface permits communications with a host computer system for the purpose of image loading and readback. The host will also be responsible for archiving and retrieving appropriate data files, and converting formats to the internal object representation. The system controller is responsible for coordinating the activities of the 64 PEs by generating the sequence control tables for each desired object orientation. The control table includes X, Y, and Z position offsets for each of the subcubes making up object space. This information is used to recursively compute the absolute offsets for every point in the output image as well as the order of processing of voxels for the hidden surface removal algorithm.

Object Memory System

To display any set of objects with a scale factor of 1:1 within a 256-cube object space requires 16 M Bytes of high speed RAM (assuming 8 bit quantization for each point). While this may seem to be an extremely large amount of high speed memory, it should be recognized that the steep decline in MOS memory prices is expected to continue for some time. In addition, even at current prices, the cost of the overall display device (which is dominated by the cost of this memory) should be relatively small compared to the cost of a complete medical imaging system such as a CAT scanner.

Since the object space is divided into 64 equal subcubes, each PE requires 256 K Bytes of associated memory. Suitable memory management hardware located between the host and the PE-memory system facilitate direct computer accesses to restricted regions of object space such as X, Y, or Z planes or variable size cubical areas.

In each memory module, data are organized into groups of eight voxels occupying a pair of 32 bit words. Each such group constitutes a 2-cube. Five bits are required to specify a 2-cube index for each coordinate axis. Each memory access retrieves a word pair which is buffered in a register between the memory and the processing element permitting an entire 2-cube to be traversed in any order. The sequence control table is used to generate addresses for the memory access controller.

Display Processing Elements

Each processing element consists of a pipelined arithmetic processor, input density lookup table, its copy of the sequence control table, and a dual 128x128 image buffer memory. Figure 4 illustrates the overall organization of the PE and its associated 64-subcube input object memory module.

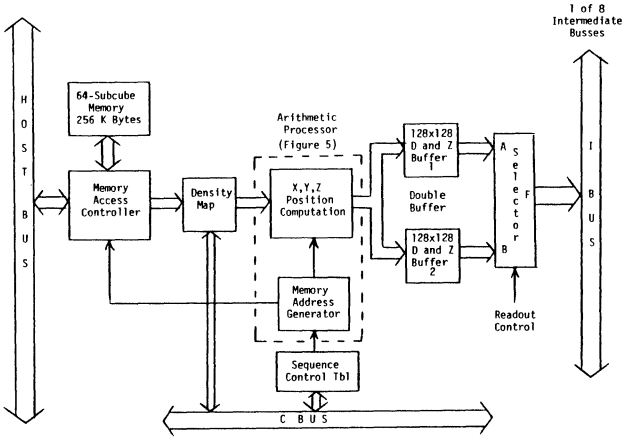


Figure 4 - Processing Element (PE) Organization

The density lookup table is used to preprocess the voxels retrieved from memory for various purposes including selective masking, thresholding, or image enhancement based on density value.

The arithmetic processor is responsible for computing X, Y, and Z offsets for each pixel of the image based upon the position of the corresponding input voxel. This is accomplished with no multiplies, divides, or other time consuming arithmetic or logical operations. As can be seen in Figure 5, the most complex operation is arithmetic addition. To obtain successively finer position offsets requires shifts but these are performed within the wiring of the pipelined system. Only the data paths for position computation along one coordinate axis are shown - the other two are similar.

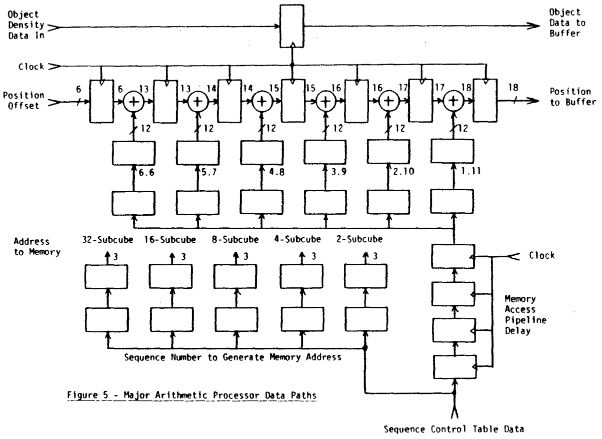


Figure 5 - Major Arithmetic Processor Data Paths

The operation of the arithmetic processor is based on the time-ordered display algorithm used for hidden surface removal. The fundamental concept which simplifies the hardware implementation is that regardless of object orientation, each subcube is entirely independent and all 64 subcubes may be processed in parallel since for any given subcube, every other subcube is either entirely in front of it or entirely behind it. The same characteristic also permits the overall computation of X and Y positions to be accomplished recursively, starting with the largest subcubes and working down to individual voxels, dividing by 2 at each step. For any particular voxel, the position offset along a given axis (X, Y, or Z) can be computed by simply adding the appropriately shifted control table entries. Thus, for a single orientation, only one control table of position offsets is sufficient for computation of all X, Y, and Z positions.

The precise X and Y destination coordinates in the 128x128 buffer are computed and converted to a memory address where the video (density and Z value) will be stored. A double buffer enables computation to proceed while the alternate buffer is being merged in subsequent stages of processing.

Anti-Aliasing

For magnifications from object space to image space greater than 1.732:1 ($1/\sqrt{3}$), holes would appear in the output image at certain orientations if anti-aliasing techniques were not utilized. Two methods have been investigated thus far: display of the centers of the visible faces of each voxel (1-cube) and double resolution interpolation with resampling. The first of these represents the singular case of the DISPLAY algorithm where the object cube faces have a size of exactly one pixel. At most, there will be three faces visible from any orientation. Interpolating out to a double resolution 3-D grid and resampling is similar to anti-aliasing techniques used in graphics display processors. Both of these require more sophisticated processing and additional buffer memory in each of the PEs, but can be accomplished within the pipelining time constraints.

Sequence Control Table (SCT)

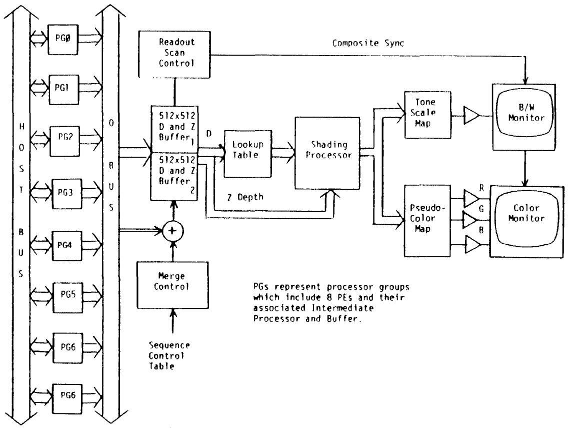
The SCT contains 8 entries sorted in the required time-order defining the X, Y, and Z offsets of the centers of the 8 largest subcubes with respect to the center of object space. Offsets for successively smaller subcubes are determined by shifting the table entries by an appropriate amount (between 0 and 7 places to the right). Adequate precision must be maintained in the table to achieve consistency of the offsets and prevent objectionable boundary errors from appearing in the final image. In addition to 3-D rotation, many other interactive capabilities can be implemented through modifications of the entries in the sequence control table and simple additions to the display processor hardware. A few of these are described below.

General anamorphic scaling is accomplished by simply multiplying the X, Y, and Z values stored in the table by the appropriate scaling factors with suitable interpolation of the input density data. Translation in 3-space is easily supported by adding X, Y, and Z offsets to the addresses of the output image buffer.

The display of up to 64 independently configurable objects can be achieved by loading object specific SCTs into each of the individual PEs or selected groups of PEs and modifying the implementation of the merge algorithms. This would permit complete control for objects within their own subcubes. These "sub object spaces" could include any 3-D rectangular region comprising multiples of the basic 64-cube. Other display parameters can be associated with the individual PEs including a translation offset and the tone scale mapping to be used for the input data.

Intermediate Processors and Buffers

Each of the images produced by the display processing elements consists of a two dimensional array of 112x112 points (in a 16384 word - 128x128 point memory - 1:1 scale factor) corresponding to the largest possible 2-D projection of the 64-subcube. These 64 images must eventually be merged into the output 512x512 frame buffer. This is accomplished in two steps. First the 64 images are combined 8-fold into the 256x256 point intermediate buffers, and then these are combined again 8-fold into the final output buffer. The first step is performed in parallel by the eight intermediate processors. Each of these merging processes requires the computation of position offsets as described for the individual PEs, above. Double buffers permit the merging and readout operations to be taking place concurrently. As described above, the same SCT determines both the order of computation within a PE and the order of combining for the merge operations. Figure 6 shows the second stage merge operation. The first stage data flow is similar.



PGs represent processor groups which include 8 PEs and their associated Intermediate Processor and Buffer.

Figure 6 - Second Stage Merging - Intermediate Buffer to Output Buffer

Output Frame Buffer

The final buffer stores the output image and Z depth values for use by the shading hardware. The major function of this memory is to permit scan conversion to standard video format for display on a monochrome or color raster scan TV monitor. This memory is directly accessible by the host. Since the output buffer is larger than the size of any projection of the 256-cube object (assuming no scaling), space is available to display other pictorial data or text. Any displayed image may be read back to the host for archiving or further processing.

Computation Pipeline Timing

Using the architecture outlined above, we can calculate the expected performance and throughput of the system. We assume that the required processing time is 100 ns per primitive calculation. This represents a conservative design guideline for discrete TTL or high performance NMOS VLSI technology.

* The time required to generate a subimage (by the PE) from the 64-subcube is 256 K x 100 ns or ~25.6 ms.

* The time required to merge groups of 8 subimage buffers into a 256x256 intermediate buffer is 8 x 112 x 112 x 100 ns or ~11 ms.

* The time required to merge 8 intermediate buffers into the output buffer is 8 x 224 x 224 x 100 ns or ~40 ms.

Thus, the limiting time is the last - corresponding to a frame update rate of 1000/40 or approximately 25 frames/second. Note that because of the pipeline latency, however, a response to a change in orientation will require a total of three frame times to become visible.

Assuming output to a standard NTSC compatible video monitor, the full 25 frame per second throughput rate can be exploited by switching buffers whenever a new frame has been completely loaded. Alternately, a dual port memory system may be used for the output buffer. However, visible image changes (breaks) may occur for fast changing objects during the frame display. An effective update rate of 20 frames per second can be easily achieved by displaying each frame generated by the display system 1-1/2 times corresponding to 3 video fields using 2:1 interlaced scanning.

Postprocessing

Two types of postprocessing are to be implemented in real time: tone scale lookup tables for the video intensity and other display parameters, and some form of shading to enhance the appearance and realism of the image.

Tone scale transformation hardware will permit the class of point type image processing functions which are traditionally used with image processing systems to be implemented on the output image in real time. Examples of these operations include contrast enhancement, interactive thresholding, and pseudo-color processing.

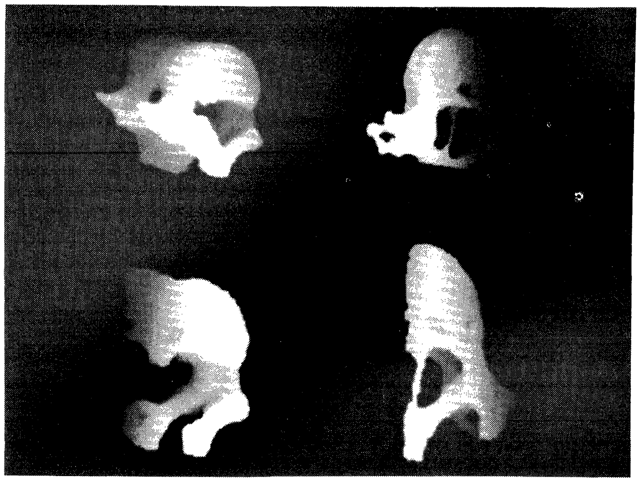


Figure 7 - Examples of Software Simulation using Depth-only Shading

Realistic shading of the output image is essential to provide depth cues and other visual information about object structure. We have conducted software simulations using "distance-only" shading, "constant" shading, "contextual" shading [3] and Phong shading [11]. In distance-only shading, the intensity of a point of the image is determined by the distance of the corresponding point of the object from the light source. This is simple to compute and gives pleasing results (Figure 7). The other shading models take direction into account by computing the inner product of the normal to the surface with a unit vector along the light ray reaching it: this provides curvature information. In constant shading, the normal assigned to a point is independent of its neighbors, whereas in contextual shading the overall configuration is taken into account. Phong shading, which is an extension of Gouraud shading [12], is an attempt to make the intensity vary smoothly from point to point.

The distance-only shading algorithm simply uses the Z coordinate (depth) to obtain the brightness of each output point. Most other shading schemes are more difficult to implement since they are non-local operations requiring knowledge of neighboring voxels. One solution would use a gradient operator on the Z coordinates to obtain the surface normal at each point. Alternatively, local direction information can be stored in each voxel (along with the density) and passed to the shading postprocessor. This approach has bee used effectively for the DISPLAY software package. A hardware implementation of directional shading would, however, necessitate expanded object and buffer memory capacity and wider data paths in addition to the shading postprocessor.

Summary and Conclusions

The hardware architecture for a high speed image display system which would permit the real-time manipulation of 3-D objects obtained from real world data has been developed. A key feature of this system is support for interactive manipulation of the object in 3-D space including rotation, translation, and scaling. Straightforward algorithms containing no complex arithmetic or logical operations are utilized throughout. The hierarchical organization and high degree of modularity will facilitate an effective implementation with VLSI technology.

Functional simulations of the display processor have been performed based on the existing DISPLAY software package. Current efforts are directed toward development of effective 3-D anti-aliasing techniques and investigation of more sophisticated shading algorithms which are appropriate for hardware implementation.

We are planning to prototype real-time hardware for one 64-subcube of the overall system using Schottky TTL devices and MOS dynamic RAMs. The prototype will permit experience to be gained from actual use by medical researchers and clinicians. For some types of imaging techniques where the current attainable resolution is relatively low, this subset of the overall system may prove adequate. The prototype effort will be followed by the full system implementation with heavy reliance on VLSI technology.

Acknowledgements

We wish to express our appreciation to Professor Gabor T. Herman and Dr. J.K. Udupa for their helpful comments and for permission to make use of the DISPLAY software package. The hidden surface removal algorithm was implemented and tested in software with the help of Mark Lipshutz and Roger Sessions. Part of our work was supported by NIH grant HL28438.

References

[1] Udupa, J.K., "DISPLAY82 - A System of Programs for the Display of Three-Dimensional Information in CT Data", Medical Image Processing Group Technical Report MIPG67, Department of Radiology, University of Pennsylvania, April 1983.

[2] Herman, G.T., and Liu, H.K., "Three-Dimensional Display of Human Organs from Computed Tomograms", Computer Graphics and Image Processing 9 (1979), pp. 1-21.

[3] Herman, G.T., and Udupa, J.K., "Display of Three-Dimensional Discrete Surfaces", Proceedings SPIE 283 (1981), pp. 90-97.

[4] Artzy, E., Frieder, G., and Herman, G.T., "The Theory, Design, Implementation, and Evaluation of a Three-Dimensional Surface Detection Algorithm", Computer Graphics and Image Processing 15 (1981), pp. 1-24.

[5] Batnitzky, S., Price, H.I., Lee, K.R., Cook, P.N., Cook, L.T., Fritz, S.L., Dwyer, S.J., and Watts, C., "Three-Dimensional Computer Reconstruction of Brain Lesions from Surface Contours provided by Computed Tomography", Neurosurgery 11 (1982), pp. 73-84.

[6] Sunguroff, A., and Greenberg, D., "Computer Generated Images for Medical Applications", Computer Graphics 12 (1978), pp. 196-202.

[7] Meagher, D.J.R., "High Speed Display of 3D Medical Images using Octree Encoding", Rensselaer Polytechnic Institute Technical Report, September 1981.

[8] SOLIDVIEW System, Lexidata Corporation, Billerica, Massachusetts.

[9] Fuchs, H., Keden, Z.M., Naylor, B.F., "On Visible Surface Generation by A Priori Tree Structures", Computer Graphics 14 (1980), pp. 124-133.

[10] Herman, G.T., Reynolds, R.A., and Udupa, J.K., "Computer Techniques for the Representation of Three-Dimensional Data on a Two-Dimensional Display", Proceedings SPIE (1982), to appear.

[11] Phong, B-T, "Illumination for Computer Generated Pictures", Communications ACM 18,6 (June 1975), pp. 311-317.

[12] Gouraud, H., "Continuous Shading of Curved Surfaces", IEEE Transactions Computers, C-20 (June 1971), pp. 623-628.

A PARALLEL ARCHITECTURE FOR LABELING, SEGMENTATION, AND LEXICAL PROCESSING IN SPEECH UNDERSTANDING

Edward C. Bronson
Leah Jamieson Siegel

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

Abstract -- Speech understanding is a complex task which requires extensive computation. To increase the processing speed, a speech understanding system is decomposed into tasks which can be performed by a series of distributed processing sub-systems. An architecture to perform labeling, segmentation, and lexical processing is described. Using a parametric characterization of the speech signal, this system divides an utterance into labeled homogeneous regions. The system then performs dictionary lookups based on all probable labelings and segmentations in order to generate a complete set of word hypotheses. Using realistic assumptions from existing speech understanding systems, a statistical model of speech input, and simulations of the speech processing algorithms, the attributes of the parallel system to perform labeling, segmentation, and lexical processing for real-time speech understanding are derived.

I. Introduction

A speech understanding system accepts speech input, derives a conceptual understanding of the input, and generates an appropriate response. In a typical system, a number of knowledge source components interact to resolve the errors and ambiguity inherent in human speech. These knowledge sources perform operations such as acoustic parameterization, phonetic interpretation, lexical processing, syntactic analysis, semantic interpretation, and response generation. Existing speech understanding systems that have been developed are described in [19, 21, 25].

The extensive computation required precludes real-time speech understanding on a conventional serial computer. To improve the processing speed, the different knowledge sources can act in parallel (possibly on different portions of an utterance), and in addition, computational tasks within each knowledge source can be performed in parallel. Advances in technology have made it realistic to consider large-scale parallel processing systems [e.g., 1, 10, 24, 29]. By designing multiprocessor knowledge sources, real-time speech understanding (with a constant delay) should be achievable. The next section briefly outlines a general configuration for a multiprocessor system for speech understanding. In sections III and IV, the speech understanding operations of labeling, segmentation, and lexical processing are outlined. Section V describes the organization of the system described in this paper. An example simulation of labeling, segmentation, and lexical processing is presented in section VI. Section VII describes the parallel algorithms and the parallel architectures that comprise the system.

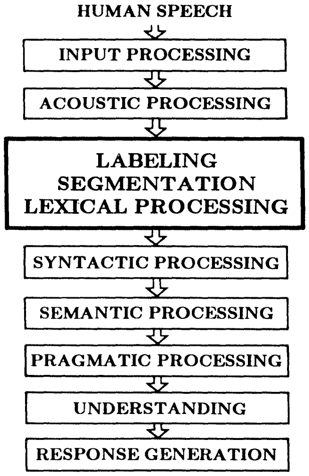


Fig. 1. The distributed speech understanding system.

II. A Parallel Architecture For Speech Understanding

An architecture proposed to handle the speech understanding task consists of a distributed series of *compustations* [2, 3]. Each compustation corresponds roughly to a speech understanding knowledge source. This distributed parallel architecture is diagrammed in Fig. 1. The interconnection of knowledge sources forms a linear pipeline.

Each compustation may itself be a parallel system. A typical compustation consists of an input memory buffer *(MB)*, an output MB, control units *(CUs)*, and processing elements *(PEs)*. The organization of the PEs within each compustation is selected to exploit whatever parallelism is inherent in the specific task being performed by that station. The processing time for each station is a function of the computational complexity of the tasks to be performed and the amount and arrival rate of input data. Assuming a maximum input rate, the processing speed requirements can be met by employing parallelism within the task algorithms and also among the tasks to be performed. Minimum processing time for the compustation will be insured when the data in the input MB is processed as soon as it is available. When a subset of PEs has finished a processing task and stored its result in the output MB, it is available to be assigned another task by the compustation's control unit.

Each compustation is specialized to meet performance (speed) requirements of the overall system. Processing proceeds asynchronously with respect to adjacent compustations. When the processing time for each station is approximately equal, then no bottlenecks occur

This material is based upon work supported by the National Science Foundation under Grant ECS-8120896.

0190-3918/83/0000/0275$01.00 © 1983 IEEE

and data flow through the system will be continuous, providing real-time performance (with a constant delay). Because the parallelism within each compustation permits processing of all probable utterance hypotheses simultaneously, there is no need to backtrack once any particular hypothesis has been determined improbable. Thus, extensive parallelism is being used at each stage of the speech understanding process in order to simplify the interaction among the various knowledge sources.

III. Labeling and Segmentation

Speech labeling is the task of analyzing a speech utterance and assigning identifying labels to regions of the utterance. Speech segmentation is the task of locating boundaries between homogeneous regions *(segments)* in a speech utterance. In some systems, labeling precedes segmentation; in others, segmentation precedes labeling. Together, the labeling and segmentation processes divide a speech utterance into labeled homogeneous regions. Speech labeling and segmentation are described in [27, 28, 36]. Detailed descriptions of methods used in specific systems are presented in [8, 13, 34].

The input to the labeling and segmentation process will be characteristic parameters computed for each frame of speech during acoustic processing. In general, each set of characteristic parameters represents an interval of speech ranging from 5 to 20 ms. The acoustic parameters describe each speech frame in terms of its characteristic time and frequency domain features [4, 36]. Labels can be assigned either to fixed duration frames (before segmentation) or to larger homogeneous segments (after segmentation). The labeling is typically done using pattern classification techniques. The criteria used and the number of different labels assignable to the regions must be consistent with the word representations used within the lexical database. The labeled segments produced will be used by the lexical processing component of the system to obtain word hypotheses.

The labels used correspond to elements of a finite set of speech sounds which can be identified by definite acoustic characteristics. The most commonly used set of speech sounds are phonemes [12]. A *phoneme* is a group of sounds that function similarly and are not meaningfully distinctive among the group for a given language. A language will contain from 20 to 60 phonemes. Examples of different phonemes are the "oo" sound in "foot," the "oo" sound in "boot," and the "s" sound in "sit." It is often difficult to identify phonemes and their boundaries accurately from acoustic data [27]. Many intervals of speech will appear similar to more than one phoneme. These problems are handled in existing speech understanding systems by permitting multiple classifications for a given speech interval and then applying a more extensive lexical analysis.

Techniques to perform labeling and segmentation are either context independent or context dependent. A context independent scheme looks at the acoustic parameters for the speech frame, compares these parameters to previously stored templates corresponding to the speech sounds label set, and assigns a label to the speech frame. A correctness score could be associated with the label. In some systems [34], the multiple labels for each frame are stored in a structure called a segment lattice. This preserves all probable frame labels and provides the option for the system to backtrack and consider less probable labels when the word hypothesis based upon one labeling becomes unlikely.

A context dependent labeling and segmentation scheme uses information from the current and surrounding frames or segments when labeling a given speech interval. Some systems use a combination of both context dependent and context independent methods.

IV. Lexical Processing

Lexical processing is the task of combining various lengths of contiguous sequences of segmented and labeled speech data, comparing these intervals to a word dictionary or lexicon, and generating probable word hypotheses. Noise, variable pronunciations, mispronunciations, regional dialects, and variable speaking rates make it difficult to map acoustic events directly to word hypotheses. Lexical processing must take into account possibilities such as mislabeled input segments and variations in pronunciation when attempting to match an input utterance to entries in the word dictionary. Lexical processing is described in [18, 32]. Detailed descriptions of methods used in specific systems are presented in [11, 31, 34].

A number of lexical processing approaches exist. One method for determining possible words within continuous speech, called "bottom-up" word hypothesizing, uses acoustic information from the input utterance to propose words. In one possible implementation, the system includes a data structure constructed from acoustic descriptions of all words comprising the system's vocabulary. The acoustic information obtained form the input utterance will control the search through the data structure to hypothesize a word.

The output of lexical processing will be word hypotheses accompanied by likelihood scores. This information is passed to other knowledge sources such as syntax and/or semantics for further processing.

V. The Distributed/Parallel Organization

As described in sections III and IV, there are many different methods that can be used to perform the operations of labeling, segmentation, and lexical processing, and the boundaries between these operations are not necessarily clear. It is therefore appropriate to consider a single architecture to perform all of the operations involved in labeling, segmentation, and lexical processing.

The merging of compustations is depicted in Fig. 1. and the organization of the architecture is shown in Fig. 2. The portion of the figure marked by Ⓐ indicates the PEs performing parallel labeling and segmentation. These PEs accept sets of time and frequency domain characteristic acoustic parameters which represent a 12.8 ms frame of speech. These parameters are stored within the input MB by the Acoustic Processing Compustation. (A design of an Acoustic Processing Compustation was presented in [4].) The labeling and segmentation operations produce a *segment lattice*. The segment lattice will contain multiple phoneme labels for each frame of speech. This data set is indicated by ① in Fig. 2 and would reside in a MB. The second set of PEs, indicated by Ⓑ, will use the segment lattice data to perform parallel word hypothesization. The multiple word hypotheses, indicated by ②, are stored in the output MB and will be used by the Syntactic Processing Compustation to perform syntactic analysis.

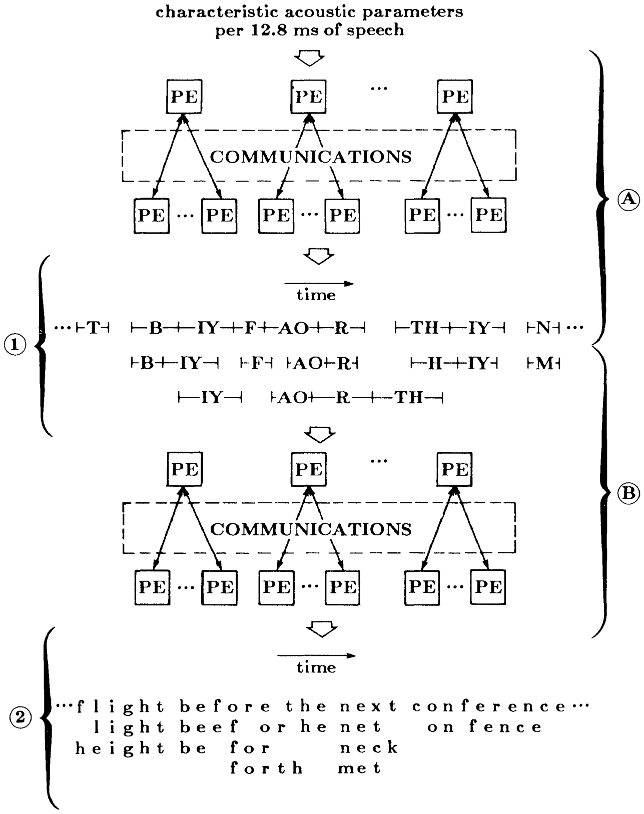

characteristic acoustic parameters
per 12.8 ms of speech
PE
COMMUNICATIONS
Ⓐ
time
①
···⊢T⊣ ⊢B⊣⊢IY⊣⊢F⊣⊢AO⊣⊢R⊣ ⊢TH⊣⊢IY⊣ ⊢N⊣···
⊢B⊣⊢IY⊣ ⊢F⊣ ⊢AO⊣⊢R⊣ ⊢H⊣⊢IY⊣ ⊢M⊣
⊢IY⊣ ⊢AO⊣⊢R⊣⊢TH⊣
Ⓑ
②
···flight before the next conference···
light beef or he net on fence
height be for neck
forth met

Fig. 2. Organization of the parallel architecture. Ⓐ Parallel labeling and segmentation. Ⓑ Parallel word hypothesization. ① Phoneme segment lattice. ② Multiple word hypotheses.

VI. An Example Simulation

To determine the requirements of a parallel architecture to perform real-time labeling, segmentation, and lexical processing, a specific set of algorithms was chosen and the operations of the algorithms were simulated. In this section, the algorithms and operation are described and the approach used in the simulation is outlined.

Algorithms and Operation

For each 12.8 ms frame of input speech, the compustation obtains a set of characteristic acoustic parameters from the input MB. The compustation determines the similarity of these input parameters to stored phonetic templates and produces a set of probable phonetic labels for the frame based upon this similarity. In the algorithm considered here, the probable labels are determined by classifying the speech-spectra using a minimum distance measure with linear mean correction [30]. This technique computes a distance measure between 40 power spectrum values representing the input speech and a set of 40 template values stored for each of 33 possible phoneme labels. The distance measure results are then sorted to determine the most probable labels for the frame. Each distance measure is compared to a threshold and from one to five most probable labels are assigned to the frame. By keeping up to the five most probable labels for the frame, the correct label will be included 95% of the time [9]. This step is a form of context independent labeling on fixed duration frames.

Once labels have been assigned to a frame, the previous and following frames are examined for occurrence of each phoneme label. If a phoneme label does not occur in either of the neighboring frames, the label is eliminated from the set of labels for the frame. If all of the labels for a frame are non-existent in neighboring frames, then a new symbol indicating an unknown phoneme label is assigned to the frame. This context-dependent label removal introduces an additional one frame (12.8 ms) delay to the speech processing.

Considering an interval of speech containing a number of segments, the multiply labeled frames over that interval form a phoneme segment lattice (e.g., ① in Fig. 2). The paths through the lattice define the possible words within the speech input. The lattice is searched to generate all possible strings of phonemes. Bottom-up lexical processing is then performed, in which these strings are compared to a lexical database of phoneme strings which represent the words known to the system. From this comparison, word hypotheses are determined.

The lexical database assumed represents a word vocabulary of 6,843 common English words. Each word is stored by its phonetic representation, with the longest phonetic representation consisting of seven phonemes. These words were taken from a vocabulary of 10,065 English words with phoneme string representations of up to fifteen phonemes [16, 26]. In the simulation, the strings generated from the speech input range in length from three to seven phoneme segments. (One and two phoneme strings are handled as special cases.) When the frequency of use of the words within the database is taken into account [16, 26], the 6,843-word vocabulary (i.e., the words of seven or fewer phonemes) comprise 95% of the words actually used. Multiple representations of a single word are included in the lexicon. It is assumed that these are generated by applying contextual and phonological rules to the theoretical word representation. The rules will account for pronunciation deletions, insertions, coarticulation, and multiple mispronunciations [7, 35]. This lexical expansion will result in an approximately three-fold increase [35] in the lexical database size. The resulting database contains 20,529 entries.

Because all paths through the phoneme lattice are compared to the database, multiple conflicting word hypotheses may be generated. In addition, every point in the lattice at which there is a transition from one phoneme to another represents a potential end of one word and beginning of the next. Word hypotheses based on all such word boundaries will be generated. A special case of this may give rise to multiple possible start and stop times for a single word. The multiple word hypotheses and the range of start and stop times will be stored in the output MB and resolved by a later compustation.

Simulation of the Algorithms

The operations of the algorithms to perform labeling, segmentation, and lexical processing were simulated on dual-processor Vax 11/780 computers [14] which are part of the Engineering Computer Network of Purdue University [17]. The components of the simulation are diagrammed in Fig. 3, and are discussed in the following paragraphs. The bold arrows indicate the flow of the simulation.

The input to the simulation was a stream of phonemically labeled frames which model speech input.

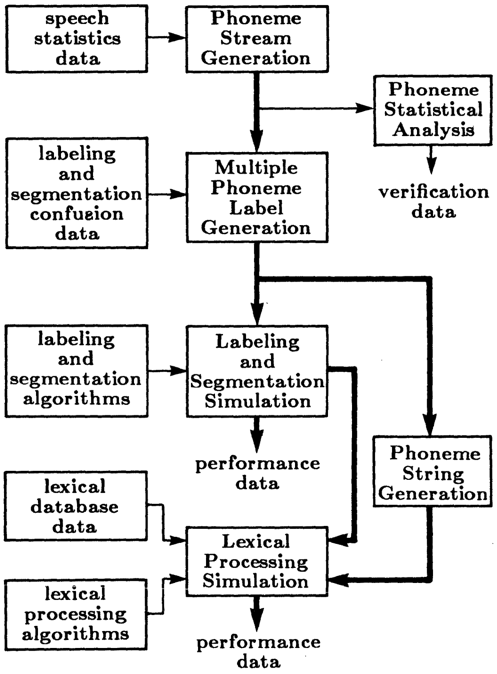

speech statistics data
Phoneme Stream Generation
Phoneme Statistical Analysis
verification data
labeling and segmentation confueion data
Multiple Phoneme Label Generation
labeling and segmentation algorithms
Labeling and Segmentation Simulation
performance data
Phoneme String Generation
lexical database data
Lexical Processing Simulation
lexical processing algorithms
performance data

Fig. 3. Components of the simulation.

The statistical distribution of the phonemes in the input is based upon a statistical linguistic analysis [26] of a vocabulary of 10,065 English words [16]. This information, combined with the average duration of each phoneme [15, 33], was used to model the input stream as a stochastic process [6]. A 33-state Markov chain was used to generate the phoneme stream. The phoneme stream statistically models the percent occurrence of each phoneme or silence, the duration of each phoneme or silence, and the transition probability of two adjacent phonemes. A statistical analysis was performed upon the generated phoneme stream to verify the accuracy of the speech model. The statistically generated phoneme input was used to avoid the difficulty of performing computationally intensive acoustic parameterization on the enormous amount of speech input data which would be required to obtain representative phoneme distributions and patterns. Details of the phoneme stream generation can be found in [5].

The generated phoneme stream contains one phoneme label per input frame of speech. The results of the labeling algorithms are simulated by generating multiple phoneme labels for each frame, based on labeling performance data which indicates the probability of confusing the true input phoneme with acoustically similar phonemes [30].

The resulting multiple phoneme label input stream corresponds to the segment lattice data and is used as an input to simulate the complexity of the labeling and segmentation algorithms. The segment lattice also provides the input to the lexical processing simulation. In particular, the "Phoneme String Generation" process provides statistics on the number of strings of various lengths that will be compared to the lexicon.

Complexity data from the labeling and segmentation simulation, results of the phoneme string generation, and the lexical database statistics are used to simulate the operations of lexical processing.

VII. The Parallel Architectures

In this section, the parallel sub-systems to perform labeling, segmentation, and lexical processing are described. The PE model used in the architecture design is presented in [20] and is based upon the Motorola MC68000 microprocessor [23]. Processor timing calculations were made with the MC68000 running Motorola's 68343 fast floating point software [22]. The execution times were calculated for the MC68000 running with a 12.5 MHz clock frequency. 13,100 frames of speech data were simulated to obtain the operations counts and database sizes.

For the architecture design presented here, it was assumed that each sub-system must attain real-time performance based on the average arrival rate of data to its input MB. This implies that, on the average, real-time performance will be achieved as long as excess data can be buffered and processed later. A more stringent analysis would require the processing to meet maximum data rates. Because of the very large variance in the data rates from one frame to another, this approach is not practical and is not considered here.

Labeling

The labeling sub-system must compute the distance from the set of input acoustic parameters to the templates for each of the 33 possible phonemes, select the (up to) five most probable phonemes, and perform the context dependent removal of isolated (single frame) labels. The context independent labeling must be performed within 12.8 ms so that this labeling is completed by the time the next set of input parameters is available. Once the subsequent frame is labeled, the context independent labeling must also be completed within 12.8 ms. Based on the operations required, the architecture to perform labeling is an eight PE MIMD system with one CU. The comparison of the input parameters to the 33 templates is distributed across the PEs, with each PE computing the distance to four templates. The CU computes the 33rd distance and performs the final phoneme selection and context dependent labeling. Communication is between the CU and the PEs, first to disseminate the acoustic data then to collect the computed distances. No inter-PE communication is required.

Segmentation

Once a frame is labeled, the segmentation sub-system must determine what new strings exist in the phoneme segment lattice as a result of the possibly multiple labels for that frame. This. involves appending each new label to every length six or shorter string in the lattice and determining what new strings result. The phoneme segment lattice is updated (i.e., the new strings are added to the lattice) and the strings which are identified as possibly complete are passed to the lexical sub-system as potential word hypotheses. Processing should be completed by the time the next set (frame) of labels is available. In general, the architecture to perform segmentation consists of an input CU and seven sub-systems, each an MIMD system consisting of a CU and a set of PEs. The input CU monitors the arrival of new input label sets and broadcasts the input MB contents to the sub-system CUs. Sub-system ℓ holds the lattice entries of length ℓ, $1 \leq \ell \leq 7$. Using the labels for the new frame, sub-systems 1 through 6 form length $\ell+1$ strings and pass these to sub-system $\ell+1$ for possible addition to the length $\ell+1$ lattice database. Similarly, sub-systems 2 through 7 receive length ℓ strings formed in sub-system $\ell - 1$ for possible addition to the length ℓ lattice database.

If sub-system ℓ has P_ℓ PEs, then the length ℓ strings will be distributed approximately evenly evenly across the P_ℓ PEs. Each PE forms new strings using the new labels and its portion of the database. Similarly, each PE searches its portion of the database when determining if a string already exists in the database. Although the PEs perform essentially the same algorithm, MIMD operation is used to allow the string comparisons to abort as soon as possible. No communication is required among the PEs. The CU must be able to broadcast data and control signals to the PEs and must be able to poll the PEs for the status of their operation. Communication from sub-system ℓ to sub-system $\ell+1$ is performed through an intermediate MB between the CUs. Finally, CU ℓ will pass length ℓ strings to the lexical processing sub-system for dictionary lookup. This communication also proceeds through an intermediate MB.

Using the simulation results, the processing for the length one, two, three, and four strings can be handled in real-time by a single processor. In order to maintain the average processing rate in real-time, sub-systems 5, 6, and 7 are MIMD systems with $P_5 = 2$, $P_6 = 5$, and $P_7 = 21$. Each of these sub-systems has its own CU. The architecture for segmentation therefore consists of four sub-systems. The maximum amount of PE memory required for storage of the phoneme segment lattice ranges from 5.6K 16-bit words for the 1/2/3/4 sub-system to 117K words for the length 7 sub-system. The total memory needed for storage of the lattice is 169K words.

Lexical Processing

The lexical processing sub-system compares strings from the phoneme segment lattice to the stored lexicon and outputs word hypotheses. For real-time operation, the rate at which the sub-system can search the lexicon must equal the rate at which strings arrive from the segmentation sub-system. Based on the simulations, the average rate will be 177 strings per 12.8 ms frame. The general architecture for lexical processing consists of an output CU and seven sub-systems, each an MIMD system consisting of a CU and a set of PEs. Sub-system ℓ holds the length ℓ portion of the lexicon, $1 \leq \ell \leq 7$. Within sub-system ℓ, the lexicon entries are ordered and are distributed across the P_ℓ PEs, so that PE p holds item p mod P_ℓ, $0 \leq p < P_\ell$. A MB resides between segmentation CU ℓ and lexical processing CU ℓ and receives the search request strings. Lexical CU ℓ broadcasts each string to its PEs, each of which performs a binary search. The CU must be able to broadcast data and control signals to the PEs and must be able to poll the PEs for the status of their operation. No inter-PE communication is needed. Located word hypotheses are passed from the sub-system CU via an intermediate MB to the output CU, which writes the hypothesized word and its associated information in the compustation output MB.

Based on the simulations and the assumptions about the lexicon size, and requiring lexical processing to match the average arrival rate of strings from the segmentation sub-system, a single PE is capable of performing the dictionary searches in real-time. Four factors contribute to this result. (1) Relatively few strings need to be looked up (on the average, 177 per frame); (2) the binary search is very efficient; (3) within the binary search, it is rare that all of the characters in the two strings need to be compared; (4) the time for a symbol comparison on the MC68000 is quite short. Therefore, for the specific assumptions made here, the lexical sub-system can consist of one PE which scans the intermediate MBs holding the segmentation output, performs the lookups, and writes the located strings into the output MB. If a different lexical database were assumed, allowing, for example, longer strings, larger vocabulary, or more variability in pronunciation, the multiprocessor architecture would be applicable. Using the assumptions outlined here, the memory required for storage of the lexical database is 139K 16-bit words.

VIII. Summary

A parallel architecture to perform the speech understanding operations of labeling, segmentation, and lexical processing has been described. The overall architecture consists of a total of 43 (42 + 1) processors organized into three major components, corresponding to the three principal operations performed. Within the component systems are MIMD or uniprocessor sub-systems. The architecture characteristics were derived using realistic assumptions about both speech data and speech processing. Extensive simulation was performed to determine the required processing and data rates.

The algorithms chosen represent one set of the many different techniques used in labeling, segmentation, and lexical processing. Clearly the architectural details will reflect the algorithms chosen. This specific study demonstrates the viability of the use of parallelism in speech understanding systems and explores techniques for determining system characteristics from specific applications requirements. Future work includes analysis of additional speech processing methods and knowledge sources.

IX. Acknowledgments

The authors thank E. J. Coyle and J. T. Gandour for their comments and assistance with this paper.

X. References

[1] K. E. Batcher, "The design of a massively parallel processor," *IEEE Trans. Comp.*, Vol. C-29, Sept. 1980, pp. 836-844.

[2] E. C. Bronson and L. J. Siegel, "A parallel architecture for speech understanding," *1981 IEEE Int. Conf. Acoust., Speech, Signal Processing*, Mar. 1981, pp. 1176-1179.

[3] E. C. Bronson and L. J. Siegel, "Overview of a distributed parallel architecture for speech understanding," *15th Hawaii Int. Conf. System Sciences*, Jan. 1982, Vol. I, pp. 350-359.

[4] E. C. Bronson and L. J. Siegel, "A parallel architecture for acoustic processing in speech understanding," *1982 Int. Conf. Parallel Processing*, Aug. 1982, pp. 307-312.

[5] E. C. Bronson and L. J. Siegel, "Modeling of English speech for the design of a distributed speech understanding system," in preparation.

[6] E. Çinlar, *Introduction to Stochastic Processes*, Prentice-Hall, Inc., Englwood Cliffs, NJ, 1975.

[7] T. C. Diller, "Automatic lexical generation for speech recognition," *1977 IEEE Int. Conf. Acoust., Speech, Signal Processing*, May 1977, pp. 803-806.

[8] R. N. Dixon and H. F. Silverman, "A general language-operated decision implementation system (GLODIS): its application to continuous speech seg-

mentation," *IEEE Trans. Acoust., Speech, Signal Processing,* Vol. ASSP-24, Apr. 1976, pp. 137-162.

[9] N. R. Dixon and H. F. Silverman, "The 1976 modular acoustic processor (MAP)," *IEEE Trans. Acoust., Speech, Signal Processing,* Vol. ASSP-25, Oct. 1977, pp.367-379.

[10] M. J. B. Duff, "Parallel algorithms and their influence on the specification of application problems," in *Multicomputers and Image Processing,* K. Preston and L. Uhr, eds., Academic Press, NY, 1982, pp. 261-274.

[11] L. D. Erman et al., "The HEARSAY-II speech understanding system: integrating knowledge to resolve uncertainty," *Computing Surveys,* Vol. 12, June 1980, pp. 213-253.

[12] W. N. Francis, *The Structure of American English,* The Ronald Press Company, NY, 1956.

[13] G. Gill et al., "A recursive segmentation procedure for continuous speech," *Dept. Comp. Sci. Tech. Rep. CMU-CS-78-134,* Carnegie-Mellon Univ., Pittsburgh, PA, May 1978.

[14] G. H. Goble and M. H. Marsh, "A dual processor VAX 11/780," *9th Annual Symp. Computer Architecture,* Apr. 1982, pp. 291-298.

[15] J. N. Holmes, I. G. Mattingly, and J. N. Shearme, "Speech synthesis by rule," *Language and Speech,* No. 7, 1964, pp. 127-143.

[16] E. Horn, *A Basic Writing Vocabulary,* University of Iowa Monographs in Education, No. 4, The College of Education, University of Iowa, Iowa City, IA, Apr. 1926.

[17] K. Hwang et al., "A Unix-based local computer network with load balancing," *Computer,* Apr. 1982, pp. 55-65.

[18] D. H. Klatt, "Word verification in a speech understanding system," in *Speech Recognition,* D. R. Reddy, ed., Academic Press, NY, 1975, pp. 321-341.

[19] D. H. Klatt, "Review of the ARPA speech understanding project," *J. Acoust. Soc. Am.,* Vol. 62, Dec. 1977, pp. 1345-1366.

[20] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 Int. Conf. Parallel Processing,* Aug. 1982, pp. 353-362.

[21] W. A. Lea, ed., *Trends in Speech Recognition,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.

[22] Motorola, "68343 fast floating point source/object for MC68000," M68KFFP specification sheet, Nov. 1981.

[23] Motorola, *MC68000 16-bit Microprocessor User's Manual,* third edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[24] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comp.,* Vol. C-26, May 1977, pp. 458-473.

[25] D. R. Reddy, "Speech recognition by machine: a review," *Proc. IEEE,* Vol. 64, Apr. 1976, pp. 501-531.

[26] A. J. Roberts, *A Statistical Linguistic Analysis of American English,* Mouton & Co., The Hague, 1965.

[27] R. Schwartz and J. Makhoul, "Where the phonemes are: dealing with ambiguity in acoustic-phonetic recognition," *IEEE Trans. Acoust., Speech, Signal Processing,* Vol. ASSP-23, Feb. 1975, pp. 50-53.

[28] J. E. Shoup, "Phonological aspects of speech recognition," in *Trends in Speech Recognition,* W. A. Lea, ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980, pp. 125-138.

[29] H. J. Siegel, et al., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comp.,* Vol. C-30, Dec. 1981, pp. 934-947.

[30] H. F. Silverman and N. R. Dixon, "A comparison of several speech-spectra classification methods," *IEEE Trans. Acoust., Speech, Signal Processing,* Vol. ASSP-24, Aug. 1976, pp. 289-295.

[31] A. R. Smith and L. D. Erman, "Noah - a bottom-up word hypothesizer for large-vocabulary speech understanding systems," *IEEE Trans. Pattern Anal. Machine Intell.,* Vol. PAMI-3, Jan. 1981, pp. 41-51.

[32] A. R. Smith and M. R. Sambur, "Hypothesizing and verifying words for speech recognition," in *Trends in Speech Recognition,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980, pp. 139-165.

[33] Votrax, "SC-01 speech synthesizer data sheet," 1980.

[34] W. A. Woods, et al., *Speech Understanding Systems - Final Technical Progress Report,* Report No. 3438, Bolt Beranek and Newman, Inc., Cambridge, MA, 1976.

[35] W. A. Woods and V. W. Zue, "Dictionary expansion via phonological rules for a speech understanding system," *1976 IEEE Int. Conf. Acoust. Speech, Signal Processing,* Apr. 1976, pp. 561-564.

[36] V. W. Zue and R. M. Schwartz, "Acoustic processing and phonetic analysis," in *Trends in Speech Recognition,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980, pp. 101-124.

ON THE ALGEBRAIC SPECIFICATION OF CONCURRENCY AND COMMUNICATION

FINANCE J.P. & OUERGHI M.S.
C.R.I.N. Campus Scientifique B.P 239
54506 - VANDOEUVRE-LES-NANCY Cédex FRANCE

ABSTRACT : The absence of well suited tools for defining semantics of parallelism in programming languages is one reason of this study.
We suggest using, in this framework, the algebraic abstract data type techniques, due to the advantages they offer and which have been justified in several papers.
A simple and deterministic language with certains features of parallelism is presented. And an abstract data type associated with it is defined to make its semantics precise.

KEYWORDS : Concurrency, communication, algebraic abstract data type, semantics, serial time.

1. INTRODUCTION

Abstract data types are suggested by several authors ([5] , [7] , [12] ,..) to be a flexible, uniform, powerfull and abstract tool for specifying formally data structures, processed by programming languages.
Applying the same technique, especially the algebraic methods, to sequential applicative and procedural languages had been already done to define the semantics of program variables and assignements [9] , procedures [4] , or recursive functions [1] .
In this approach, a programming language can be described by an abstract (data) type, as described for example in [1] , for which the meaning is generally explained by particular models such as initial or terminals ones, or the set of all possible models (which may be described by sets of congruences of the term algebra). Thus it would be possible to have several semantic models to programs according to the choice of the model associated with the abstract type.
The object of the present paper is to apply and extend the techniques of algebraic semantics (for more explicative notions see [8]) to describe parallelism, using a simple and deterministic language, named CSPD, emboding explicit forms of concurrency and communication.

2. CSPD : SYNTAX AND INFORMAL MEANING

The main features of this language are :

- Processes are disjoint, don't have any shared variables. They only interact by means of communication.
- Communication is achieved by means of send and receive operations which are primitives of the language. Besides passing messages, communication may play a role of synchronisation.
  RECEIVE x FROM P_j (in P_i) is an input command, expressing an input request of the process P_i from the process P_j and assignment of the input value to the (local) variable x.
  SEND y TO P_i (in P_j) is an output command, meaning a request of P_j to output the value of y to P^j_i.
- Processes are deterministic and concurrency between them is explicit by a parallel operator.

The other components of the language are the nop and abort constructions, the sequential composition (stmt1; stmt2), the assignement (variable := expression), the conditional (IF boolexp THEN stmt1 ELSE stmt2 FI), the while loop (WHILE boolexp DO stmt OD) and typed declarations (variable : type).
An example of CSPD programs is the following :
PROGRAM Toy; ...

```
PROCESS Div ;
VAR x, y, quot, rem : INTEGER;
BEGIN
      RECEIVE x FROM User; RECEIVE y FROM User;
            quot := 0; rem := x ;
      WHILE rem ≥ y DO
           rem := rem-y; quot := quot+1;
             OD;
      SEND quot TO User; SEND rem TO User;
END;
PROCESS User;
VAR a,b,q,r : INTEGER;
BEGIN
      IF a > 0 AND b > 0
```

0190-3918/83/0000/0281$01.00 © 1983 IEEE

```
    THEN SEND a TO Div;SEND b TO Div;
         RECEIVE q FROM Div;
         RECEIVE r FROM div
    ELSE NOP
  FI
 END;
PARBEGIN  ...Div//User ... PAREND.
```

The language, if summarized, corresponds to the following syntax, where id, procid, declarations, var, exp and boolexp are non terminals to which we give no syntax rule and which represent respectively the program identifier, a process identifier, a local declarations, a variable, an expression and a boolean expression.

```
program —> PROGRAM id;{process};
   PARBEGIN procid {//procid} PAREND.
process —> PROCESS procid; declara-
  tions; BEGIN stmt END
stmt —> NOP | ABORT | var := exp |
    iF boolexp THEN stm1 ELSE stmt2 FI|
         WHILE boolexp DO stmt OD |
  stmt1;stmt2 | SEND exp TO procid |
         RECEIVE var FROM procid
```

3. FORMALIZATION OF THE SEMANTICS OF CSPD.

3.1. CSPD abstract type : the hypothesis.

We suppose :

- defined in a formal manner the context conditions of CSPD. From now on, we deal only with legal programs designed in CSPD (no static semantics are given).
- defined the basic (primitive) types that occurs in CSPD.
  We have in particular :
  VALUE : the type of value sorts, including the sort of identifiers(*) and the data sort Data, for which equality operations are predefined.
  EXP : The type of expressions with a general sort Exp (including in particular the sort of boolean expressions Bexp) for which a substitute operation is given.

According [8] , to define the semantics of a programming language, a method consists of extending the semantics, supposed to be known, of the primitive objects to the other abjects, that is the constructs of the language. The hierarchical construction of the type is given below.

3.2. CSPD abstract type : the systematic hierarchical description

To make explicit the definition of CSPD other types are required, in particular those concerning the language phrases and (the interpretor) states. Thus we have :

STATE : The type of states with a sort State generated by
- a null operation :
  init : —> State
- a two aries operation connecting states and the language phrases :
  apply : Modif x State —> State

To evaluate expressions into a value, we introduce an operation expressing this requirement :

eval : Exp x State → Data

MODIF : The type corresponding to phrases (statements and declarations) occuring in the language, with a general sort Modif generated by the following operations(*), to which we give a notation convention on the left hand side of the operations profile.

nop, abort : —> Modif to explicit the meaning of empty and abortion statements,	nop, abort
seq : Modif x Modif —> Modif to explicit the definition of the sequential composition,	s1 ; s2
if : Bexp x Modif x Modif —> Modif to define explicitly the conditional statement,	if b then s1 else s2 fi
while : Bexp x Modif —> Modif to define explicitly the while loop statement,	while b do s od
Send : Procid x Procid x Exp —> Modif to explicit the meaning of a send statement between a receiver and a sender, two concurrent processes,	P.Q!e
receive : Procid x Procid x varid —> Modif to explicit the meaning of a receive statement between a sender end a receiver, two concurrent processes.	P.Q?v

(*) The sort Ident maps variable identifiers Varid and process identifiers Procid.

(*) We concentrate our attention only on statements forgetting object declarations.

AGENT : The type corresponding to a process definition, with a sort Agent generated by : P::S

declagent : Procid x Modif → Agent

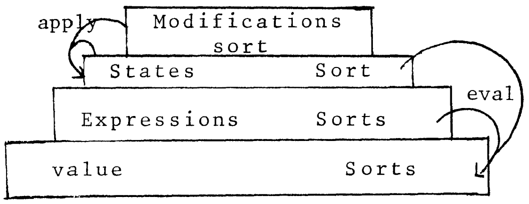
PROGR : The type corresponding to a CSPD program description, with a sort Progr generated by putting together processes in a concurrent way

parall : Agent x Agent → Progr P1::S1 || P2::S2

3.3. Remarks

(1) Defining the abstract data type for CSPD, semantical descriptions remains to be explicited. They will be certain terms of this abstract type. For instance, the term corresponding to an expression is of the sort Exp and the term corresponding to a statement is of the sort Modif. The mapping of concrete programs into terms is specified using semantic (conditional) equations.

(2) To be more explicit, the hierarchical construction of the type can be schematized as follow.



(3) For the CSPD type the following basic equalities are required :

- $s\ ;\ \underline{nop} = s$
- $\underline{abort}\ ;\ s = \underline{abort}$
- $(s_1 ; s_2)\ ;\ s_3 = s_1\ ;\ (s_2\ ;\ s_3)$
- $\underline{While}\ b\ \underline{do}\ s\ \underline{od} = \underline{if}\ b\ \underline{then}\ s\ ;\ \underline{while}\ b\ \underline{do}\ s\ \underline{od}\ \underline{else}\ \underline{nop}\ \underline{fi}$
- $a_1 \ || \ a_2 = a_2 \ || \ a_1$ where a_i is of the form $P_i :: s_i$ $(i = 1..2)$

With the help of this equalities every CSPD-term of sort Modif can be brought into the normal form : $s_1 ; s_2$, where s_1 is either nop, assign, send, receive or if operation.
Every term of sort Progr is equivalent to a term of the form :

$X::x_1; x_2 || \ldots || Y::y_1; y_2$

Where $x_1, \ldots, y_1$ corresponds to an elementary or conditional instruction. In our study we consider only the systems which are designed by two CSPD processes, a generalisation with n processes remaind to treat.

(4) Following the nature of modifications we consider them in a stratified manner. We note at least two kinds of modifications :

- Elementary modifications, including :
. The basic modifications which correspond to the nop, abort and assignement constructions. We characterize them by the predicate :
bstm? : Modif ―> Bool
. The input/output modifications which correspond to the send and receive commands. We characterize them by the predicate :
iostm? : Modif ―> Bool
- Composed modifications which correspond to the sequential composition, conditional and while loop constructions. We characterize them by the predicate :
cstm? : Modif ―> Bool

From now on, we preserve this decomposition (and go on with), because it will help us to clarify some difficult notions in parallel programming theory and to master the complexity originated from a large number of functions and axioms.

3.4. CSPD abstract type : a formal specification

3.4.1. A specification framework

3.4.1.1. The problem position

To specify (i.e, give semantics for) the full type two attitudes are possible :

(1). Consider that couples generated by the function
config : Modif x State → Conf
can be reduced by a semantic function
reduce : Conf → Conf
This is an approach of operational semantics of systems, in the form of transitions systems. For more explanations see [2] , [11] .

(2) Consider that a statement transforms the state by the "apply" operation, which we specify its properties by using conditional axioms and substitutions on the formulas.
To deal with this approach,

- At first, if we are not interested by the communication features, a single process (taken out of a set of communicating processes) is regarded by itself to be a semantically meaningful entity [3] .
- Therefore, a binding operator is introduced to combine a set of all separate "a priori" meanings of all separate processes to a joint meaning of the system designed by them in a parallel composition, and to settle communication features. This definition uses the important notion of (execution) time introduced below.

3.4.1.2. States and temporal specification

A state of a given system designed in CSPD can be viewed as a tuple of processes state (the processes are these composing the system). Recall that the processes state are disjoints because of the lack of global variables in the language and that each one can be viewed as a (linear) computation from initial state at an initial date (say t_0). If we are interested in describing, at a given date t, the general state of such a system (say Σ_t at t), two possibilities can take place :

- Σ_t is perfect : the processes finish at the same time t the execution of their current instruction.
- Σ_t is imperfect : at least one process finishes at the time t the execution of its current instruction, while the other not.

To consider similar situations, we introduce the predicate :

perfect? : State —> Bool.

To any particular process, one wish to attach a date (involving on the other hand all execution sequences which a program generates on the general state) and a modification. From now on, a state is considered as a tuple :
<memory, data, modification>
The perfectness of a process state is thus given by :

perfect? : (<σ,t,s>) = *if* s = nop *then* true
else false *fi*

We suppose that the initial states (at initial date t_o) are perfect.
To coop up the time notions, we suppose as known the primitive type DATE with the sort Date, ranged over by t, and give below profiles and informal meanings of the needed operations :

(1) dateof : State —> Date
defines the second projection of a state seen as a tuple of the form <σ,t,m> where σ is a process memory of the sort Memory and t is the associated date of the sort Date.

(2) datend : State —> Date
defines the date at which the current modification, submitted to a state, takes off. The state at that date is complete. The computation of that date depends in particular on the duration of the (current) statement execution. Note that this operation cannot be applicable to the input/output modifications.

(3) precedes : Date x Date —> Bool
defines a total order on Date.

Remark :
The "dateof" and "datend" operations can express, to a given state, equal dates. The relative equations can be written as follow :

dateof (<σ,t,s>) = t

datend (<σ,t,s>) = *if* perfect? (<σ,t,s>)
then t
else dateof (apply (s, <σ,t, nop>)) *fi*

3.4.2. The "a priori semantics" equations

In this section we want to focus our interest in defining the equations which correspond to the "apply" operation, connecting a process state and its modifications, which are of the profile : Modif x State —> State.
Notice that nothing will be said about the communication operations, treated in the following section.
The computation of a sequential process P starts at an initial perfect state, say $\Sigma_0 = < \sigma_0(P),\ t_0, nop>$ where t_0 is the initial date.,

. "failure_{tn}", which record an erroneous computation ;
. "abortion_{tn}" which record an abortion ;
. $\Sigma_n = <\sigma_n (P),\ t_n,\ nop>$ otherwise (t_n being the date of the computation end).

(1) Suppose to be known at time t the perfect state $\Sigma_t = <\sigma_t (P),\ t,\ nop>$ of the process P.

The following are the equations, induced on the syntax, of the apply operation.

apply (nop, Σ_t) = Σ_t

apply (abort, Σ_t) = abortion_t

apply (v:=e, Σ_t) = $<\Sigma'_t,\ t',$ nop>
such that Σ'_t = subst (val (eval (v, Σ_t)), eval (e,Σ_t)) *and* precedes (t,t')

apply ($s_1;s_2,\Sigma_t$) =
apply (s_2, apply (s_1, Σ_t))

apply (if b then s_1 else s_2 fi, Σ_t) =
if eval (b, Σ_t) = true *then*
apply (s_1, Σ_t)
elsif eval (b, Σ_t) = false *then*
apply (s_2, Σ_t)
else failure_t *fi*

(2) Suppose now that the given state of the process P, at that time t, is an imperfect one. In a such state, an elementary modification is being executed.
This fact can be described elegantly by intro-

ducing the operation

$$\text{in} : \underline{\text{Modif}} \times \underline{\text{State}} \longrightarrow \underline{\text{State}} \ (*)$$

To that imperfect state no modification can be applied (and no expression can be evaluated into it) until knowning its perfect state.
So, we have

$$\text{apply } (s_2, \text{ in } (s_1, \Sigma_{t1}))= \text{apply } (s_2, \text{ apply } (s_1, \Sigma_{t1}))$$

$$\text{eval } (e, \text{ in } (s_1, \Sigma_{t1})) = \text{eval } (e, \text{ apply } (s_1, \Sigma_{t1}))$$

Where Σ_{t1} is the perfect required state on which the modification s_1 was submitted. The computation of the modification s_1 halts at a perfect state apply (s_1, Σ_{t1}), which we can describe by an operation

$$\text{statend} : \underline{\text{State}} \longrightarrow \underline{\text{State}}$$

with the equations

$$\text{statend } (\text{apply}(s_1, \Sigma_{t1})) = \text{apply } (s_1, \Sigma_{t1})$$

$$\text{statend } (\text{in}(s_1, \Sigma_{t1})) = \text{apply } (s_1, \Sigma_{t1}).$$

3.4.3. The "binding-operator" equations

3.4.3.1. Communication axiomatisation

In the above section no equations was given to define the "send" and the "receive" operations (ie modifications predicated by the "iostm?" hidden operation). This is due to the approach adopted and to the fact that the communication type [10] requested by the language still intentionally unknown. So different CSPD can be taken into account, according to the choice of the communication type (which may be synchrone, asynchrone, deterministic, nondeterministic and so on) ; and it seems possible to give a communication axiomatisation by means of the same convenient tool.
To be succintly, we treat below two communication types :
- a deterministic and synchronised communication like it occurs in [6] : the rendez-vous protocol oblige the process reaching a communication command (in other words, able to communicate) to wait for the corresponding complementary command in its interlocutor.
- a nondeterministic and asynchronised communication, where the producer and the consumer are totally independent.

Remarks :
(1) No equations can be given to axiomatize the communication synchronised by a rendez-vous protocol because it depends on the processes capabilities.
(2) A communication of the second type can be seen as a basic modification on a predefined identifiers attached to the processes to hold messages.

$$\text{apply } (P.Q!e, <\sigma_t\ (P),t,m>) = <\Sigma'_t,t',\underline{\text{nop}}>$$

$$\textit{let } \Sigma_t = <\sigma_t\ (P),t,m> \textit{ st } \text{precedes } (t,t')$$

$$\text{and } \Sigma'_t = <\text{subst } (\text{val } (\text{eval } (\pi_p\ ,\Sigma_t), \text{eval } (e,\Sigma_t))>$$

$$\text{apply } (P.Q?v, <\sigma_t\ (P), t,m>) = \text{apply } (v:= \bar{\pi}_q, <\sigma_t\ (P),t,m>)$$

Where π_p and π_q are those predefined identifiers attacherd respectivelly to the processes P and Q states.
(π_p and π_q type depends on the communication type. It can be a queue, an infinite queue, a buffer with one element, and so on).

In the above equation $\bar{\pi}_q$ is a copy of π_q delivered by the producer to the consumer : that will permit handling a totally independent production and consommation. We suppose that to be brief in describing the concurrency axiomatisation.

3.4.3.2. Concurrency axiomatisation

The computation of a system designed by the communicating processes X and Y starts in a initial general (and perfect) state $\Sigma_0 = < \Sigma_0(X), \Sigma_0(Y)>$ and may produce a set of such couples as final state. We also include as possible final state : "failure$_{tn}$" and "abortion$_{tn}$" which record respectivelly an erroneous computation and an abortion in one component.
This computation is described by transformations of the general state. To tackle the semantic equational definition of this parallel computation an (binding) operation is introduced (at the interpretation level of the hierarchical construction of the CSPD type) :

$$\text{exec} : \underline{\text{Progr}} \times \underline{\text{State}} \longrightarrow \underline{\text{State}}$$

Knowning at time t the general state Σ_t composed at t by $\Sigma_t(X)$ and $\Sigma_t(Y)$ the states of the disjoints processes X and Y, we want to describe formally

$$\text{exec } (X :: x_1 \ ; \ x_2 || \ Y :: y_1 \ ; \ y_2, <\Sigma_t(X), \Sigma_t(Y)>)$$

This formalization depends on the form of the term x_i and y_i $(i = 1...2)$ and on $\Sigma_t\ (X)$ and $\Sigma_t\ (Y)$ which can be perfect or not perfect.

Before tackling this formalization, we want to point out the fact that it is a little cumbersome. It is the reason why we induce it on the structure of modification terms.

$$\textit{let } \Sigma_t \in \underline{\text{State}} \ ; \ d1,d2,d3,d4 \in \underline{\text{Date}}$$

$$\textit{st } \Sigma_t = <\Sigma_t(X), \Sigma_t(Y)>,$$

$$d1 = \text{datend } (\text{apply } (x_1, \Sigma_t(X)))$$

$$d2 = \text{datend } (\text{apply } (x_2, \Sigma_t(Y)))$$

$$d3 = \text{datend } (\Sigma_t(X))$$

$$d4 = \text{datend } (\Sigma_t(Y))$$

in

(*) Notice that we have the following equality :

$$\text{in } (s',<\sigma,t,s>) = \textit{if } s = \text{nop } \textit{then } <\sigma,t,s> \textit{ else } \text{in } (s', \text{statend } (<\sigma,t,s>)) \textit{ fi}$$

(1) Basic modification

```
bstm? ($x_1$) $\wedge$ bstm? ($y_1$) $\Rightarrow$
exec (X :: $x_1$ ; $x_2$ ||Y :: $y_1$ ; $y_2$, $\Sigma_t$) =
    exec (X :: $x$ || Y :: $y$ , $\Sigma'_t$)
 st case
perfect? ($\Sigma_t$(X)) $\wedge$ perfect? ($\Sigma_t$(Y)) $\Rightarrow$
    . $x$ = $x_2$
    . $y$ = $y_2$
    . if precedes (d1, d2)then $\Sigma'_t$=<apply
          ($x_1$, $\Sigma_t$(X)), in ($y_1$,$\Sigma_t$(Y))>
      elsif precedes (d2, d1) then $\Sigma'_t$ = <in
          ($x_1$, $\Sigma_t$(X)), apply ($y_1$,$\Sigma_t$(Y))>
      else $\Sigma'_t$ = apply ($x_1$, $\Sigma_t$(X)), apply ($y_1$,
          $\Sigma_t$(Y))> fi;
perfect? ($\Sigma_t$(X)) $\wedge$ $\neg$ perfect? ($\Sigma_t$(Y)) $\Rightarrow$
    . $x$ = $x_2$
    . $y$ = $y_1$ ; $y_2$
    . if  precedes (d1, d4) then $\Sigma'_t$ =
         <apply ($x_1$, $\Sigma_t$(X)), $\Sigma_t$(Y)>
      elsif precedes (d4, d1) then $\Sigma'_t$ =
         <in($x_1$, $\Sigma_t$(X)), statend ($\Sigma_t$(Y))>
      else $\Sigma'_t$ = <apply ($x_1$,$\Sigma_t$(X)),
         statend ($\Sigma_t$(Y))>  fi


$\neg$perfect? ($\Sigma_t$(X)) $\wedge$ perfect? ($\Sigma_t$(Y))
    . $x$ = $x_1$; $x_2$
    . $y$ = $y_2$
    . if  precedes (d3,d2) then $\Sigma'_t$ = <$\Sigma_t$(X),
          apply ($y_1$, $\Sigma_t$(Y))>
      elsif precedes (d2,d3) then $\Sigma'_t$= <statend
          ($\Sigma_t$(X)), in ($y_1$, $\Sigma_t$(Y))>
      else $\Sigma'_t$ = <statend ($\Sigma_t$(X)), apply ($y_1$,
          $\Sigma_t$(Y))>  fi .

    esac ;
```

(2) Communication modification

(2.1) Case of deterministic and synchronous communication

```
iostm?($x_1$) $\wedge$  bstm?($y_1$) $\Rightarrow$
exec (X :: $x_1$ ; $x_2$|| Y :: $y_1$ ; $y_2$, $\Sigma_t$) =
    exec(X :: $x_1$ ; $x_2$ || Y :: $y$, $\Sigma'_t$)
 st case
perfect? ($\Sigma_t$(X)) $\wedge$ perfect? ($\Sigma_t$(Y))> $\Rightarrow$
    .  $y$  = $y_2$
    . $\Sigma'_t$ = <$\Sigma_t$(X), apply ($y_1$, $\Sigma_t$(Y))> ;
perfect? ($\Sigma_t$(X))$\wedge\neg$perfect?($\Sigma_t$(Y)) $\Rightarrow$
    . $y$ = $y_1$; $y_2$
    . $\Sigma'_t$= <$\Sigma_t$(X), statend ($\Sigma_t$(Y))> ;
$\neg$perfect? ($\Sigma_t$(X)) $\wedge$ perfect? ($\Sigma_t$(Y)) $\Rightarrow$
    . $y$ = $y_2$
    . if precedes (d3,d2) then $\Sigma'_t$ = <statend
         ($\Sigma_t$(X)), in ($y_1$,$\Sigma_t$(X))>
   elsy  precedes (d2,d3) then $\Sigma'_t$ = <$\Sigma_t$(X),
         apply ($y_1$,$\Sigma_t$(Y))>
   then $\Sigma'_t$ = <statend ($\Sigma_t$(X)), apply ($y_1$,
         $\Sigma_t$(Y))> fi ;
esac ;
% bstm ? ($x_1$) $\wedge$ iostm? ($y_1$) $\Rightarrow$ Simular to the
 preceding equation, due to the communicati-
 vity of the "parall" operation %
iostm? ($x_1$) $\wedge$ iostm? ($y_1$) $\Rightarrow$
exec(X :: $x_1$ ; $x_2$|| Y :: $y_1$ ; $y_2$, <$\Sigma_t$(X),
    $\Sigma_t$ (Y)>) =
if perfect? ($\Sigma_t$(X)) $\wedge$ perfect? ($\Sigma_t$(Y)) then
     if  $x_1$ = X.Y! $e_1$ $\wedge$ $y_1$ = Y.X?$v_2$
      then exec (X :: $x_2$ ||Y :: $y_2$, <$\Sigma_t$ (X),
         apply ($v_2$ := $e_1$, $\Sigma_t$(Y))>)
     elsif  $x_1$ = X.Y?$v_1$ $\wedge$ $y_1$ = Y.X!$e_2$
    then exec (X:: $x_2$ || Y :: $y_2$,
         <apply ($v_1$ := $e_2$, $\Sigma_t$(X)), $\Sigma_t$(Y)>)
    elsif  $x_1$ = X.X!$e_1$$\vee$ $x_1$ = X.X?$v_1$ $\vee$
           $y_1$ = Y.Y!$e_2$$\vee$ $y_1$ = Y.Y?$v_2$
           then  $\text{failure}_t$
           then  $\text{abortion}_t$
     fi

else if   perfect? ($\Sigma_t$(X))
     then  exec (X :: $x_1$ ; $x_2$|| Y :: $y_1$ ; $y_2$,
         <$\Sigma_t$(X), statend ($\Sigma_t$(Y))>)
     else  % perfect? ($\Sigma_t$(Y)) %
         exec (X : :  $x_1$ ; $x_2$ || Y :: $y_1$ ; $y_2$,
         <statend ($\Sigma_t$(X)), $\Sigma_t$(Y)>)
  fi
fi
```

(2.2) Case of nondeterministic and asynchronous communication

```
iostm? ($x_1$) $\wedge$  bstm? ($y_1$) $\Rightarrow$
exec (X :: $x_1$; $x_2$ || Y :: $y_1$; $y_2$, $\Sigma_t$) =
    if  $x_1$ = X.X!e $\vee$ $x_1$ = X.X?v then $\text{failure}_t$
  else  exec (X :: $x$ ||  Y :: $y$ ,$\Sigma'_t$ ) fi
```

```
st case perfect? ($\Sigma_t$ (X)) $\Rightarrow$
    . if  $x_1$ = X.Y!e then $x$ =($\pi_x$:= e; $x_2$)
      else % $x_1$ = X.Y?v %  $x$ =(v := $\bar{\pi}_y$ ; $x_2$)
    .  $y = y_1$
    . $\Sigma'_t = \Sigma_t$;
      perfect? ($\Sigma_t$(Y)) $\Rightarrow$
    . $x = x_1 ; x_2$
    . $y = y_2$
    . if precedes (d3,d2) then $\Sigma'_t$ = <statend
          ($\Sigma_t$(X)), in ($y_1$, $\Sigma_t$(Y))>
      elsif precedes (d2,d3) then $\Sigma'_t$= <$\Sigma_t$(X),
          apply ($y_1$,$\Sigma_t$(Y))>
      else $\Sigma'_t$=<statend ($\Sigma_t$(X)), apply ($y_1$,
          $\Sigma_t$(Y))> fi  ;
esac ;

% bstm ? ($x_1$) $\wedge$ iostm? ($y_1$) $\Rightarrow$ similar to the
  preceding equation %

iostm? ($x_1$)  $\wedge$  iostm? ($y_1$) $\Rightarrow$
exec (X :: $x_1$ ; $x_2$ || Y :: $y_1$ ; $y_2$, $\Sigma_t$) =
    if $x_1$ = X.X!$e_1$ $\vee$ $x_1$ = X.X?$v_1$ $\vee$ $y_1$ = Y.Y!$e_2$
        $\vee$ $y_1$ = Y.Y?$v_2$
     then failure$_t$ else
              exec (X :: $x$; $x_2$||Y :: $y$ ; $y_2$, $\Sigma_t$) fi
 st case perfect? ($\Sigma_t$(X))$\wedge$perfect? ($\Sigma_t$(Y))$\Rightarrow$
    . if $x_1$ = X.Y!$e_1$ then $x$  =($\pi_x$ := $e_1$) else
          % $x_1$ = X.Y?$v_1$ % $x$ =($v_1$ := $\bar{\pi}_y$)fi
    . if $y_1$ = Y.X!$e_2$ then $y$=($\pi_y$ := $e_2$)else
        %$y_1$ = Y.X?$v_2$ % $y$ = ($v_2$ := $\bar{\pi}_x$) fi
        perfect? ($\Sigma_t$(X))$\wedge\neg$ perfect? ($\Sigma_t$(Y))$\Rightarrow$
    . if $x_1$ = X.Y!e then $x$= $\pi_x$ := $e_1$else
        % $x_1$ = X.Y?$v_1$% $x$ = $v_1$ := $\bar{\pi}_y$ fi
    .  $y = y_1$
       $\neg$perfect? ($\Sigma_t$ (X)) $\wedge$ perfect? ($\Sigma_t$ (Y))$\Rightarrow$
    .  $x$  = $x_1$
    . if $y_1$ = Y.X!$e_2$ then $y$=($\pi_y$ := $e_2$) else
        % $y_1$ = Y.X?$v_2$% $y$ =($v_2$ := $\bar{\pi}_x$)fi
esac ;
```

(3) Conditional modification

```
cstm? ($x_1$) $\wedge$ perfect?($\Sigma_t$(X)) $\Rightarrow$
    exec (X :: $x_1$ ; $x_2$ || Y :: $y_1$ ; $y_2$,
          <$\Sigma_t$(X),  $_t$(Y)>) =
    let $x_1$ = if b then $s_1$ else $s_2$ fi
    in
    if  eval (b, $\Sigma_t$(X)) = true
      then exec (X :: $s_1$ ; $x_2$ || Y :: $y_1$ ; $y_2$,
            <$\Sigma_t$(X), $\Sigma_t$(Y)>)
      elsif eval (b, $\Sigma_t$(X)) = false
        then  exec (X :: $s_2$; $x_2$ || Y :: $y_1$; $y_2$,
            <$\Sigma_t$(X), $\Sigma_t$(Y)>)
       else failure$_t$
fi

% cstm? ($y_1$)  $\wedge$ perfect? ($\Sigma_t$(Y)) $\Rightarrow$ similar to
 the preceding one.
  cstm? ($x_1$) $\wedge\neg$perfect ($\Sigma_t$(X)) $\Rightarrow$ not given
 here to shorten the paper.
 This equation depends on the form of $y_1$.
  cstm? ($y_1$) $\wedge$ $\neg$perfect? ($\Sigma_t$(Y)) $\Rightarrow$ similar to
 the preceding one %
```

3.4.3.3. General remarks :

(1) The formalization is involved in three kinds of equations (analoguous one are also written) depending on the form of the modification considered.

(2) The evaluation of boolean expression results in some cases undefined values. This is necessary for describing loops within a single process.

(3) Successful termination of the system designed by two processes is noted by a couple of complete states different from "failure" and "abortion".

(4) Deadlock arise when the processes are involved in some cyclic deterministic and synchronuous communication.

(5) The definitions above presuppose that exec will never be supplied with a pair of states <Σ_t(X), Σ_t(Y)> both of which are imperfect. The partial function can easily be extended to cover the missing case :

```
    $\neg$perfect ($\Sigma_t$(X)) $\wedge\neg$perfect? ($\Sigma_t$(Y)) $\Rightarrow$
      exec (X :: x || Y :: y, <$\Sigma_t$(X), $\Sigma_t$(Y)>) =
          if precedes (datend ($\Sigma_t$(X)),
              datend ($\Sigma_t$ (Y)))
          then exec (X :: x || Y :: y,
            <statend ($\Sigma_t$(X)), $\Sigma_t$(Y)>)
          else exec (X :: x || Y :: y,
              <$\Sigma_t$(X), statend ($\Sigma_t$(Y))>) fi
```

This would hold for all forms of x and y.

4. CONCLUSION :

Switching on to concurrency and communication, some inherent computational complexity arises and makes necessary the introduction of time notion.

Commonly this notion is considered as an implementation tool, we regard it to be a high abstraction level in specifying parallel programs and formalizing its semantics (similarly to the sequence notion introduced to formalize semantics of sequential programming).

Defining the meaning of systems of communicating processes in a "mathematical" way, the main problem is to find appropriate domains for the semantical models.

To avoid these crucial problems, purely algebraic methods are used. The presented abstract type takes together into account the phrases of the language and elements allowing to express their meaning, like states, "eval", "apply" and "exec" operations.

Remark that if the type is relatively complex, the description of phrases are more simple (language level of the hierarchical construction, the other levels being used to write axioms). In addition, different communication types can be easily handled in an elegant and a simple manner.

Extending the method to treat more complex constructs, especially those inducing nondeterminism and showing its complementarity with axiomatic methods (proof rules), remains to developp.

References :

[1] M. BROY & M. WIRSING, "*Programming languages as abstract data types*", 5th CAAP, Lille 1980.

[2] M. BROY & M. WIRSING, "*On the algebraic specification of finitary infinite communicating sequential processes*", IFIP-TC2, Garmish-Partenkirchen 1982, pp 135-162.

[3] N. FRANCEZ & al, "*Semantics of non determinism, concurrency and communication*", JCSS 19, 1980, pp 203-308.

[4] M.C. GAUDEL, "*Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation*", Thèse d'état INPL, Nancy, 1980.

[5] J.V. GUTTAG, "*Notes on data abstraction*", LNCS69, 1979, pp 593-615.

[6] C.A.R. HOARE, "*Communicating sequential processes*", CACM 20,1978, pp 666-677.

[7] B. LISKOV & al, "*Abstraction mechanism in CLU*", CACM 20, 1977, pp 546-576.

[8] C. PAIR, "*Abstract data types and algebraic semantics of programming languages*", TCS 1, 1982, pp 1-38.

[9] P. PEPPER, "*A study on transformational semantics*", LNCS 69, 1979, pp 382-405.

[10] G.R. PERRIN, "*Specification and verification of communication of processes*", CRIN 82-R-020, Nancy 1982.

[11] G.D.PLOTKIN, "*An operational semantics of CSP*", IFIP-TC2, Garmish-Partenkirchen 1982, pp 185-208.

[12] S.N. ZILLES, "*A look at algebraic specifications*", IBM San José Research Laboratory, U.S.A. 1981.

Introduction to the Poker Parallel Programming Environment

Lawrence Snyder

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

The Poker Parallel Programming Environment is a graphics-based, interactive system for programming the Configurable, Highly Parallel (CHiP) Computer. Designed to support nearly all aspects of parallel programming in one integrated system, Poker has been implemented as a (~35,000 line) C program on the VAX 11/780 under UNIX. It provides a number of novel features including graphics programming of parallel processor communication.

Although much sequential programming can be accomplished with only the support of a programming language compiler, loader and run-time system, parallel programming is too complex to be done with such rudimentary facilities alone. The Poker System is an interactive programming environment to support the Configurable, Highly Parallel (CHiP) Computer [1]. The Poker System is not itself a parallel program, but rather it is a "front end", implemented on the VAX/780 under UNIX. It is a front end to a preprototype version of the CHiP hardware, called the Pringle, which is a 64 processor parallel computer emulating the CHiP [2]. In addition, Poker is a front end for a complete software emulator for the Pringle.

The Poker System enables the programmer to define a large family of CHiP architectures with 4 to 4096 processors. Programs can be written and emulated for any family member. Facilities are provided to define processor interconnection structures graphically, to program the processing elements, to compile, coordinate [3], and load, to perform single and multistep execution, and to peek and poke (whence the name) at the memory of the architectures.

CHiP Programming is Something Else

The programming environment provided by Poker is somewhat unconventional due partly to novel properties of the CHiP Computer and partly to novel properties of the system itself. To increase the accessibility of subsequent sections, we discuss here the *activity* of CHiP programming and the *role* Poker plays.

Programming, of course, is the conversion of an (abstract) algorithm that is "machine independent" into a form suitable for execution on a particular computer. Thus, to begin programming a CHiP machine, we need to have a parallel algorithm in mind. The algorithm is presumed to have the form of a graph whose vertices are processes and whose edges specify the communication paths among the processes.

For example, Figure 1 gives an algorithm that uses a binary tree as the communication graph. The algorithm finds the maximum of a set of numbers (stored one per process in a local variable called "val") and then multiplies each number by the maximum. The maximum is found by "floating" the largest value in each subtree to the root of that subtree. Then the global maximum is broadcast back through the tree where each process multiplies it times its local "val." Notice that although there are fifteen processes in the tree, there are only three *types* of processes used.

The conversion of this algorithm to run on a CHiP computer, i.e., the programming, is straight forward.* It involves

(a) embedding the communication graph into the switch lattice,

(b) programming the process types in a sequential programming language,

(c) assigning one of the process types to each processor,

(d) naming the data path ports, and

(e) compiling, assembling, coordinating, and loading the program.

We consider each of these activities in turn.

Embedding the communication graph into the switch lattice requires that we program the switches of the lattice so that the processors have a topology that matches (or is a super set of) the topology of the communication graph. This embedding operation is done graphically (rather than symbolically) in the Poker System using the Switch Settings mode. Figure 2 illustrates a particular embedding of the fifteen node binary tree into the lattice. Processor (1,2) is the root of the processor tree, processor (1,1) is a leaf, and processor (1,3) is unused.

Next we program the three process types in a sequential language, XX. Each process is viewed as a procedure with (optional) parameters and local variables. In addition to the usual declarations we must specify the *port names*, symbolic names used by a process to refer to other processes with which it communicates. Figure 3 shows the XX code for the three process types. In the programs the symbol '<-' is used for input/output; assigning to a port name, e.g., PARENT <- val, causes output, and assigning from a port name, e.g., max <- PARENT, causes input.

The construction of the processor tree in the switch lattice to match the communications graph gives an implicit association between the processes of the algorithm and the processors. We make this relationship

This work is part of the Blue CHiP Project. It is supported in part by the Office of Naval Research Contracts N00014-80-K-0816 and N00014-81-K-0360. The latter is Task SRO-100.

*Assuming familiarity with the CHiP Computer [1].

0190-3918/83/0000/0289$01.00 © 1983 IEEE

explicit by assigning process names to the appropriate processors using the Code Names mode of the Poker System. Figure 4 gives the result.

Next, the port names mentioned in each process must be associated with a specific data path. Each processor has eight ports corresponding to the compass points. Only those connected by an active data path to another PE need be named. This activity is performed using the Port Names mode of Poker. Figure 5 shows the result of naming the ports.

The algorithm is now programmed. Next, each process type mentioned in the Code Names specification is compiled into assembly code. The assembly code is then "coordinated," i.e., modified so that the CHiP Computer can run it synchronously. The coordinated programs are assembled to produce processor object code. The interconnection structure is "compiled" to produce switch object code. The object codes are loaded into the machine and executed.

Description of the Poker Environment

In the last section we used the Poker Programming Environment to embed graphs, to define processes, to assign processes to processors, and to declare port names. The discussion implied the existence of certain facilities in the Poker System. Now we give a more complete description of those facilities.

The Poker System is an interactive programming environment that uses two displays: The primary display is a high resolution (1024 × 768 pixel) bit-mapped display, and the secondary display is a conventional character display. The two displays are used to increase the amount of information available to the programmer. Most activity takes place on the primary display; XX programming is usually done on the secondary display.

The primary display has the form illustrated in Figures 2-5. The bottom square region, called the *field*, is where most of the programming activity takes place. The field always displays some schematic representation of the two dimensional array of processors being programmed. The exact form of the representation changes depending on whether the programmer is performing a graph embedding, a process assignment, a port declaration, etc. Since the field is not always large enough to show the whole schematic representation, a map of that portion being displayed is given in the upper left-hand corner. Status information, diagnostics and miscellaneous data are given in the upper right region of the display, called the *chalkboard*. The bottom line of the chalkboard is the *command line*, used for specifying the few textual commands* required by Poker, such as reading library files.

The logical structure of the Poker Environment is shown in Figure 6. It provides an integrated set of facilities to

- define architecture characteristics (CHiP Parameter),
- embed communication graphs (Switch Settings),
- program process code segments (XX language),
- assign processes to processors (Code Naming),
- declare port names (Port Naming),
- compile, coordinate, assemble and load (Command Request),
- execute, trace, peek and poke (Trace Values).

We now describe each of these facilities in detail.

Architectural definition. Because Poker is intended to be a laboratory tool for studying CHiP programming, it has been designed to support a number of CHiP family architectures. Programs can be written for logical CHiP machines with from 4 to 4096 processors. All of these logical machines can be emulated using a software emulator, and one family member, the 64 processor version, will be able to be run on a hardware emulator, the Pringle [2], when it is completed. Consequently, the programmer begins using Poker by specifying the characteristics of the underlying logical architecture. These include the number of processing elements and the amount of routing capability needed for the lattice (corridor width [1]). The default parameters are those that match the machine defined in the previous session, or, if there was none, then the parameters of the Pringle hardware.

Graph embedding. The field of the primary display shows the lattice of the current architecture, as illustrated in Figure 1. The activity is largely that previously described; the programmer connects the processors (represented as boxes) with line segments to define edges. Graphics primitives based on cursor keys permit edges to be drawn and erased. Facilities are available for following graph edges, managing the display (e.g., centering), saving embeddings, reading in library embeddings, etc.

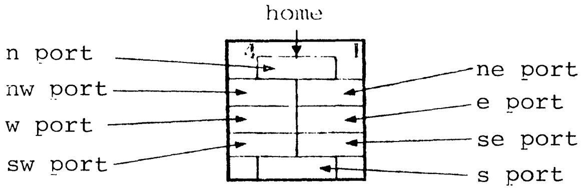
Programming the process code segment. The XX (Dos Equis) sequential programming language is a simple scalar language for defining processes. The language has four data types (Boolean, character, integer and real), the common control structures (**while, for, if-then-else**, etc.), vectors and the usual supply of scalar arithmetic and logical operators. In addition to data type declarations, one can also declare scalar variables to be port names, procedure parameters, or variables to be traced. Input/output is performed by assigning from or to a port name. The semantics are "data-driven:" writes occur immediately and reads wait on the arrival of data, if necessary. XX process codes are generally developed on the secondary display using a standard editor.

Process assignment. The processors are assigned processes using a field display on the primary terminal like those in Figure 4. The programmer enters the name of the process procedure on the first line of the processor box. If the procedure has formal parameters, then values for the actual parameters can be entered on the following (four) lines. Facilities are provided for buffering the contents of a box and then automatically depositing the contents of the buffer into processors in whole regions of the processor array. In this way the programmer is saved from manually entering repeated information when the algorithm exhibits uniformity.

Port declarations. The field of the primary display has the form illustrated in Figure 5. Each processor has up to eight incident edges as a result of the graph embedding, and it has been assigned a process which refers to up to eight port names. These are matched

*Poker is completely interactive; most actions are given as a single key stroke and have immediate effect.

using the port declaration. The processor box is divided into eight windows:



The programmer enters the names used by the assigned process code into the window for that edge. The names are clipped to the first five characters. Facilities are provided for displaying unclipped names in the chalkboard, and like the process assignment, it is possible to buffer port assignments and deposit them automatically in whole regions of the processor array.

Program translation. The preceding facilities provide a means of specifying the elements of a Poker program. They are then converted into executable form. The XX compiler converts each process to assembly code. The coordinator [3] then attempts to convert the process assigned to each processor into a form that permits the entire program to run with synchronous (i.e., not data-driven) execution. [This step can be by-passed and the processes can be run in data-driven form.] If coordination is successful, the processors may all have different assembly codes associated with them. In any event the assembly converts the assembler code to object form. The connector "compiles" the graphical representation of the communication graph into an object form. The object code and the object graph as well as the actual parameter values are loaded into the emulator (or the Pringle).

Execution. The resulting program is executed and the traced variables are displayed; the field is similar to that used for process assignment. The execution can proceed for a given number of steps, or until a displayed value changes. When the execution is suspended, any of the displayed values can be changed. When execution resumes these new values are poked back into the processor memory.

Further detail about the Poker Environment can be fo:ınd in the references [4,5].

References

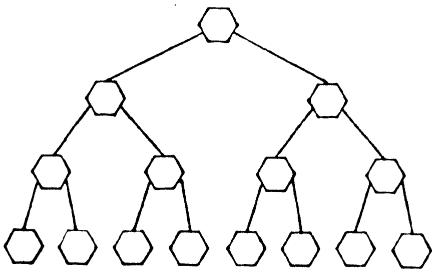
[1] Lawrence Snyder
Introduction to the Configurable, Highly Parallel Computer
Computer, 15(1): 47-56, January 1982

[2] J. Timothy Field, Alejandro A. Kapauan, and Lawrence Snyder
Pringle: A Parallel Processor to Emulate CHiP Computers
Technical Report CSD-TR-433, Purdue University, 1983

[3] Janice E. Cuny and Lawrence Snyder
Compilation of Data-driven Programs for Synchronous Execution
Proceedings of the 10th Symposium on the Principles of Programming Languages, ACM, pp. 197-202, 1983

[4] Lawrence Snyder
Parallel Programming and the Poker Environment
Technical Report CSD-TR-443, Purdue University, 1983

[5] Lawrence Snyder, Steven S. Albert, Carl W. Amport, Brian G. Beuning, Alan J. Chester, John P. Guaragno, Christopher A. Kent, John Thomas Love, Eugene J. Shekita, Carleton A. Smith
The Poker Programming Environment and its Implementation
Technical Report CSD-TR-410, Purdue University, 1982

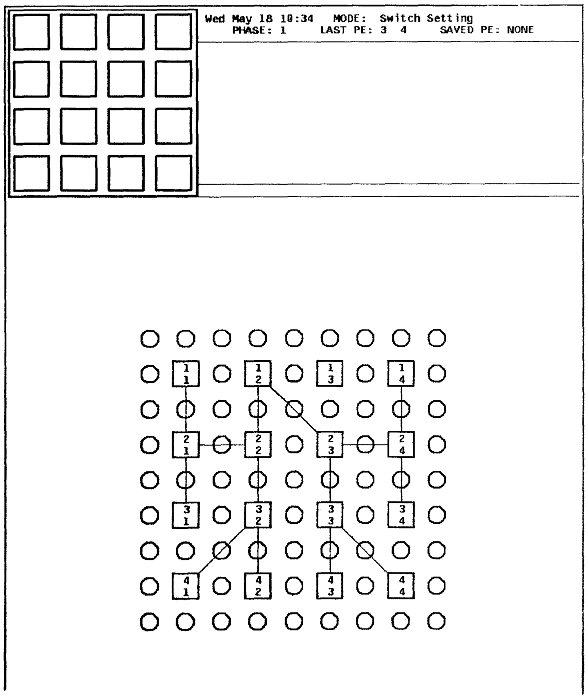


```
leaf process:
    write val to parent;
    read max from parent;
    val ← val · max;

root process:
    read x from left child;
    read y from right child;
    max ← max (x ,y , val);
    write max to left child;
    write max to right child;
    val ← val · max;

ancestor process:
    read x from left child;
    read y from right child;
    write max (x ,y , val) to parent;
    read max from parent;
    write max to left child;
    write max to right child;
    val ← val · max;
```

Figure 1. An algorithm; each leaf is an instance of the leaf process, the root is an instance of the root process and all other nodes are instances of the ancestor process.



Wed May 18 10:34 MODE: Switch Setting
PHASE: 1 LAST PE: 3 4 SAVED PE: NONE

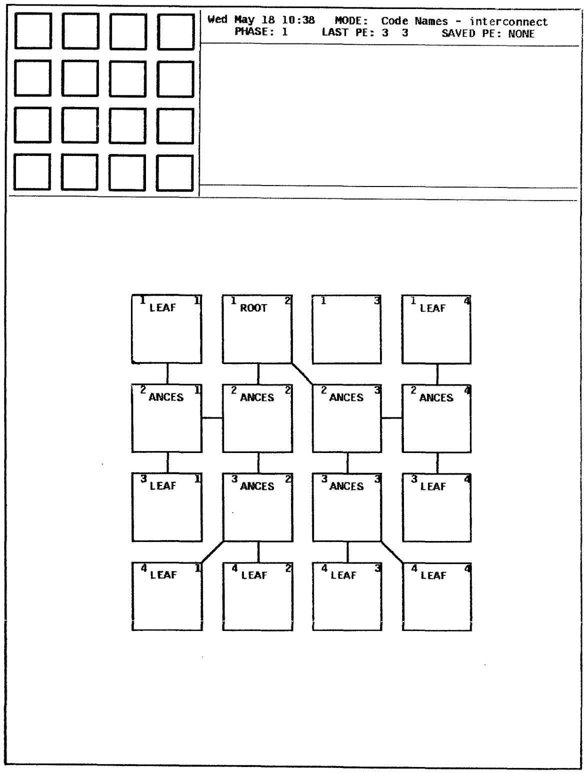
Figure 2. An embedding of the 15 node binary tree.

```
code leaf (val);
ports PARENT;
begin
int max, PARENT;
PARENT <- val;
max <- PARENT;
val:=val * max;
end.


code root (val);
ports LCHILD, RCHILD;
begin
int x,y, max, val,
  LCHILD, RCHILD;
x <- LCHILD;
y <- RCHILD;
if x> y then max:=x
  else max:=y;
if val > max then max:=val;
LCHILD <- max;
RCHILD <- max;
val:=val * max;
end.
```

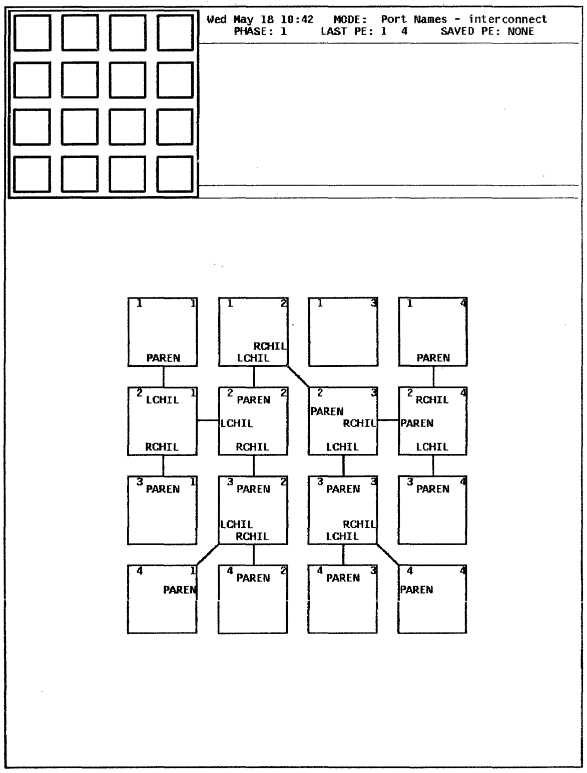
```
code ancestor (val);
ports PARENT,LCHILD,RCHILD;
begin
int x,y, max, val,
  PARENT, LCHILD, RCHILD;
x <- LCHILD;
y <- RCHILD;
if x> y then max:=x
  else max:=y;
if val > max then max:=val;
PARENT <- max;
max <- PARENT;
LCHILD <- max;
RCHILD <- max;
val:=val * max;
end.
```

Figure 3. Code for the three process types.



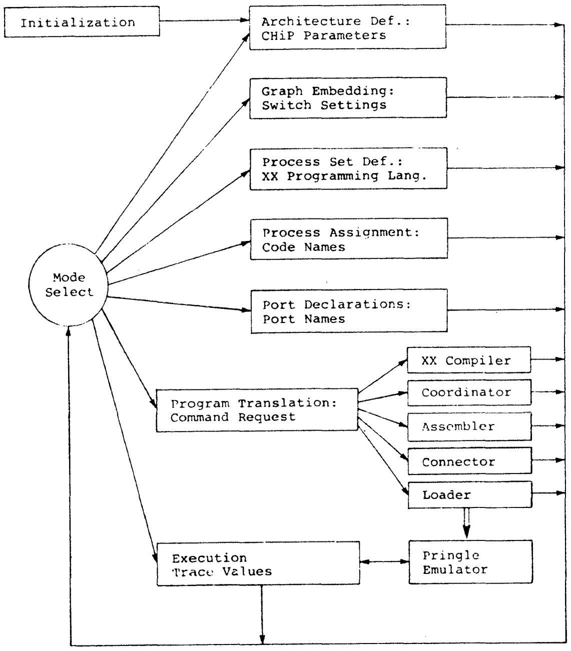
```
Wed May 18 10:38   MODE:  Code Names - interconnect
       PHASE: 1     LAST PE: 3  3      SAVED PE: NONE
```

Figure 4. Assignment of process names to processors; note that the name "ancestor" has been clipped to five characters.



```
Wed May 18 10:42   MODE:  Port Names - interconnect
       PHASE: 1     LAST PE: 1  4      SAVED PE: NONE
```

Figure 5. The specification of the port names; note that the names have been clipped to the first five characters.



Initialization
Architecture Def.: CHiP Parameters
Graph Embedding: Switch Settings
Process Set Def.: XX Programming Lang.
Process Assignment: Code Names
Mode Select
Port Declarations: Port Names
XX Compiler
Coordinator
Program Translation: Command Request
Assembler
Connector
Loader
Execution Trace Values
Pringle Emulator

Figure 6. The logical structure of the Poker Environment.

A High Level Analysis Tool For Concurrent Programs

Paolo Mancarella and Franco Turini
Dipartimento di Informatica, Università di Pisa
C.so Italia,40, I-56100 Pisa

ABSTRACT

The design of a tool which analyzes Concurrent Sequential Processes-like programs is presented. The purpose of the analyzer is to favor the understanding of a concurrent program via its simplification with respect to constraints interactively provided by the user. Constraints can be either on the input data or on possible synchronizations. The analyzer is based on parallelism removal techniques and symbolic evaluation techniques.

1. INTRODUCTION

Much research effort has been recently devoted to the design of languages for parallel processing and to the study of their formal semantics /5,6,11/. Several basic principles, like message passing and separation of workspaces are now widely accepted. It is then time to begin the design of suitable programming environments for such a class of languages. Among the tools one would like to have there are the so called static analyzers, i.e. programs able to favor the discovery of properties of the run-time behavior of parallel programs.
The paper reports on the design of a tool for analyzing concurrent programs written in a CSP-like language. The basic idea is to apply the technique of symbolic evaluation, which has been successfully applied to sequential programs, to parallel ones. The process of symbolically evaluating a program is based on the ability of executing it on partially specified input data. Such an ability can be exploited to several purposes:

(1) To obtain symbolic statements, i.e. logical assertions involving the variables of the program, about the run-time behavior /1,3/.
(2) To prove the correctness of a program, when it is annotated by suitable assertions /7/.
(3) To partially compile a program in order either to obtain a more efficient one on a subset of possible input data or to get a better understanding of the behavior of the program /4,13/.

Our approach intends to exploit symbolic evaluation techniques to obtain a better understanding of the behavior of parallel programs under explicit hypotheses on the behavior of its environment. Such hypotheses include restrictions on inputs received from outside.

Basically, the system accepts a parallel program and transforms it in two respects:

- It tries to resolve part of the parallelism by substituting it with sequential nondeterministic code.
- It annotates control points of the resulting program with assertions about the state at those points.

The transformations are interactively guided by the user, who, acting as the external environment, can force the behavior of the program either providing restrictions on the input data or forcing certain process synchronizations instead of others.

The attempt of eliminating parallelism is motivated by the desire of obtainig a better understanding of the program. In fact, on one hand we expect that sequential reasoning is easier than parallel reasoning, on the other hand putting together two parallel modules allows to move symbolic information from one process to the other, possibly leading to the simplification of both. The design of parallelism removal has been deeply influenced by several approaches to the semantics of CSP-like languages. Indeed, several authors /5,11/ propose to reduce the notion of paralleism to the notion of sequentiality plus nondeterminism,

This work was partly supported by CNR-PFI under contract No. 81.02053.97.

ultimately denoting a parallel program by a multi-valued function.

Our analysis tool is organized in three steps:

- First of all, a process is analyzed in isolation in order to annotate it with assertions about its local symbolic states.
- Second of all, processes are iteratively matched two by two in order to eliminate parallelism.
- Finally program transformation techniques are applied to the resulting program.

The paper is organized as follows: section 1 describes the parallel language we are dealing with. Section 2 describes the tool in some detail, while section 3 contains an example which puts in focus the characteristics and the possible use of the tool.

2. THE LANGUAGE CSP-L

The language is a slight modification of Communicating Sequential Processes /6/. The differences do not concern the basic features of the language, i.e. message passing, nondeterminism and static configuration of the program.

The syntax of CSP-L is described in a BNF-like form, where {x}* denotes 'n' occurrences of 'x', n≥0, and {x} denotes at most one occurrence of x.

```
<system> ::= <process> {|| process}*
<process> ::= <name> :: [ <command> ]
< command> ::= skip | < assignment> | < input> |
               < output> | < composite_command> |
               < alternation> | < repetition>
< assignment> ::= < variable> :=< expression>
< input > ::= < channel_id> ? < variable>
< output > ::= < channel_id > ! < expression >
< composite_command> ::= < command > ; < command >
< alternation > ::= [ <guarded_command>
                     {□ < guarded_command> } * ]
< guarded_command > ::= < guard > → < command>
< guard > ::= <boolean_expression >
< repetition > ::= *[<do_guard_command>
                     { □ <do_guard_command>}*]
<do_guard_command> ::= <guard> → < do_body >
<do_body > ::= <command> |{ < command>; } EXIT
```

The most important differences between CSP and CSP-L are the use of channels for communication and the definition of alternative and repetitive commands. In fact, the former does not deal with input guards and the latter has an explicit exit condition. Some motivations for similar modifications can be found in /12/.

3. THE ANALYSIS TOOL

The input of the analysis tool is the abstract syntax tree of the program. Each step performs some kind of tree transformation. The transformations come in two categories:

- annotating the arcs of the trees with assertions.
- transforming the trees by merging two of them together, by eliminating subtrees and so on.

Before tackling a description of the phases of the analyzer it is necessary to spend some words about what we mean by **symbolic constants** and **symbolic states.** By symbolic constants we mean a subset of a data domain. Symbolic constants are represented by predicates. If

$$p : D \rightarrow Bool$$

is a predicate on the data domain D, "p" represent the symbolic constant

$$A = \{ x / p(x) = true \}.$$

Applying a basic operation to a symbolic constant is then a predicate transforming operation. For example, let

$$A = \{ x / x > 10 \} ;$$

then "A + 1" must result into the predicate

$$A = \{ x / x > 11 \}.$$

An important issue in designing a symbolic evaluator along these lines is then to provide a "predicate transformer semantics" to each primitive operation of the data types allowed by the language. This issue will not be considered further in this paper and it is dealt with in full in /13/.

A symbolic state is a set of bindings among identifiers and symbolic constants.

From now on by symbolically evaluating a (sequential) program we intend the ability of computing the symbolic state associated to every control point of the program by propagating the initial symbolic constants through the control paths.

The three phases of the tool are named A1, A2, A3 respectively.

3.1. First Phase : A1

The purpose of A1 is to simplify a tree eliminating assignments and sequentialization nodes. Assignment nodes are eliminated by annotating the arcs of the tree with symbolic states which keep track of their effects. Sequentialization nodes are eliminated by embedding the control flow into the structure of the tree.

For example the fragment of process

... c?x; x := f(x); y := g(x); d!y ...

where "c" and "d" are channel names and "f" and "g" are elementary operations, is transformed by A1 into the subtree:

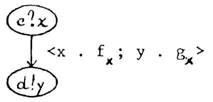


fig. 1

where "f_x" and "g_x" are the symbolic constants resulting from the computation of "f" and "g" respectively.

The elimination of assignments nodes is particularly important because it reduces the necessity of interleaving the independent actions of two processes during their merging.

During this phase the symbolic evaluation of the process is performed **locally.** This is to say that A1 is unable to propagate symbolic constants beyond certain points of control because of the lack of information. First of all, input nodes are considered by A1 points after which the evaluation is restarted with an undefined symbolic state (i.e. the symbolic value associated to each variable of the program is a predicate representing the whole data domain). A1 just assumes that the communication will effectively take place, but it has no information about the value assigned to the input variable by the communication.

The choice of making the whole state of the computation undefined may appear a bit too extreme. Indeed A1 could propagate the part of the symbolic state which is not affected by the communication. On the other hand, A3 will recompute symbolically the whole program to get advantage of the information gained by A2, hence the full symbolic evaluation is deferred to A3.

Secondly, loops in the process induce another class of cut-points for obvious reasons. The subtree representing a loop, produced by A1 is of the kind:

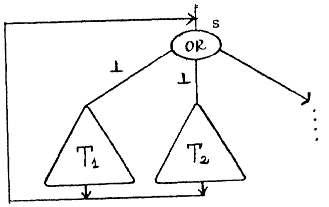


fig. 2

where "s" is the symbolic state valid before entering the loop and "$\perp$" is the undefined symbolic state.

It is worth noting that at this point it does not make any sense trying to apply techniques for synthesizing loop invariants, since the control structure of the program obtained by merging several processes may be radically different.

In summary the tree resulting from the application of A1 has paths of the kind:

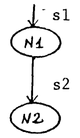


fig. 3

where the symbolic states "Si" are independent. For example "s2" retains the effect of the possible assignments between "N1" and "N2" but no information derived from "s1".

3.2. Second phase : A2

A2 is designed to merge together two symbolic trees, yielding a new one: the idea is to apply A2, step by step, to pairs of symbolic trees constructed by A1 or by previous applications of A2 itself, in order to obtain a symbolic tree describing the control structure of the whole program. Given two symbolic trees T1 and T2, the result of evaluating A2[T1;T2] is a symbolic tree T embedding the control structure of the compound process P1||P2.

The purpose of A2 is to solve the internal communications of two processes, keeping track of them in the symbolic states of the new tree. In some sense A2 simulates the parallel execution of the two processes in that it maintains and updates two continuation points in the two processes and produces new nodes of the final tree, depending upon the currently examined nodes of the input trees.

A2 is recursively defined by cases and we show here the most important ones. As before, a graphical representation of trees is used.

Let T1 and T2 trees like:

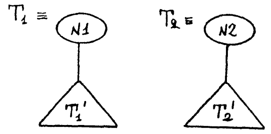


fig. 4

- case i)

let N1, N2 be two matching I/O nodes, i.e. they name the same internal channel. Assume, for example, that N1 corresponds to the input command "c?x" and N2 to the output command "c!exp". The result is simply to modify the current symbolic state to take into account the "assignment"

x := exp

and recursively apply A2 on T1' and T2' without generating any new node for the resulting tree. In all the other cases A2 does not modify symbolic states. In other words, the symbolic evaluation is performed locally also by A2.

- case ii)
Let N1 be an I/O internal node (i.e. it names a channel common to the two processes) and N2 an external I/O node (i.e. it names a channel unknown to the other process). The only way the system of the two processes can proceed is that the communication named in N2 occurs. As a consequence A2 assumes the occurrence of such a communication re-applying to T1 and T2' and creating a copy of N2 for the resulting tree. Graphically:

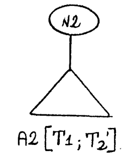


fig. 5

- case iii)
Let N1, N2 be internal unmatching nodes. N1 and N2 may not match either because the processes name different channels or because the communication requests are not complementary. This can be classified as a deadlock situation. The aim of our analyzer, at least in the present design, is to point out the behavior of the program in the absence of deadlock conditions. Hence A2 is designed to abort the execution path. The reason for this design decision is that the deadlock situations detected by A2 may be only apparent, i.e. such events will never occur in an actual execution, but they are generated by the order of application of A2. In other words, deadlock detection can be performed only by global reasoning on the whole program. Indeed, such a design choice guarantees the associativity of A2 /10/.

- case iv)
Let N1 and N2 be external I/O nodes (i.e. they name channels connecting the two processes to other processes or to the external world). In this situation no assumptions can be made about the order of the two communications. Consequently A2 generates a nondeterministic branch in the resulting tree as follows:

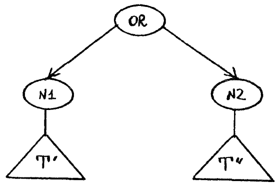


fig. 6

where T' = A2[T1' ; T2]
T" = A2[T1 ; T2'].

- other cases
The behavior of A2 in the other cases is much simpler and will be made evident by the example.

- termination conditions
It is instead very important to discuss the termination of A2, i.e. the reasons why A2 is able to yield a finite tree.
Obviously, A2 halts when it encounters the end of two paths of the input trees, i.e. the termination of execution paths for the two processes.
Furthermore, A2 reaches a termination point when it detects a deadlock condition, as pointed out earlier.
Finally, A2 halts when it is considering two subtrees which have already been merged. In this case a re-entring arc to the subtree previously generated is added to the resulting tree, creating a cyclic path (i.e. a loop). Note that the presence of such paths implies that we are actually dealing with graphs.

3.3. Third phase : A3

When A2 has been iteratively run on pairs of processes to obtain a single symbolic tree, A3 is run to simplify it.
A3 is the phase which embeds most of the power of symbolic evaluation. Indeed it tries to propagate symbolic information through the whole tree and, furthermore, it tries to find simple invariants of loops. The symbolic information is also used to simplify the tree dropping the paths which can be taken by no actual execution. The two tasks of A3, i.e. tree simplification and finding invariants, reflect into two differents modes of working: **forcing** and **non-forcing** mode.

- FORCING MODE
When working in forcing mode, A3 propagates symbolic states and tries to drop paths; in this case the application of A3 to a boolean node can result into different actions:

i) The current symbolic state implies that the guard is always true. The path is simplified eliminating the node.

ii) The current symbolic state implies that the guard is always false. The path is eliminated from the tree.

iii) None of the above implications is true. The only consequence is that the symbolic state propagated by A3 incorporates the guard (see the example).

The application of A3 to an input node results into establishing an interaction with the user. The user is requested to simulate the external environment and to provide an input to the system. The input must be a symbolic constant which will be propagated by A3 through the program. It is worth recalling that a concrete value is a special kind of symbolic constant.

- NON FORCING MODE

When A3 encounters the initial node of a loop (which has been marked by A2 on creating a re-entering arc), it switches to non-forcing mode in the attempt of finding simple invariants for it. All the paths exiting from the node are walked through, starting with the current symbolic state and propagating states around the loop without affecting the tree (i.e. without forcing boolean guards or eliminating paths). Whenever a re-entring arc is found, A3 associates the current symbolic state to the loop node. In this way, when all the cyclic paths have been traversed the loop node mantains all the final states obtained by symbolically executing the loop starting with the symbolic state immediately preceding it (taken as an "attempt" state). At this point A3 intersects these states in order to find possible invariants for the loop. Finally A3 switches to "forcing" mode and restarts the analysis from the loop node, taking as current symbolic state the invariant one.

Some other details about A1, A2 and A3 and their formal description in a LISP-like formalism can be found in /10/.

4. AN EXAMPLE

The previous sections have somewhat vaguely described the design of the analyzer. This section tries to complete the description showing how the analyzer works on a concrete example. In this example the user is not supposed to provide any constraint on the external environment. However, the example is interesting because, also in this case, the process provide a better insight of the run-time behavior of the program.

Let P the CSP-L program:

P :: [P1||P2]

where P1 and P2 are described as follows:

```
P1 ::[  guard := true ;
        r1 ? I1;
        * [ guard  →
             c ! min(I1) ;
             d ? x ;
             [ x ≤ min(I1)  →  guard : = false
             □ x ≥ min(I1)  →
               I1 := ins (x; del(I1; min(I1)))]
          □ ~guard  →  EXIT] ]

P2 :: [  test := true ;
         r2 ? I2 ;
         *[ test  →
             c ? y ;
             d ! max(I2) ;
             [ y ≥ max(I2)  →  test := false
             □ y ≤ max(I2)  →
               I2 := ins (y; del(I2; max(I2)))]
          □~test  →  EXIT] ]
```

Intuitively the program P is designed to transform two sets of numbers, say S1 and S2, into two new sets, say S1' and S2', such that:

$$S1 \cup S2 = S1' \cup S2'$$

and

$$\forall\, x,y\ (\ x \in S1' \text{ and } y \in S2'\ ) \rightarrow\ x \geq y$$

At the very beginning of the computation P1 and P2 read the initial sets on the external channels "r1" and "r2". Then they exchange elements on the internal channels "c" and "d". Let t1 and t2 be the syntactic trees of P1 and P2 respectively.

The symbolic tree shown in fig.7 is the result of A1[t1]. The structure of A1[t2] is quite similar. The effect of assignment statements is embedded into the symbolic states associated with some arcs: they are the only relevant states in the tree (we choose to use a notation for symbolic expressions similar to the syntactic one for simplicity reasons: an effective implementation of our tool must deal with some concrete representation for symbolic constants, as in /13/).

Let now be

T1 = A1 [t1]

T2 = A1 [t2] ;

the next step of the analysis is the merging phase, yielding the symbolic tree for the whole program; in other words we compute

$$T = A2 \, [\, T1 \; ; \; T2 \,] \, .$$

The initial nodes of T1 and T2 are two external communicating nodes: in this case A2 generates a nondeterministic branch and the resulting partial tree is pictured in fig.8.
The subtrees T' and T" are very similar because the symmetry between T1 and T2: Therefore we will consider only the structure of T'.

During the analysis A2 reaches a point such that the intermediate tree has the structure shown in fig.9.
The path labeled (*) corresponds to the situation in which A2 tries to visit the remaining path of T2, while a terminal node of T1 has already been reached. Just after the construction of the node (*), A2 visits the internal communication node of T2

c ? y.

This is obviously a deadlock situation. Hence A2 aborts the path yielding the tree pictured in fig.10.

The final tree computed by A2 is shown in fig.11.
It is worth noting that T has one re-entering node and that the subtree T" is very similar to the other one. The subtrees T3, T4, T5, T6, T7 and T8 have been omitted because they will be suppressed by the third phase.

First A3 works in forcing mode trying to simplify the tree constructed by A2. When it reaches the situation pictured in fig.12, the current symbolic state for the paths labeled (1) and (2) is:

```
s = < guard . true ;
      test  . true >
```

It is worth noting that "s" implies the truth of the boolean guard

p1 = 'guard'

and, conversely, the falseness of the boolean guard

p2 = '~guard'.

Consequently A3 deletes the path corresponding to "p2" (which will never be traversed in an actual execution) and simplifies the other path, obtaining the tree shown in fig.13.

Let us now explain the behavior of A3 when the loop node is detected: the current symbolic state is:

```
s1 = < test  . true ;
       guard . true ;
           y . min (I1) ;
           x . max (I2) ;
          I1 .  ⊥ ;
          I2 .  ⊥ >
```

which is bound to the loop node. At this point the remaining subtree is the one shown in fig.14.
A3 works now in non_forcing mode in the attempt of finding invariant assertions for the loop.
The path with boolean guard

$$G1 = (\; y \geq \max(I2) \wedge \; x \geq \min(I1) \;)$$

is not explored because the current symbolic state is such that

$$S \rightarrow \sim G1.$$

Analogous arguments are valid for the guard

$$G2 = (\; y < \max(I2) \wedge x > \min(I1).$$

The path starting with the boolean node

$$G3 = (y \geq \max(I2) \wedge x \leq \min(I1)$$

is explored but it leads A3 to a terminal node (i.e. it is an exit path). On the other hand, on visiting the remaining path, A3 binds another symbolic state, say S4, to the loop node. The contents of S4 are the following:

```
S4 = < guard . true;
        test . true;
           y . min(I1);
           x . max(I2);
          I1 . ins(x; del(I1; min(I1)));
          I2 . ins(y; del(I2; max(I2)))>
```

At this point of the computation A3 gets the invariant state

```
S5 = S1 ∩ S4 =
 <guard . true;
   test . true;
      y . min(I1);
      x . max(I2)>
```

and restarts visiting the loop with the symbolic state S5 in forcing mode. Finally, note that the information provided by S5 leads to delete some paths in the symbolic tree: in particular those starting with the boolean guards G1, G2, G5 which are always false.

The final tree resulting from the analysis is pictured in fig.15. The symbolic states are the following:

```
S1 = < guard . true;
        test . true >

S2 = < guard . true;
        test . true;
           x . max(I2);
           y . min(I1) >
S3 = < guard . false;
        test . false;
           y . min(I1);
           x . max(I2)>
```

```
S4 =  <guard . true;
        test . true;
          I1 . ins(x'; del(I1; min(I1)));
          I2 . ins(y'; del(I2; max(I2)));
           y . min(I1);
           x . max(I2) >
```

Note that the symbolic expressions in state S3 must represent in some way the relationship

$$\min(I1) \geq \max(I2).$$

Furthermore, by x' and y' we mean the symbolic values hold by variables x and y before the symbolic execution of the matching communications

$$c \;!\; \min(I1) \dashrightarrow c \;?\; y$$

$$d \;!\; \max(I2) \dashrightarrow d \;?\; x.$$

The analyzer has solved the internal communications and the final tree contains I/O nodes naming external channels only. This fact has allowed to move symbolic information from one process to the other and then to the resolution of some boolean guard during the forcing mode analysis of A3.

5. CONCLUSIONS

This paper reports on a two people paper project. The project is a small and "advanced" part of a larger one, whose aim is to build an integrated software environment for developing ADA programs in a distributed computing environment /9/. The data structures and algorithms for an implementation of the analyzer are described in full in /10/ along with other examples. A simplified CSP-like language has been chosen instead of ADA for simplicity reasons. Indeed the principal goal of the authors was to demonstrate that high level analysis based on the paradigm of symbolic evaluation is of some use also in a parallel programming framework. Planned developments of the project are:

- An experimental implementation in Lisp. The reasons for choosing Lisp are its orientation to tree manipulation and, more importantly, the availability of a simplifier already written for a symbolic interpreter.

- Refinements of the aesign including improving the analysis of loops following ideas published in /1/, designing a friendly interface to make the user able to display and inspect the symbolic trees produced by the analyzer and adding to the system other composition operators besides the symmetric composition. It seems interesting, for example, to merge an user process and a server process from the viewpoint of the user, in order to gain information about the behavior of the user process when it gets the service without having to consider the other parts of the server.

References

1. Asirelli,P., Degano,P., Levi,G., Martelli, A., Montanari, U., Pacini,G., Sirovich,F. and F.Turini." A flexible environment for program development based on a symbolic interpreter". **Proc. 4th Int. Conf. of Soft. Eng.** (1979), pp. 251-263.
2. Cheatham,T., Holloway,G. and J.Townley. "Symbolic evaluation and the analysis of programs". **IEEE Trans. on Soft. Eng.,** Vol.SE-5,no. 4 (1979).
3. Darringer,J. and J.King." Applications of symbolic execution to program testing". **IEEE Trans. on Soft. Eng.,** Vol SE-4, no.4 (1978).
4. Ershov,A. "On a partial compilation principle". **Inf.Proc. Letters** 6,2 (1977).
5. Francez,N., Hoare,C.A.R., Lehmann, D.J. and W.P.DeRoever. "Semantics of Nondeterminism, Concurrency and Communication". **JCSS** 19,3 (1979).
6. Hoare, C.A.R. "Communicating Sequential Processes". **CACM** 21,8 (1978).
7. King,J. and S.Hantler. "An introduction to proving the correctness of programs". **Comp. Surveys** 8,3 (1976).
8. King,J. **Program reduction using symbolic execution.** IBM Res. Rep. RJ-3051 (1981).
9. Lijtmaer,N. **Cnet: proposta di ricerca 1981.** (in Italian) Collana Cnet 16 (1981).
10. Mancarella,P. **Uno strumento per l'analisi di programmi concorrenti.** (in Italian) Tesi di Laurea ISI, Universita' di Pisa (1982).
11. Milner,R. **A Calculus of Communicating Systems.** LNCS 92, Springer-Verlag (1980).
12. Queille,J.P. **The CESAR System: an aided design and certification system for distributed applications.** RR 264, IMAG, Grenoble (1980).
13. Turini,F., Ambriola,V., Giannotti,F., and D. Pedreschi. **Symbolic semantics and transformations of applicative programs.** Res. Rep. Dipartimento di Informatica, Univ. di Pisa (1983).

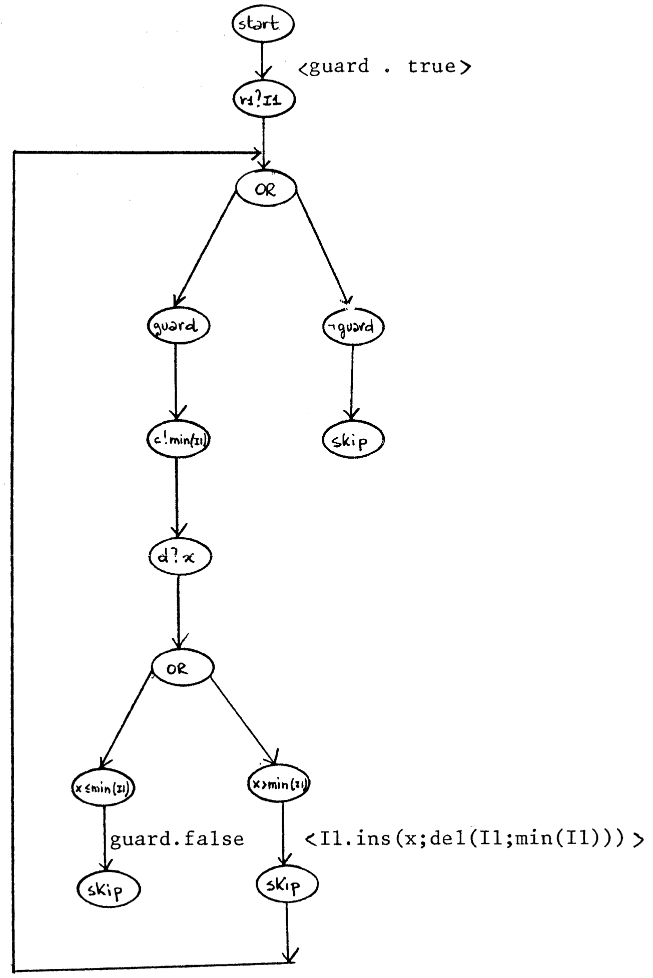


fig. 7

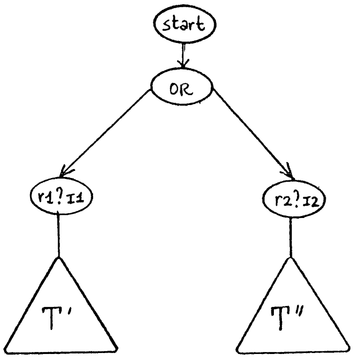


fig. 8

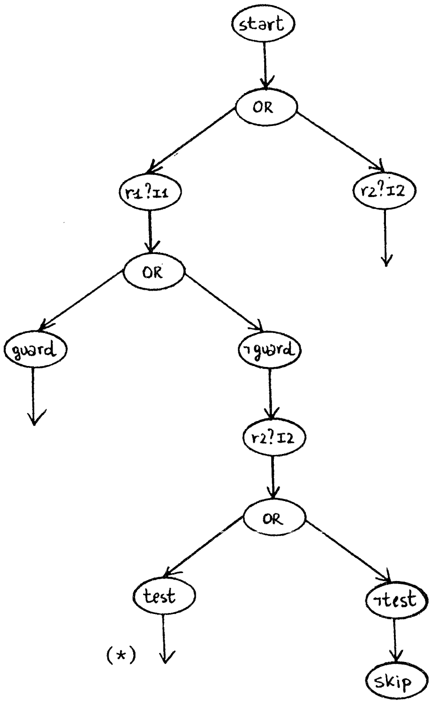


fig. 9

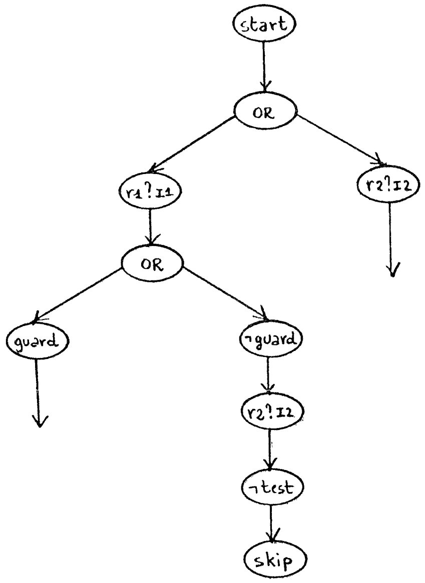
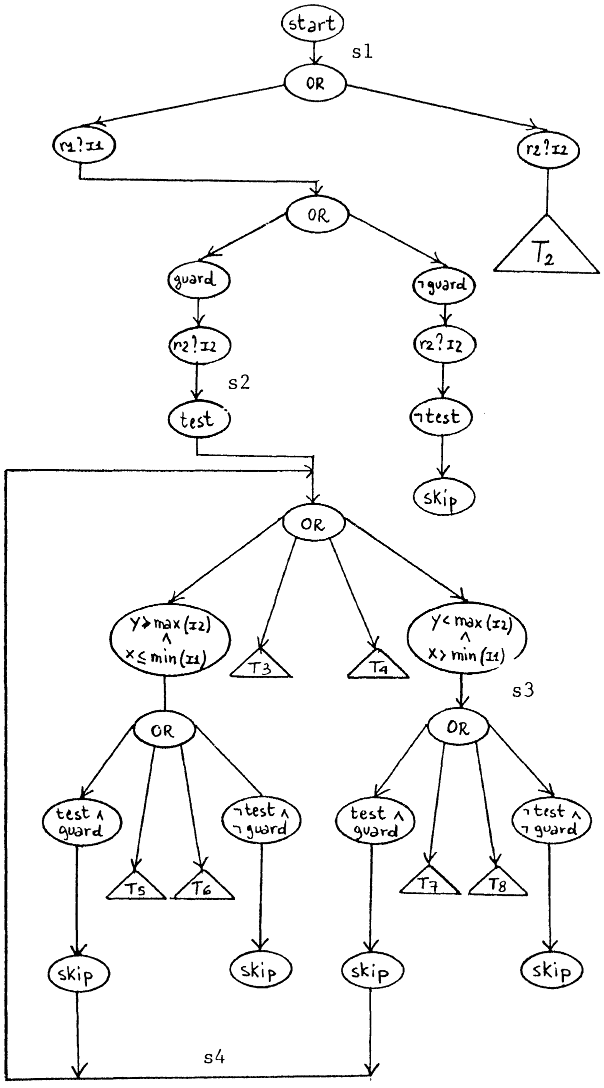


fig. 10



```
s1 = < guard . true
       test  . true >
s2 = < x . max(I2)
       y . min(I1) >
s3 = < I1 . ins(x; del(I1; min(I1)))
       I2 . ins(y; del(I2; max(I2))) >
s4 = <x . max(I2)
      y . min(I1) >
```

fig. 11

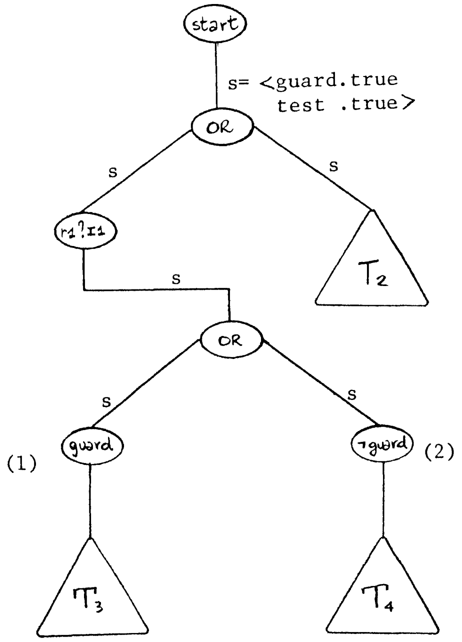


fig. 12

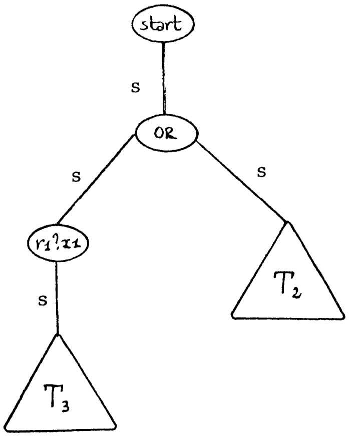
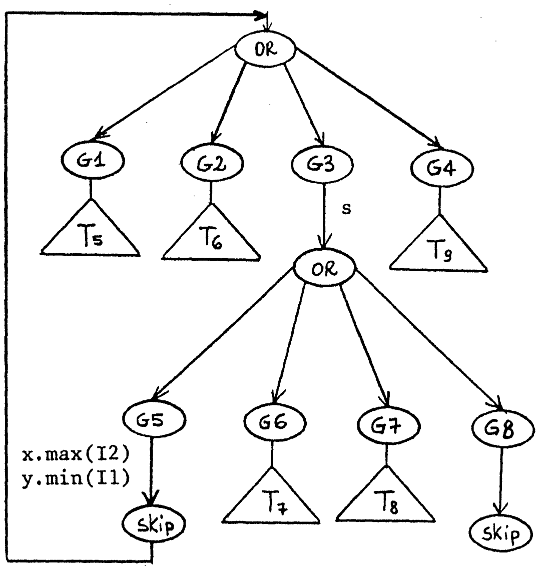


fig. 13



s=<guard.false
test .false>

fig. 14

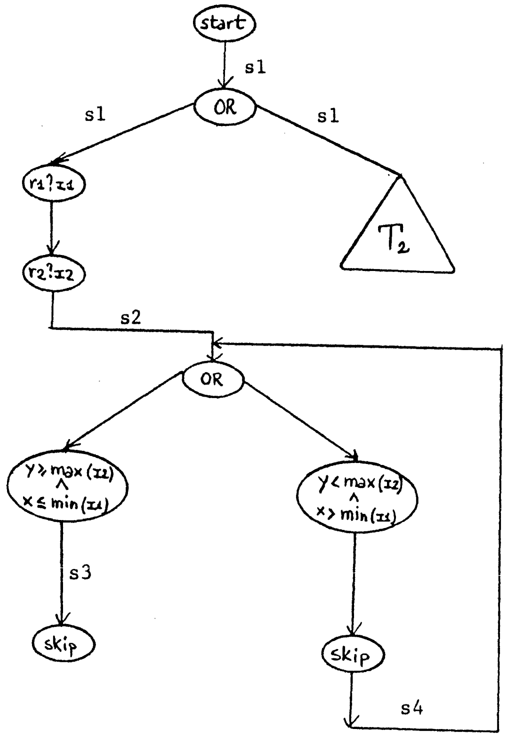


fig. 15

A STREAM DEFINITION FOR VON NEUMANN MULTIPROCESSORS

S. J. Allan and R. R. Oldehoeft
Computer Science Department
Colorado State University
Fort Collins, CO 80523

Abstract

Streams are data structures proposed for inclusion in several research programming languages, including VAL, to promote parallel execution and to implement input-output in applicative systems. To avoid paying a large overhead cost in near-term multiprocessor systems, we propose a special version of streams whose implementation efficiency potential does not impair their usefulness in typical applications. Special streams require no dynamic storage management during element production and consumption. They are part of a VAL implementation effort for the Denelcor HEP multiprocessor system.

Introduction

A stream is a data structure containing an ordered sequence of values. A stream differs from a vector because elements are accessible only in the given order. It differs from a list in that some leading elements may be missing (having been consumed) and some trailing elements can be absent (not having yet been produced). In its general definition a stream also differs from a queue because each consumer of a stream obtains a complete stream of all the produced values. The distinguished value "end of stream" appears after the last ordinary stream element. Streams are important for introducing general pipelined computations as well as input-output capabilities in parallel applicative or data flow languages [2, 3, 4, 7, 10].

Our interest in streams results from an ongoing project to implement a version of the VAL data flow language [1, 6] on the Denelcor HEP multiprocessor system [9]. The addition of streams to VAL provides for general pipelined computation yielding the potential for greatly increased parallelism and more effective performance on the HEP architecture, as well as forming the basis for defining input-output facilities.

In this report we will describe the architecture that forms the hardware base for the project, outline general streams as defined in the VAL proposals, and define special streams that show promise for efficient implementation and high speed performance in near-term multiprocessor systems.

The HEP Multiprocessor System

Denelcor, Inc. produces the Heterogeneous Element Processor (HEP), a shared-resource MIMD multiprocessor system. A HEP computer consists of one or more process execution modules (PEMs) connected by a packet-switched network to data memory modules and a high speed input-output cache. Data memory contains up to 1024K of 64-bit words. Each data word has, in addition to a value, an empty-full state that may serve to provide producer-consumer synchronization among processes sharing data memory. A PEM includes program memory (executable code), constant memory (read-only values), and register memory (general purpose fast storage, also with the empty-full property). All memories are allocatable via base and bounds registers to "tasks," groups of cooperating processes. A process is embodied as a process state word (PSW) in a queue containing up to 128 entries as part of PEM hardware. Each 100 nanoseconds a PEM examines the PSW for a process in an "active" task, examines its next instruction and tests the state of the source and destination registers. If source registers are full and the destination is empty, then one of several eight-stage pipelined function units initiates the operation and the PEM increments the instruction address in the PSW. If not, the PSW rejoins the tail of the queue for retry later. Note that in the presence of an ample quantity of work, this results in a "not very busy wait" solution to the process synchronization problem. A load or store instruction initiates interaction with data memory via the switch network. Because of the length of pipelined function units, a PEM must be executing instructions for at least eight processes to take advantage of the potential 10 MIPS instruction execution rate. Although PEM hardware currently supports 56 user processes, parallel processing speedup does not increase linearly up to this value because the number of independently executable function units in a PEM is much smaller. The remainder of the PSW queue slots hold PSWs for supervisor processes managing each user process, or are reserved for worst-case situations when hardware process creations are still in progress.

In data flow architectures, an operator unit executes to produce a result as soon as all operands are present, and the result is broadcast to all other operator units that use this value. Data dependencies are easy to find in VAL programs (lack of side effects, single assignment) and a translator can produce data flow graphs effectively. A translator could produce HEP processes that perform atomic arithmetic or logical operations sharing data or register memory cells for input and output but the overhead would

This research is supported by ARO Contract DAAG29-82-K-0108.

preclude satisfactory parallel execution. Instead, we intend to implement two forms of concurrency at a higher level. First, a VAL function invocation will initiate a HEP process that executes in parallel with the invoking function and with functions that it calls in turn. Second, some parallel loops will have their bodies executed simultaneously by several processes. This level of process granularity will, for sizable programs, use the entire capability for parallel operation in a single PEM machine. It will be easy to adjust this granularity if appropriate. Since a HEP PEM supports 56 processes maximally, some limitations and refinements to this intent are required. (Solving the process management problem is the subject of another report.) In the next section we will describe how general streams are defined, produced, and consumed.

General Streams

VAL functions in which parallel or sequential loop control structures yield individual stream elements produce values of type stream. The same control structures use up stream values element by element. In addition, they consume input file values and produce output file streams.

The operations associate with general streams in VAL are either implicit in the language semantics or explicitly represented in operators of the language. Implicit operations are the creation and deletion of streams, appending the end-of-stream value, and the production of copies of entire streams for new consumers. We will discuss the last operation later. The invocation of a producer function (process) instantiates a stream. VAL does not admit the possibility of more than one producer for a particular stream. Stream deletion results when all consumers have terminated and future existence of additional consumers is not possible. Producer termination causes the implicit appending of the end-of-stream value.

Explicit stream operations include copying the front stream element, "first," replicating the strea except for the first element, "rest," testing for end of stream, "empty," and adding an element to the end of a stream, "returns stream of."

Some simple examples of stream usage follow. The syntax is that for a revision of VAL described in [8].

(1) A function that produces the odd positive integers up to a given limit in a stream (uses the sequential "for" loop)--

```
type IS = stream[ integer ];
function INTS( LMT: integer returns IS )
 for I := 1
 while I <= LMT
 repeat
  I := old I + 2
  returns stream of I
 end for
end function
```

(2) A function that produces the negation of the contents of an integer stream--

```
function NEGATE( S: IS returns IS )
 for I in S
  returns stream of -I
 end for
end function
```

(3) A function that accepts a stream of integers and emits a stream whose elements come from the input stream, excepting those that are multiples of another integer parameter (includes the phrase "unless boolean expression" to exemplify conditional stream element production)--

```
function FILTER( S: IS; P: integer
                 returns IS )
 for I in S do
  returns stream of I
          unless mod(I,P) = 0
 end for
end function
```

(4) A function to merge two ordered streams into a single output stream (uses "first" and "rest" to consume streams)--

```
function MERGE( SA, SB: IS returns IS )
 for TA := SA; TB := SB
 while not (empty( TA ) and empty( TB ))
 repeat
  TA, TB :=
   if empty( old TA ) then
    old TA, rest( old TB )
   elseif empty( old TB ) then
    rest( old TA ), old TB
   elseif first( old TA ) <=
    first( old TB ) then
    rest( old TA ), old TB
   else old TA, rest( old TB )
   end if
  returns stream of
   if empty( TA ) then first( TB )
   elseif empty( TB ) then first( TA )
   elseif first( TA ) <=
    first( TB ) then first( TA)
   else first( TB )
   end if
 end for
end function
```

(5) A function that accepts an integer stream parameter and produces an output stream whose N-th element is the sum of the first N values in the input stream--

```
function COMPOUND( S: IS returns IS )
 for T := S; V := 0
 while not empty( T )
 repeat
  V := old V + first( old T );
  T := rest( old T )
 end for
end function
```

(6) A function that produces both the sum and the product of the elements of its integer stream argument (each "for" loop consumes the entire stream)--

```
function SUMPROD( S: IS
         returns integer, integer )
 for I in S
  returns value of plus I
 end for,
 for I in S
  returns value of times I
 end for
end function
```

(7) A function that accepts a stream of integers and an integer value K, and emits the original stream with each element multiplied by the K-th element of the input stream (the unwritten auxiliary function KTH consumes through the K-th value, then the "for" loop consumes the entire stream to produce the output stream)--

```
function MAGNIFY( S: IS; K: integer
                  returns IS )
 let V := KTH( S, K )
 in
  for I in S
   returns stream of V * I
  end for

 end let
end function
```

Observe that in each of these examples the function operates asynchronously as a "pump" to take in stream values or to push out computed values on an output stream. This is reminiscent of "systolic architectures" [5] and has the same potential for large-scale parallel execution. However, on the HEP system, software processes (instead of fixed hardware components with static interconnections) will implement stream producers and consumers.

The last two examples show the effect of the implicit "copy an entire stream" operation. In spite of the existence of a consumer using up stream elements, another consumer must be able to access all elements generated by the producer. This means that the decision about when a stream element is discardable, and when an entire stream can be deleted, is a difficult run-time issue. In near-term, von Neumann architectures, it seems that general stream implementations will require dynamic storage management for linked lists of stream values and the capacity to store arbitrary numbers of unconsumed stream elements. The cost of such management stands in opposition to our goal of promoting high speed parallel execution.

In the next section we restrict our definition to "special streams" whose implementation may allow high speed processing on the kinds of architectures available now and in the near future.

Special Streams

These definitions of operations on streams are meant to promote a more efficient implementation than is possible for general streams in near-term von Neumann multiprocessor systems. In particular we propose changes in the semantics of general streams to avoid dynamic storage management during the lifetime of a stream. Programs use the explicit operations "first," "rest," "empty," and "returns stream of" as for general streams. The implicit (automatic in language semantics) operations for stream creation, appending the end-of-stream value, and deletion are unchanged. Implicitly copying an entire stream is no longer possible: the use of the same stream name by multiple consumer loops in a function results in each consumer loop obtaining a disjoint subsequence of the stream of values. (There can be programs in which this is a desired effect.) If more than one consumer must share the entire stream, then an explicit preceding assignment to another stream value is necessary. We define passing a stream as a parameter to another function as another form of the explicit stream copy operation.

The stream copying via assignment or parameter passing may be actual: a new stream structure receives a copy of the current stream contents. Then the stream producer emits a new value to the tail of each copy of the stream it produces (or blocks if at least one of the stream data structures is full), and appends the end-of-stream value to each when it terminates. Each consumer works via "first," "rest," and "empty" operations on its own private data structure. When a consumer terminates, deletion of the associated stream copy occurs. On the other hand, the stream copying can be realized by associating a reference count with each stream buffer (holding a single stream element). The producer adds a reference count (initialized to the current number of consumers of the stream) to each emitted stream value. Each consumer works on the same data structure, and so must maintain its own pointer to the first value. The "rest" operation decrements the reference count of the first value, and moves the first pointer. The producer may fill a buffer when its reference count is zero, and must block if there is no empty (zero reference count) buffer. A stream copying operation via assignment or parameter passing increments the current number of consumers and the reference counts of each nonempty stream buffer. Consumer termination decrements the current number of consumers and the reference counts on unconsumed elements. A stream is deletable when the current number of consumers reduces to zero. No advantage accrues to the first approach, and since the second requires less copying of information, we will use it.

These special streams of course behave differently than general streams, and some programs using streams differ from those using previously defined versions. We believe that special strea~s are as useful as general streams in most applications. The last two program examples above using the implicit stream copying operation cannot work in that form. The former can be rewritten with an auxiliary assignment:

```
function SUMPROD( S: IS
         returns integer, integer )
 let T := S
 in
  for I in S
   returns value of plus I
  end for,
  for I in T
   returns value of times I
  end for
 end let
end function
```

The latter presents a difficulty for special streams. No matter how we implement special streams to avoid dynamic storage management (copying for each consumer or reference counts on values), the "first" pointers for all the consumers must be within a contiguous substream. The size of this substream cannot exceed the number of buffers in the stream data structure because the producer cannot get more than that number of elements ahead of the slowest consumer. No performance problems should arise in the expected kinds of of "systolic" stream applications when the executing program has ample work to keep the components of a multiprocessor system active. In MAGNIFY, if K is greater than the number of buffers available, the subsequent "for" loop will not be able to consume some prefix stream values, and the function value will differ from that for general streams. Problems such as this may arise in real programs, but such occurrences may suggest that randomly accessible structures (arrays) are more appropriate.

Summary

Streams have been recognized by several researchers to be useful for promoting parallel computation. There is, however, no experience in production systems with general streams. Our purpose is to implement a version of the parallel language system VAL including streams on a currently available multiprocessor system. Special streams as described here seem to be a valuable compromise preserving the potential for parallel execution while enhancing the possibility of efficient implementation.

References

[1] William B. Ackerman and Jack B. Dennis, VAL--A Value-Oriented Algorithmic Language: Preliminary Reference Manual (1978), Laboratory for Computer Science, MIT.

[2] William B. Ackerman and Jack B. Dennis, VimVal: The Base Language for a Value-Oriented Computer System: Changes to the VAL Preliminary Reference Manual (1982), Laboratory for Computer Science, MIT.

[3] Arvind, Kim P. Gostelow and Wil Plouffe, An Asynchronous Programming Language and Computing Machine (1978), Department of Information and Computer Science TR114a, University of California at Irvine.

[4] Gilles Kahn and David B. MacQueen, "Coroutines and Networks of Parallel Processes," Proc. IFIP 1977, (B. Gilchrist, ed.), North-Holland, pp. 993-998.

[5] H. T. Kung, "Why Systolic Architectures?" IEEE Computer, 15, No. 1 (1982), pp. 37-46.

[6] James R. McGraw, "The VAL Language: Description and Analysis," ACM Trans. on Programming Languages and Systems, 4, No. 1 (1982), pp. 44-82.

[7] James R. McGraw and Steven K. Skedzielewski, "Streams and Iteration in VAL," Int. Conf. on Distributed Processing (1982).

[8] James R. McGraw, et. al., SAL: A Single Assignment Language, Language Reference Manual Version 1.0 (1983).

[9] Burton J. Smith, "A Pipelined, Shared-Resource MIMD Computer," Proc. Int. Conf. on Parallel Processing, (1978) pp. 6-8.

[10] K. Weng, Stream-Oriented Computations in Recursive Data Flow Schemes, M.S. thesis, Laboratory for Computer Science, MIT (1975).

A DATABASE MACHINE
FOR VERY LARGE RELATIONAL DATABASES

G. Z. Qadah and K. B. Irani
Computing Research Laboratory
The University of Michigan
Ann Arbor, MI. 48109

ABSTRACT -- In this paper, we present an architectural design for a Back-End Database Machine (DBM) suitable for supporting concurrent, on-line, very large relational database systems. This machine is called Michigan Relational Database Machine (MIRDM). In designing such a machine, a structured approach has been followed. First, the DBMs proposed so far have been reviewed using a novel classification scheme. Next, this review, the very large relational database system requirements and the restrictions imposed by the current and near future state of technology has been used to formulate a set of design guidelines. Consequently, an architecture for a cost-effective DBM that meets the latter set of guidelines has been synthesized.

1. Introduction

The collection of data in the form of an integrated database is a sound approach to data management. The conventional implementation of the database system-that is, the augmentation of a large general purpose conventional von Neumann computer with a large complex software system, the Database Management System (DBMS)- suffers from several disadvantages. These disadvantages are:

(1) Low reliability due to the large complex software system,

(2) Poor performance due to the fact that the underlying hardware is a general purpose von Neumann processor with insufficient processing power, little parallelism, and

(3) Inability to meet the demands for increased processing power and throughput to fulfill the current and anticipated large increases in database size and usage.

The limitations of the conventional database systems, the continuous advancements in memory-processor technology, and the continuous reduction in its fabrication cost have inspired a new approach to the database system implementation. This approach replaces the general purpose von Neumann processor with a dedicated machine, the Database Machine (DBM), tailored to the data processing environment. Mostly it utilizes parallel processing to support some or all the functions of the DBMS. This approach improves the system's reliability through software complexity reduction and improves the system's performance through specialization, increased parallelism and increased processing power.

Most of the DBMs proposed so far have been organized as "back-end" machines to one or more general purpose computer(s), called the Host(s). While the host is responsible for interfacing the users to the DBM, the DBM itself is responsible for the database access and control. The "back-end" design concept for the DBM was first introduced in [1].

The general objective of this work is the design of a back-end DBM suitable for supporting the concurrent on-line, very large relational database systems. This machine will be called the Michigan Relational Database Machine (MIRDM). The design of MIRDM is done in two steps. In the first step, the DBMs proposed so far are reviewed. This review is based on a novel scheme for classifying these machines. The new scheme helps us understand the previous organizations of the DBMs as well as their design trade offs. It also provides us with a systematic way to qualitatively analyze and compare such organizations.

In the second step, the above analysis coupled with the requirements of the very large relational database systems as well as the current and near future state of the hardware technology is used to arrive at a set of guidelines along which our DBM must be designed. Consequently, an architecture for a cost-effective DBM that meets the latter set of guidelines is synthesized.

This paper is divided into five sections. Section 2 qualitatively reviews, analyzes and compares the various previously proposed designs for the relational DBMs. Section 3 formulates a set of guidelines based on the qualitative analysis of section 2, the current and near future state of technology and the very large relational database systems requirements. It also outlines a new DBM, called MIRDM, synthesized along the latter guidelines, and it presents a brief introduction to the algorithms that implement the primitives of this machine. Section 4 provides qualitative comparisons between MIRDM and some other DBMs already proposed. Finally, Section 5 gives some concluding remarks.

2. Review of the Previously Proposed DBMs

During the past decade, a large number of DBMs have been proposed. Some of them have also been implemented. Others have been commercialized. All of these machines have been designed to partially or totally support the relational database or to support the relational database together with the other database types, namely, the network and the hierarchical. In the following, a novel scheme for classifying the set of the DBMs proposed so far will be presented. This scheme will next be used to qualitatively evaluate and compare the respective DBMs.

2.1. A Classification Scheme for the Previously Proposed DBMs

The new scheme views the DBMs as points in a three dimensional space, the DBM space. The coordinates of this space are the indexing level, the query processing place and the processor-memory Organization.

The most fundamental and important operations the DBMs were designed to support are the selection (from a permanent relation) and the modification operations. In the early designs of the DBMs these operations

0190-3918/83/0000/0307$01.00 © 1983 IEEE

qualification expressions), however, they perform poorly in executing more complex database operations that require many disk revolutions (the ϑ-Join and the projection operations, for example). This was evident in the performance evaluation of RAP[5].

Recall that a query can be thought of as a tree whose nodes represent a set of database operations [3]. The leaves of the tree reference only permanent relations of the database. In a real database environment, the leaf nodes are mostly of the selection and update types. A hybrid DBM processes the leaf selection operations and, in some machines, the update operations on the disk. The resulting relations(referred to as the temporary relations) are then moved to a fast processor-memory complex where the rest of the query operations (if any) are executed. In most cases, execution of the leaf selection and update operations on the disk largely reduces the volume of data needed to be moved to the fast processor-memory complex.

The above discussion indicates that the performance of the DBMs of the Hybrid-DB design approach is superior to that of both the On-Disk-DB and the Off-Disk-DB DBMs. However, the Hybrid-DB design approach compounds the cost problem because it combines two very expensive technologies, namely, the logic-per-track disks and the associative memories.

In general, the number of tuples which are selected/modified as a result of executing a typical selection/modification operation is a small fraction of the number of tuples of the database. In the light of the current processor-memory technology and the vast amount of data in the very large databases, it is clear that the need to scan the whole database, at least once, for carrying out these operations is very cost-ineffective. The design approach which provides the DBM with a mechanism that eliminates this need is highly desirable.

In the context of the DBMs, an index table defined for the permanent database has been used as a mechanism to reduce the amount of data to be processed for a given selection or modification operation. For every relation of the database the index table defined for a DBM of the DBMs-relation category store the set of addresses of the minimum access units(MACUs) which store the corresponding relation. Although the size of the index table (relative to that of the database) is a function of the number of relations in the database and the size of the MACU, nevertheless, it is very small. Its maintenance, storage cost and access time are negligible relative to that of the database. To execute a selection/modification operation on an On-Disk-Relation/Hybrid-Relation DBM, only the data units which store the relation referenced by the operation are searched/modified. For those machines which use a logic-per-track disk as a storage media, only the tracks which correspond to these units are processed. On the other hand, to execute these operations on an off-disk-relation DBM, only the data units which store the relation referenced by the operation need be moved to the processor-memory complex for processing.

The above argument suggests that the DBMs which support only a relation level index table perform differently for different types of databases. In the case of databases dominated by relations of small size, indexing on the relation level substantially improves the DBM cost-effectiveness. For the on-disk/hybrid organized DBMs, the logic-per-track disk can be replaced by a less expensive mass storage media. For the Off-Disk DBMs, a less expensive I/O channel can be utilized. On the other hand, in the case of the databases dominated by relations of relatively large size, indexing on the relation level does not improve the DBM cost-effectiveness. It is very clear that such DBMs suffer from the same problems as the DBMs-DB type.

The index tables supported by the DBMs which belong to the DBMs-Page category are defined for the set of the most frequently referenced attributes of the database. For every value of each of these attributes, the index table stores the set of addresses of all the pages which contain tuples having that value. Although the size of the index table (relative to that of the database) is a function of the size of the page itself, nevertheless it, as well as its storage maintenance cost, is substantial. A clustering mechanism has to be used [6] in conjunction with the page index table. This mechanism is used to store the set of tuples which are frequently referenced together in as few pages as possible. In general, the use of the page indexing coupled with an efficient clustering mechanism improves the performance of the selection and possibly the modification operations. On the other hand, the index search introduces some overhead for executing these operations as well as the insertion and deletion operations.

In the case of the very large database systems, the use of small size pages results in a relatively large page index table which requires a huge amount of storage, large access time and high maintenance and update cost. The improvement in the execution time for the selection and modification operations does not justify this overhead and the additional storage cost. On the other hand, the use of large size pages, in conjunction with a DBM, results in a relatively small page index (~1% of the total database size [6]). Thus its storage cost, access time and maintenance cost are substantially lower. Moreover, the performance of the selection and the modification are enhanced especially if a page is processed by a number of processors in parallel. Using the page index,large size pages and efficient clustering mechanism have allowed designers to replace the expensive logic-per-track disk with the relatively cheap moving-head-disk (or a slightly modified version of it) as a unit for the mass storage.

In the scheme, presented earlier, the previously proposed DBMs have been organized as SISD, SIMD and MIMD machines. In general, the execution of the database operations, on a DBM of the first/second group, are done serially. That is, one operation (possibly two for the Hybrid DBMs) is the maximum number of operations that a DBM of these types can execute at any given time. While a database operation is executed by one processor on the SISD DBMs, it is executed by more than one processor on the SIMD DBMs. The MIMD DBMs, on the other hand, execute one or more database operations simultaneously. The operation itself also gets executed by more than one processor.

Because of the relatively low cost of the processor and memory devices, the use of parallel processing for very large databases enhances the effectiveness of the DBMs. Although the SIMD organization of the DBMs reduces the execution time of a database operation, it does not offer a real solution to the concurrent user problem. The MIMD organization is more effective in supporting databases where fast concurrent accesses to the databases is a basic requirement. This is due to the fact that the MIMD organization has the ability not only to execute a database operation in parallel but also to execute more than one operation (from same or different queries) simultaneously and in parallel.

One important drawback in the MIMD organization is the associated overhead, namely, the large amount of processing needed for controlling the simultaneous execution of different operations and the management of were carried out using an entirely associative approach. That is, the whole database (a set of permanent relations) was scanned and the data items which satisfied the selection/modification criteria were retrieved/modified. Soon, researchers in the DBMs field came to realize that such an approach is not a cost-effective one since it requires that the whole database be scanned, at least once, for every selection or modification operation regardless of the size of the operation's response set.

To achieve a more cost-effective DBM's design, a new approach for performing the selection and the modification operations has been followed. It is called the quasi associative approach. In this approach only a relatively small portion of the database need to be processed for every operation (rather than all). To support such an approach, the database is divided into a set of data units. In order to perform the selection operation, for example, the machine first maps the corresponding selection qualification expression to the set of data units which contain the data which satisfy the qualification expression. Then each data unit of the latter set is searched and the data items which satisfy the selection qualification expression are extracted.

In most of the DBMs proposed so far the structure used to map the qualification expression of the selection or the modification operation, to the corresponding data units of the permanent database, is the index tables [2]. These tables are defined for the database and need to be stored and maintained. The data unit, the smallest addressable unit of data, can be logical (i.e., the database, no indexing, or a relation) or physical (i.e., a set of tracks, a track or part of a track of a moving-/fixed-head-disk). The physical data unit is called a page.

The first coordinate in the proposed scheme is the indexing level defined for the permanent database and supported by the particular DBM. Along this coordinate, the DBMs can be grouped into three categories, namely, the DBMs with database indexing level (DBMs-DB), DBMs with relation indexing level (DBMs-Relation) and DBMs with page indexing level (DBMs-Page). The first category includes all the DBMs which support only the entirely associative approach. The DBMs of the second category support the quasi-associative approach. The index tables of the machines in this category are defined with the permanent relations as the minimum addressable units.* The DBMs of the third category also support the quasi associative search approach. However, in addition to supporting the relation level index tables, they also support index tables defined for pages (containing tuples from the permanent relations) as the minimum addressable units.

The second coordinate in the proposed scheme is the query processing place. Along this coordinate, the DBMs can be grouped into three categories, namely, the Off-Disk,** the On-Disk and the Hybrid categories. The DBMs of the first category process the database query off the disk where the database is stored. In doing that the DBMs of this category need to move the data which are relevant to the query from the disk to a separate processor-memory complex where the query processing takes place.

The DBMs of the second category execute the database query on the disk. The machines of this category need not move data from the disk to a different memory for processing. The disk is provided with logic units and the query processing is done directly on the disk where the database is stored.

The DBMs of the third category execute part of the query the selection operations and, in some machines, the update operations on the disk, and move the resulting data to a separate processor-memory complex where the rest of the query execution (if any) takes place.

The third coordinate in the proposed scheme is the processor-memory organization. This coordinate characterizes the hardware of the DBMs. For the On-Disk/Off-Disk machines, this coordinate characterizes the way the processor-disk/processor-memory complex executes the database operations. For the hybrid machines, this coordinate characterizes the way both the processor-disk and the processor-memory execute the database operations. Along the third coordinate, the DBMs can be grouped into three categories: 1) the single instruction stream-single data stream (SISD), 2) the single instruction stream-multiple data stream (SIMD), and 3) the multiple instruction stream-multiple data stream (MIMD) categories.

2.2. A Critical Look at the Previously Proposed DBMs.

Figures 2.1 through 2.3 show most of the previously proposed DBMs grouped according to the classification scheme presented earlier. For more information about these machines one can refer to [3] and to the references presented in the latter figures.

Historically speaking, the first DBMs to be proposed were organized as Off-Disk machines and were provided with only the associative access to the database (Off-Disk-DB DBMs). In general, these machines suffer from many drawbacks, in particular, their ineffectiveness in handling the very large database systems. A DBM of this type has to move all of the database from the slow disks, where it is stored, to a fast (associative) memory where the execution of the selection or the update operations take place. In a large database system environment, the I/O channels easily become a system bottleneck as a result of the mass data movement.

The On-Disk-DB design eliminates the data movement problem. This design approach avoids that problem by processing the database operations on the mass storage device where the database resides. The most important drawback of this approach is the fact that it stores the database on a set of logic-per-track disks. Using this disk as a mass storage device is very costly, in fact orders of magnitude more expensive than the moving-head-disk. The high cost of the logic-per-track disk is attributed mainly to the high cost of its two basic components, namely, the storage component, the fixed head-per-track disk and the processing units attached to each head of the head-per-track disk. The anticipated trends in the mass storage technology [4] show that the logic-per-track disk or its electronic counter part will not challenge the speed/cost level of the moving-head-disk for at least the near future.

Another important problem in the DBMs which adopt the On-Disk-DB design approach is the fact that they perform relatively well in executing the simple database operations which require few disk revolutions (the selection and update operations with simple

*To facilitate the parallel processing as well as the data movement, some DBMs of the first/second category store the corresponding minimum addressable unit of data (DB/relation) on a set of physical units, the minimum access units (MACUs). Each could be moved separately. However when a data item needs to be retrieved, all the MACUs containing the DB/relation are processed. In the DBMs of the third category, the page (the minimum addressable unit) is contained in one MACU.

**The disk here implies a moving-head-disk, a fixed-head-disk or an electronic disk such as the magnetic bubble memory (MBM) or the charge coupled devices memory (CCD). The disk(s) stores the database.

the various system components. In most MIMD DBMs, such overhead puts an upper limit on the number of queries that can be executed and on the resources that can be simultaneously active in the system. Therefore, controlling and minimizing such overhead must be a basic objective for the MIMD DBM designer.

3. The Architecture of the New System

The most important characteristics of the contemporary and anticipated very large scale relational database systems are the vast amount of data in such systems and the large number of users requiring simultaneous access to this data. These basic characteristics impose two important requirements on any design for a relational DBM, namely:

1. Availability of large capacity store,
2. Ability to handle the on-line concurrent access to the database with adequate response time and throughput.

Based on the above requirements as well as the state of the current and anticipated mass storage technology and in the light of our study for the previously proposed DBMs, a set of guidelines along which a DBM should be designed, have been formulated. These guidelines are:

(1) The mass store is to consist of moving-head-disks. This disk type has been selected for its ability to provide a vast amount of on-line storage at a relatively low cost and moderate performance. Currently the magnetic fixed head-per-track disk is considered obsolete as a mass storage device. The electronic disk (the MBM and the CCD memory devices) technology is at least one order of magnitude more expensive than that of the moving-head-disk. A look at the future directions in the mass storage technology shows that the electronic disk will not challenge the speed/cost level of the moving-head-disk for at least the near future [4].

(2) Support the page level indexing. This type of indexing greatly improves the execution time of the selection and the modification operations. On the other hand, it introduces some overhead in executing these operations, in the form of index table access delay and maintenance, and increases the execution time of the other update operations. To minimize the drop in performance due to this overhead, the page must be selected to have a large size (multiple tracks of the moving-head-disk) and be processed in parallel, by a number of processors. Also, the access to the page level index table should be supported at the hardware level.

(3) Organize the DBM as an off-disk type. Although this organization introduces some increases in the execution time of the database operations (due to the movement of data to the processor-memory complex), it nevertheless avoids providing the moving-head-disk with a high speed, specially designed logic units capable of processing data "on the fly", thus keeping the mass storage cost at its minimum. The amount of data to be moved can be reduced by taking advantage of the local and sequential references in the database. The processor-memory complex should be designed to effectively support not only the relational algebra operations, such as the selection, the projection, and the ϑ-Join, but also the primitives that manipulate the page level index.

(4) Organize the DBM as an MIMD type. This is very important for providing the machine with the capability of handling concurrent access to the database. The proposed design must be able to handle, at the hardware level, the excessive overhead associated with the MIMD organization.

A DBM which follows the above guidelines has been designed. This machine is called Michigan Relational Database Machine (MIRDM). However, before presenting its organization, the way the data is organized in MIRDM will be next outlined and discussed.

3.1. The Data Organization

In general MIRDM stores two types of data, namely:

1. The Database

The database is organized as a collection of time varying relations. The database is divided into a set of large data units. Each, called a page (or the minimum addressable unit** (MAU)), represents the smallest addressable unit of data. The only tuples which are allowed in the same MAU are those that belong to the same relation.

2. The Database Directory

The database directory contains the information needed to map a "data name" to the set of MAU addresses which store the named data. In the proposed machine, data is named at two levels, namely, the relation level (relation-name) and the tuple level (tuple-name: <relation name, attribute name, value>). The tuple-name may not be unique.

The database directory consists of two indices, namely the relation index and the MAU index. The relation index maps the "relation-name" to a set of MAU addresses. These MAUs contain all the tuples of the relation whose name is "relation-name". The relation index contains entries for all the relations in the database.

The MAU index maps a "tuple-name" to a set of MAU addresses. Each of these MAUs contains at least one tuple which has the "tuple-name" as its name. The MAU index is organized as a three level index. The first level is the MAU master index, the second level is the attribute index and the third level is the index-term index. The index-term index is a collection of index terms, each of which is an ordered quadruple of the form:

< relation name, attribute name, value, MAUA >

where MAUA is an MAU address which contains at least one tuple with the name "< relation name, attribute name, value >".

In general, the index terms are only defined for those relations and attributes which are frequently referenced by users. The index terms in the proposed system are grouped and stored in units equal in size to an MAU, called the index MAU (IMAU). Although the IMAU can contain index terms defined for different attributes of different relations, a clustering mechanism is used to cluster, into the same IMAU, those index terms which are defined for the attributes of the same relation. This improves the storage cost and the processing efficiency of the index terms.

The MAU master index and the attribute index have been introduced in order to reduce the number of IMAUs which need to be processed for a selection/modification operation. The MAU master index maps a "relation-name" to its attributes. The attribute index maps an attribute name to a set of IMAU addresses. The IMAUs contain the set of index terms which are defined for the corresponding attribute.

Before leaving this section, an important operator, the *index-select*, which performs the retrieval operation

**As is seen later, the MAU occupies one minimum access unit (MACU).

on the index-term Index, must be mentioned. The index-select operator is executed in conjunction with the relational algebra operation selection and the modification operation. It retrieves the addresses of those MAUs which contain at least one tuple that satisfies the corresponding qualification expression (QE).

3.2. The Michigan Relational Database Machine Organization

The proposed Michigan Relational Database Machine (MIRDM), shown in figure 3.1, consists of four components, namely, the master back-end controller (MBC), the processing cluster subsystem (PCS), the mass storage subsystem (MSS) and the interconnection network subsystem (INS). In the following the organizations of these components will be outlined.

3.2.1. The Master Back-End Controller

In cooperation with the front-end computer system(s), the master back-end controller (MBC) interfaces the users to the database system, translates the user queries to the primitives of MIRDM, schedules and monitors the query execution, manages and controls the different components of MIRDM, stores and maintains the system's dictionary, stores, maintains, and manipulates part of the database directory (the relation, the MAU master and the attributes indices) and provides for security checking, integrity maintenance and users' views.

The implementation of the MBC is strongly dependent on the way the above stated functions are partitioned between the front-end computer system and the MBC. Based on this partition, the MBC can be implemented using a powerful mini/micro computer.

3.2.2. The Mass Storage Subsystem

The mass storage subsystem (MSS), shown in figure 3.1, is the repository of the database and its index-term index. The MSS is organized as a two-level memory, namely, the mass memory (MM) and the parallel buffer (PB). While the MM helps MIRDM to meet the large capacity storage requirement, the PB helps it to take advantage of the local and sequential references to the database. In the following, the architecture of both levels will be outlined.

The Mass Memory

The mass memory is organized as a set of moving-head-disks, controlled and managed by the mass storage controller (MSC). Each disk is provided with the capability of reading/writing from/to more than one track in parallel. Tracks which can be read/written in parallel, from one disk, form what is called the minimum access unit (MACU). The tuples within this Unit are laid out on a moving-head-disk's track in a "bit serial-word serial" fashion. The MACU is the smallest accessible unit of data as well as the unit of data transferable between the MM, the PB and the PCS. The MACU in MIRDM stores only one MAU. We expect the MACU to have the size of a moving-head-disk cylinder.

In addition to the database relations, the MM stores another type of data, namely, the index terms In general, the index terms which are defined on attributes of different relations can reside in the same IMAU. In order to improve their retrieval cost, the index terms are clustered together according to their relation and attribute names.

An IMAU is stored in one MACU. Every track, within the latter unit, contains a set of blocks of suitable sizes (~ 2 - 4 K bytes). Each block contains index terms defined for the same relation and attribute. For storage as well as processing efficiency, the <relation name, attribute name> common to all the index terms of the block is stored once, at the beginning of the block. The rest of the block stores only the <value, MAUA> part of the corresponding index terms.

The Parallel Buffer

The parallel buffer (PB), shown in figure 3.1, is organized as a set of blocks, each of size equal to that of an MACU. A block is further partitioned into a set of subblocks. Each subblock can buffer one track of a moving-head-disk. The PB is managed by the mass memory controller.

The PB implementation can take advantage of the technologies of both the magnetic bubble memory and the charge coupled device memory. Both technologies currently have off-the-shelf memory chips which can buffer an entire disk track.

3.2.3. The Processing Clusters Subsystem

The processing clusters subsystem (PCS) is organized as a multiple single instruction stream-multiple data stream (MSIMD) system. The PCS (figure 3.1) consists of a set of processing clusters (PCs) which share a common buffer, the parallel buffer. A PC, shown in figure 3.2, has a single instruction steam-multiple data stream (SIMD) organization. A PC consists of a set of triplets, each of the form:

<I/O controller (IOC), triplet processor (TP),
local memory unit (LMU)>

The set of triplets within a PC is controlled and managed by the cluster master processor (CMP). The latter accesses its triplets through a broadcast bus, the master bus (MBUS). The MBUS permits the CMP to write the same data to all the LMUs of its cluster triplets simultaneously. On the other hand, the MBUS permits the CMP to sequentially read data from any one of its triplets' LMUs.

Within a PC, the data is moved between its triplets via a bus, the triplets bus (TBUS), controlled by a high speed DMA controller, the data mover (DM). Under instructions from the CMP, the DM moves data items between the LMUs of the cluster's triplets. The TBUS is provided with both point-to-point as well as broadcast capabilities.

In general, the i^{th} LMU, in a PC, is accessible directly by the CMP, through the MBUS, by the DM, through the TBUS, and by both the i^{th} TP and IOC. We expect the LMU of a triplet to have a relatively large capacity (multiple of the size of a moving-head-disk track) and to be implemented using RAM technology. It is suggested that the IOC of a triplet be implemented as a high speed DMA controller and that a TP be an off-the-shelf microprocessor.

3.2.4. The Interconnection Network Subsystem

The interconnection network subsystem (INS) is designed to fulfill two basic requirements, namely, the ability to allow any two PCs or two moving-head-disks to read/write from/to any two blocks of the PB simultaneously and enable two or more PCs to read from the same PB block. The latter requirement is needed to provide the proposed machine with the capability of handling simultaneous processing of the database operations.

The INS, shown in figure 3.1, is a modified version of an interconnection network proposed by Dewitt [7]. The network consists of a set of buses, each associated with one subblock and having one bit width. The subblock continuously broadcasts its contents over the corresponding bus. Only one triplet in each PC as well as one head of each moving-head-disk, in the MM, is connected to the same bus(subblock). Thus the complexity of the logic at the triplet, disk head or subblock interface is (1/ number of triplets in a PC) that of the one proposed by Dewitt. Whenever a PC(s)/(MM disk) needs to read a given PB block, its IOCs(disk heads) need only switch themselves to the appropriate set of data buses. If the parallel buffer block contains a data MAU, then the IOCs(disk heads) can begin to read at a tuple boundary. However, for an index MAU, the IOCs(disk heads) proceed to read it at an index block boundary. Whenever a PC(MM disk) needs to write to a given parallel buffer block, its IOCs(disk heads) need only switch themselves to the appropriate set of data buses. The writing then follows immediately. Notice that the MSC (figure 3.1) is responsible for preventing any two PCs, or disks, or a PC and a disk from writing to the same parallel buffer block.

3.3. Algorithms for the Relational Algebra and Index Retrieval Operators

The newly proposed MIRDM supports the Parallel processing of the most important relational database operators- namely, select, project and ϑ-join- as well as the index retrieval operator index-select. In most retrieval queries, the operator project follows the select operator. For this as well as for reasons of performance improvement, the newly proposed machine combines the two operators, select and project, to form a new operator (the select-project Operator). The latter operator is processed as a non-decomposable operator.

In MIRDM, one or more PCs are used to execute the select-project and the ϑ-join operators. On the other hand, only one PC is used to execute the operator index-select. In general, the number of PCs assigned to execute a select-project or a ϑ-join operator is an MBC decision. This decision is based on many factors, such as the operator type, the size(s) of input relation(s), the expected size of the output relation, the number of available PCs and the priority class to which the operator's query belongs. By accessing the appropriate directory, possibly with the help of a PC, the MBC determines the set of all MAUs relevant to a given operator.

The flexibility and generality of MIRDM architecture permits the implementation of a powerful set of algorithms for the above operators. These algorithms are presented in [3].

4. Discussion

In the previous section, we have presented an architecture for a back-end database machine which is capable of supporting the on-line, concurrent, very large relational database systems. Our approach rests on a set of fundamental design principles. This set includes two principles followed by previously designed DBMs, namely, the MIMD organization of DIRECT [7] and the "page level indexing" of DBC [6]. While the MIMD organization is very valuable in handling the concurrent user environment, the "page level indexing" is equally important in supporting the very large database environment as well as in the reduction of the system data volume needed to be moved for the selection/modification operation. In contrast to the DBC, MIRDM stores the database structure information on the relatively inexpensive mass storage devices (on moving-head-disks rather than the much more expensive electronic ones), manipulates this structure information using the same units (the processing clusters) which manipulate the database (thus distributing the systems workload uniformly among its various components) and provides the machine with the MIMD capability as well as the additional parallelism and processing power which are essential for meeting the requirements of the contemporary and anticipated database systems. Finally, our proposed architecture removes the restriction imposed by the DBC on processing the ϑ-join operation [8], namely, that both the source and target relations of the ϑ-join operation must fit in the local memory of a processor-memory complex, designed specifically to carry out the ϑ-join operation. The newly proposed machine has the capability of joining relations of any sizes.

In contrast to DIRECT, MIRDM groups the processing elements into a set of clusters, each cluster with its own controlling processor. The data transfer, in the new machine, is done in relatively large units (the MACUs). This organization not only improves the management and control of the processing elements and distributes the overhead caused by the processing of requests for the movement of the data units (this overhead is causing a system bottleneck in DIRECT), but it also reduces the complexity of the interconnection network. Providing the newly proposed machine with "page level index" as well as supporting its primitive operations, at the hardware level, helps to improve the performance of the selection operation. The latter operation is performed poorly, on DIRECT, relative to other DBMs [9]. In the case of very large databases, our machine permits the implementation of a set of algorithms for the equi-join operation, more powerful than the one implemented in DIRECT. To demonstrate that fact, we have compared [10] the performance of the equi-join algorithms recommended for the newly proposed machine with that adopted by DIRECT. These algorithms have been selected [3] from a large set of algorithms, based primarily on the behavior of one performance measure, the equi-join "total execution time". This comparison shows that in a typical very large databased environment MIRDM, is (1.5 - 5) times faster in executing the equi-join operation than DIRECT.

5. Conclusion

In this paper, we have proposed a back-end database machine suitable for supporting the concurrent, on-line, very large relational database systems. The new machine is designed to satisfy a set of guidelines. These guidelines have been formulated based on reviewing the previously proposed DBMs, the current and the near future state of technology and the requirements of the concurrent very large relational database systems. The previously proposed DBMs were reviewed using a novel scheme for their classification.

At the present, our research activities are centered around prototyping the new machine. Currently a processing cluster with 8 triplets is being implemented at the Computing Research Laboratory of the University of Michigan, Ann Arbor.

References

[1] R. H. Canaday, et al., "A Backend Computer for Database Mangement," CACM, Oct. 1974, Vol. 17, No. 10.

[2] J. Martin, "Computer Database Organization," Prentice Hall, 1977.

[3] G. Z. Qadah, "A Relational Database Machine: Analysis and Design," Ph.D. Thesis, 1983. The Electrical and Computer Engineering Department, the University of Michigan, Ann Arbor.

[4] D. K. Hsiao, "Data Base Computers." Advances in Computers, Vol. 19, June 1981.

[5] A. Ozkarahan, S. A. Schuster and K. C. Sevcik, "Performance Evaluation of a Relational Associative Processor," ACM Trans. on Database Systems, Vol. 2, No. 2, June 1977, pp. 175-195.

[6] J. Banerjee, D. Hsiao and K. Kannan, "DBC-A Database Computer for Very Large Databases," IEEE Transaction on Computers, Vol. C-28, No. 6 June 1979, pp. 414-429.

[7] D. J. Dewitt, "DIRECT-A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Trans. on Computers, Vol. C-28, No. 6, June 1979, pp.395-408.

[8] M. J. Menon and D. K. Hsiao, "Design and Analysis of a Relational Join Operation for VLSI," Proceedings VLDBS, 1981, pp. 44-55.

[9] P. B. Hawthorn and D. Dewitt, "Performance Analysis of Alternative Database Machine Architectures," IEEE Trans. on Software Engineering, Vol. SE.8, No. 1, Jun. 1981, pp. 61-75.

[10] G. Z. Qadah and K. B. Irani, "A Database Machine for Very Large Relational Databases,"to be Submitted for Publication to the IEEE Transaction on Computers.

[11] C. Defiore and P. B. Berra, "A Data Mangement System Utilizing an Associative Memory." AFIPS Conference Proceedings, Vol. 42, June 1973, pp. 181-185.

[12] R. Moulder, "An Implementation of a Data Management System on Associative Processor," AFIPS Conference Proceeding, Vol. 42, 1973, pp. 171-179.

[13] R. Linde, R. Gates and T. Peng, "Associative Processor Application to Real-Time Data Mangement," AFIPS Conference Proceeding, Vol. 42, 1973, pp. 187-195.

[14] D. Shaw, "Knowledge-Based Retrieval on a Relational Database Machine," Ph.D. Thesis, Aug. 1980, Dept. of Computer Science, Stanford University.

[15] D. L. Slotnick, "Logic Per Track Devices," Advances in Computers, Academic Press, 1970, pp. 291-296.

[16] J. L. Parker, "A Logic per Truck Retrieval System," Proceeding IFIP Congress 1971, pp. TA4-146 to TA-4-150.

[17] B. Parhami, "A Highly Paraliel Computing System for Information Retrieval," AFIPS Conference Proceedings, Vol. 41, Part II, 1972, pp. 681-690.

[18] G. J. Liposki, "Architectural Feature of CASSM: A Context Segment Sequential Memory," Fifth Annual Symp. Computer Architecture Proceedings, Palo Alto, CA, April 1978, pp. 31-38.

[19] E. A. Ozkarahan, S. A. Schister and K. C. Smith, "RAP-An Associative Processor for Database Mangement," AFIPS Proceedings, Vol. 45, 1975, pp. 379-387.

[20] C. S. Lin, C. P. Smith and J. M. Smith, "The Design of a Rotating Associative Memory for Relational Database System," ACM Trans. Database Systems, Vol. 1, March 1976, pp. 53-65.

[21] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM Trans. Database System, Vol. 4, No. 1 (March 1979), pp. 1-29.

[22] F. Bancihon and M. Scholl, "Design of a Backend Processor for a Database Machine," Proc. of the ACM SIGMOD, 1980, International Conference of Mangement of Data, (May 1980).

[23] S. A. Schuster, H. B. Nguyen, E. A. Ozkarahan and K. C. Smith, "RAP.2- An Associative Processor for Database and its Applications," IEEE Trans. on Computers, June 1979, Vol. C-28, No. 6.

[24] E. Oliver, "RELACS, An Associative Computer Architecture to Support a Relational Data Model," Doctoral Dissertation, Syracuse University, 1979.

[25] H. Boral, "On the Use of Data-Flow Techniques In Database Machines," Ph.D. Thesis, 1981. The Computer Sciences Department, University of Wisconsin, Madison.

[26] R. Epstein and P. Hawthorn, "Design Decisions for the Intelligent Database Machine," Proceedings of NCC 49, AFPS, 1980.

[27] J. R. Goodman, "An Investigation of Multiprocessor Structures and Algorithms for Database Mangement," Memo No. UCB/ERLM81/33 (May 1981), Electronic Research Lab., College of Engineering, University of California/Berkeley.

[28] S. E. Madnick, "INFOPLEX - Hierarchical Decomposition of a Large Information Management System Using a Microprocessor Complex," Proceeding of the NCC, 1975, pp. 581-586.

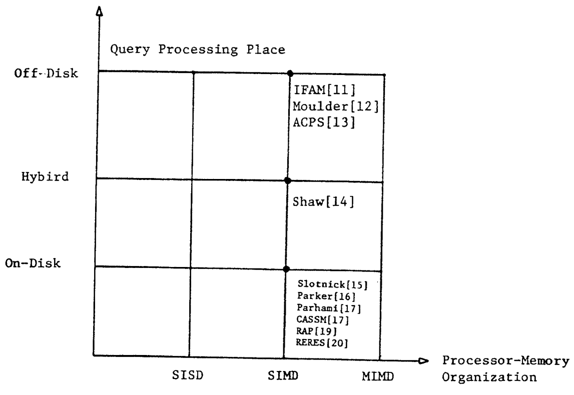


Figure 2.1 The DBMs with DB Indexing Level

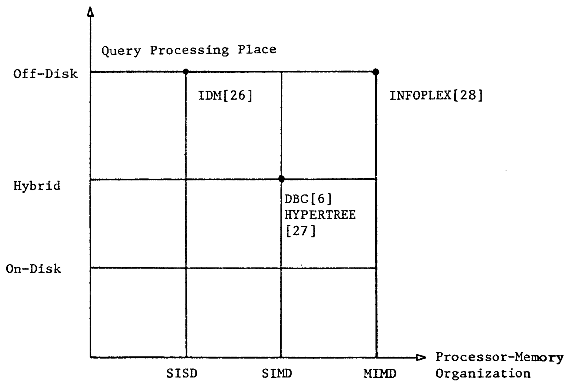


Figure 2.3 The DBMs with Page Indexing Level

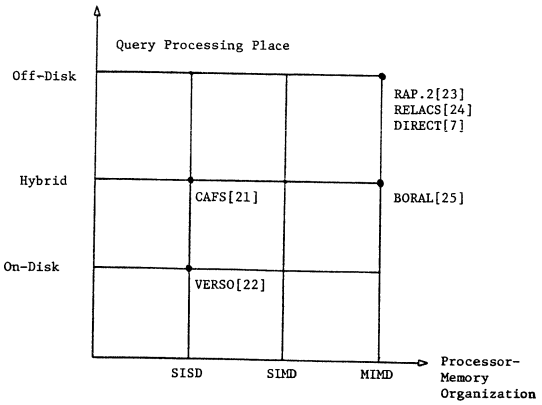


FIGURE 2.2 The DBMs with Relation Indexing Level

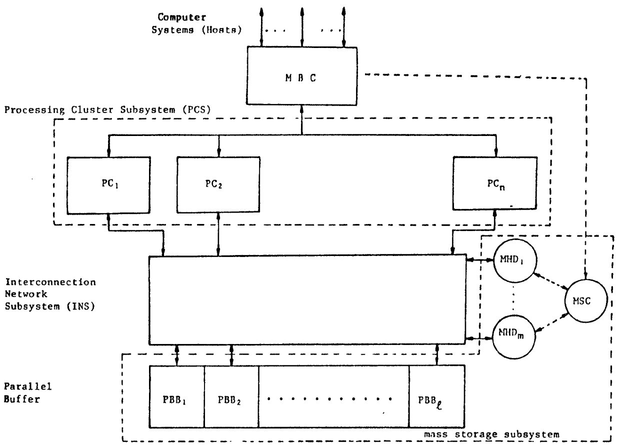


Figure 3.1 The Organization of MIRDM

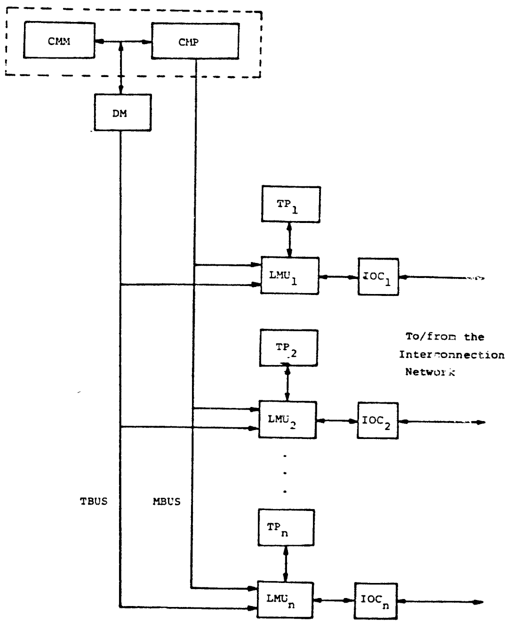


Figure 3.2 The Processing Cluster Organization

EFFICIENT COMPUTING OF RELATIONAL JOIN OPERATIONS BY MEANS OF SPECIALIZED HARDWARE(a)

Yang-Chang Hong
Department of Mathematics
University of California
Riverside, CA. 92521

Abstract -- A hardware architecture is presented which can provide powerful join capabilities to associative processing (AP) systems. The main feature of the hardware is a bit- and word-addressable store which can rapidly remember or recall data. The data might be the values or tuples selected from one relation, in which case the store helps to perform the joining of these values or tuples with the tuples in the second relation. For general case of the join, the store can help in dividing the tuples of the relations being joined into blocks, according to their join column values. The concatenation of tuples in the corresponding blocks is then done by an array of servers. This hardware design emphasizes parallelism in the cross referencing between tuples of the relations being joined, giving considerable performance improvement over existing AP systems. The paper finally gives an analysis of the results of hardware performance under different applications.

Introduction

Limitations of AP Systems

Previous designs of associative processing hardware [1,2,3,5,6,8,9,11] for relational databases have concentrated on searching through data contained within a table. The search involving more than one table has been focused on extensively. This form, called the "implicit" join [1,3,8, 9,11], does not create a derived relation; instead, values selected from one relation are transferred to select the tuples in the second (or original) relation that have the same values in their join columns (i.e., columns on which the joining is based). The joining of tuples of relations (referred to as the explicit join) is, however, carried out primarily by the MFC, to which the AP system is a backend. It will not be very effective if the number of tuples to be joined is large.

The AP systems were based on the parallel processing of a segmented sequential search. While the join operation generally requires a great deal of corss checking between tuples of the relations being joined (which in turn results in a breakdown of parallelism) it is not, in itself, sufficient to make a high performance database machine, especially when a join-dominated database application is involved. Separate hardware which can perform a large amount of cross referencing in parallel must be sought. It is the purpose of this paper to augment an AP system with hardware that will aid in the join operation.

Approach

The main feature of the hardware seems to be a bit- and word-addressable store which can rapidly remember or recall data. The data might be the values or tuples selected from the first relation being joined. They are then used to select the tuples in the second relation. For the general case of join, the store can help to divide the tuples of the relations being joined into blocks of tuples according to their join column values. The concatenation of tuples in the corresponding blocks is then done by an array of servers. The approach emphasizes parallelism in the cross referencing between tuples of the relations being joined, providing a considerable performance improvement over existing AP systems. A hardware simulator was developed on the PDP-11/70 computer for determining hardware parameters - the number of servers and their associated queue length. It also had the ability, given a fixed number of servers, to tell the performance of the hardware under different applications.

Organization of the paper

The body of the paper is divided into three parts: In this first part the hardware architecture is described. The second part is concerned with the computing of relational joins on the proposed architecture. The third part is concerned with the performance analysis of the architecture, which is followed by a summary and conclusion.

Hardware Architecture

The architecture depicted in Figure 1 suggests that the selection of column values or tuples in a relation is done by the AP, while the joining of the tuples is performed by the extended hardware. The command and control processor (CCP) receives data requests from the MFC, translates them into commands, distributes those commands to the AP and extended hardware for execution, receives and formats the resulting data, and outputs the formatted data to the MFC. Like CASSM and RAP [9,11], we assume that data in the AP are stored in encoded form and that the encoding and decoding processes are done by the CCP (i.e., E.D.U.).

The extended hardware consists of five major parts: IP, MB, RAM, S and CP. They are described as follows:

(1) IP is an input processor which serves as

(a) The work was in part supported by the National Science Council, Taipei, Grant #NSC70-0404-E001-02. The author is currently visiting the Department of Electrical Engrg. & Computer Science, Univ. of Santa Clara, Santa Clara, CA 95053.

0190-3918/83/0000/0315$01.00 © 1983 IEEE

a buffer between AP and extended hardware. It accepts the column values or tuples selected from the AP and stores them into queue Q. The registers H and T are used to hold the locations of the first and last entries of Q, respectively. The flag F_Q, when set to 1, indicates Q is full. The IP starts its operation whenever Q is not empty.

(2) MB is a memory bank for storing the tuples of the first relation being joined. It is divided into P modules, designated as M(i), $0 \leq i \leq p-1$. Each module has q words. P and q are design parameters.

(3) RAM consists of single bit array stores (rA, rB, etc.) and an array r of words. The bits of the store and r-words are addressed by encoded values. The addressed bit can be set to 1 or 0 and can be tested for being 1 or 0. The addressed r-word can hold an encoded value or a pointer pointing to a particular word of a particular module, or its contents can be fetched for various joining purposes.

(4) S is a set of servers which forms new tuples of the join. Associated with each server S_i is a queue Q_i, $0 \leq i \leq p-1$, for holding tuples of the second relation being joined. Like queue Q, each Q_i has two registers, T_i and H_i, and a flag F_i. Each S_i is designed to form new tuples from Q_i and M(i), without any memory conflicts. Thus, there are as many S_i's as M(i)'s. A buffer is provided for each server to hold the new tuples it produces. The new tuples can then be either output to the MFC or stored back to the AP for further processing. This is accomplished by the output mechanism.

(5) CP is a central processor which reads data stored in Q, addresses the RAM using encoded values as indices, allocates the storage space in MB for the tuples of the first relation being joined, and deposits the tuples of the second relation into the appropriate server queues. Registers D, T, and BR(i), $0 \leq i \leq p-1$, are provided for allocating storage space for storing the tuples of the first relation being joined.

Computing of Join Operations

This section describes the implementation algorithms on the extended hardware. As suggested above, the joining of relational tuples is done by the extended hardware. In our implementation, implicit and explicit joins are treated in different ways. We will first discuss the implementation of implicit joins and then discuss the explicit joins.

Queries Involving the Implicit Joins

We use an example to illustrate how the single bit array store is used to implement implicit joins.

Example 1. Print all the green items sold by the D1 department.

To answer this query, a simplified database with tables SALES and TYPE is assumed in Figure 2. The query can be implemented in various ways. One way is to apply the selection process to SALES to select the items sold by D1. The selected items are then used as a disjunctive condition to match the TYPE tuples. The green items of the matched tuples are output to the MFC. The procedure implemented by the single bit array store rA is outlined below:

(1) Reset rA.

(2) Scan table SALES by the AP and output the items sold by D1 to the extended hardware, specifically to the queue Q of the IP. The items are fetched and used as indices to set the appropriate rA bits to 1. After this step, all the items sold by D1 are recorded in rA, each having one corresponding set bit in rA.

(3) Scan table TYPE and output all the green items to Q. These items are then checked against items sold by D1. This is done by using the items in Q as indices to address the corresponding rA bits and outputting the items whose corresponding rA bits are set to 1. (We neglect the encoding and decoding processes here.)

The discussion above assumes that all the encoded ITEM values are within the address space of rA. If not, the procedure is repeatedly applied. The ith ($i \geq 1$) repetition is applied to the value range between 2^{t+i-1} and $(2^{t+i}-1)$, where t is the number of bits required for the address space rA.

If the values selected from the second relation are again transferred to match tuples in the third relation, another single bit array store is required. In general, two stores are sufficient, which can be alternatively used for a query involving a chain of implicit joins.

Implementation of Explicit Joins

There have been proposed several different approaches to this type of join [4,7,10,12]. One way is to sort the tuples of each relation being joined into blocks of tuples based on the join column values. The tuples in a block have the same join column value. Each tuple in one block is then concatenated with the tuples in the corresponding block of the second relation. The concatenation is rather straightforward.

Our approach is very similar to this approach. However, it does not actually perform the sorting. Instead, tuples in one relation are first divided into blocks of tuples, according to their join column values, and then are stored in the MB (see Figure 3). No block is allowed to be stored in more than one module. The information about the location of each block of tuples is stored in RAM. In Figure 3, a block of 3 tuples with join column value B1 is stored at the location 100 of module 0, i.e., M(0). The location information of this block is stored in an r-word addressed by B1 (assume B1 is encoded as 1). The setting of the corresponding rA bit to 1 indicates that there is a block of tuples with join column value B1 in the first relation.

After the first relation is stored in MB and the location information is entered into RAM, the AP system starts outputting the tuples of the second relation to the CP. The join column value of each incoming tuple of the second relation is extracted and used as an index to address the RAM. If the addressed rA bit is 0, then the tuple is discarded because there is no match. If set, the module number of the addressed r-word determines the queue in which the incoming tuple will be deposited. It is concatenated to the location number of the addressed r-word so that, when the tuple is processed, the server would know the location of the block the tuple will be concatenated to. The arrangement described above permits the array of servers to produce the concatenated tuples of the join from their queues and the corresponding modules, without any memory addressing contention.

The algorithm assumes that MB is large enough to hold an entire relation to be joined and the block size is less than or equal to the module size. If not, additional effort is needed.

Analysis

The analysis has concentrated on how the number of servers and the length of the server queues affect the hardware performance in computing the explicit join operation. It is divided into two aspects: one is to, given an application, determine the number of servers required and the length of their associated queues so that tuples deposited to the array of servers will not be blocked (theoretically). The application can be characterized in terms of many factors. In our analysis, it is defined by the ratio of the number of tuples in a relation to the number of distinct join column values and will be referred to as the "multiplicity".

The analysis is based on a hardware simulator developed on the PDP-11/70 which simulates the functions of the proposed hardware. The applications are generated using a random number generator.

Our analysis results are stated below. The number of servers required for each application $N_S \doteq (\text{multiplicity})^{1.5}$. If each application runs on its required number of servers, the analysis indicates that only a few units of tuple are required for each server queue to achieve good performance. If the number of servers is fixed, then the performance logorithmically degrades if the multiplicity of an application is greater than that of the application running on that number of servers.

Summary and Conclusion

A hardware architecture which can efficiently compute the relational join operation has been described. The main feature of this architecture is a RAM which can rapidly remember or recall data for computing implicit joins. It can help dividing the tuples of the relations into blocks of tuples for computing explicit joins. The analysis results show that the number of servers required for a tuple deposited to the array without being blocked (theoretically) is a function of multiplicity, independent of the cardinality of the resulting relation.

This hardware provides powerful join capabilities to AP systems, especially when applied to join-dominated database applications. We believe that it can be adapted to the current VLSI technology.

References

[1] Bobb, E., "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol.4, 1, March 1979, pp. 1-29.

[2] Banerjee, J., and Hsiao, D.K., "DBC - A Database Computer for Very Large Databases," IEEE Trans. on Computers, Vol.C-28, 3, 1979.

[3] Chang, H., "On Bubble Memories and Relational Data Base," Proc. 4th Int'l. Conf. on VLDB, West Berlin, 1978, pp. 207-229.

[4] Dewitt, D.J., "Direct - A Multiprocessor Organization for Supporting Relational Database Management System," IEEE Trans. Comp., C-28. June 1979.

[5] Hong, Y.C., and Su. S.Y.W., "Associative Hardware and Software Techniques for Integrity Control," ACM TODS, Vol.6, 3, Sept. 1981, pp. 416-440.

[6] Hong, Y.C., and Su, S.Y.W., "A Mechanism for Database Protection in Cellar-Logic Devices, IEEE Trans. Software Engineering, Nov. 1982.

[7] Menon, M.J., and Hsiao, D.K., "Design and Analysis of a Relational Join Operation for VLSI," Proc. 7th VLDB, Paris, France, 1981, pp. 44-55.

[8] Lin, C.S., Smith, D.C.P., and Smith, J.M., "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS, Vol.1, 1, March 1976, pp. 53-65.

[9] Ozkarahan, E.A., Schuster, S.A., and Smith, K.C., "RAP - An Associative Processor for Database Management," Proc. 1975 NCC, Vol.44, AFIPS Press, Montvale, N.J., pp. 379-387.

[10] Shaw, D., "A Relational Database Machine Architecture," Proc. 5th Annual Workshop on Computer Architecture for Non-Numeric Processing, Pacific Grove, CA, March 1980.

[11] Su, S.Y.W., and Lipovski, G.J., "CASSM: A Cellular System for Very Large Databases," Proc. Int'l. Conf. on VLDB, Sept. 1975, pp. 456-472.

[12] Tanaka, Y., Nozaka, Y., and Masuyama, A., "Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer," Proceedings of IFIP Congress 80, pp. 427-432.

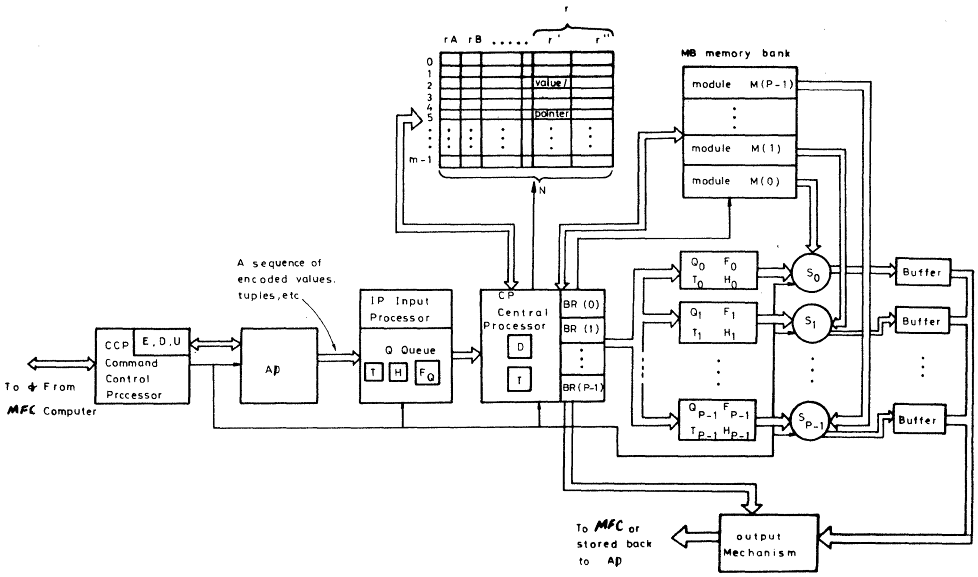


Figure 1. Hardware Architecture

SALES

DEPT	ITEM
<D1>	<CAM>
<D1>	<GEAR>
<D5>	<CAM>
<D5>	<NUT>
<D8>	<CAM>
<D10>	<NUT>

TYPE

ITEM	COLOR	PRICE
<BOLT>	<GREEN>	<5p>
<CAM>	<RED>	<2p>
<COG>	<RED>	<4p>
<GEAR>	<GREEN>	<4p>
<NUT>	<BLACK>	<8p>
<SCREW>	<YELLOW>	<7p>

Figure 2. A Simplified Database with Two Tables SALES and TYPE Linked by ITEM

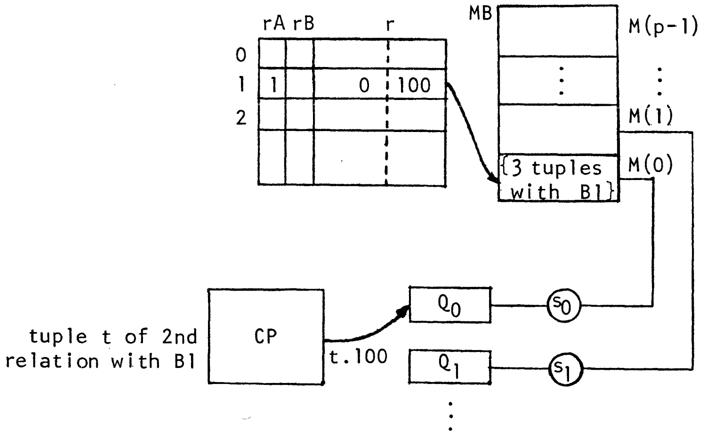


Figure 3. Implementation of Explicit Joins

A VLSI MODULAR ARCHITECTURE METHODOLOGY FOR REALTIME SIGNAL PROCESSING APPLICATIONS

Hungwen Li
RCA Advanced Technology Laboratories
Camden, NJ 08102

Abstract — computer system architecture is currently under development for realtime processing applications having widely varying throughput requirements. Modularity at the VLSI chip level is therefore the highest priority attribute in the design process. In support of this modularity concept, two VLSI primitives — one for computing, the other for interconnecting — were defined for use as "building blocks" to customize the amount of hardware required. To maximize VLSI modularity, hardware system architecture was established incorporating the two VLSI primitives mentioned above and an existing microprocessor for two groupings of hierarchies, cluster and system. The software system architecture adheres to the dataflow concept. Synchronized at the functional level, the signal processing operations that constitute the software are data-driven and data-independent. The software architecture was developed orthogonal to the hardware system architecture to ensure independence and transportability. In addition, the orthogonality between the hardware and software system architectures provides the flexibility to adjust the hardware/software mixture as required to the application without major redesign of the system components. A system architecture simulator, to evaluate various candidate system architectures in response to a set of system parameters and constraints, is being implemented to aid in the tradeoff of the hardware/software mixture.

Introduction

The development of computer system organization (system architecture) for future realtime signal processing applications including radar, sonar, telecommunications, and speech [1] is underway. Modularity at the VLSI chip level is the chief criterion and the hierarchical use of two VLSI modules in the system architecture appears highly effective logistically.

The higher-level attributes of the signal processing system vary from application to application. Obviously, the system requirements of a radar system to simultaneously track 400 targets are far removed from the specifications of a speech terminal providing secure voice transmission; the algorithm for detecting a target via sonar signal processing bears no resemblance to the modulation algorithm of the telecommunication signal processing.

However, the lower-level attributes of such systems, upon which the higher-level attributes are built, are functionally similar over a wide range of applications and are well defined. On this basis, a set of VLSI primitives and a unified system architecture can be defined for a host of applications, each with differing performance requirements. Such primitives and architecture will provide a sound foundation on lower-level attributes and allow the system designer to focus on the design of the higher-level layer efficiently.

Two VLSI primitives were identified to facilitate the developing of lower-level attributes. The first, a Programmable Signal Processor (PSP) [2], supports versatile vector operations which provide a structure for various higher-level algorithms. The second, a Signal Processor Interconnection Switch (SPIS) [3], offers a full connection among participating multiple PSPs, allowing algorithms to be synthesized for applications having different performance requirements. The architecture and characteristics of the VLSI primitives will be discussed in the next section.

The hardware system architecture relies heavily on the SPIS interconnection mechanism. Using the SPIS according to the cluster and system hierarchy, which is described under the Hardware System Architecture section, maximizes the VLSI modularity. The software system architecture follows the data flow concept [4,5] at the functional level and was developed separately from the hardware system architecture to ensure independence and transportability. Orthogonality between the hardware and software system architectures additionally provides the flexibility to adjust the hardware/software mixtue as required by the applications without major redesign of the system primitives. To aid the tradeoff of hardware/software mixture, a system architecture simulator (SARSIM) is being implemented to evaluate various candidate system architctures in response to various system parameters and constraints.

VLSI Primitives

To fulfill the computational requirements of signal processing on an incremental basis, a high throughput programmable signal processor with modularity at the VLSI chip level is essential; furthermore, a unified interconnection mechanism that constructs a multiprocessor system and can be implemented preserving VLSI modularity is very desirable. This concept identified and defined two VLSI primitives, the PSP and SPIS.

Programmable Signal Processor (PSP)

The PSP is a two-chip device consisting of the controller and the Register Arithmetic Logic Unit (RALU). It is designed for the 132-pin package and 1.25 μm CMOS/SOS technology. A clock rate higher than 50 MHz, operating at 5 volts, is required. Figure 1 illustrates the major PSP components and their interfaces to the program memory, the data memory, and the control unit which will be described in the section on system architecture.

0190-3918/83/0000/0319$01.00 © 1983 IEEE

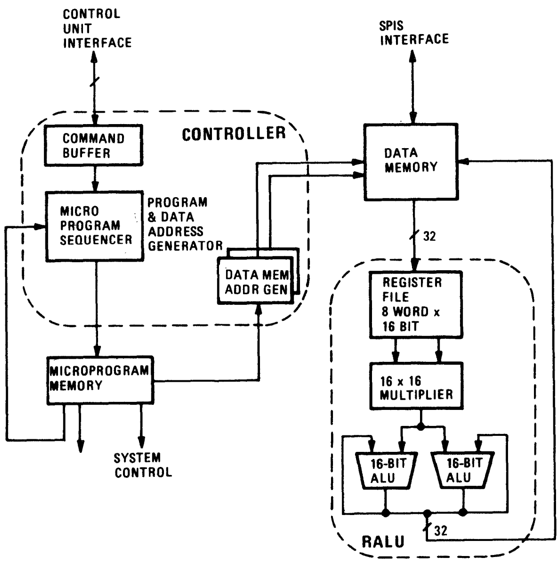


Fig. 1. Block Diagram of Programmable Signal Processor.

RALU. The RALU has a horizontally microprogrammable, pipelined architecture which performs computation-intensive signal processing functions. As shown in Fig. 1, the data path of RALU is organized as a multiplier/dual-adder structure to optimize the Fast Fourier Transform (FFT) computation as well as popular signal processing algorithms such as filtering and convolution.

Major components of the RALU include eight 16-bit registers, a 16-bit-by-16-bit multiplier, and two ALUs. Separate input and output lines support simultaneous memory read and write. Controlled by a 16-bit μ-instruction, all activities in these components occur concurrently yielding a throughput of more than 100 million operations per second (MOPS).

A list of operations directly supported by RALU are given in Table 1. These operations are supported in 16-bit integer, 32-bit complex, 16-bit block floating point, and 32-bit complex block floating point formats.

TABLE 1. FUNCTIONS SUPPORTED BY PSP

Function	
FFT MULTIPLY/ACCUM	CONVOLUTION INTEGRATE
ADD/SUB DIVIDE	POLYNOMINAL POISSON
SQUARE ROOT LOGICAL	GAUSSIAN MIN
CONJUGATE FILTERING	MAX THRESHOLD
CLIP LIMIT	SUM OF ABSOLUTE AMDF

The VLSI modularity is further enhanced by the RALU. For example, one RALU can execute the FFT butterfly in 4 clock cycles. However, two RALUs can be arranged side-by-side to execute a 2-cycle butterfly; more significantly, four RALUs can be arranged in parallel to accomplish a 1-cycle butterfly for applications requiring a very high throughput.

Controller. The PSP controller supports all the control mechanisms required by the RALU. It handles the program sequencing, the data memory address generating and control, and the interface to the control unit.

The controller contains three components: The SEQuencer (SEQ), two Data Address Generators (DAGs), and the system control. The sequencer generates one program address per cycle to pre-fetch one instruction for controlling the DAGs, the RALU, and the remainder of the system. The sequencer also handles the command buffer, the interface to the control unit for passing the signal processing descriptors (described in the system architecture section). The DAG generates one data memory address per cycle to read or write memory data. Along with the address, the timing and the control signals are generated on-chip to simplify the system interface circuit.

The sequencer supports immediate, register, and relative addressing modes and convenient branching functions such as IF and CASE desired by the program control. Two stacks, the iteration stack and the program counter stack, cooperatively achieve FOR and WHILE looping actions described in Ada. Special LOOP action allows a program to operate continuously without resetting th iteration count. This feature is particularly useful in a clock-driven, high data-rate environment.

Flexible addressing modes and a wraparound mechanism provided by DAG handle wraparound for circular buffers and corner-turning of two-dimensional memories. In this way a wide spectrum of data memory accessing patterns (such as bit reversal of FFT and window movement) generic to the signal processing applications can be managed..

Signal Processor Interconnection Switch (SPIS)

The most challenging problem faced in implementing VLSI modularity when a large number of PSPs are interconnected is overcoming the pin limitation of the package. Recognizing this, the SPIS separates the passing of data from the controls so that the data path can be bit-sliced, which allows more PSPs to be connected in one module.

Figure 2 depicts the data path of SPIS. Each signal processor is equipped with a lccal memory and is allowed to have its own port attached to the shared mrmory via SPIS. The shared memory is organized as a B-word-wide memory with word size W (where B is the block size and is the smallest addressable unit of the shared memory). The block

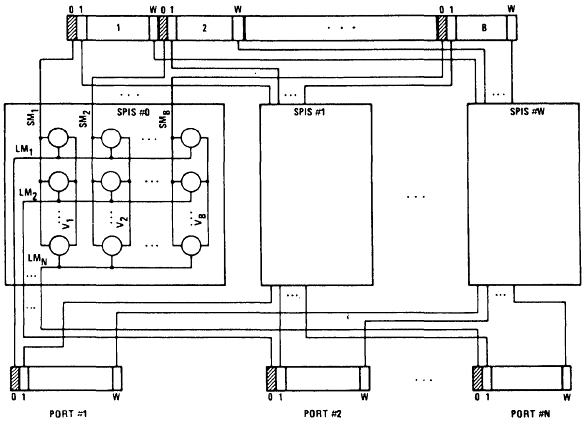


Fig. 2. Schematic Diagram of SPIS Data Path.

size B is programmable and is subject to the number of ports (N) and the relative speed between the shared memory and the local memory accessing.

A control unit (not shown) is dedicated to control the SPIS operation via a control bus (CBUS) to which every signal processor attaches and sends commands. The control unit accepts commands in a round-robin fashion, with a fixed time slot allocated to each signal processor. During each time slot, the control unit may perform one of the three operations as summarized in Table 2. These operations are performed by sending control signals from the control unit to the shared memory unit, SPIS chips, and each signal processor.

In any operation, a parallel transfer (B words) occurs at the shared memory end while serial transfer (one word) occurs at the local memory end. This requires one bit storage per crosspoint and one vertical connection per column (Fig. 2), which enhances the crossbar switch and increases the complexity of SPIS accordingly. A first-cut logic design indicated that a 32 x 32 SPIS chip is at the complexity of about 60K transistors.

TABLE 2. BASIC COMMAND TYPES

Operation	Source Operand	Destination Operand	Iteration Count
READ	SMU-BLOCK #	PORT #	# OF BLOCKS
WRITE	PORT #	SMU-BLOCK #	# OF BLOCKS
COPY	PORT #	PORT #	# OF BLOCKS

The bit-sliced SPIS architecture overcame the pin limitation of the VLSI package and made it feasible that at least a 32 x 32 configuration can be supported with today's technology (<100K transistors and 132-pin package). With a large number of interconnections supported per chip, off-chip switching delay is reduced. This leads to a higher data-transfer rate, which contributes significantly to the performance of the data flow software architecture (discussed in the next section). In addition to the above-mentioned advantages, the bit architecture matches ideally to the bit-organized memory, leading to easier implementation of error-correcting codes which ensures higher reliability.

The major drawback of the SPIS architecture is that any crosspoint failure will potentially affect the overall system because of its bit-sliced structure and full connectivity. This, however, can be easily corrected by having redundant SPISs for each bit path.

System Architecture

The system architecture development is divided into hardware and software architecture, both of which are developed orthogonally to provide not only independence and transportability, but also the flexibility to adjust the hardware support as required by the application or as restricted by the physical or cost constraints.

Software System Architecture

The software system architecture follows the concept of data flow because signal processing functions are basically data-driven and data-independent. Unlike the proposed data flow concept [4], which synchronizes the operations at the arithmetic level, the data flow advocated here for signal processing synchronizes the operations at the functional level. This was chosen because most signal processing functions operate on vectors of reasonably large size. Consequently, functional-level data flow reduces the overhead of passing both the data and the descriptors.

A detailed software system architecture has been defined [6]. Due to its complexity, only the portions sufficient to address the methodology are presented here. Similar work [7] has been presented recently.

Data Flow Model. As shown in Fig. 3, a signal processing function is represented by a node and its Input-object (I-object)and Output-object (O-object). These objects are represented by the direct links going in and out of the node. The Input-objects and Output-objects can be data or controls. A node is initially in the "wait" state and can be converted into the "executable" state only when all its associated data I-objects have arrived and control I-objects are in a "true" state.

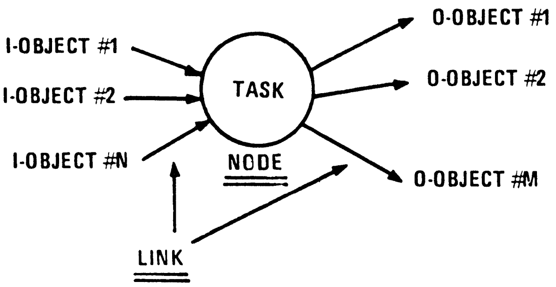


Fig. 3. Data Flow Model.

Data Flow Graph (DFG). Based on the flow model, an algorithm or application can be typified by a data flow graph consisting of a set of nodes representing the processing elements of the graph, a set of links or queues representing the directed information flow through the graph, and a command program carrying out control functions. Graph input queues provide a means of transporting data into the graph; graph output queues provide a means of transporting data out of the graph; and command programs interface to the data processing subsystem.

Each node in the graph represents a specific signal processing operation called the underlying operation of the node. The underlying operation may be either a subgraph or a predefined primitive operation (macro). Subgraphs allow hierarchical structuring in graph definition. The expansion is complete when a graph contains only nodes with macros.

A node has a set of input data queues supplying data to the node. Associated with each data input queue are:

a) A threshold value representing the minimum number of data elements that must be present on the corresponding queue before the macro can be executed.
b) A read amount representing the number of data points that the macro will use as input data.
c) A consume amount representing the actual number of data points to be removed from the queue after the macro has been executed.

These queue parameters are managed by a data flow schedule algorithm to maintain the DFG execution.

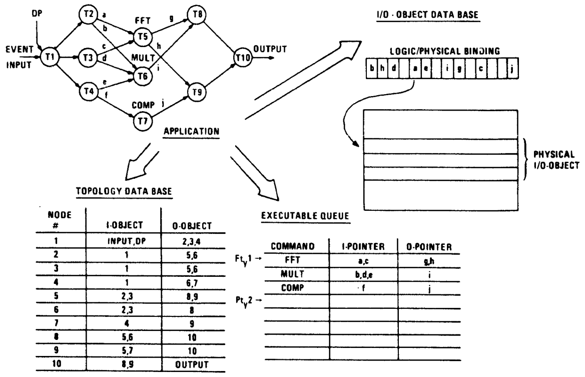
A link or queue of a DFG represents the directed flow of information from node to node within a graph or from a node to another graph. There are two types of queues — data queues carrying data and control queues for synchronization. A command program, which carries out control functions in an application and serves as the interface between signal and data processing, can be associated with a graph.

Automatic application partitioning into a DFG is a difficult issue and is currently under study. Even with the aid of some partitioning programs, it is believed that manual partitioning will still play the most important role in constructing a DFG.

Object Database, Data Flow Schedule Algorithm, and Synchronization. The DFG contains two types of information: the topology which illustrates the input/output relationship between nodes, and the descriptor which contains the node name and the queue parameters. Both topology and descriptor information must be described in Signal Processing Language (SPL) [6] and translated into the object database for the graph execution. An example of the object database can be found in Fig. 4.

The descriptor includes the name of the node and the queue description for each link. Each queue descriptor includes queue type (control or data), input/output type, capacity, consume amount, produce amount, input/output pointer, etc.

The schedule algorithm utilizes the object database to execute the data flow model. It is this mechanism that synchronizes the nodes in a DFG. Each node has an indication of its state (wait, executable, processing, or finished) in the object database. Some hardware entities examine these states; change them from "wait" to "executable;" and move the descriptor of the executable nodes to the PSP for execution. The synchronization is automatically established by the data flow model and the object database. All nodes are executed asynchronously; however, maximum parallelism is allowed when sufficient numbers of processes are available.



NODE #	I-OBJECT	O-OBJECT
1	INPUT,DP	2,3,4
2	1	5,6
3	1	5,6
4	1	6,7
5	2,3	8,9
6	2,3	8
7	4	9
8	5,6	10
9	5,7	10
10	8,9	OUTPUT

COMMAND	I-POINTER	O-POINTER
FFT	a,c	g,h
MULT	b,d,e	i
COMP	· f	j

Fig. 4. Topology and Object Database Generation.

Deadlock. A deadlock situation is possible when there exists a feedback path in a DFG; however, it can be prevented by creating a control object along with the feedback path. The feedback control is initially "true"; it becomes "false" after the execution of the node that accepts the feedback path. This feedback control object will be triggered to be "true" via the execution of the node generating the feedback. By constructing a logically sound DFG, the deadlock can be totally prevented.

Hardware System Architecture

Hierarchical approach and modularity are the highest priority factors in configuring hardware system architectures. Two levels of hierarchy, system and cluster, are established as the system architecture (Fig. 5) using the same SPIS primitives.

A cluster consists of one control unit, a set of Signal Processor Modules (SPM), and a set of Input/Output Processors (IOP). These modules are connected by a SPIS network, a CBUS, and a SBUS. The cluster control unit is attached to the object database and is responsible for executing the data flow schedule algorithm described previously. The SPM is responsible for computation and consists of the PSP primitive, the local memory, and a local control unit interfacing with both CBUS and SBUS.

The IOP consists of a local memory and a local control unit interfacing not only with CBUS and SBUS but also with the Cluster CBUS and Cluster SBUS. The IOP is responsible for inputting data from the sensor, outputting data to the data processing subsystem, and transferring data among the clusters. The IOP can be implemented with existing microprocessors from which the local/cluster/system control unit may also be constructed without further dedicated VLSI primitives.

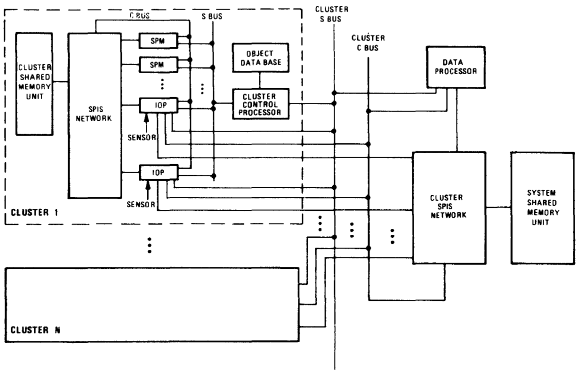


Fig. 5. Hardware System Architecture.

In the hierarchical approach, several clusters grouped together constitute a system. At the system level of hierarchy, identical interconnection methods and modules are adopted, with the exception of replacing the SPMs by the cluster block in which one or more IOPs may communicate via the Cluster SPIS network and Cluster CBUS. A Cluster SBUS and a system control unit with an object database are also available for the execution of the data flow. After significant data reduction by the signal processing front end, data processing is performed. The data processor, which also houses the command program, is most appropriately connected at the system hierarchy.

In light of the realtime signal processing applications, the hierarchical architecture is the ideal structure for the two developed VLSI primitives. Hierarchical approach groups tightly-coupled nodes into a cluster and several clusters into a system, localizing the communication traffic and maximizing the utilization of the SPIS. This type of grouping is best depicted by several channels of identical signal processing and is a natural and logical way of mapping the signal processing problem to the hardware. Linked to the software architecture, the hierarchical hardware approach is almost a one-to-one mapping to the hierarchical expansion of the subgraph, which strongly indicates the high modularity of this methodology in both hardware and software dimensions.

The choice of the hierarchical architecture is also driven by the physical limitation of the packaging. Using standard chassis and printed-circuit boards, about 32 programmable signal processors can be assembled in one chassis as a cluster. The physical interconnection of clusters can be conveniently done in one independent chassis. This simplifying approach achieves high modularity even at the chassis level.

Many issues in the hardware system architecture (e.g., number of SPMs and IOPs in a cluster, the communication bandwidth of CBUS and SBUS, and the structure of the object database, etc.) remain undetermined. These issues are, more or less, application dependent and should not be totally predetermined until the application requirements are specified. To resolve these issues, a design automation tool at the system architecture level is needed and will be discussed in the next section.

System Architecture Simulator (SARSIM)

The system architecture simulator is a tool for evaluating the performance of a candidate system architecture before building the prototype hardware. The idea behind SARSIM is to input the parameterized architecture attributes — including the hardware system architecture in terms of PMS notation [8], the software system architecture in terms of the data flow schedule algorithm, and the application in terms of DFG — into the simulator and to allow the system designers to observe and collect the statistics of the interesting parameters. SARSIM consists of six software modules. The modules' functions are described below.

The graph translater converts the DFG codes in SPL to the object database for use by the policy handler in the execution of a data flow model.

The topology handler inputs the system configuration described in PMS notation and its associated parameters (e.g., delay of a bus). The handler then builds a network of queues with corresponding queue disciplines for the manipulation of the global clock handler.

The policy handler module, where the data flow schedule algorithm resides, can implement a variety of algorithms following the same model. By observing the output of the statistic handler, the algorithm performance can be measured to aid in choosing the algorithm.

The task handler mimics the PSP execution, generates the appropriate delay information to the statistic handler, and transmits execution status to the policy handler.

The statistic handler collects and reports the interesting parameters such as the CBUS delay caused by the contention, SPIS performance as a function of the number of ports, and the impact of the object data base structure on performance.

The global clock handler mimics the parallel events sequentially and drives the simulator. It examines every queue generated by the topology handler, services the pending requests in the queues, and adjusts the global timing information for each queue so that the statistic handler can perform the statistic calculation.

Status and Future Study

In the aspect of the VLSI primitive, the PSP has been defined at the register transfer level (RTL) and an RTL simulator has been implemented to validate the correctness of the architecture and the completeness of the instruction set by implementing a set of signal processing macros. Furthermore, the logic designs of the RALU and the SPIS were completed.

The definition of the software system architecture and the signal processing language have been completed and documented [6], while the detailed hardware system architecture needs to be investigated.

The conceptual definition of the system architecture simulator has been finished and its implementation is currently proceeding. After its completion, a series of hardware system architectures, data flow schedule algorithms, and different object database structures will be tested. Expected test results are a family of performance curves serving as the guidelines of the design space for the hardware/software tradeoff.

The fabrications of PSP and SPIS are planned, from which a hardware testbed will be constructed as a signal processing system prototype.

References

[1] A. Oppenheim, ed., Application of Digital Signal Processings, 1978, Prentice Hall, Englewood Cliffs, NJ.

[2] RCA internal report.

[3] P. Sawker, T. Forquer, E. Schernecke, and H. Li, "A Multi-Port Memory Organization for Use in Distributed Computing Systems," Proc. of 3rd Int'l Conf. on Distributed Computing Systems, Miami/Ft. Lauderdale, FL, Oct. 18-22, 1982.

[4] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings on 2nd International Symposium on Computer Architecture, pp. 126-322., IEEE, New York, 1975.

[5] J.R. Heath, G.D. Broomell, and A. Hurt, "A Distributed Computer Architecture for Real-time, Data Driven Applications," Proc. of 3rd Int'l Conf. on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, Oct. 1982.

[6] RCA internal report.

[7] Y.S. Wu, "A Common Operational Software (ACOS) Approach to a Signal Processing Development System," Proceedings of ICASSP83, Boston, MA, April 1983.

[8] C. Bell and A. Newell, Computer Structures: Readings and Examples, New York, McGraw Hill, 1971.

EMSY85 - The Erlangen Multi-Processor System for a Broad Spectrum of Applications.

G. Fritsch, W. Kleinoeder, C.U. Linster and J. Volkert

Institute of Computer Science (IMMD)
University of Erlangen - Nuremberg
Federal Republic of Germany

ABSTRACT

A new Erlangen multiprocessor system, EMSY85, will consist of a grid-like array of microprocessors operating asynchronously, each of which is coupled via memory with a limited number of its neighbors. We intend to demonstrate that for a broad spectrum of applications the system's performance can grow nearly in proportion to the number of processors in the array. The operating system is based on UNIX*. A programming environment for parallel programs makes the system attractive to the users. Many design decisions have been based on the results of an existing pilot project.

1. Introduction and Motivation

Numerous attempts have been made in the past few years to increase computing power by means of multiprocessor systems with various architectures. Two such systems have been implemented in Erlangen, SYMPOS [12,15], which is a symmetrical system, and EGPA** [5], which is hierarchical.

The experience gained with applications on these two projects led scientists at the Computer Science Department (IMMD) of the University of Erlangen - Nuremberg to conceive EMSY85, which will be implemented in the next few years. EMSY85 consists of a grid-like array of microprocessors operating asynchronously, each of which is coupled via memory with a limited number of its neighbors. Because of the large number of processors, a symmetrical system (each processor has access to each memory modul) is unrealistic [6]. On the other hand because of the many computation-intensive user applications with a matrix structure, a processor field with a grid structure was chosen. Above this array there is a hierarchy of processors, whose job it is to supervise the array and to transport data between processors that happen not to be neighbors. The higher-level processors can also perform tasks other than supervision. Thus EMSY85 is also well suited for tree-like user applications. In addition, results from the pilot project EGPA lead us to believe that many user applications with a subtask structure that is neither grid-like nor tree-like can easily be mapped onto, and computed efficiently on EMSY85.

Therefore the project's main goal is to show that for a broad spectrum of applications, system performance can indeed grow nearly in proportion to the number of processors in the array. We are thus planning to test a large number of algorithms from such fields as physics, chemistry, operations research and image-processing, many of which have already shown large speedups on the pilot project. In order to make the system attractive to potential users, the complexity of programs written for the system must not be essentially greater than that of programs written for a monoprocessor. Not only must the system's higher-level language contain constructs for asynchronous programming, there must be a programming environment capable of supporting the development of

* UNIX is a Trademark of Bell Laboratories.
** EGPA was supported by the Federal Ministry for Research and Development, F.R.Germany

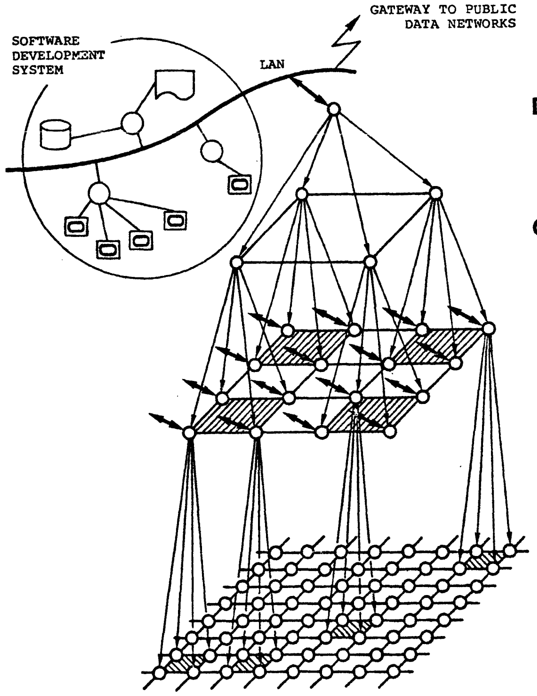
0190-3918/83/0000/0325$01.00 © 1983 IEEE

parallel software.

The paper describes five aspects of the EMSY85 project: the hardware, the operating system, the programming environment, measurement and performance aspects and applications.

2. The EMSY85 - Architecture

The Erlangen multiprocessor system EMSY85 will consist of identical Processor-Memory Modules (PMMs). Each PMM will consist, in turn, of an iAPX 286/287 microprocessor and a one-half megabyte multiport memory. The PMMs are arranged hierarchically in four levels (A,B,C,D) as shown in Fig. 1.



○ Processor - Memory - Module (PMM) of EMSY 85

—— symmetric multiport-memory connection between neighboring PMMs

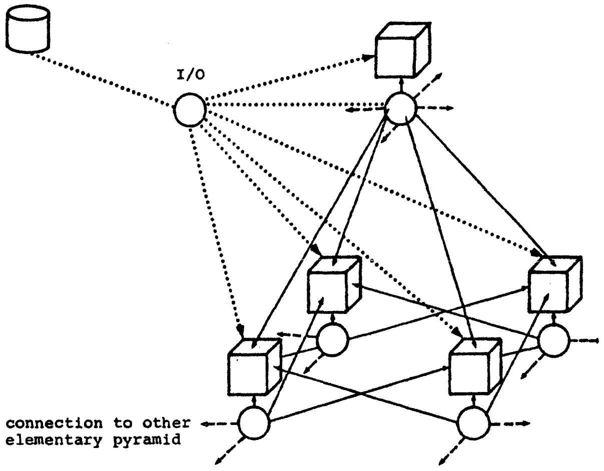
—► asymmetric multiport-memory connection between PMMs of different hierarchical level

◄► I/O communication to elementary pyramid, supported by I/O processor

Fig. 1: EMSY 85 - Hardware-Structure: Four hierarchical levels A, B, C, D. Some of the elementary pyramids are highlighted.

At the topmost level there is only a single PMM, for which there will be several standby processors to increase the system's reliability. At each lower level, each PMM is connected to exactly four neighbors at the same level, i.e. each processor has access to the memories of its neighboring PMMs. Thus at each level the hardware has a grid structure.

In addition to the horizontal accesses, each PMM, except of course the very lowest, has access to four PMMs at the next lower level. The vertical connections are equipped to broadcast data downward to all four of the lower PMMs simultaneously, say, to transmit code segments. On the other hand, though the lower PMMs do not have memory-access to those at higher levels, they are able to interrupt their supervising PMM. These substructures, consisting of four processors and their supervisor, are called elementary pyramids c.f. Figure 2.



○ Processor - □ Memory - Module (PMM)

Fig. 2: EMSY 85 - Elementary pyramid

Several of them are highlighted in Figure 1. Each elementary pyramid has an I/O processor with the corresponding I/O devices, for the most part, Winchester disks. The I/O processor, which has access to all of its elementary pyramid memories, is controlled by the supervising PMM.

The overall arrangement of EMSY85's PMMs is, as the preceding discussion suggests, pyramidal. The topmost PMM is connected to a network containing several software-development processors on which the operating system and applications are in development.

An EMSY85 pyramid can, of course, be extended downward arbitrarily, by adding new levels. Such an extension increases significantly the computing power of the system.

For many kinds of computations, the main computing load will be carried by the lowest level, leaving the higher levels underutilized. For such applications it would be reasonable to taper the pyramid so that each lower level in an elementary pyramid has nine, or even sixteen, rather than merely four PMMs.

Experience with the pilot project, EGPA, which was built using five powerful miniprocessors, has shown however that excessive tapering can lead to bottlenecks at the higher levels that restrict the system's overall performance. EMSY85 will therefore have two manually switchable configurations, one more strongly tapered than the other:

Lowest level PPMs:	8 x 8	9 x 9
Element. pyramid PPMs:	4 + 1	9 + 1
Number of levels	4	3
Total PPMs:	64+16+4+1 = 85	81+9+1 = 91

3. EMOS - the EMSY85 Operating System

The operating system, which is based on UNIX will be structured in a hierarchy analogous to the hardware. The operating system consists of more or less autonomous subsystems, one per processor. Each of the subsystems has a common kernel, but the subsystems on the lowest level are rudimentary and increase in power toward the top of the pyramid.

The opinion, often found in the literature, that UNIX is unsuited to multiprocessor systems can no longer be maintained without qualification. The multiprocessor project "SYMPOS - An Operating System for Homogeneous Multiprocessor Systems" has shown that UNIX can be modified with relatively little effort and essentially without changing its structure. The effort for such a modification depends on the complexity and homogeneity of the hardware, the desired user-friendlyness, and the planned spectrum of applications. In addition, the question is relevant whether the user should have the possibility of implementing genuinely concurrent processes, or merely quasi-concurrent processes. Considering all of these parameters, we estimate a total effort of 4 to 16 man-years.

In order to avoid the phenomenon of processor-thrashing in memory-coupled multiprocessor systems, we adopted and expanded on an idea that found limited application in the CMU multiprocessor projects [8]. The problem consists of relieving the bottleneck involved in common memory-access; the solution consists of maximizing the amount of code and data for local functions loaded into local memory. The procedure leads to an operating system that consists of a number of local subsystems based on a common kernel. This approach was implemented so successfully in SYMPOS that it will be adopted for EMSY85. Subsystems of varying power for the different hardware levels are easily provided since the system is partitioned into modules that can be freely combined to form a complete system.

In spite of the fact that EMSY85 is not a symmetrical system, similar problems arise since every memory is equipped with seven ports, independently of the size of the elementary pyramid. In view of the number of accessing processors, measures similar to those in SYMPOS will certainly be necessary to prevent bottlenecks.

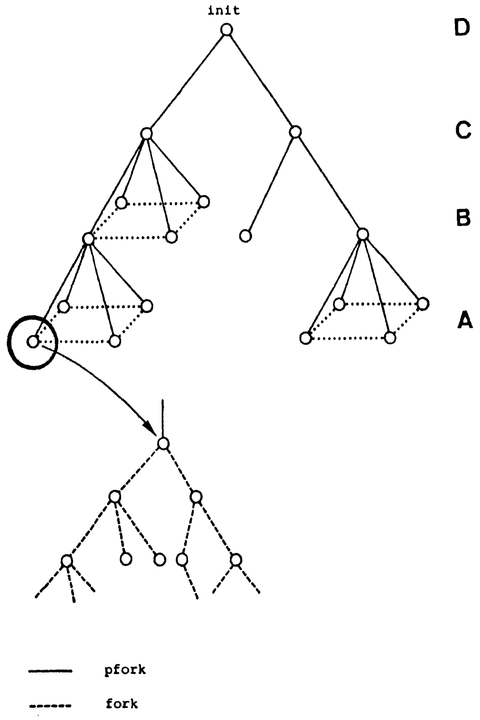


Fig. 3: EMSY 85 - Process-Structure

UNIX insiders will note the congruence between the hardware structure and UNIX's process structure. It is thus fairly easy to extend the original process-management to a distributed system. We shall use the **fork** function (spawn a process) as an example.

In the local environment, i.e. on a single processor, **fork** functions as usual. There is in addition a distributed version, **pfork** whose effect is analogous to its local cousin. The difference is merely that for a **pfork** process a new hardware environment is initialized. The two processors involved must process a common physical memory space; in case more than one generation level is involved the memory spaces may be disjunct. The hardware to which a **pfork** process is assigned can be influenced by parameters such as access to non-local data segments, processor assignment, and so forth.

Figure 3 shows a typical process structure resulting from multiple **pfork** and **fork** operations. The process identifiers consist of a triple identifying hardware level, processor number, and local process identifier.

In addition to the multiprocessor-specific analogues **fork, wait, exit, alarm,** etc. there will be a number of completely new functions for the coordination of asynchronous processes and management of global files. The new coordination functions include mechanisms such as semaphores, lock/unlock and message switching.

In this section we have discussed several aspects of process management in the EMSY85 Operating System. Further research areas relevant to the project include resource management in multiprocessor-systems, online reconfiguration after hardware failures and adaptable management strategies, among others.

4. A Programming Environment for Parallel Programs

The operating-system interface in the EMSY85 multiprocessor system permits the implementation of a user's algorithm as a system of cooperating concurrent processes. In order to permit the use of such a multiprocessor system by non-specialists, tools are provided in a parallelprogramming environment oriented to the user's problem rather than to the system's architecture.

Of course it would be ideal if the task could be distributed automatically to the various processors. The user could then program his application as if it were a sequential process. Current experience with the EGPA system leaves us skeptical about the success of completely automatic analysis of sequential into parallel algorithms.

The parallel programming environment therefore assumes the following model: an application consists of sequential subtasks and a description of their interdependencies, either concurrent or sequential. The individual subtasks are formulated in a powerful higher-level programming language (C, because of the use of UNIX). There is a programming package that permits easy formulation of the required synchronization by means of such calls as

"execute subtask x on processor y"

"wait for the termination of subtask x on processor y"

The programming package implicitly includes the generation of the requisite system of processes, handles the individual calls necessary for process communication (e.g. using messages), and initiates the subtasks on their respective processors. This approach has been successfully tested in the pilot project [13].

Another approach that is much easier to use and which has proved efficient as well, has been borrowed from data-flow theory. The theory that applies to elementary operations such as "+" or "*" yields an elegant generalization to subtasks, viewed as complex operations, which we call macro data-flow. The synchronization dependencies between subtasks can be expressed in analogous fashion. In the EGPA pilot project we have shown that these tools are especially attractive for applications whose asynchronous structure is extremely complex [10].

5. Measurement and Performance Evaluation

The measurement and evaluation subproject of EMSY85 is intended to support the users in optimizing performance, whether they be implementing the operating system, the programming environment, or applications. Since the simultaneous operation of nearly one hundred processors is beyond human comprehension without some kind of visual support, there is a software-measurement system applicable to all levels of the system, whose measurement points can be selected arbitrarily. A trace of the selected events is recorded along with timing information.

Then a direct evaluation of this information is made possible by an online visual-display package. Not only can the system's current status be displayed, the trace data can be used to display the flow of events in slow-motion.

Since the measurement points are

chosen using the programming environment, the data can be displayed in the user's notation, i.e. with his symbolic names, rather than as hexadecimal numbers.

It would, of course, involve excessive implementation effort to attempt to find an "optimal" version among several candidate implementations by measuring their performance. However a modeling and evaluation procedure for complex tasks based on stochastic analysis of the measurement data permits a choice among interesting variants [9]. The task model used to implement this subsystem is based, as are the environment's other programming-support tools, on the data-flow approach.

A survey of research on hardware and software measurement as well as performance evaluation in the EGPA pilot project, which constituted the planing basis for the measurement and evaluation software in EMSY85, can be found in [3].

6. Applications

EMSY85 will be capable of implementing a broad spectrum of applications. Among them will be tasks that require intensive computation, e.g. problems from physics, operations research and pattern-recognition.

Such problems can often be reduced, either directly or via discretization, to problems in linear algebra. An important research area is the discovery of appropriate asynchronous algorithms, especially for the solution of large systems of linear equations with either sparse matrices (10**6 unknowns), e.g. finite elements or systems of differential equations, or dense (10**3 to 10**4 unknowns). These problems are not difficult to implement on EMSY85 because of the close match between the structure of the tasks and the system's array. For other classes of tasks, the match is not as felicitous. For such problems, strategies must be developed for mapping the problem onto EMSY85's hardware structure. Not all applications algorithms can be adapted without modification, and must in fact sometimes be re-developed from scratch. Such problematic tasks, e.g. from pattern-recognition, nonlinear programming (say for technical installations), simulation of complicated systems (telephone networks, multiprocessors, VLSI circuits), and so forth, are also to be investigated. These will require research in the decomposition of tasks, the adapting of asynchronous algorithms, and the design and implementation of parallel programs. The computing demands of several of these tasks, especially those from pattern-recognition, comprise both high computation speed and high data-transfer rates.

By means of these applications, we intend to show that, for problems of sufficient size, the system's performance improves nearly linearly in the number of array-processors (level A). Considering the breadth of the applications areas, EMSY85 will have thus proven itself to be a multi-purpose system.

As a result of experience with the pilot-project EGPA (Erlangen General-Purpose Array), we are convinced that we shall indeed be able to demonstrate the expected improvement. EGPA's hardware structure corresponds to a single elementary pyramid in EMSY85, consisting of four array PMMs and one supervisory PMM. Thus the limiting speed-up for an algorithm run on EGPA (versus a monoprocessor) is four-to-one.

We list below a number of applications actually tested on EGPA along with their speed-up factors.

Subject:	Speed up:
Linear algebra [7]:	
-Matrix inversion (200 x 200 dense)	
Gauss-Jordan	3.8
column-substitution	ca. 4.0
-Matrix multiplication (200 x 200)	3.7
-Solving of linear equations	
Gauss-Seidel	ca. 4.0
Differential equations [2]:	
-Relaxation	ca. 3.5
Image processing and graphics:	
-Topographical representation [10]	3.6
-Illumination of the topographical model	2.4
-Line following (vectorizing of a grey-level matrix)	ca. 2.9
-Distance transformation [4]	ca. 3.0 - 3.3
Non linear programming [1]:	
-Search for minima of a multidimensional object function given by an algebraic term	ca. 3.2
Graph theory:	
-network flow with neighborhood support (each idle processor helps one of its neighbors)	3.5
Text formating [14]:	2.6

These encouraging results, which span a broad spectrum of applications, were an essential factor in the design of EMSY85. On the basis of theoretical investigations, speedups proportional to the number of processors at the lowest level can be expected on systems of the same type.

7. Conclusions

The University's Institute of Computer Science is working hand in hand with an industrial partner, Siemens' Corporate Laboratories for Information Technology in Munich. The cooperation is in the design, development and production of the hardware, as well as in the development and testing of parallel algorithms for computation-intensive applications such as the simulation of complex systems.

The EMSY85 project employs about fifty scientists at the University from the following academic chairs in the Computer Science Department: Computer Architecture, Performance Evaluation, Operating Systems, Programming Languages and Pattern Recognition. The German Federal Ministry for Research and Development is supporting EMSY85.

8. Acknowledgements

The authors gratefully acknowledge the contributions of all members of the Erlangen Group to this paper. They also would like to thank Mrs. L. Lange for drawing the figures.

References

1. Fritch,G., H. Mueller
"Parallelization of a Minnimization Problem for Multiprocessor Systems"
CONPAR'81, Lecture Notes in Computer Sience No.11,453-463,Springer Verlag Berlin-Heidelberg-New York 1981

2. Fromm, H.J.
"Multiprozessor-Rechneranlagen: Programmstrukturen, Maschinenstrukturen und Zuordnungsprobleme"
Arbeitsberichte des IMMD, Univ. Erlangen-Nuernberg, Band 15, Nr.5 '82

3. Fromm, H.J., U. Herksen, U. Herzog, K.H. John, R. Klar and W. Kleinoeder
"Experiences with Performance, Measurement and Modeling of a Processor Array"
IEEE Transactions on Computers, vol. C-32, no. 1, Jan. 1983

3. Goessmann, M., J. Volkert und H. Zischler
"Image Proc. and Graphics on EGPA"
EGPA-Int. Paper(to be published)

5. Haendler, W., F. Hofmann, H.J. Schneider
"A General Purpose Array with a Broad Spectrum of Applications"
Computer Architecture, Workshop of the G. I. Erlangen/R.F.Germany, May 1975

6. Haendler,W.
"Aspects of Parallelism in Computer Architecture"
Parallel Computers - Parallel Mathematics Feilmeier, M. (ed)
North Holland Publishing Company, Amsterdam, 1977

7. Henning, W., M Vajtersic and J. Volkert
"Matrix Inversion Algorithm for the Parallel Computer EGPA"
EGPA-Int. Paper(to be published)

8. Jones, A.K. and P. Schwarz
"Experience Using Multiprocessor Systems - A Status Report"
Computing Surveys, Vol.12, No.2, June 1980

9. Kleinoeder, W.
"Stochastische Bewertung von Aufgabenstrukuren fuer hierarch. Mehrrechnersysteme"
Arbeitsberichte des IMMD Univ. Erlangen-Nuernberg Band 15 Nr.11 '82

10. Kneissl, F.
"Realisierung von Datenflussmech. auf hierachische Mehrrechnersysteme"
Arbeitsberichte des IMMD Univ. Erlangen-Nuernberg, Band 15 Nr.12 '82

11. Kneissl, F.
"Macro Data Flow on EGPA Config."
EGPA-Int. Paper(to be published)

12. Linster, C.U.
"SYMPOS/UNIX - Ein Betriebssystem fuer homogene Polyprozessorsysteme"
Arbeitsberichte des IMMD Univ. Erlangen-Nuernberg, Band 14 Nr.3 '81

13. Rathke, M.
"Benutzung der Parallel-Schnittstelle des EGPA-Rechners"
EGPA-Int. Dokumentation

14. Rathke, M.
"SAP: Ein optimistischer Algorithmus fuer die parall. Textverarbeitung"
EGPA-Int. Paper(to be published)

15. Wurm, F.X.
"Auftragssystem fuer eine Multiprozessoranlage"
Arbeitsberichte des IMMD, Univ. Erlangen-Nuernberg, Band 13 Nr.7 '80

Maximum Pipelining of Array Operations on Static Data Flow Machine*

Jack B. Dennis
Gao Guang Rong

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Data flow computers are a radical departure from conventional computer architecture, and new methodologies are required for generating efficient machine-level programs from high-level user programming languages. In this paper, we show that, for certain programs in the Val language, it is possible to construct machine-level data flow programs that support fully pipelined computation. A Val program in the class considered consists of blocks of code each of which defines a new array value either by a **forall** expression in which each element may be computed independently, or by a **for-iter** expression that defines array elements by a first-order recurrence relation.

1. Introduction

In this paper we study the translation of the program structures used to express array computations in the programming language Val [1], a functional programming language designed for expressing computations to be executed by computers capable of highly concurrent operation, data flow computers in particular.

The organization of data flow computer that appears most attractive to us for high performance computation is the static data flow supercomputer described in [2] [3]. A machine level program fo such a computer, regarded as a collection of instruction cells, is essentially a directed graph, with nodes corresponding to instructions and an arc for each instruction destination field. We will use such diagrams to present data flow machine code structures in the remainder of this paper.

Two constructs in the Val programming language are of major importance in expressing scientific computations. A **forall** expression can be used to express the construction of an array where each element of the array is specified by the same computational rule and all elements may be computed independently. Example 1 is a Val **forall** expression which uses values from two arrays B and C to construct a new array A.

```
A : array[real] :=
  forall
    i in [0, m+1]    % range specification
    P : real :=      % definition
      if (i = 0)|(i = m+1) then C[i]
      else 0.25*(C[i-1]+2.*C[i]+C[i+1])
      endif;
  construct B[i] * (P * P)    % accumulation
  endall
```

Example 1. A Val forall Construct

The **for-iter** expression in Val is the construct used to express iteration—the computation of sequences of values in which the value produced in one cycle depends on the value or partial results produced by the proceeding cycle. Example 2 is a **for-iter** loop which constructs an array X.

```
X : array [real] :=
  for     i : integer := 1; % initialization
      T : array[real] := [0: 0.]
  do
  let
  P : real := A[i]*T[i-1]+B[i] % definition
  in if i < m then    % body
      iter T := T[i : P]
         i := i+1
      enditer
    else T
    endif
  endlet
  endfor
```

Example 2. for-iter Construct

The Val programs of interest in this paper are those made up of program blocks, each of which is a **forall** or a **for-iter** block. Each block may be thought of as a *producer* of an array value, and a 'consumer' of other array values produced by other blocks. This simple structure matches the main body of many practical programs of computational physics.

<u>Definition</u> A *pipe-structured* program is a Val

* This research was supported by the Department of Energy under grant number DE-AC02-79ER10473 and the National Science Foundation under grant number MCS-7915255.

0190-3918/83/0000/0331$01.00 © 1983 IEEE

program in which all array constructions are defined by non-nested blocks such that: (1) each block is either a **forall** block or a **for-iter** block, (2) the index ranges of the arrays generated by the blocks are fixed.

Pipe-structured programs are attractive candidates for implementation as fully pipelined machine code structures for data flow computers.

2. Pipelined Mapping of Primitive Expressions

Pipelined execution of computations is very natural on the static data flow computer. We first study the pipelined implementation of a restricted class of Val expressions. which contains no nested forall or for-iter expressions and no array constructor operations.

Definition Let i be an identifier called an index variable. Then a *primitive expression* (PE) on i is any Val expression which may be constructed using only the following rules:

(1) A scalar literal constant is a PE.
(2) An identifier of a scalar value is a PE.
(3) If *E1* and *E2* are PEs, then (*E1* op *E2*) is a PE, where op is an arithmetic or relational operator.
(4) If A is an identifier that denotes an array, then A[i+m] is a PE, where m is an integer constant.
(5) Let *E* be a Val let-in construct expressed as Let ⟨*definition*⟩ in *E0* endlet. If the definition part, contains only PEs and *E0* is also a PE, then *E* is a PE.
(6) If *E1*, *E2*, *E3* are PEs, then **if** *E1* **then** *E2* else *E3* endif is a PE.

If a primitive expression is formed using only rules (1), (2), (3), and (5), its implementation as an acyclic data flow instruction graph is straightforward, and the methods developed by Montz [6] may be used to balance the instruction graph so that it supports fully pipelined computation. For the array access operations (rule (4)) Two matters must be addressed to make pipelined operation work correctly: (1) the elements of the incoming array not used in the computation must be discarded so they do not cause jams; (2) buffering must be inserted to introduce any skew needed to balance the pipeline. As an example, consider the expression 0.25 * (C[i-1] + 2. * C[i] + C[i+1]) from the body of Example 1 in Section 1. The corresponding fully pipelined instruction graph is shown in Figure 1. Here we suppose the array C is represented by m+2 result packets for the index set {0,..., m+1}. The boolean control sequences select just those array elements needed for the computation. The two FIFOs balance the pipeline by holding values of array elements between their arrival at the identity instructions and the time when they must enter the arithmetic pipeline.

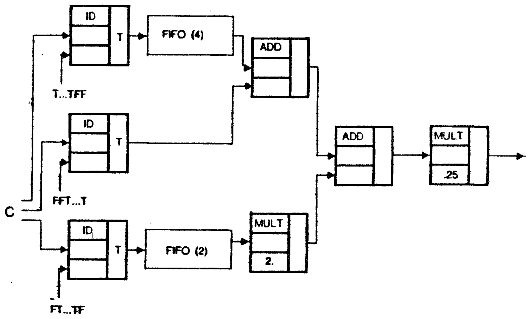


Fig. 1. Pipelining for Array Selection Operations

The final case is that of conditional expressions. The general technique is illustrated in Figure 2 for the following example:

```
if C[i] then (A[i]+B[i])
else 5.*(A[i]*B[i]+2.)
endif
```

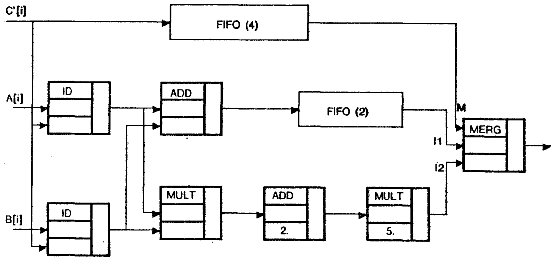


Fig. 2. Pipelining for an if-then-else expression

This instruction graph makes use of instruction cells (identity operations in this case) in which a boolean operand directs a result packet to destinations according to a tag (T or F) on the destination arc. The control input M directs the merge instruction to forward one or the other of its data operands. Note that to keep the program fully pipelined, it may be necessary to add FIFO buffers to both the data and the control arms.

This discussion and examples lead us to the following theorem which provides a basis for the constructions presented in the next two sections.

Theorem 1 For any primitive expression, a fully pipelined data flow instruction graph can be constructed.

3. Pipelined Mapping of forall Constructs

In this section we will present the *pipeline* *scheme* where the array elements are generated in sequence by implementing the body of the forall construct as a pipelined instruction graph.

Definition A *primitive* forall expression is a forall expression in which: (1) The index range is specified as [p,q] where p and q are integer constants. (2) The right hand side of the definitions and the expression in the accumulation part are all primitive expressions in i, where i is the index variable of the forall expression.

The fully pipelined implementation of a primitive forall expression (Example 1 in Section 1) is shown in Figure 3. It is essentially the instruction graph obtained by cascading the instruction graphs for the definition expression and the accumulation expression. We suppose the input arrays B and C are fed to the instruction graph element by element for the index set {0,..., m+1}. The identity instructions select from the input arrays those elements needed for the computation, and the merge instruction combines results computed by different rules into the sequence of values that represent the constructed array. Further details of this implementation scheme can be found in [4]. As a result we have

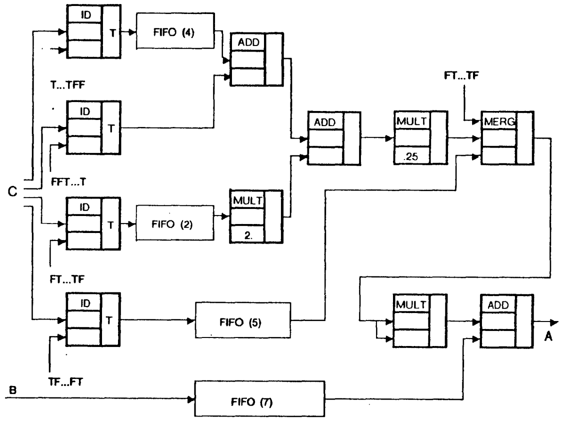


Fig. 3. Pipelining of A Primitive forall Expression

Theorem 2 For any *primitive* forall expression, a corresponding fully pipelined data flow instruction graph can be constructed.

4. Pipelined Mapping of for-iter Construct

To study the pipelined implementation of iterative programs we first define a class of for-iter constructs which are built on primitive expressions.

Definition A *primitive* for-iter construct is a for-iter expression with two loop variables–let them be i and X–such that: (1) Loop variable i takes on successive integer values p, p+1,..., q for successive evaluations of the for-iter body, and the loop terminates after i = q. (2) The loop variable X is initialized to the empty array or by X := [r: *E*] for some integer r and some primitive scalar expression *E*. Each iteration appends to the array by X := X [i: *E*]. (3) The result expression on loop termination is X which will be the array constructed by the for-iter expression.

One scheme for implementation the for-iter construct is to introduce feedback in data flow instruction graphs [7]. Due to the presence of cycles, the instruction graph corresponding to such a scheme can not, in general, be fully pipelined.

The most common problems involving for-iter array operations are recurrences. Example 2 shows the general form of a first order recurrence function expressed in Val. In fact, it is exactly the Val code for the following mathematical notation,

$$\begin{aligned} x_i &= A_i x_{i-1} + B_i \\ &= F(a_i,\ x_{i-1}) \qquad (2) \end{aligned}$$

where a_i is the ordered pair (A_i, B_i), and the function F is composed of add and multiply. Based on a solution first proposed in [5], we note that (2) can easily be transformed into

$$x_i = F(c_i,\ x_{i-3})$$

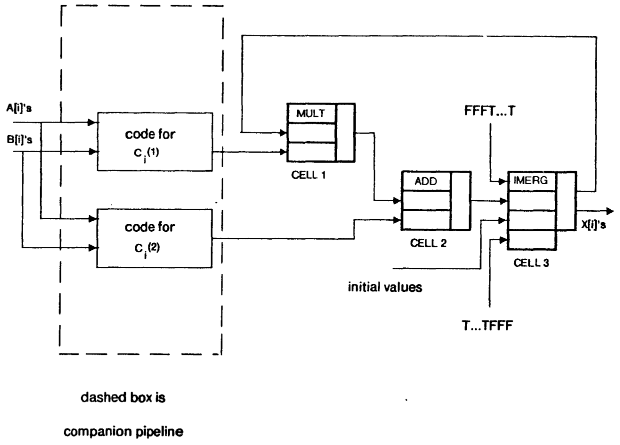


Fig. 4. Pipeling of A Simple for-iter Expression

where the pair c_i is computed from the a's by

$$\begin{aligned} c_i(1) &= A_i A_{i-1} A_{i-2} \\ c_i(2) &= A_i A_{i-1} B_{i-2} + A_i B_{i-1} + B_i \end{aligned}$$

This transformation is useful to us because x_i now depends on x_{i-3} instead of x_{i-1}, and we note that the function F has an execution delay of 3. Therefore we can compute F by using an auxiliary pipeline that computes c_i from a_i using the scheme shown in Figure 4. This added pipeline (see the dashed-line block in Figure 4) will be named the *companion pipeline* in the rest of this paper. Also the function computed by the companion pipeline is named the *companion function* of the recurrence function. By constructing the companion pipeline properly, it is possible to keep the whole pipeline running at maximum throughput.

The previous related work on the use of companion functions [5] has been on conventional architecture, where the pipeline configuration is wired into hardware. It is impractical, however, to construct a separate hardware companion pipeline for each possible recurrence relation in the computation. In contrast, the pipeline for a data flow machine is software implemented. It is more flexible to introduce a piece of data flow program which acts as a particular companion pipeline. Hence, it is much more attractive to apply this scheme on a data flow machine. Now let us return to the problem of classifying Val for-iter constructs which have good mapping schemes.

Definition A *simple* for-iter expression is a primitive for-iter expression such that (1) the recurrence function it denotes has a companion function and (2) the Val expression which computes the companion function is a PE.

Using the scheme presented above, we have theorem :

Theorem 3 A simple for-iter expression can be mapped into a fully pipelined instruction graph.

Some other techniques for pipelined implementation of iteration expressions are known, generally involving trading off delay in exchange for achievement of computation at the maximum rate.

5. Fully Pipelined Pipe-Structured Programs

A pipe-structured program in which each forall expression is primitive and each for-iter expression is simple has an elegant structure; each component is a consumer and producer of array values and has an implementation as a fully pipelined data flow instruction graph. Due to the applicative nature of the Val programming language, the data dependencies among the forall and for-iter expressions define an acyclic directed graph in which each edge represents a path over which an array value is sent from producer to consumer. Since the component instruction graphs are fully pipelined, the balancing algorithm [4] may be applied to the acyclic interconnection to produce a fully pipelined instruction graph for the complete pipe-structured program.

Theorem 4 For any pipe-structured program in which each forall expression is primitive and each for-iter expression is simple, a fully pipelined data flow instruction graph can be constructed.

6. Conclusion

We have developed a formal model of pipe-structured programs for use in studying algorithms for balancing and optimizing corresponding data flow instruction graphs for fully pipelined operation. Interested readers will find a rigorous formulation and analysis in [4]. Investigation of the design of a compiler that will automatically construct fully pipelined code for a large class of Val programs are subjects for further study.

References

[1] Ackerman, W. B. and J. B. Dennis. Val -- A Value-Oriented Algorithmic Language Preliminary Reference Manual.'' Technical Report 218, LCS, MIT, Cambridge, MA, 13 June 1979.

[2] Dennis, J. B. Data Flow Supercomputers'' IEEE, Computer, Nov. 1980.

[3] Dennis, J. B., Gao, G. R., and Todd, K. A Data Flow Supercomputer'' Computation Structure Group Memo 213, LCS, MIT, Cambridge, MA, Jan 1982.

[4] Gao, G. R. An Implementation Scheme for Array Operations in Static Data Flow Computer'' MS Thesis, LCS, MIT, Cambridge, MA, June 1982.

[5] Kogge, P. M. A parallel Algorithm for Efficient Solution of a General Class of Recurrence Equations.'' IEEE Trans. Comput., Vol. c-22, no. 8, Aug. 1973.

[6] Montz, L. B. Safety and Optimization Transformations for Data Flow Programs.'' Technical Report 240, LCS, MIT, Cambridge, MA, January 1980.

[7] Todd, K. W. High Level Val Constructs in A Static Data Flow Machine'' Technical Report 262, LCS, MIT, Cambridge, MA, June 1981.

A DIRECT MAPPING OF ALGORITHMS ONTO VLSI PROCESSING ARRAYS BASED ON THE DATA FLOW APPROACH

Israel Koren

Computer Science Division
University of California
Berkeley, CA 94720
on leave from the
Dept. of Electrical Engineering
Technion - Haifa 32000, Israel

Gabriel M. Silberman

Dept. of Computer Science
Technion
Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

A new approach to the utilization of VLSI processing arrays by means of the algorithms running on them is presented. The idea is to represent algorithms as *data flow graphs*, and then map these graphs onto the array. This approach obviates the need to develop new concurrent algorithms to utilize the parallelism inherent in the array, while offering a general environment for the realization of algorithms on semi-custom VLSI.

1. INTRODUCTION

The approach taken in this research to achieve parallelism within a special purpose VLSI chip, without developing new concurrent algorithms, is the *data flow* approach [3-5]. In it, concurrency of activities is achieved at the lowest possible level by treating each machine instruction as an independent activity. This enables "fine grain parallelism" [3], not achievable when scheduling and synchronization of concurrent activities are controlled by software.

However, we do not propose to use one of the known general-purpose architectures of data flow machines [3-5]. Instead, we suggest to map the data flow graph which describes the problem in hand, on a regular array implemented in VLSI. These regular arrays of identical cells take considerably less time to design and manufacture [1,2]. Also, the mapping should not be fixed but changeable, enabling the user to map various data flow graphs (algorithms) on the same chip. Regularity and flexibility are thus obtained, increasing the number of potential applications for the chip and thereby making it more appealing to the semiconductor industry.

In the following we consider the hexagonal array as a basis for illustrating our approach. This array has a flexible structure [1,6], simplifying the task of mapping. In addition, fault-tolerance may be introduced into it [6,7] allowing it to recover from errors by reconfiguration. We then propose an architecture for the processing element (PE) which constitutes the basic cell in the array. Also presented is an outline of the general graph-to-array mapping process.

2. PRELIMINARIES

In contrast to control flow computers, data flow computers have no program counter. In the latter, an instruction is ready for execution when all its operands have arrived. Consequently, all such instructions may be executed in parallel. If the processing capabilities of the data flow computer are sufficient, the highest degree of parallelism may be achieved.

The program is represented by a data flow graph. The vertices correspond to operators, and data tokens move along the arcs. Parts of the graph may have to be executed iteratively. This might cause tokens to accumulate on certain arcs and result in the presence of tokens belonging to different iteration steps at the input arcs of an operator. This problem may be solved by either labeling (coloring) the tokens [4] or by preventing the accumulation altogether [3]. The latter is achieved by preventing an operator from producing a new output token until the previous one has been consumed [3,8]. This approach still enables pipelining through the data flow graph.

Maximum pipelining is not however, always possible. Bottlenecks may appear in parallel segments of the graph [8] (e.g., paths of a conditional expression), thus severly limiting the amount of concurrency. To eliminate these bottlenecks an optimization technique has been suggested in [8], inserting buffers (delay operators) in some of the parallel paths. However, these buffers may result in either an increase in the overall delay through the pipeline or a reduction in the throughput [8].

In the architecture suggested here dynamic length FIFO queues are employed. In this way, the level of concurrency is increased without the penalty of an increase in the overall delay. The labeling scheme as presented in [4] might be inappropriate for our purposes due to the additional hardware complexity.

3. PE ARCHITECTURE AND PRINCIPLES OF OPERATION

The basic PE, shown in Figure 1, is connected to its six immediate neighbors by dedicated busses, in an hexagonal processor array. The PE contains six registers $x_0, x_1, \cdots, x_5$ which are connected to the six communication busses. Each of these registers can either receive or transmit tokens and will accordingly be defined either as a primary input register or a primary output register.

Each basic PE contains in addition, an arithmetic and logic unit (ALU) and a *Pseudo Associative Memory unit* (PAM). Each location within the PAM contains a key and a data element. The PAM has therefore, two input registers k_i (key-in) and d_i (data-in), and two output registers k_o and d_o (Figure 1).

The ALU is capable of performing the basic arithmetic and logic operations. Its inputs may be connected to any primary input register, or to the PAM data-out register d_o. The ALU result is routed to either a primary output register or the PAM data-in register d_i.

The overall function of the PE is specified by the designation of each of the x_i registers as primary input or output register, by the internal connections of these registers to the ALU and the PAM unit registers, and by the operation performed by the ALU. Thus, the operation of a PE may be defined by a set of statements like

$$PE : \begin{cases} d_i := x_0 + x_1 & x_4 := d_0 \\ x_3 := x_1 & x_5 := x_2 \end{cases}$$

The PAM unit has two modes of operation, *First-in First-out* (FIFO) mode and *Associative* mode. In the FIFO mode the PAM unit serves as an input or output buffer for token accumulation. In the associative mode the PAM serves as a queue in which a key is attached to each data element. This mode of operation is useful when implementing recursion and in it data elements are accessed by using keys instead of addresses. However, a fully associative memory is not necessary. Instead, a sequential access memory unit with added logic can be employed, using shift registers, CCD's, or magnetic bubbles.

In the FIFO mode of operation the PAM unit may serve as a queue for accumulating either incoming or outgoing tokens. The purpose of this FIFO queue is to dynamically equalize the length of parallel paths in the graph in order to achieve maximum pipelining. The fixed capacity

0190-3918/83/0000/0335$01.00 © 1983 IEEE

of the PAM might limit the maximum length of the FIFO queue. However, the PAM units in two or more neighboring PEs may be chained and used for accumulating tokens from a single source.

For the sake of brevity, the exact principles of operation of the proposed PAM unit are not detailed here.

4. IMPLEMENTING BASIC DATA-FLOW STRUCTURES

This section shows how to use an array of PEs in the implementation of data-flow structures. We begin by examining the basic data-flow elements, which can be directly mapped onto a single PE. These are the *Arithmetic and Logical Operators* (like addition, negation, And, complement, etc), and the *Conditional Operators* (like arithmetic comparisons, test for zero, etc). Also requiring only a single PE are the *Flow Control Operators*. They do not affect the contents of the token, but rather its progress and/or destination.

The simplest such operator is *Copy* (denoted by C). It makes two identical copies of its input token.
Merge (M) operator is used when a data token may come from two different sources (paths), and it is to be merged into a single path for further processing. This operator is also capable of producing a Boolean token whose value depends on the input path which supplied the token for the output.
The *Router* (R) operator receives two inputs, a data token and a logical value. The data token is copied into exactly one of two output registers, depending on the value of the logical input. This is analogous to "distribute" in [9].
The *Gate* (G) operator transfers the incoming token to an output register if a second token is present at another input register. The G and M operators may be used to implement the "Select" operation [9].
Self-Iterating Operator (L) is used in those cases where the result produced by the PE is immediately used as argument to the next operation. This saves the need to create "external" loop structures (e.g., [10]).

Figure 2 depicts a data flow graph which calculates the factorial function, using C, M, R and L operators. Notice the labeling of the outputs of the R operator by T and F. This is used to specify which output path corresponds to each logical value. The *L* operator receives two inputs, one being the initial value for the iteration (in this case n), and the other a Boolean value which determines, for each iteration, whether to load a new initial value, or use the one from the latest iteration.

Having defined the basic data-flow elements, we now show how these may be combined to yield the basic data-flow *structures*.

4.1. Conditional (if-then-else) -

This construct, has the general format:

if <condition> then <expression1>
else <expression2>,
endif

and is evaluated as <expression1> *or* <expression2>, depending on the logical value of <condition>. The above statement may be implemented in general as shown in Figure 3.

Notice that when a certain branch of the conditional is taken, the tokens corresponding to the other branch are not produced at all. This is achieved by using an R operator with only one output; this way tokens corresponding to different computations are not mixed.

We also deal here with the problem of keeping in correct sequence the results being produced by a conditional construct. The ordering is achieved by using an extra R and two Gs as shown in Figure 3. The initial token present at the R input is routed to the G in the appropriate branch of the conditional, thus allowing only its result to flow through. When a result arrives at the M, a token (its value is immaterial) is recirculated to the R to enable further output tokens.

Note that conditional constructs may result in token accumulation, because of different path lengths between the two branches. Here we can benefit from the PE's capability of dynamically adjusting to token traffic, by using the FIFO mode of the PAM.

4.2. Iterative (Do-While, Repeat-Until) -

Iterative data-flow constructs make use of conditionals much in the same way traditional programming languages do. In general, we have the two iterative constructs, *do-while* and *repeat-until*, depending on whether the test for loop repetition is placed before or after the loop body, respectively. Figure 4 shows how a repeat-until loop is used to approximate the square root of a (positive) value c, using Newton's iterative method.

4.3. Recursion -

This construct is by far the most involved in the data-flow context. Actually, most current data-flow architectures do not handle recursion at all. However, recursion is generally recognized as a good programming technique. When used, it leads in many cases to simpler and shorter algorithms which are easier to understand and to prove correct.

For the sake of brevity we do not present here the way recursion is implemented, we would like however to indicate that any possible implementation of the recursion construct will be substantially more complex than all previous ones. The benefits of its use should therefore, be carefully examined before incorporating it.

5. DATA FLOW GRAPH MAPPING ALGORITHM

In the following, we show a simple (by no means optimal) scheme for mapping complete data flow graphs onto an hexagonally connected PE array. The mapping algorithm presented is executed externally by some host, and the results are then fed into the array for distribution. The graph mapping process is clearly dependent on the array topology. Therefore, different such topologies result in different mapping algorithms. Nevertheless, they all must tackle the same problem, namely, the non-planarity of the graph, arising from both ordering of operands and iterative constructs (loops).

We begin by assigning levels to the vertices (operators), where an operator (mapped on a PE) is at level i if all its operands come from operators at level $i-1$ or above. Clearly, our objective is to find minimal levels for all operators. In the case of loops, we do not consider the target of the loop to be a descendant of the source.

A second pass is now made to insure that no two operators which are either a loop source or target, are at the same level. If this is the case, the level is split until the condition no longer exists. The reason for this is to enable the use of the horizontal busses between PEs for connecting the source to the target.

In the next step of the mapping we connect the operators in the various levels. The outputs of level i have to be ordered so as to fit the inputs to level $i+1$.

After ordering the operands, (possibly by introducing extra levels which exchange operands), we connect the loop source with its target by using the horizontal connections between PEs. We first route the operand from the source to the boundary of the current graph. Then, we route it to the level of the loop target. Finally, we use again the horizontal connections to reenter the graph up to the target operator.

An example of the mapping of the factorial function from Figure 3 is shown in Figure 5(a). After the mapping process, has been completed, reduction techniques may be applied to reduce the size of the final mapping. For example, two levels may be collapsed into one, taking advantage of unused horizontal connections. Such a reduction procedure, when applied to the example in Figure 5(a), results in the mapping shown in Figure 5(b).

The mapping algorithm described above is by no means optimal and the only purpose it serves is to show the feasibility of mapping arbitrary data flow graphs onto hexagonal arrays. More efficient mapping algorithms for hexagonal arrays as well as for other array topologies are clearly needed.

Once the mapping algorithm is completed, we have to convey its results toward every relevant PE in the array. A simple way of doing this is to input a "configuration string", each component of which is addressed to a specific PE, and contains the setup information for it. Another possibility that is being investigated, is to execute the mapping algorithm within the array itself in a distributive fashion.. This may enable a dynamic mapping, allowing PEs which have completed their current processing task, go into a configuring phase, change their function and execute another part of the data flow graph. Such a dynamic mechanism may allow the mapping of larger data flow graphs on a given VLSI chip. It will also facilitate the handling of faulty PEs and/or connections.

6. CONCLUSIONS

The idea of directly mapping an arbitrary algorithm on a VLSI array has been shown to be feasible. However, further research has to be carried out before the effectiveness and practicality of this approach are established.

Clearly, not all algorithms that can be mapped on a array will use it effectively. Some algorithms may require a too large chip area, other may not execute fast enough. Consequently, methods have to be developed for estimating the chip area that will be used by a given algorithm, and its execution time.

REFERENCES

[1] C.Mead and L.Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

[2] M.J.Foster and H.T.Kung, "Design of Special-Purpose VLSI Chips: Examples and Opinions," *Proc. of the 7th Symp. on Comp. Arch.*, April 1980, pp.300-307.

[3] J.B.Dennis,"Data Flow Supercomputers," *Computer* Vol.13, Nov.1980, pp.48-56.

[4] I.Watson and J.R.Gurd,"A Practical Data Flow Computer," *Computer*, Vol.15, Feb.1982, pp.51-57.

[5] A.L.Davis,"The Architecture and System Methodology of DDM1," *Proc. 5th Symp. Comp. Arch.*, April 1978, pp.210-215.

[6] D.Gordon, I.Koren, and G.M.Silberman,"Embedding Tree Structures in Fault-Tolerant VLSI Hexagonal Arrays," submitted for publication.

[7] I.Koren,"A Reconfigurable and Fault-tolerant VLSI Multiprocessor Array," *Proc. of the 8th Symp. on Comp. Arch.*, May 1981, pp.300-307.

[8] J.D.Brock and L.B.Montz,"Translation and Optimization of Data Flow Programs," *Proc. of the 1979 Int'l Conf. on Parallel Processing*, Aug.1979, pp.46-54.

[9] A.L.Davis and R.M.Keller,"Data Flow Program Graphs" *Computer*, Vol.15, Feb.1982, pp.26-41.

[10] Arvind and K.P.Gostelow,"The U-Interpreter" *Computer*, Vol.15, Feb.1982, pp.42-49.

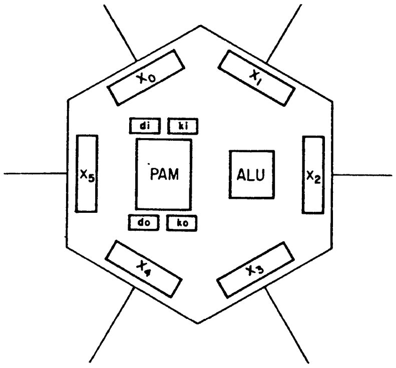


Fig. 1: The basic processing element.

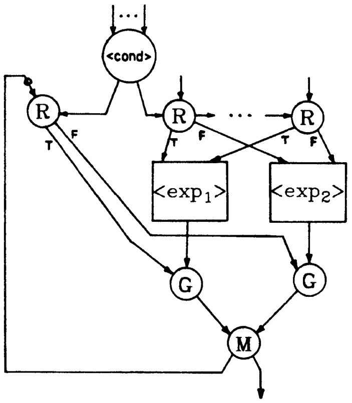
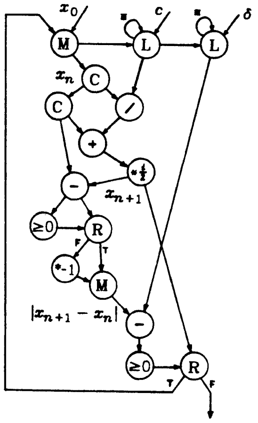


Fig. 3: Conditional construct.



repeat $x_{n+1} = \frac{1}{2}(x_n + \frac{c}{x_n})$

until $|x_{n+1} - x_n| < \delta$

Fig. 4: The Newton method.

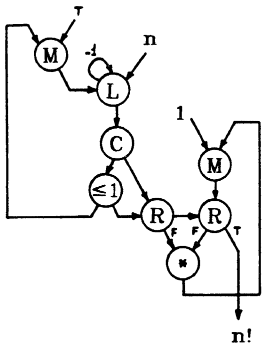


Fig. 2: The data flow graph for the factorial function.

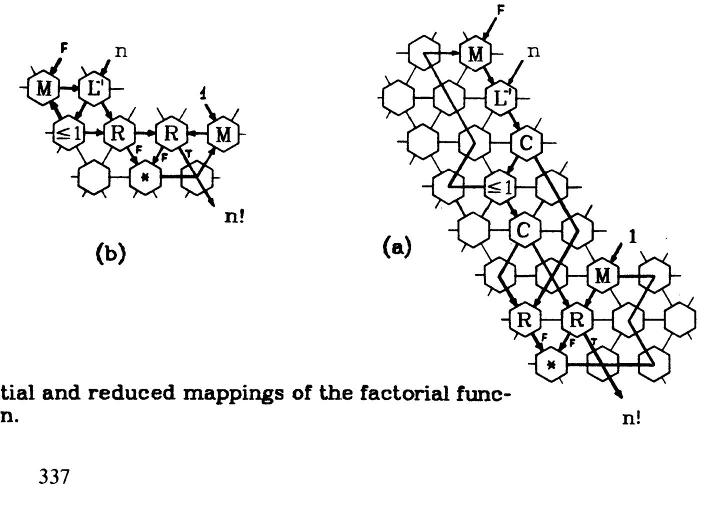


Fig. 5: Initial and reduced mappings of the factorial function.

An Algorithm For Processor Allocation In A Dataflow Multiprocessing Environment

Lawrence Y. Ho and Keki B. Irani
Computing Research Laboratory
The University of Michigan
Ann Arbor, Mi. 48109

ABSTRACT: This paper deals with the problem of associating the operations of a program with the processors in a distributed dataflow environment. The objective is to develop an algorithm which will partition a given dataflow program and map the blocks of the partition onto the processing elements of a general dataflow multiprocessing system. System performance is measured by the total processing time of the program.

INTRODUCTION

The total execution time of a program can be broken down into the actual computation time and the time for interprocessor data communication. When a program is partitioned among the processors of a dataflow system, the communication time is dependent on several factors such as the interprocessor connection network and the allocation scheme which maps the program operations onto the processing element.

Different restricted versions of program decomposition and processor allocation schemes for dataflow programs have been proposed, [2], [3], [4], [5]. These proposed algorithms, however, are all limited in several respects which make them inapplicable to an actual dataflow system. These defficiencies are attributable to one or more assumptions or restrictions, such as:

(a) The communication cost negligible or constant,
(b) A uniform computation time for all operations,
(c) A limited set of common language constructs,
(d) Unlimited system resources such as the number of processors, and the memory sizes, and
(e) Suppressing concurrency at lower levels.

Our research attempts to develop a mapping scheme which provides a reasonably good solution to the main problem, and at the same time avoids the above mentioned restrictions. Our eventual goal is to establish algorithms which are implementable for a large class of realistic dataflow machines.

THE PROCESSOR ALLOCATION PROBLEM AND SOLUTION STRATEGIES

The general architecture studied is that of a machine consisting of a large number of small asynchronously operating processing elements which can communicate with one another. Each processor in the system is capable of accepting a task generated by the program, performing the indicated operations, producing partial results, and transmitting these results to the other processing elements in the system. Figure 1 and 2 show the general architecture of the dataflow multiprocessor system.

Dataflow programs are directly translatable into directed graphs because these simple structures are intuitively simple to understand and directly convey the program semantics. Such a program is partitioned into parts which are distributed to and stored in the processing elements. The intermediate results of these partitions of the program are retained in the local memory. Since the amount of communication may vary considerably from one pair of nodes to another, the way in which the nodes of the dataflow graph are allocated to processors can change the overall processing time of the program dramatically.

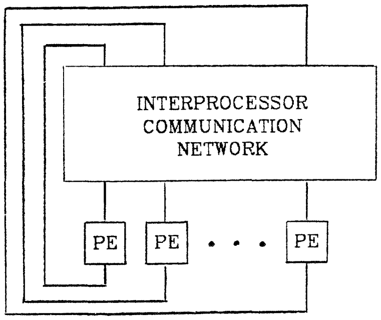


Fig. 1 Architecture of Dataflow Multiprocessing System

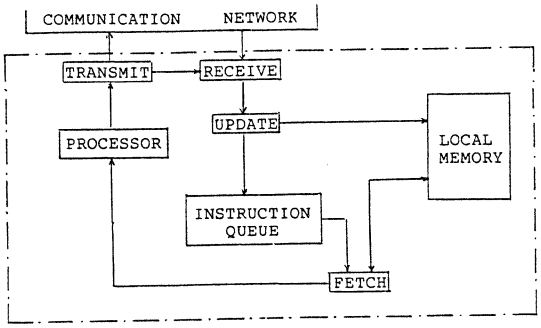


Fig. 2 A Processing Element

Minimizing the interprocessor communication time and balancing processor loads are the two factors that influence the program partition strategy for optimal system performance. It is clear that they are also two conflicting factors. While the total execution time can be minimized by distributing the dataflow nodes on all the available processors, the total communication time is minimized by concentrating the nodes in as few processors as possible.

The algorithm we propose maps the program nodes onto the processing elements by simulating a run time allocation environment during the compile phase. There are many criteria which may be used to decide how a node is to be allocated to a processor. Some of these are: length of time an enabled node has been waiting for a processor; the number of output edges of the node; the length of time required for the primitive operation represented by the node; the position of the node in the graph (as represented, for example, by the length of its longest output path); the amount of data generated by the node (which is directly related to the communication cost), or the number of data items at the input of the node. Besides these, there are other more complex criteria which may lead to more accurate estimation of the execution priority of each node. We have not

0190-3918/83/0000/0338$01.00 © 1983 IEEE

considered them because we want the algorithm to have reasonable complexity.

SUMMARY OF THE ALGORITHM

The dataflow program to be partitioned is translated into a dataflow graph whose nodes represent the operators and edges correspond to the data dependencies among the instructions. Every edge is regarded as a channel through which data items carrying values from the output of a node to an input of another node. In order that an instruction be enabled as soon as its operands are available, the data dependencies of a dataflow graph must always be exactly the same as the sequencing constraints. Hence, applicative languages that obey the single assignment rule have been the basis of most of the proposed dataflow languages, [1]. In this paper, we too are assuming the use of such a language. Since the algorithm attempts to simulate a run-time status of the program, any node that is enabled for execution is also enabled for allocation. The following information is required by the algorithm and hence must be known in advance:

(1) The number of available processing elements, and the processor type of each.

(2) A table of interprocessor communication (IPC) times. This is necessary in order to take the cost of communication between processors into consideration. The IPC reflects the topology of the interprocessor communication network.

(3) The expected execution time of each instruction. Each of these values is a constant throughout the program execution. The only exception may be the I/O operations which are non-deterministic in nature. The execution time of an I/O operation is modelled by a probability distribution function.

(4) Each individual procedure, iterative loop, and block structure has a unique code block name which is associated with every node within it so that nodes belonging to the same code block can be identified.

The mapping of the nodes of a dataflow graph onto the memories of the processing elements is accomplished in a single pass through the program graph. The algorithm maintains several pieces of data:

(1) For each node n_i, there is a token counter, TC, which indicates the number of tokens still needed before the node is enabled. Initially, $TC(n_i)$ is set equal to the indegree of n_i. This TC is decremented every time a token is supplied to the node through an edge that has no token on it.

(2) For each node n_i, there is an enabled time $t_e(n_i)$. When the node is put in the set of firable nodes, this time is used to determine the order in which a node can be allocated to a processor. When the node is not yet in the set of firable nodes, the $t_e(n_i)$ is updated each time $TC(n_i)$ is decremented to a value equal to the time a token is supplied to the node. Initially, $t_e(n_i) = 0$.

(3) There is a set of firable nodes, F, whose elements are ordered pairs $(n_i, t_e(n_i))$. A node is inserted in this set if and only if its corresponding TC has a value of 0.

(4) For each processor, there is a ready time, t_r. The value of $t_r(p_i)$ indicates the time at which the processor p_i will be idle next. Initially, $t_r(p_i) = 0$.

The algorithm can be analyzed by considering the information available at each node as it is enabled for allocation. Since each code block has its name, nodes within the block can be uniquely identified. The algorithm starts by traversing the graph from its outermost block. As soon as a node (say, n_i) is enabled, which will be indicated by its token counter $TC(n_i)$ going to zero, that node is assigned to the "best suited" processor. The question being asked at this point is: which processor can start the execution of the node n_i earliest if it were to be allocated to this processor? The answer involves checking the next ready time t_r, of each processor while taking into consideration the communication time between the target processor and the processors to which the predecessor nodes of n_i have been assigned. This allocation step is completed by updating the system status change caused by the allocation.

When a node with a different code block name is encountered, the algorithm is applied recursively to allocate this inner block. This recursive step eventually ends in the innermost block and completes the allocation as described above. The resulting schedule is passed back to the embedding block. Processors are then assigned to these schedules while preserving their relative start times.

In order to handle control structures such as a procedure call, a conditional construct or an iterative construct, special procedures are developed to reflect the effects of these various structures. These procedures, as well as the criteria used for tie-breaking situation when more than one node are available for allocation are summarized as follows:

Note 1: Iterative constructs: the loop body is allocated as described above to a set of processors. The resulting schedule of each of these processor is treated as if it were the schedule of a "single" node. The starting time of these "nodes" are compared to obtain the earliest starting time among themselves. Similarly, the latest finishing time can be determined. The difference between these two time values is used as the execution time of each of these "nodes" and is multiplied by the expected number of iterations. This parameter is the final schedule for these processors. In the case when the number of iterations is a run time variable, a value is obtained from a probability distribution function.

Note 2: Conditional constructs: After the allocation of each block that corresponds to a condition, and for each processor which has been utilized in more than one of these blocks, the schedules of each block for this processor are compared and the one representing the "worst case" execution time is used. This step is necessary in order to guarantee that sufficient processing time is reserved regardless of the condition selected during run time.

Note 3: Procedure Call: The procedure, which is essentially a dataflow graph itself, is allocated as a code block. The algorithm maps the variables used and created in the procedure to those passed and returned in the procedure call.

Note 4: The tie-breaking criteria used when more than one node are simultaneously enabled for allocation is the *Length of the Longest Output Path, (LLOP)*. Each node contributes a value of 1 to the length of the path in which it occurs except for the procedure node which contributes a value equal to the length of the longest path in the graph of the procedure called by the node. Clearly, since the program graph allows iterative constructs, which may be executed a number of times, the length of the longest output path is not well defined. In this case, the longest output path obtained by traversing the loop once is multiplied by the expected numeber of iterations of the loop. Another criteria which has been considered is the largest outdegree of the successors of a node, (LOD). It is used as a guidline because a node with a larger outdegree has a greater potential for enabling the firing of other nodes. As we shall discussed in the next section, results obtained by simulation indicate that LLOP yields a better performance than LOD in most cases. Therefore, perference is given to LLOP as the tie-breaking criteria in our algorithm.

RESULTS AND COMPARISONS

In order to evaluate the performance of the algorithm, simulation runs were made on data consisting of randomly generated program graphs. Programs with 10 to 100 nodes, 1 to 5 levels of code blocks in a 3 to 8 processor system were simulated. The number of nodes in the graphs, the interprocessor communication costs, the individual node execution costs, and the number of active processors were all generated from uniform random distribution. The table below shows the distribution of the ratio R of execution time for the algorithm using two different "tie-breaking" measures (LLOP and LOD) to the optimal execution time.

Summary of simulation results				
	$R=1$	$1<R<11/10$	$11/10<R<5/4$	$R>5/4$
LLOP	*74%*	*18%*	*8%*	*0%*
LOD	*65%*	*19%*	*11%*	*5%*

The most important property of the algorithm is its effect on the execution time of the program graph. A good algorithm will attempt to minimize the execution time regardless of the number of available processors. Figure 3 and 4 show the total execution time of 2 different dataflow programs as a function of the number of processors for our algorithm and for the ones reported in [4]. The two programs chosen are a trapezoidal quadrature program and a matrix multiplication program. The two appraoches of the algorithm in [4] are identified as MLC1 and MLC2. The two versions of our algorithm are identified as LLOP and LOD. The best result is obtained using the LLOP version of the proposed algorithm in both the programs. While the algorithm MLC2 gives a shorter execution time than the LOD version of our algorithm when there is a small number of processors increases.

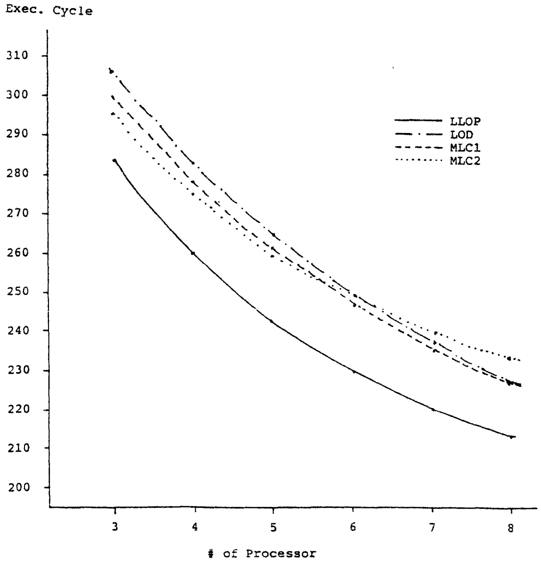


Fig. 3 Algorithm Performance with the Trapezoidal Program

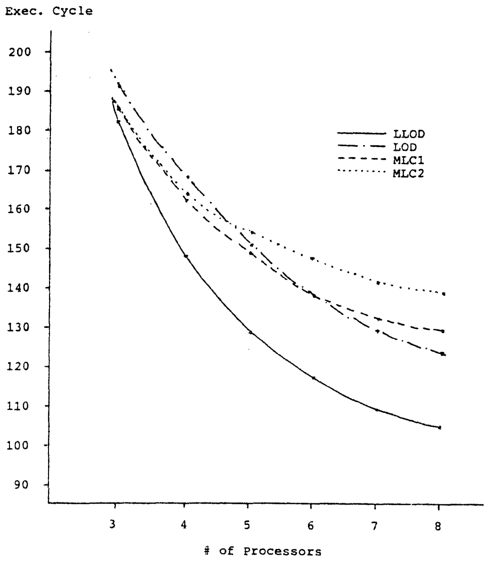


Fig. 4 Algorithm Performance with the Matrix Multiplication Program

The effect is even more pronounced in the matrix multiplication program. The reason for this can be attributed to the fact that each node in both MLC1 and MLC2 algorithms represents a complete statement and hence the inherent parallelism of the statement is not taken advantage of.

CONCLUSION

The problem of decomposing and mapping a dataflow program graph onto a set of processing elements is discussed. An algorithm has been developed for achieving efficient program execution while maintaining a high degree of concurrency. It is summarized in this paper. Research is proceeding for the investigation of the effect on the algorithm of putting limiataions on several system resources. These include a finite memory size for each processing element, a communication network that is not totally connected, and a set of nonhomogeneous processors each of which is characterized by the functions it performs.

REFERENCES

[1] Ackerman, W.B., "Dataflow Languages", AFIPS Conf. Proc., Vol. 48, 1979 NCC, New York, p.1087-1095, June 1979.

[2] Arvind, "Decomposing a Program for Multiple Processors System", Proc. Int'l. Conf. on Parallel Processing, p.7-14, 1980.

[3] Davis, A.L., "A Dataflow Evaluation System based on the Concept of Recursive Locality", Proc. AFIPS NCC, Vol. 48, p. 1079-1086, 1979.

[4] Mundell, K.J., Linder,M.W. and Conry, S.E., "Processor Allocation in Data-Driven Systems -- Two Approaches", Proc. Int'l. Conf. on Parallel Processing, p.156-157, 1981.

[5] Nelson, E.C., "Resource Allocation for Free Running and Resource Limited Program Graph", Ph.D. Thesis, Department of Computer Science, Stanford Unviersity, CA., 1973.

A Small, High-Speed Dataflow Processor

Wm Leler
Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina 27514

Abstract

Dataflow processors show much promise for high-speed computation at reasonable cost, but they are not without problems. This short paper will discuss a processor design which combines ideas from dynamic dataflow architecture with those from reduced instruction set computers and proven large computers with parallel internal structures. The resulting processor includes a number of innovations, including *operand destinations*, *killer tokens*, *I/O streams* and *closed-loop computation*, which result in a small, relatively inexpensive processor capable of high-speed computation. The expected application areas of the processor include interactive computer graphics, signal processing, and artificial intelligence.

1. Introduction

Many new architectural proposals seem to assume the need for a new architecture. As a result, these designs end up as "solutions in search of a problem." This design started from the opposite viewpoint. We already had a problem -- the need for quick, easy to program interactive graphics. Its characteristics are common: the demand for fast computation at reasonable cost; use of algorithms that not only offer high degrees of parallelism, but are difficult to code sequentially; and a good fit into a message passing paradigm. The solution will involve some sort of parallel processor.

Note that the expected application area was used to guide the design decisions, not to cripple the machine into special purpose oblivion. An example of this sort of decision is the need to keep the processor as simple as possible; most graphics applications simply cannot afford a large, expensive processor. Another design decision was to avoid global bottlenecks such as centralized schedulers. As the design progressed it was decided to ignore questions of purity; issues such as whether it is possible to construct indeterminate graphs are best dealt with at the language level.

2. Dynamic dataflow

The architecture chosen was the dynamic dataflow architecture [Tr82] first proposed by Arvind. The dataflow machine running at Manchester is an example of this type of architecture, however, there are still a number of problems to be solved [Ga82]. These problems can be broken down into three classes: problems with *all* parallel computers; problems with *dataflow* computers; and problems with *current dynamic dataflow* designs.

An example of the first type of problem is processor scheduling. On any multiple processor machine individual instructions must be scheduled to specific processing elements for execution. If this scheduling is done at any time other than run time there is a danger that some elements will be idle while others are swamped. In order to solve this problem the scheduling must be done at run time, but this requires global knowledge of the dynamic state of the system. Any global resource scheduler, however, will become a bottleneck on a multiple processor machine.

An example of the second type of problem is the handling of data structures. The data storage of a conventional computer is arranged as a large array, so arrays have become the dominant data structure used in computer science. In order to implement randomly updatable arrays on a dataflow machine, either the entire array must be passed around as the value of a token (i.e., array copying), or else it must be stored as a tree with the accompanying overhead on access. Dataflow machines are able to handle alternative data structures such as streams. It may be that our heavy dependence on arrays is an artifact of current practice, and other data structures might also be sufficiently powerful and efficient.

Any new type of computer will have the third type of problem. For example, early von Neumann computers did not have index registers. This meant that address manipulation had to be done with self modifying code and other tricks, until someone thought up a better way to do it. Most of the design innovations in this paper are aimed at the third class of problem, but the first two types of problems will also be addressed. These problems include the following:

- Conditional instructions affect data flow and not control flow. All values circulating in a loop, even loop constants, must pass through branch instructions. This degrades execution time as well as code density.
- A further problem with dynamic dataflow designs is that all values circulating in a loop must have their tags updated.
- Conventional processors can take advantage of program locality by holding data and status in registers. The tradeoff is that this information must be saved during a context switch. Dataflow processors have no overhead on context switches, but as a result have difficulty utilizing program locality.
- Functional architectures such as dataflow have conceptual problems with nondeterminate and history sensitive operations, especially I/O operations.
- In many parallel processor designs there is much duplication of hardware, or hardware that cannot be utilized concurrently.
- Most proposed parallel processors perform poorly under low degrees of parallelism. One of the strengths of the Cray-1 was that it performed scalar operations reasonably well, not just vector computations.
- Some parallel processors depend upon massive amounts of parallelism, in the thousands or even millions, to realize their full performance. By studying the flow graphs from optimizing compilers we can see that most programs tend to have a parallelism of between 30 and 100 [TI80] [Fi82]. Impressive exceptions do exist, but they too will have bottlenecks where the parallelism is not quite as impressive. It is not even clear whether a computer with millions or even thousands of processing elements could be built. Certainly it would be difficult to keep such a system running for any length of time.
- In a parallel processor, if one process is producing values to be consumed by another process, the producer can get far ahead of the consumer and fill up the interprocessor queues unless some sort of handshaking is used between the processes or processors.
- As discussed earlier, there are problems with instruction scheduling and data structures.

0190-3918/83/0000/0341$01.00 © 1983 IEEE

3. Processor Structure

The processor is divided into a number of separate processing elements connected through a network. In order to avoid the duplication of hardware, and to keep the processing elements simple, different elements of the processor may have different instruction sets. Currently there are five types of processing elements (see below), although more could be added at any time. A processor can contain as many as 32 elements, but there must be at least one of each type.

Instructions can take either one or two operands, and those that take two require a matching buffer. These buffers are expensive, so on this processor the processing elements are divided into those whose instructions take one operand, and those that take two. This allows the matching buffers to be eliminated on elements that only execute monadic instructions, as well as the field in a token that indicates the number of operands required by its instruction.

Each processing element then consists of two or three stages: an optional matching buffer, an instruction fetch stage, and an execution unit. The instruction fetch section fetches the appropriate instruction from a conventional memory, and the execution unit performs the desired operation, and updates the output tokens.

4. Tokens

32	14	5	12	1
DATA	IADDR	ITER	INVN	P

Tokens consist of a DATA field and a tag, both of which are 32 bits long. The tag field is further divided into several fields: the IADDR field gives the address of the destination instruction; the ITER field is used to keep up to 32 active iterations of a loop separate; the INVN field is used to specify which invocation of a subroutine this token is from; and the P (or PORT) field tells two operand instructions whether this is the left or right argument, or for one operand instructions it can indicate the end of a stream. Matching buffers match tokens on all bits of the tag except for the PORT bit.

A unique feature of this processor is the use of *killer tokens* to remove tokens from the matching buffers. Killer tokens can be used to increase parallelism in loops, perform elementary stream operations, and form the basis for non-determinate computation. For all this utility they are extremely easy to implement. A killer token is a token that has the same tag as the token it is to kill, and can be created by any instruction. If two tokens match in a matching buffer and their PORT bits are the same, then both are discarded. This situation corresponds to the illegal condition of having, say, two right operands for the same instruction, and so does not require any extra bits. Even so, it is very general, since killer tokens can arrive before the token to be killed, and killer tokens can even kill other killer tokens. The uses of killer tokens will be discussed later.

5. Instructions

An instruction consists of a 32 bit operation phrase (opcode plus literals) followed by one or more 32 bit destination phrases. The ability to have an arbitrary number of destinations eliminates the need for explicit copy instructions, as well as the waste when space is provided for a fixed number of destinations and fewer are needed.

Destination phrases on this processor are different from those in other dataflow designs in that they include fields to control data branching, token relabeling, output data source, and token priority, in addition to the fields containing routing information for the new token. A destination phrase consists of the following fields:

field	*bits*	*contents*
LAST	1	Since there can be an arbitrary number of destination phrases, the last destination has this bit set, otherwise it is cleared.
SOURCE	2	The source of the DATA field for the outgoing token can be either of up to two results of an operation (i.e., low order or high order product for multiplication), or either of the operands. For single operand instructions, the data source can be the single operand, or one of up to three results.
SENSE	2	An operation may have a condition associated with it. This field tells the processing element whether to send this token out: regardless of the condition; only if the condition is true; only if the condition is false; or only if the processor is in debug mode.
INCR	5	Tells how much to increment the ITER field of the token before sending it out. This is used to send values from one iteration of a loop to another.
SMODE	1	If this bit is clear, the addition to the ITER field by the INCR field is done modulo 32. If set, overflow of the ITER field increments the INVN field.
PE	5	Processing element number to send this token to.
IADDR	14	New destination instruction address.
PORT	1	New PORT bit. Invert for a killer token.
PRIO	1	If set, then this token has priority in the communication network.

6. The Processing Elements

6.1 Arithmetic Processing Element. The arithmetic processing element does simple arithmetic and logical operations such as adds, exclusive ors, comparisons, and shifts. Combined with the SENSE field these instructions can be *fancy* loop closers in that arithmetic, test, and branch operations are combined in the same instruction.

6.2 Multiply and Divide Processing Element. This processing element is based on some monolithic multiplication chip, with a standard iterative algorithm used for division.

6.3 Constant Element. Constants too big to fit in the operation phrase of the arithmetic element, as well as constants for other operations must be generated explicitly by this monadic processing element. It also has the ability to do lookups into tables of constants, which can be used for sine and cosine tables, for example. This element can also be used to watch for the last token of a stream.

6.4 Context Element. The context element executes instructions which manipulate the INVN field of a token. It is thus used for subroutine calls and returns, but, unlike conventional machines, instructions are not associated with any control flow, they are associated only with the parameters passed and returned.

6.5 Input/Output Element. The interface between the history sensitive outside world and this applicative processor is done by the I/O element using streams. The I/O processor contains a block of random access memory called the buffer memory. This buffer memory is mapped into the address space of the host, or if there is no host, it can be used as I/O buffers for DMA I/O devices such as disks, tapes, or frame buffers. Streams of data are written into and read out of the buffer memory.

7. Operand Destinations and Conditions

This processor uses the SOURCE and INCR fields to avoid passing values circulating in a loop through branch or tag update instructions. The INCR field updates the ITER field to send its destination to the next iteration of a loop. Loop constants can be specified as operand destinations using the SOURCE field to automatically send the constant to the next iteration. Unfortunately, this constant will be sent to the next iteration regardless of whether that iteration will ever occur, so there will be a token left over. A loop must have at least one branch instruction, and it can be used to generate a killer token for the leftover constant. As a result, only a single branch instruction is needed for any loop, just as in conventional architectures. Using the SENSE field very compact loops can result. For example, an iterative factorial program on this processor is two instructions, a multiply and a decrement and branch.

8. Streams

One of the more popular features of the UNIX operating system [Ri78] is the ability to make a connection between the input and output of two concurrently running processes using a mechanism called a *pipe*. This allows concurrent computation, although on a uniprocessor the concurrency is simulated using an I/O buffer and multiprogramming. Streams on this processor perform a similar function, but since there is no extra overhead associated with process switching processes can be made much smaller, and streams become much more pervasive. Even loops can be considered as processes operating on streams of data, and so streams become the major data structuring facility of this processor.

Streams can also be used to do I/O concurrently with processing. In order to keep read operations from generating too many tokens and overflowing the ITER field, all operations are done on a closed-loop basis. Write operations generate a dummy result token that can be sent to a read operation to enable the next read. This scheme works in much the same way as acknowledgement tokens on static dataflow processors, except that they are explicit in the program, and are used around entire computations, not for every instruction.

Large arrays of data, such as pixel arrays in a frame buffer, can be read and written using streams. To do reads from a frame buffer, a stream of pixel addresses is sent to the frame buffer, and a stream of pixel values is returned. If read and write operations to the frame buffer are mixed then it is up to the user (or the compiler) to insure that there are explicit data dependencies to keep proper order to the operations. Since all operations are done on a closed-loop basis, this is fairly easy to guarantee. This method can be used for any array of data, not just pixels, using the buffer memory in the I/O processor.

9. Resource Control

Functional architectures usually have problems with history sensitive and non-determinate computations. Most history sensitive computations can be handled using streams. For example, if a resource is being requested by a stream of incoming requests, but only one of them is allowed access at a time, the requests can be made to queue up in a matching buffer of a two operand element. When the present request is finished using the resource, the requisite matching token for the next request is sent to the matching buffer.

This mechanism is not always enough; consider the situation where a request for a resource has been made, but the process requesting the resource changes its mind and wants to abort the request. This is done using killer tokens. The process sends two tokens containing null requests to the resource request queue. If the request has already been granted access to the resource, then the two tokens will act as killer tokens and kill each other. If not, the first token to arrive will act as a killer token for the original request, and the second will wait for the request acknowledgement token, and then perform the null operation. In either case, the resource will correctly enable the next request, and will send back some result (either the original one or a null), which the requesting process can ignore.

10. Program Locality and Parallelism

The PRIO bit in a destination is used to indicate computations that are on the critical path. This is used to give a token priority in the communications network. The matching buffers also contain a small set of high-speed registers, much like registers on a conventional processor. The PRIO bit is used to indicate tokens which are given priority in these registers. If a token enters the matching buffer stage of a processing element and the instruction fetch section is free, then an instruction prefetch can be initiated using the IADDR field of the incoming token. If the match succeeds, then the instruction is available immediately. These features improve performance under a low degree of parallelism, and allow the processor to take advantage of program locality. This results in a processor which does not depend on large amounts of parallelism for impressive performance.

11. Supervisory Functions

INVN value zero is reserved for supervisory functions such as loading the instruction memories (there is also an I/O instruction to do this), clearing the matching buffers, and monitoring processing element load and utilization of matching buffer space. By monitoring system load, the operating system can shift instructions around when imbalances occur, while avoiding the global bottleneck of an instruction scheduler. The debug mode of the SENSE field can be used to count the number of times an instruction or loop is executed, print intermediate values, and other functions useful in debugging.

12. Conclusions and Status

This paper presents a dataflow design that has been evolving over the last several years [Le81]. It offers new solutions to many of the problems associated with dataflow processors. The design is currently running on a simulator which has been used to verify that the schemes proposed here actually work and can be used in small programs. To the best of my knowledge, the following features are unique to this processor:

- Non-homogeneity of the processor. Increases modularity, eliminates matching buffers in some elements, ALU's in others, simplifies control and instruction decoding.
- Operand destinations with automatic token relabeling. Eliminates extra branch and relabeling instructions. Only explicit relabeling instructions are for parameters passed to subroutines.
- Killer tokens. Clean up after operand destinations, perform stream operations, help non-determinate computations.
- Loop closing instructions for fast execution.
- Ability to take advantage of program locality and give priority to computations on the critical path.
- Instruction prefetching and matching buffer registers. Improves performance when parallelism is low.
- Arbitrary number of destinations per instruction. Eliminates need for copy instructions.
- Elimination of explicit branch instructions by associating a condition with most operations.
- Ability to stream I/O directly from peripheral devices such as disks and tapes with no host intervention.
- Computations are done on a closed-loop basis, which keeps producer processes from racing too far ahead of their consumers.

In my opinion, the problems plaguing current dataflow designs will turn out to be no more severe than the problems that plagued early von Neumann computers. This paper proposed solutions to some of these problems, and solutions to other problems will undoubtably be discovered as more dataflow processors are built and experience with them is gained. One way to help make sure that dataflow processors will be built is to design them for application areas where traditional architectures are weak. This processor is aimed at such areas.

13. References

[Fi82] Personal communications with J. Fisher, 1982; also A. Nicolau and J. Fisher, "Using an Oracle to Measure Potential Parallelism in Single Instruction Stream Programs," *Micro 81*, pp. 171-182.

[Ga82] D. Gajski, D. Padua, D. Kuck and R. Kuhn, "A Second Opinion on Dataflow Machines and Languages," *IEEE Computer*, (Feb. 1982), pp. 58-69.

[Le81] W. Leler, *A High-Speed Dataflow Architecture*, Department of Computer Science, University of North Carolina at Chapel Hill, (Dec. 1981) 26 pp.

[Ri78] D. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal*, (July-August 1978, part 2).

[Ti80] Research done at Texas Instruments on the ASC Compiler; also as reported in course *Data Flow Concepts in Computer Language and Architecture*, MIT, (June 1980).

[Tr82] P. Treleaven, D. Brownbridge and R. Hopkins, "Data-Driven and Demand-Driven Computer Architectures," *Computing Surveys*, (March 1982), pp. 93-143.

PROGRAMMABLE MODULAR SIGNAL PROCESSOR — A DATA FLOW COMPUTER SYSTEM FOR REAL TIME SIGNAL PROCESSING

Prashant S. Sawkar, Timothy J. Forquer, and Richard P. Perry
RCA Government Systems Division
Moorestown, New Jersey 08057

Abstract

Classical signal processor design techniques are very expensive, time consuming, and result in a custom hardware and software that may not be capable of meeting the wide ranging requirements of signal processing applications other than the one for which it was intended.

This paper presents an organization of a Programmable Modular Signal Processor. A data flow concept of control is advocated in order to take advantage of the run-time parallelism inherent in most applications. The attractive features of the system are that it is capable of being realized with a small number of hardware functional units, and it allows the hardware to be independent of the signal processing application. The system is organized to support a high-level graph-oriented signal processing application description capability in order to simplify the user interface.

INTRODUCTION

Many areas of scientific computation — aerodynamic simulation, weather forecasting, real-time radar signal processing — have immense computational requirements with instruction execution rates greater than 2 x 10^9 operations per second. Conventional computer structures are unable to meet the demand because they cannot exploit the high degree of parallelism exhibited by most applications.

Considerable research has been done in parallel computer structures and their suitability for applications exhibiting high degree of concurrency. The two basic structures are parallel array and the pipeline. The parallel arrays (array processors, such as ILLIAC IV, and multiprocessors, such as C.mmp) have various problems such as job partitioning and memory conflicts. There is overwhelming evidence that only a fraction of their potential can be realized for a broad spectrum of applications [1,2,3]. The pipeline structures (IBM 360/91, CDC-STAR, CRAY1), although capable of high performance, are prone to extremely low performance when the pipeline flow is disrupted by branching, non-availability of data, or interdependence between stages.

The data flow concept promotes a fundamentally different way of executing instructions — an alternative to sequential program execution in conventional computer structures. In a data flow machine, an instruction is ready for execution when its operands are available. No control flow is indicated either implicitly or explicitly and there are no program counters. The concept of data-activated instruction execution allows multiple instructions in a data flow program to be executed concurrently. This expression of parallelism in terms of data dependencies, rather than in spite of them, leads to a far more natural and flexible picture of parallel program execution [4].

In view of the nature of the parallel hardware structures and the practical difficulties encountered in running them at full speed, it seems likely that the data flow architectures, with their fundamentally different approach to instruction execution, have a better chance of exploiting the high degree of parallelism inherent in most applications.

In this paper a data flow architecture for a Programmable Modular Signal Processor (PMSP) machine is presented. The PMSP system is being designed for solving a broad range of real-time signal processing applications, with particular emphasis to radar signal processing. The architecture is developed by repeated use of a small number of basic building blocks, which are:

1. A Multiport Memory System (MMS) that features concurrent access to memory at its multiple data ports, high throughput per port, and a programmable interconnection mechanism.
2. A General Processing Element (GPE) that has scalar/ vector arithmetic processing capability, matrix arithmetic, transforms, and function generation capability.
3. An Input/Output Processing Element (IOPE) to support all the input/output interfaces.

These building blocks can support multiprocessor systems that have very high throughput and which are reliable and dynamically reconfigurable. The MMS, GPE and IOPE are coupled with a data flow control mechanism to provide the PMSP architecture with characteristics needed to support real-time radar signal processing applications.

DATA FLOW ARCHITECTURES AND PMSP SYSTEM ORGANIZATION

The objective of this section is to introduce the basic data flow architecture and identify the modifications meeded to support a wide range of real-time signal processing applications.

A number of data flow architectures have been proposed [5-9]. We have chosen the Manchester data flow model proposed by Gurd and Watson [8] to illustrate the approach and the modifications needed. The Manchester data flow machine has five units interconnected (see Figure 1). The node store contains node descriptions, which specify the operation to be performed, and the node store address to which the results of the node execution must be directed. Data values circulate around the ring as tokens that consist of a data value and the address in the node store. When a token or a pair of tokens arrive at the node store, the corresponding instruction (node description) is fetched from the node store. The node description (instruction) when coupled with its tokens (operands) becomes an executable instruction that is routed to the processing subsystem.

The matching store is an associative storage mechanism in which tokens directed to the same node are grouped together. The token queue is a buffer between the switch and the matching unit for equalizing disparities in the rate of production and consumption of tokens. The switch provides a means for communicating with the external world.

The data flow mechanism may be viewed as a circular pipeline that carries tokens. The degree of concurrency possible is limited by two factors: 1) the number of processing elements in the processor subsystem and the degree of pipelining within each unit,

0190-3918/83/0000/0344$01.00 © 1983 IEEE

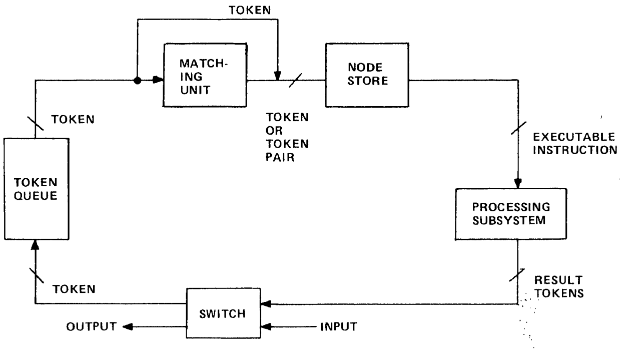


Figure 1. Basic Manchester Data-flow Architecture

and 2) the capacity of the data paths connecting various subsystems in the ring.

The suitability of a data flow model for radar signal processing applications has been studied [10]. The benefits of simultaneity in an irregular, and run-time dependent data flow situation, and also the software engineering benefits when several complex functions are to be executed in parallel are very attractive [11]. A radar signal processing application is typified by very high data arrival rates, high computational rates on large volumes of data, high reliability, and fault tolerance. In order to accommodate radar signal processing applications in data flow, a number of modifications are necessary.

The first modification proposed is a higher level of data flow (macro data flow), which would synchronize operations at the functional level. The functions may be complex tasks on large vectors of data. The data flow concept described earlier, in contrast, synchronizes operations at the arithmetic level. This modification can minimize the token flow traffic on the circular pipeline of the data flow machine.

Second, the tokens in the macro data flow model would consist of a pointer to a value or set of values and also an address of the node store, instead of the conventional tokens, which carry a label, a value, and the address of the node store. By making this change, it would be possible to avoid saturating the data paths with data tokens (operands) intended for a single instruction. The time taken to route an instruction from node store to a processor in the processing subsystem would be quite small in comparison. Hence, the unused capacity of the data paths could be used to route more instructions to improve the concurrency to a level acceptable for real-time applications.

Finally, the processing elements in the processing subsystem should be capable of executing a variety of complex tasks. The architecture of these processing elements, however, need not be data-flow oriented. Any of the conventional parallel structures — array, pipeline, or hybrid, for example — can be chosen in order to efficiently execute the tasks because the concurrency achieved by data flow at this level is offset by communications and control overhead. A complete discussion of data flow performance under various conditions is available in Reference [10].

PMSP Data Flow Processor (PDFP) Architecture

The PMSP Data Flow Processor consists of five subsystems (see Figure 2): a Task Data Base, a Task Dispatcher, a Processor Subsystem, an Input/Output Subsystem, and a Task Scheduler. The five subsystems are interconnected in a ring; the I/O Subsystem and the Processor Subsystem also interact with a multiport memory system.

The task data base provides a central storage media for various tables describing graph nodes, node tasks, and the interconnectivity of the nodes. The data base also maintains status and capability lists of the various resources in the Processing and I/O Subsystems. A Task Ready List indicating the order in which the tasks became ready to execute is also maintained in the task data base.

The run-time function of the task dispatcher is to select the next task to be run from the Task Ready List, to select a processor from the I/O Subsystem or the Processor Subsystem upon which to execute the task, and to dispatch the corresponding task packet to the selected processor. The packet consists of a task identification followed by parameter, input, and output parameter information. The task dispatcher broadcasts the task packet on the Dispatcher Bus (DBUS). The task dispatcher also interfaces with the host, for purposes of statically loading the task data base with an application, and runtime control of the database by the host.

The Processor Subsystem consists of a number of General Processing Elements (GPEs), as shown in Figure 2b. The Dispatcher may select one of the GPEs to receive a task packet over the DBUS. The GPE retrieves the input operands from the Multiport Memory System (MMS) from the locations indicated by the task packet, performs the specified operation, and returns the data (output operands) generated to the MMS. (The GPE may have one or more ports to the MMS.) The GPE also generates a result packet consisting of a GPE-Identification, Task, and Task-status. The result packet is available to the Task Scheduler on the Scheduler Bus (SBUS), which is time multiplexed. The task scheduler generates the address of the next device that may place its result packet on the SBUS.

Each GPE is capable of executing a number of most commonly used signal processing algorithms or primitives, for example, FFT, MTI, and CROSS-ADD. A library of these packages exists at each GPE; there is no run-time loading of packages. It is possible to have each GPE execute every signal processing primitive or to have GPEs with different primitive execution capabilities. A capability list of each GPE is maintained in the task data base.

The Input/Output Subsystem consists of several I/O Processing Elements (IOPE), each of which has at least two I/O ports, one for communicating with the multiport memory system and the other for communicating with devices external to the PDFP. A variety of external devices, such as ADCs, DACs, IOPE of another PDFP, may be attached to an IOPE. The dispatcher may select one of the IOPEs to receive a I/O task packet over the DBUS. The selected IOPE decodes the task and performs the specified data transfer operation between the MMS and the external devices attached to the IOPE. Upon completion of the I/O task, the IOPE generates a result package consisting of an IOPE identification, Task, and Task-status. The result packet is available to the task scheduler on the SBUS. Upon initialization of the PDFP, some of the IOPEs are dedicated to bringing in external data and starting the data flow application resident in the task data base.

The task scheduler receives the result packets from the GPEs and IOPEs in a time-multiplexed fashion. From the result packet and the application graph tables residing in the task data base, the scheduler determines which nodes are ready for execution and places them at the back of the Task Ready List.

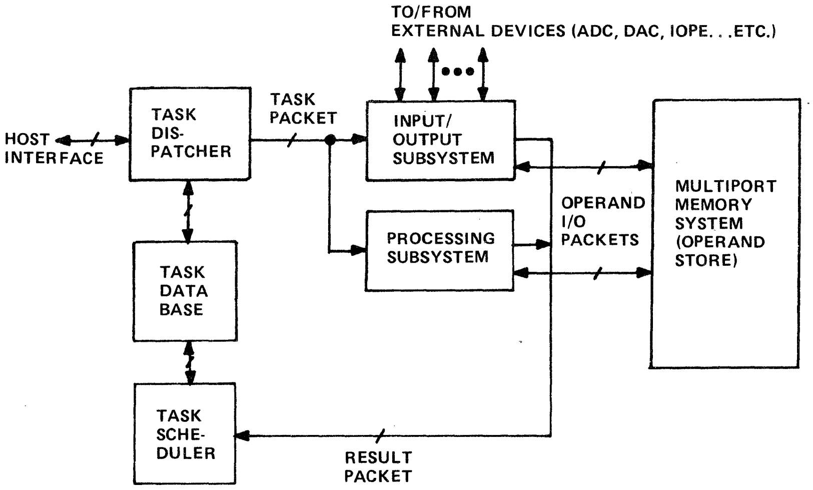


Figure 2a. PMSP Data Flow Processor Organization

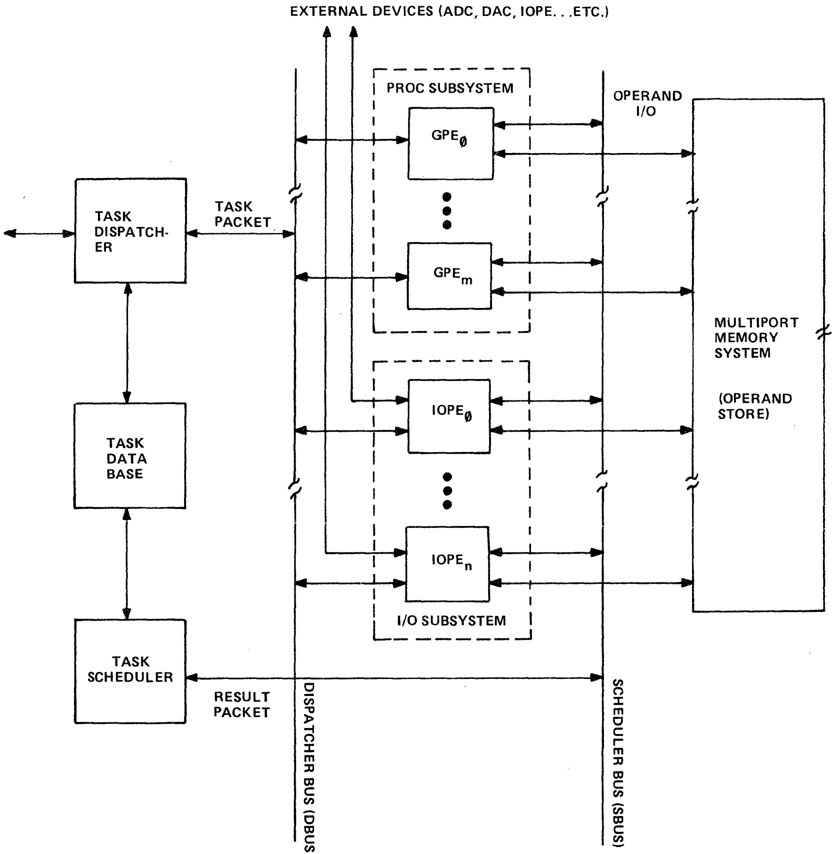


Figure 2b. PMSP Data Flow Processor Organizational Details

The Multiport Memory System supports contention-free parallel access to the bulk memory at its multiple ports. The MMS provides a programmable interconnection mechanism by supporting a set of high-level I/O commands at each port (see Table I) to interconnect bulk memory and an external device or any two external devices. The high speeds attainable at each port and programmable interconnection capability make MMS a useful tool in building reliable and adaptive multiprocessing systems for real-time applications.

The bulk memory used in MMS is block oriented: the smallest unit of data transferred between the MMS and an external device attached to it is one block. A block is transferred word-serially. The block size varies linearly with the MMS configuration; that is, an MMS with 16 ports has a block size of 16 words. The block size is not considered a limitation because the class of applications being solved by the PDFP require large volumes of data; therefore, an optimized block transfer approach, such as the one used by MMS, is preferred over other approaches. Details of MMS operation are beyond the scope of this paper and are available elsewhere [12].

TABLE I. MSS COMMAND SET

OPERATION	DEVICE-ADDR	BLOCK-COUNT	REMARKS
READ	BULK-MEMORY ADDRESS	#BLOCKS	READ SPECIFIED # OF BLOCKS FROM BULK MEMORY AND TRANSFER TO EXTERNAL DEVICE
WRITE	BULK-MEMORY ADDRESS	# BLOCKS	WRITE SPECIFIED # OF BLOCKS TO BULK-MEMORY. THE SOURCE OF THESE BLOCKS IS THE EXTERNAL DEVICE
SEND	PORT-ADDRESS	# BLOCKS	INTERPROCESSOR COMMUNICATION COMMANDS
RECEIVE	PORT-ADDRESS	# BLOCKS	

PMSP Data Flow Multiprocessor (PDFMP) Organization

The level of concurrency exploited by the PDFP may be increased enormously by connecting several PDFPs to form a multiprocessor system. The organization of a PMSP Data Flow Multiprocessor is illustrated in Figure 3. The PDFPs communicate with each other by way of their respective IOPs. The interconnection of the PDFPs may be made independent of application by using an MMS. For a small number of PDFPs, a direct interconnection of their respective IOPEs may be preferred.

The PDFPs are connected to the host by means of a host bus (HBUS). The user may use the graph description capabilities existing on the host to describe the various subgraphs that make up the application. The host assigns one subgraph to each PDFP and exercises overall control of the PDFPs.

PROGRAMMING SUPPORT AND OPERATING SYSTEM REQUIREMENTS

An application that can run on PDFP or PDFMP may be described as a data flow graph (see Figure 4). The nodes are the operators and the arcs are data queues. Operators may be large tasks, such as 1024 point FFT or matrix operation. The data queues are FIFO queues maintained in the Multiport Memory System. Each queue has a read amount (RA), a consume amount (CA), and a produce amount (PA). The produce amount determines the amount of data a node produces and places on its output queue. The read amount determines the amount of data needed from the queue for a node to start execution. The consume amount determines the exact amount of data consumed by the node. When a node has multiple input queues, it can start execution only when all the input queues have enough data on them to equal or exceed their corresponding read amounts.

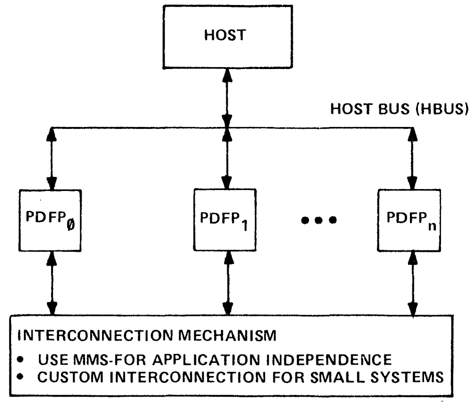

HOST
HOST BUS (HBUS)
$PDFP_{\emptyset}$
$PDFP_1$
•••
$PDFP_n$
INTERCONNECTION MECHANISM
• USE MMS-FOR APPLICATION INDEPENDENCE
• CUSTOM INTERCONNECTION FOR SMALL SYSTEMS

Figure 3. PMSP Data Flow Multiprocessor Organization

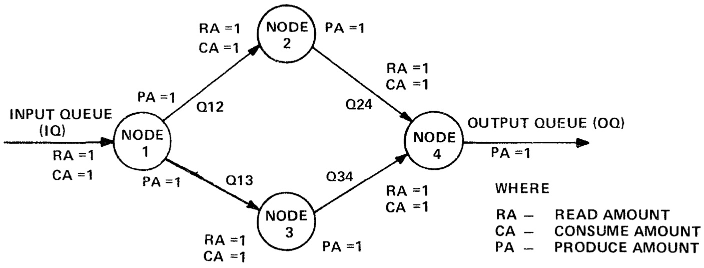


Figure 4. Example of Graphical Description of Application

The host system implements a PCOS interpreter [13], which accepts graph descriptions as a set of directives and generates an object graph. An object graph is simply a collection of data sets indicating the various nodes in the graph and their connectivity as well as various queues in the graph and their characteristics (see Figure 5).

Each GPE in the processing subsystem has an executive capable of decoding the task packet, fetching the input operands, and invoking the appropriate primitive routine to execute the task. Upon completion of the task it returns the output operands to the operand store and posts the result to the task scheduler. The most commonly encountered primitives may be coded and stored locally in its program memory.

In addition, it is possible to specify a combination of these primitives as a single task. For example, in the implementation of a radar MTI Process (Figure 6), a number of primitives (nodes) are interconnected as a subgraph to be executed by a GPE. Only the data produced by the peripheral nodes is transferred to the Multiport Memory System, and the data produced by the intermediate nodes is retained in the GPE for use as inputs by the subsequent nodes in the subgraph. This technique significantly reduces I/O, promotes better GPE performance, and reduces the data queue

NODE #	PRIMITIVE IDENT.	INPUT QUEUE POINTER LIST	OUTPUT QUEUE POINTER LIST	CONTROL VARIABLE POINTER LIST
1				
2				
•				
•				
•				
•				

a) NODE LIST

QUEUE #	QUEUE IDENT.	QUEUE DATA TYPE	READ AMOUNT	PRODUCE AMOUNT	CONSUME AMOUNT	BULK MEMORY POINTERS
1						
2						
•						
•						
•						
•						

b) QUEUE LIST

QUEUE #	PREDECES-SOR NODE #	SUCCESSOR NODE #
1		
2		
•		
•		
•		
•		

c) CONNECTIVITY LIST

GPE # \ PRIMITIVE #	1	2	•	•	•	•
1	YES	YES	NO			
2	NO	YES	•	•	•	•
•	•					
•						
•	•					
•						

d) GPE CAPABILITY LIST

NODE #	PRIMITIVE IDENT.	INPUT QUEUE INFORMATION	OUTPUT QUEUE INFORMATION

e) NODE READY LIST

Figure 5. Object Graph and Architecture Control Data

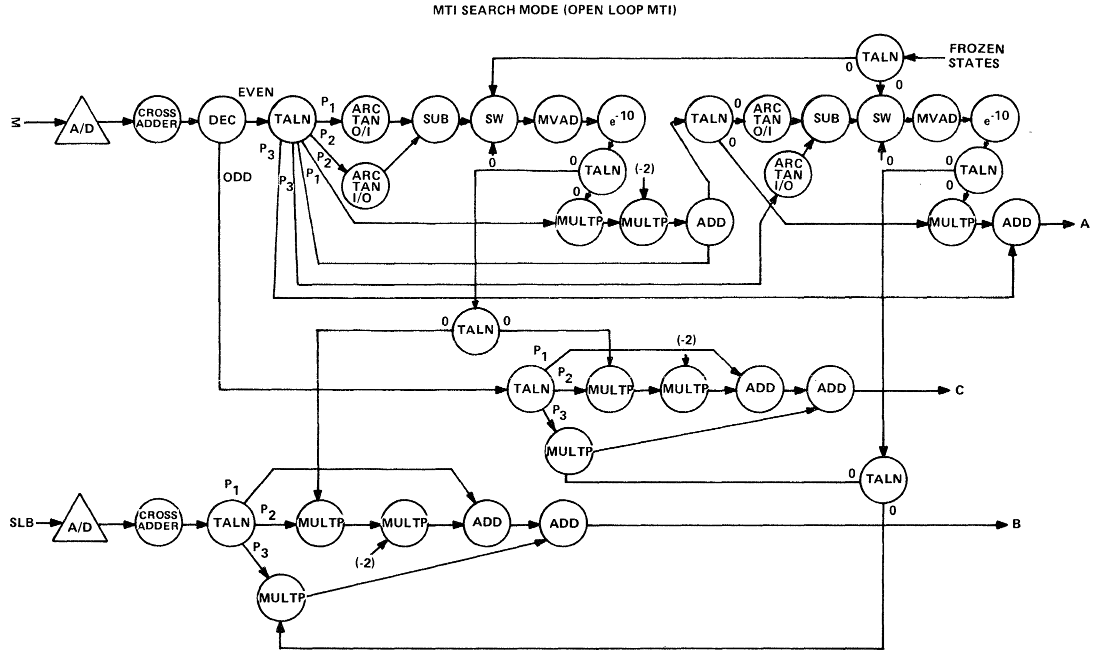


Figure 6. Subgraph of Radar MTI Process

build-ups in the Multiport Memory System. With these capabilities, most of the complex tasks encountered in radar signal processing can probably be specified directly by the user in the data flow graph, thus avoiding conventional software development.

The task dispatcher has access to the object graph data sets (see Figure 5) and the task ready list from which it selects the task as well as the processor best suited to perform the task. The task scheduler accesses the object graph data sets to determine which nodes are ready to execute, and places them on the task ready list.

CONCLUSIONS

In this paper, the organization of a Programmable Modular Signal processor that utilizes a data flow concept of control was presented. Data flow architecture was chosen to take advantage of the parallelism inherent in most real-time applications. The attractive features of this approach include:

a) Hardware that is independent of signal processing applications

b) Development of a relatively small number of hardware functional units

c) Isolation of signal processing algorithm development from software development

d) A mechanism for the development of a signal processing application comprised of a configurable set of standard hardware functional units

e) A mechanism for merging data processing control functions written in a HOL at the host, and signal processing functions expressed in graph notation.

The modifications to the conventional data flow architectures enables very high performance data flow machines to be developed. Having eliminated the data token traffic from the circular pipeline, the dispatcher can dispatch the task packets to the processors faster than normally possible. The scheduler can also perform faster because it does not have to deal with large number of data tokens. Performance can be further improved by segmenting the dispatching and scheduling operations and pipelining them.

Preliminary analysis of the architecture gives a strong indication that it is suitable for real-time signal processing applications. Detailed simulation modelling and performance evaluation under various conditions for a variety of applications is in progress.

REFERENCES

1. Minsky, M. and S. Papert, *On Some Associative, Parallel and Analog Computations,* Associative Information Techniques (E.J. Jacks, ed.), Elsevier, 1971.

2. Flynn, M.J., "Some Computer Organizations and their Effectiveness," *IEEE Transactions on Computers,* Vol. C-21, No-9, Sept. 1972, p. 948.

3. Enslow, P.H., *"Multiprocessor Organization — A Survey," ACM Computing Surveys,* Vol. 9, No. 1, March 1977, p.103.

4. Watson, I. and Gurd, J., "A Prototype Data Flow Computer with Token Labelling," *National Computer Conference Proceedings,* 1979, pp. 623-628.

5. Watson, I., and Gurd, J., "Data Driven System for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution," *Computer Design,* June 1980, pp. 91-100.

6. Watson, I., and Gurd, J., "Data Driven System for High Speed Parallel Computing - Part 2: Hardware Design," *Computer Design,* July 1980, pp. 97-106.

7. Dennis, J.B., "Data Flow Supercomputers," *IEEE Computer,* Nov. 1980, pp. 48-56.

8. Watson, I., and Gurd, J., "A Practical Data Flow Computer," *IEEE Computer,* Feb. 1982, pp. 51-57.

9. Davis, A.L., "A Data Driven Machine Architecture Suitable for VLSI Implementation," Caltech Conference on VLSI, January, 1979, pp. 479-494.

10. Perry, R.P., Private Communication.

11. Gajski, D.D., et al., "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer*, Feb. 1982, pp. 58-69.

12. Sawkar, P.S. et al, "A Multiport Memory Organization for Use in Distributed Computing Systems," IEEE 3d Int'l. Conference on Distributed Computing Systems, Oct. 1982, pp. 56-61.

13. Schernecke, E.J., Private Communication.

A SIMULATOR FOR MIMD PERFORMANCE PREDICTION-- APPLICATION TO THE S-1 MkIIa MULTIPROCESSOR[a]

T. S. Axelrod, P. F. Dubois, P. G. Eltgroth
Theoretical Physics Division
Mathematics and Statistics Division
Lawrence Livermore National Laboratory
Livermore, California 94550

Abstract -- We describe a MIMD multiprocessor simulator and application of that simulator to a multiprocessor of current interest, the S-1 MkIIa. The simulator runs on the Cray-1 and is designed so that computational physics benchmarks are actually run and produce results. Simulator output from this run are fed into a second level (hardware) simulator which calculates the behavior of the multiprocessor. The simulator can simulate multiprocessors whose basic architecture is that of a few, large processors with or without data caches, sharing global memory through an interconnection switch. The simulator is applied to investigate the behavior of SIMPLE, a benchmark physics code.

I. Introduction

Our purpose in this paper is twofold. We begin by describing a simulator we have created for predicting the performance of realistic physics calculations performed on multiprocessors. We then describe the physics code SIMPLE and present simulator results for its performance on a multiprocessor of current interest, the S-1 MkIIa [S-179]. The simulator is available for use on Cray-1 computers under the CTSS operating system, and a user's manual has been previously published [Axe82].

II. Simulation Methodology

1. Goals of the Simulator

Simulation of computer systems is performed for a great variety of purposes. Among these are gate level simulations to verify logical correctness of a design, register level simulations which allow the effects of instruction sequences to be determined, and queuing models, which are most commonly employed to predict performance of computer systems under timesharing loads. The simulations described here fall in a less common category. Our goal is to predict the performance of an MIMD computer in solving large computational physics problems by some specified algorithm.

Our simulation has an additional goal of nearly equal importance. Since usable and accessible MIMD machines are still very rare, computational physicists, numerical analysts, and programmers have little or no experience with the "parallel world". Not only is a new set of performance related issues present, but there are new issues of logical correctness and choice of language constructs. Our simulator, called MPSIM, provides a readily available tool for gaining at least some of this experience. One crucial feature of the simulator is that the algorithm being investigated is actually run in all its detail so that its numerical behavior (e.g., stability, correctness of answers) can be observed. Among other things, this allows many bugs related to multiprocessing to be found.

2. The Trace-driven Two Level Methodology

Simulation under MPSIM occurs at two levels (MPSIM-1 and MPSIM-2). These two levels represent the software and hardware, respectively, of the integrated system being modelled. This two level structure was inspired by the work of L. Cox [Cox78]. At the first level of simulation we deal with autonomous instruction streams referred to as processes, without reference to any physical implementation, while at the second level we deal with physical processors. At present we assume that there is a one-to-one mapping between processes and processors, but this assumption is unnecessary, and affects only MPSIM-2. Thus extension of the simulator to include machines such as the Denelcor HEP [Smi78], which has multiple processes per processor, is at least in principle straightforward.

The two levels of the simulation (which can be run independently) are used for different purposes. At the first level of simulation, we are concerned primarily with the correct logical operation of a program, and the details of a particular multiprocessor implementation (such as the relative speeds of private and shared memory, the speeds of various synchronization operations, and so on) may be mostly ignored. On the other hand, the performance of a multiprocessor program may depend critically on the details of the implementation, and the second level of the simulation, which is driven by output from the first level, is intended to predict this performance. In general, performance information will dictate changes in the program, which are incorporated by returning to the first level.

MPSIM-1 is an extended version of a simulator written by L. Sloan. It runs on the Cray-1 under CTSS and consists principally of a process scheduler and the machinery and data structures to save and restore process state information. The simulator provides the FORTRAN user with a number of subroutines which allow the creation and destruction of processes and implement a variety of synchronization operations. A brief summary of the available functions is given in Table 1. A complete description is available in [Axe82].

[a] Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

0190-3918/83/0000/0350$01.00 © 1983 IEEE

Table 1. Summary of MPSIM-1 Subroutines.

Name	Purpose
MPINIT	Initialize MPSIM-1
TRCINIT	Begin trace output for MPSIM-2
FORK	Start an additional process
FORKOFF	Start multiple additional processes
SYNCAL	Global synchronization barrier
PSEM	P operation on a semaphore
VSEM	V operation on a semaphore
PAWS	Wake up the MPSIM-1 scheduler
JOINAL	Terminate all but 1 process
PRCEND	Terminate a process
TRCFIN	Terminate trace output for MPSIM-2
MPFINI	Terminate MPSIM-1 operation

In operation the Level 1 simulator is a timesharing system in miniature. A single Cray-1 is multiplexed among the currently runnable processes and a simulated wall-clock is maintained. The level 1 simulator is in fact a simulation of a particular MIMD machine--a highly idealized multiple Cray-1 with both shared and local memory.

Clearly most of the characteristics of a parallel algorithm which will determine its performance on a real MIMD machine are contained in the details of its behavior on this idealized MIMD machine. Not only is the pattern of process creation, destruction, and communication present, but quite complete information is also available on the actual computation performed by each process, much of which is hardware independent. The linkage between the level 1 and level 2 simulator consists of abstracting the performance related information from the level 1 behavior and transferring it to level 2 where it may be interpreted in the context of a more realistic (and possibly quite different) MIMD machine.

What should actually be included in the information transmitted to level 2? One extreme approach would be to include the complete state of the simulated MP-Cray on every clock cycle. There are two serious problems with this, however. The quantity of information is several orders of magnitude too large to permit the behavior of a complete physics algorithm to be practically stored or processed. Even more serious, perhaps, is the fact that most of the information generated has no obvious relevance to machines other than a Cray-1 or a very close relative.

At the opposite extreme, one might simply count arithmetic operations performed by each process. This information is so incomplete, however, that the hardware model contained in Level 2 must of necessity be quite simple. In particular, details of interprocessor interactions which arise from memory conflicts and cache coherence problems cannot be modeled.

In general the choice of which information should be abstracted from the Level 1 behavior is partially subjective and must be based on a careful assessment of the characteristics of the system being modelled, and the size of the computational resources available for the simulation. The nature of the choice we have made for modelling the S-1 MkIIa is discussed in Section III. For the moment we merely note that it falls between the two extremes, preserving the details of each process' data referencing patterns and the arithmetic operations it performs, while neglecting many details of address calculations and instruction referencing patterns.

During the operation of MPSIM-1 the information needed by MPSIM-2 is gathered by machine instructions which have been automatically inserted into the user's object code. These instructions cause control to be temporarily transferred to simulator routines which write the desired machine state information to disc files, one for each process. This information gathering process is invisible to the user except for increased CPU charges.

MPSIM-2 views the events contained in each process' trace stream as requests for hardware services by a running process. Typically the services required are arithmetic operations and transfers of operands. Each request is satisfied as soon as possible within the constraints of the hardware model. We note that this may result in a time ordering for events in separate process streams that differs from that of MPSIM-1. Event ordering within a single process stream is preserved (unless the MPSIM-2 hardware model allows out-of-order execution), as are the interprocess orderings enforced by synchronization.

The output of MPSIM-2 is a detailed record of event histories for each of the simulated processors, which in practice are usually viewed in the form of graphical plots.

III. THE S-1 MkIIa Hardware Model

As the first application of our simulation techniques we chose to investigate the S-1 MkIIa multiprocessor. This machine is of great interest due to its imminent availability and supercomputer performance potential. Additionally, its design explores for the first time the use of cache memory in high speed multiprocessors.

In this section we describe the hardware model we have incorporated in SISIM-2 to model the S-1 MkIIa multiprocessor. When we created the model, the S-1 MkIIa implementation was far from complete, and a number of details were uncertain. This was particularly the case for the main memory, crossbar switch, and microcode used for interprocessor communication. Since hardware documentation was generally unavailable we have relied on conversation with S-1 project members [Far82] to fill in the gaps where possible. How successful we have been in modelling the S-1 MkIIa as it is finally implemented will not be known until the machine is available for testing. We nonetheless are presenting the model and the results from it in the hope that it will be a generally useful contribution to performance assessment of MIMD machines.

1. Summary of S-1 hardware

The S-1 MkIIa multiprocessor [S-179,Far80,Far81] consists of up to 16 processors connected to a similar number of memory banks by a crossbar switch. Each processor is extensively pipelined and microcoded and possesses the following major resources:

1. Instruction cache (4k words).
2. Data cache (16k words).

3. Address arithmetic unit.

4. Floating point adder.

5. Floating point multiplier.

6. 16 general purpose register files with 32 words each.

7. Local memory (1M word class).

The S-1 pipeline is partitioned into two major units. The IBOX fetches and decodes instructions and handles all tasks associated with the fetching and storing of operands, including management of the cache, mapping of virtual to physical addresses, and memory protection. The ABOX is responsible for the remaining steps in instruction execution and calculates all results which will be stored in the register files or memory.

The processors are implemented with ECL 10K and 100K integrated circuits, while the local and global memories are implemented with 64kb MOS chips. The design cycle time of the processor is 50 ns for the instruction fetch and decode unit (IBOX) and 25 ns for the arithmetic unit (ABOX). The word size is 36 bits.

The ABOX adder and multiplier are innovative in several respects [Far81]. They have been designed for low latency and are partitionable to allow calculation with operands of 18, 36, and 72 bits. Special attention has been devoted to achieving high speed on FFT's and other mathematical functions. In part for this reason the adder produces both the sum and difference of its operands simultaneously.

The instruction set is extensive, providing a wide variety of addressing modes, dyadic and triadic vector operations, and evaluation of mathematical functions among its most notable features.

Since the MPSIM-2 hardware model is driven by a trace stream obtained from a Cray-1, it is appropriate to contrast the S-1 MkIIa design with that of the Cray-1 [Cra76]. The most important differences include the following:

1. The design of the memory hierarchies differ greatly. The S-1 is a virtual address machine which utilizes a combination of fast ECL data and instruction caches, very large MOS main memory, and disk for demand paging. The Cray-1 is much simpler, relying on ECL technology for both its registers and 16 interleaved banks of main memory.

2. The S-1 has fewer functional units. All instructions must pass through either the adder or multiplier in the ABOX. On the Cray-1 there are thirteen functional units (including memory) which may execute instructions.

3. Vector operands on the S-1 are obtained directly from memory, while on the Cray-1 they are held in vector registers for use by the vector functional units. The vector stride must be 1 for the S-1, while it may be any value on the Cray.

4. The S-1 instruction set is more powerful than that of the Cray, so that multiple Cray instructions can often be replaced by a single S-1 instruction. The most common occurrence of this is in operand address computation, but there are many examples.

5. On the S-1, results are produced strictly in the order implied by the instruction order. On the Cray-1, although instructions are always issued in order, results may be produced out of order and by any of the functional units.

Both the number of instructions and the number of machine cycles devoted to address arithmetic are likely to differ greatly between the two machines. In favorable cases, however, the time taken to perform these operations is completely "hidden" through overlap with floating point operations and memory references. The same situation holds for conditional branches. Both machines are able to issue instructions at a maximum rate of one per cycle (12.5 ns Cray-1, 50 ns S-1 MkIIa). How closely this goal is approached is very sensitive to the optimization techniques employed by the compiler.

As sketched above, the S-1 MkIIa processor is highly complex. Our simulation models a small carefully chosen subset of its features. In most cases the model assumes that omitted features (e.g., prediction of conditional branches) work perfectly, so that the simulation results form an upper bound on performance, but there are exceptions.

In selecting the hardware features to be included in the model we began by recognizing that the effectiveness of the data cache will be a major determinant of performance. Occurrence of a data cache miss stops the IBOX pipeline until the required data can be obtained. Since main memory is much slower than the IBOX cycle time, data cache misses can easily impose a limit on performance. This is especially the case when processors share data so that cache coherence problems arise. The model must therefore be able to determine the contents of the data cache of each processor at every stage of the computation.

On the other hand, we expect the effectiveness of the instruction cache to be generally high and of less importance in determining performance. This arises from both the generally much smaller size of total program instructions relative to cache size, and from the generally much higher localities observed for program instruction references compared to data references. This is most fortunate, since the instruction cache hit rate could not be modelled accurately without actual S-1 instruction streams for the computation. These will not be usefully available until a vectorizing compiler for the machine is completed.

Our model of the ABOX is quite simplified, since it ignores delays which result from data dependencies between operations. If all required operands are in cache the simulated ABOX is ready to execute a new instruction a fixed time (T_{issue}) after beginning the previous instruction. We have chosen T_{issue} to be the shortest possible--those obtained in the absence of data dependencies. This is not a necessary restriction on the model, since all data dependencies are in fact available in the trace stream. Again, our results are expected to form an upper bound on performance.

Most of the complexity of the S-1 processor arises from the need to keep arithmetic function units at the end of a long complex pipeline supplied with operands at a continuously high rate. (See [Kog81] and [Lor72] for good discussions of pipelined processors.) To achieve this, a variety of techniques is employed, including the partial decoupling of the IBOX and ABOX with an operand queue, the prediction of conditional branches, and the prediction of values needed in address computations [Far81].

Our model assumes that in the absence of data cache misses, all of these techniques work perfectly, so that the functional units perform work at the maximum possible rate. Clearly this may be seriously in error for some computations. A Monte Carlo computation with a large number of quasi-random conditional branches is likely to run more slowly than our model would predict, for example, due to reduced effectiveness of the hardware conditional branch prediction strategy.

The model makes an additional simplifying assumption: the cost of address arithmetic instructions is ignored. As discussed above, this is equivalent to assuming that address arithmetic calculations are fully overlapped with other computations. Although the address arithmetic instructions themselves are filtered out, the memory reference instructions needed to fetch their operands are retained. This is necessary to include their effect on the data cache hit ratio (typically small).

The effect of the simplifications we have discussed so far is exclusively in the direction of predicting performance which is too high. The performance results for the Monte Carlo algorithm discussed in Section IV probably show these effects to a significant degree. The model, however, contains additional assumptions which work in the other direction. The most important of these is ignoring the chaining of vector operations on the S-1. Chaining allows triadic vector instructions to make simultaneous use of the adder and multiplier in the ABOX. This can increase the MFLOP rate by up to a factor of 2. Measurements on the Cray-1, which has a more general chaining capability, show the effect to be somewhat less than this in most cases.

Of less importance is the fact that the model does not reflect the ability of the S-1 to calculate special functions (e.g., **sin**, **log**, **exp**) nearly as fast as a single multiplication. Measurements on the Cray-1, which calculates these functions slowly relative to multiplications (120 ns vs 12 ns per result in vector mode), shows that even physics simulation codes which make intensive use (by current standards) of special functions very rarely spend more than 10% of their time performing them. It is quite possible, however, that algorithms will evolve which exploit the S-1's ability to evaluate special functions inexpensively.

IV. SIMPLE

SIMPLE [Cro78] is a widely distributed code which models the hydrodynamic and thermal behavior of fluids in two dimensions. The hydrodynamics is a standard Lagrangian formulation using an artificial viscosity. Heat transfer is performed in the diffusion approximation using a single ADI iteration on a five point implicit difference operator. Thermodynamic properties of the fluid are obtained by table lookup and biquadratic interpolation between table entries.

After an initialization phase the calculation consists of a sequence of timesteps each of which advances the solution by a time increment DT in the following manner:

1. Calculate the pressure in each zone given the temperature and density.
2. Compute the acceleration, new velocity and new position of each zone.
3. Compute new zone volumes.
4. Compute the artificial viscosity and Courant timestep limit for each zone.
5. Calculate new zone internal energy after hydrodynamic work. To maintain sufficient accuracy the new energy is first predicted using old thermodynamic quantities. The predicted energy is then used to calculate more accurate thermodynamic quantities which are used for the final calculation of the new internal energy.
6. Calculate new temperature after hydrodynamics from new density and internal energy.
7. Calculate heat diffusion coefficient for each zone.
8. Calculate the coupling constants for the column direction (one per zone).
9. Calculate an intermediate temperature by solving a tridiagonal linear system which couples zones in the same column and has the temperature from step 7 as a right hand side.
10. Calculate the coupling constants for the row direction (one per zone).
11. Calculate the new temperature by solving a tridiagonal linear system which couples zones in the same row and has the temperature from step 9 as a right hand side.
12. Calculate a heat diffusion DT for each zone from the rate of change of its temperature.
13. Calculate new zone internal energy from new temperature.
14. Calculate whole problem sums (kinetic and internal energy and heat flow across problem boundaries) and next DT by finding the minimum over the entire mesh of the zonal Courant and heat diffusion DT's.

Although space does not permit a complete discussion of these calculational steps, some comments are in order. Steps 1 through 6 constitute the hydrodynamics portion of the timestep. The method is explicit, and the new values for a zone depend only on the previous values of that zone and its six nearest neighbors. Steps 7 through 13 constitute the heat conduction portion of the timestep. The method is implicit, and the new temperature of a zone depends on the previous values of all zones in the mesh. This difference is quite important for a multiprocessor. It is also important to note that boundary zones require special treatment for both hydrodynamics and heat conduction and require more calculations than interior zones.

We began with a version of SIMPLE which is almost completely vectorized by the CFT or CIVIC compilers for the Cray-1. This program was modified for a multiprocessor in a straightforward manner. Each processor is assigned a fixed contiguous group of mesh columns for which it is responsible at each stage of the calculation. With the exception of the heat conduction row sweep (discussed below) all calculations are vectorized along columns. All arrays are stored columnwise, so this results in vector operations with unit stride, which is ideal for the S-1. Synchronization barriers were emplaced between calculation stages when required to ensure the proper data dependencies. Fourteen such barriers are required within the timestep loop. The program contains a single critical section implemented with semaphores which is used for updating scalars which depend on the global mesh (step 14).

With the exception of a single processor which is given the additional duty of performing output to the edit file, all processors execute identical code. In addition to the main data structures of the problem, which are in shared memory, each processor is supplied with local data structures which are stored in processor private memory. A few of these, such as loop indices, must be supplied to ensure logically correct operation. The majority of them, however, are made local to increase performance. These include the material property tables, arrays holding the coupling coefficients for heat conduction, and scratchpad arrays used for holding temporary results. Local scalars are used to hold each processor's contribution to global mesh quantities such as total kinetic energy and minimum timestep (step 14).

As mentioned above, the heat conduction calculation contains an exception to the column group partitioning used in the remainder of the code. The heat conduction step in fact raises some interesting partitioning issues, and deserves special discussion. Since the value of a zone temperature after the heat conduction step depends on the previous temperatures of all mesh zones, it is inevitable that this calculation on a multiprocessor will involve substantial interprocessor communication. There are at least two different ways of organizing the calculation.

1. Straightforward partitioning. During the column sweep each processor is given a group of columns, while during the row sweep each processor is given a group of rows. All interprocessor communication is handled invisibly by the cache coherence algorithm in the hardware.

2. Wavefront with row blocking. Each processor works with a group of columns for both the row and column sweep. During the row sweep processor i+1 cannot begin on its portion of a row until processor i is finished with its portion of that row.

The second method, which is advocated by Gilbert [Gil79] attempts to reduce interprocessor communication demands at the expense of reduced parallelism and increased program complexity. The idea is that each processor "owns" the data associated with a column group. During the row sweep a processor continues to work with this local data except at the boundaries of its column group, where interprocessor communication is required. For large column groups the relative cost of this communication becomes small.

This strategy is effective only if the data associated with a column group remains local to a processor throughout the calculation. On the S-1 MkIIa there are two kinds of local data - that which is contained in the data cache and that which is contained in processor private memory external to the cache. Data which is in cache remains there only until it is removed by the replacement algorithm to make way for other data. This kind of locality is transient and for large column groups is destroyed during the column sweep. For the wavefront algorithm to work as intended, therefore, the column group data must be held in processor local memory. Processors utilize the crossbar switch only to send data packets directly to other processors.

The wavefront approach therefore is a form of "distributed computing" and as such requires quite different programming constructs than employed in uniprocessor scientific programs. In particular, the extended version of FORTRAN we are using has no convenient facilities to distinguish local from shared data or for the sending and receiving of interprocessor messages. The first approach, however, allows the heat conduction part of SIMPLE to be programmed in the same style as the rest of the code.

For the simulations reported here we have chosen to use the straightforward partitioning appropriate to a tightly coupled approach to multiprocessing. It is interesting to note that for large problems the overhead directly attributable to interprocessor communication still becomes small. It is of course true that each processor writes out many cache lines to main memory which are later read by other processors. The point is that the vast majority of these reads and writes are performed by the normal cache replacement algorithm, and would occur even in the absence of other processors. The penalty of the shared memory approach is then paid mainly in bank conflicts.

The simulator output includes a large number of performance diagnostic quantities. In our discussion below, we have used the following definitions: The cache hit ratio is the ratio of the number of memory references originally finding a datum in cache divided by the total number of memory references. Efficiency is the ratio of the time one processor needs for a calculation divided by P times the time it takes P processors to finish the same calculation. Traffic ratio is the ratio of the number of bits transferred between cache and other memory to the total number of bits which flow between cache memory and its CPU.

Speed is measured in megaflops (MFLOPS), millions of floating point operations performed per second. Speedup is the ratio of the time it takes one processor to perform a calculation divided by the time it takes a specified number of processors to complete the calculation.

It is important to notice that direct comparison of MFLOPS on a parallel machine versus a sequential machine may be misleading since many parallel algorithms perform significantly more arithmetic operations than their sequential analogs. For SIMPLE, however, such redundant operations are completely negligible.

We have run problems varying in size from 20C,20R to 90C,40R (where C denotes number of columns and R number of rows) and utilizing from 1 to 16 processors. Most of these have been run in single precision mode, although a few double precision runs have also been made. The results we report here are for a single execution of steps 1--14 of the timestep advance. Runs with multiple cycles have been performed, and show that the transient effects from starting with an empty cache are quite small. Table 2 summarizes our results for a 90C,20R problem run in single precision with varying numbers of processors.

Table 2. Simulated performance of SIMPLE.

P	MFLOPS	Speedup	Efficiency
1	9.04	1.00	1.00
2	16.00	1.77	0.89
4	26.45	2.93	0.73
6	32.17	3.56	0.59
8	34.7	3.84	0.48

The efficiency drops rapidly for P > 4 and there is clearly little to be gained by running this problem on more than eight processors.

Figures 1 through 5 show detailed simulator results for a single run, a 90C,20R single precision problem run with four processors. As is evident, each phase of the calculation exhibits its own pattern of machine activity, so that the behavior during the timestep is quite complex. This pattern is recognizably similar for all multiprocessor SIMPLE runs we have performed, regardless of size or number of processors.

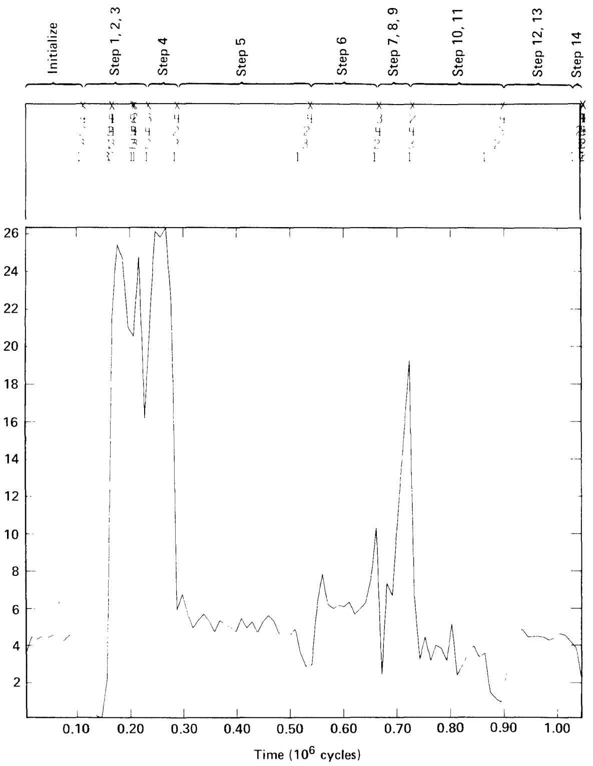


Figure 1. Simulated performance of four processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The upper portion of the figure shows the arrival times of the individual processors at the algorithm's synchronization points. All processors are restarted after the synchronization at the points marked "X". The step numbers refer to the calculational steps defined in the text. The lower portion of the figure shows the average per-processor megaflops vs time measured in 25 ns ABOX cycles.

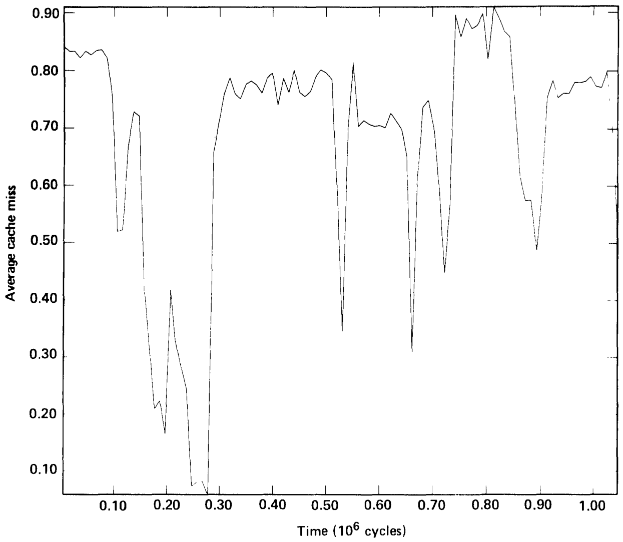


Figure 2. Simulated performance of four processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average fraction of time spent servicing data cache misses vs time in 25 ns ABOX cycles.

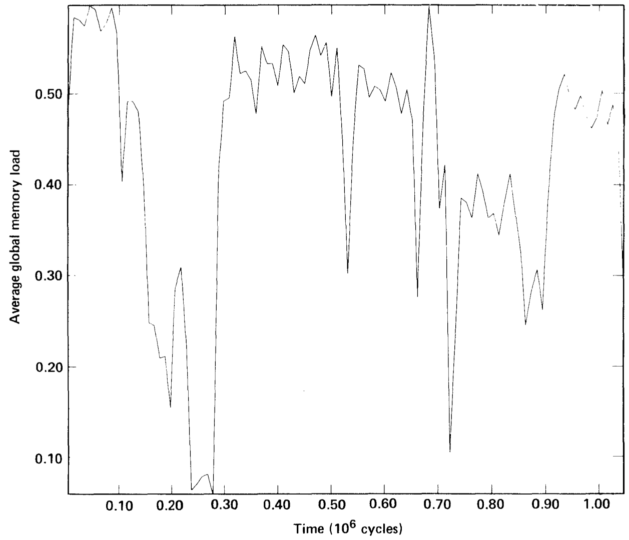


Figure 3. Simulated performance of four processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average global memory load as a fraction of total available bandwidth vs time in 25 ns ABOX cycles.

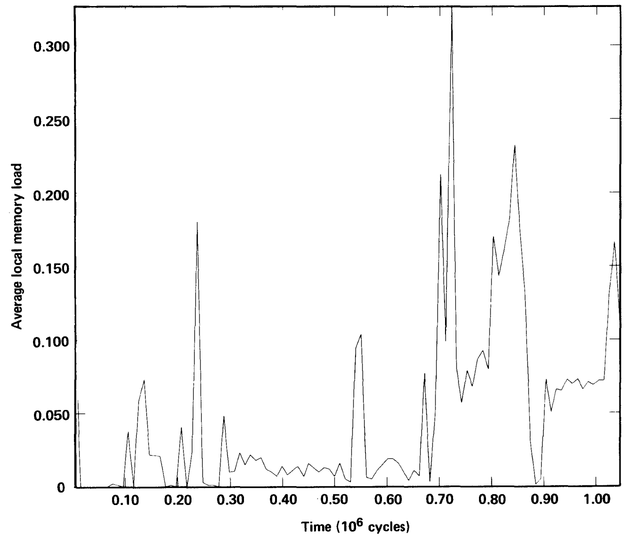


Figure 4. Simulated performance of four processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average local memory load as a fraction of total available bandwidth vs time in 25 ns ABOX cycles.

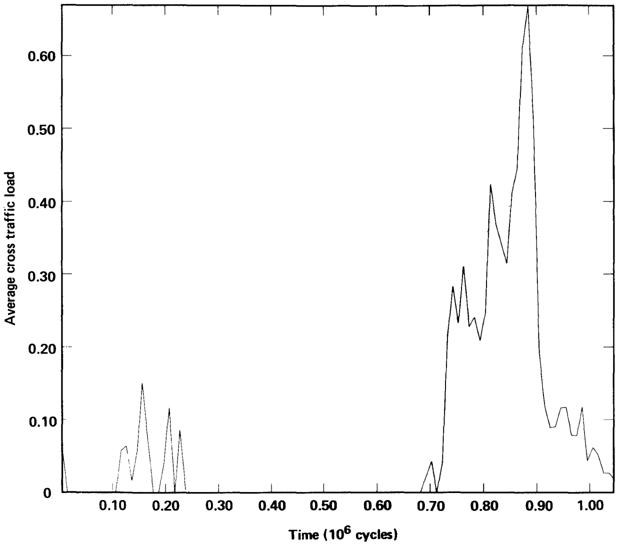


Figure 5. Simulated performance of four processor S-1 MkIIa in single precision mode on SIMPLE. The problem size is 90C, 20R. The figure plots the average fraction of time spent servicing interprocessor cache line transfers vs time in 25 ns ABOX cycles.

It is interesting to compare the machine activity during the heat conduction row sweep with that of the column sweep. The row sweep shows high cross traffic loads and takes about 2.75 times as long as the column sweep. Both column and row sweeps show extensive use of processor local memory, mainly due to the local storage of coupling coefficients. In the light of the discussion above we expect this picture to change substantially for sufficiently large problems, which should show a less pronounced performance difference between column and row sweeps.

We can easily calculate the problem size for which this transition should occur. For the 90C,20R problem, each processor requires roughly 5*90*20/4 = 2250 distinct operands from the point it begins to access the shared temperature array during the column sweep until the column sweep is finished. This is substantially smaller than the cache size of 16384 words, so that each processor's entire column group is present in cache at the end of the column sweep. The subsequent row sweep then triggers the observed burst of cross traffic. For problems larger by a factor of roughly 8 (16000 zones), however, this situation will change and each processor at the end of the column sweep will already have started writing back to shared memory the first temperature elements it accessed.

In spite of its dramatic appearance, the row sweep is not the only cause of the inefficiency shown in Table 2. Analysis of the simulator runs allows us to assign the inefficiency to three major causes, as shown below.

	P = 4	P = 8
Waiting at synchronization barriers:	0.03	0.06
Global memory conflicts:	0.15	0.25
Interprocessor line transfer:	0.06	0.12
	----	----
Total:	0.24	0.43

It is perhaps more useful to express these same numbers as "lost CPU's" by multiplying the fractional performance loss by P.

	P = 4	P = 8
Waiting at synchronization barriers:	0.12	0.48
Global memory conflicts:	0.60	2.00
Interprocessor line transfer:	0.24	0.96
	----	----
Total:	0.96	3.44

We clearly must consider how this picture will change when the mesh size is greatly increased. As argued above, the fractional cost of both synchronization waiting and interprocessor line transfer should drop steadily with increasing mesh size, leaving global memory conflicts as the principal cost of multiprocessing. Simple models of multiprocessor memories [Yen82] predict that crossbar systems with equal numbers of processors and memories show an inefficiency due to conflicts that is nearly independent of P when P is greater than about 8. These facts taken together imply that efficiencies for sufficiently large problems should be fairly high (about 0.7) even for large numbers of processors.

This is not the end of the story, however. We must note that high efficiency does not necessarily imply high performance! It merely means that performance continues to grow linearly as processors are added. On a cache based machine, such as the S-1, the performance of each uniprocessor drops as problem size is increased. Table 3 shows the effect of varying the mesh size for SIMPLE in the single processor case.

Table 3. One processor, varied problem size.

C	R	C*R	MFLOPS	Hit ratio	Tfc ratio
10	20	200	13.61	0.9956	0.078
15	20	300	12.68	0.9952	0.087
30	20	600	11.05	0.9943	0.108
60	20	1200	9.89	0.9934	0.128
90	20	1800	9.02	0.9925	0.145

One may expect that this decline will continue as problem size is further increased. The asymptotic value is difficult to predict without detailed analysis of the algorithm. The issue of performance scaling with problem size therefore becomes complex. As problem size increases performance tends to also increase, due to the decreasing relative cost of synchronization waiting and interprocessor communication. At the same time, however, performance is negatively affected by the decreasing effectiveness of cache as the data set size increases.

V. Conclusion

The simulations we have reported on were undertaken with the goals of investigating the performance issues raised by MIMD machines and gaining experience with the programming techniques required to utilize them. As yet we have explored only a limited set of algorithms and a single simulated machine. As discussed earlier, we have taken existing uniprocessor algorithms and extended them to a shared memory multiprocessor with minimal changes. Clearly future algorithms may depart radically from this approach. Our simulations are also deficient in that real problems run on fast multiprocessors will in general have many more zones than those we have been able to treat.

In view of all these limitations, what have we actually learned? Our experience to date with the simulator allows us to draw three tentative conclusions about the use of MIMD machines for solving large scientific problems:

1. The S-1 (and other similar machines) can be used as a multiprocessor in two relatively distinct modes. These are a shared memory, or tightly coupled, approach in which problem data is primarily stored in shared memory; and a distributed processing, or loosely coupled, approach in which problem data is primarily stored in private memory and communication takes place when required through the interprocessor message network. The shared memory approach, which we have used here, is relatively simple to program using a few extensions to FORTRAN. The distributed computing approach appears to offer higher performance for many algorithms. However, substantial programming effort and significant language extensions would be required to realize this potential.

2. Extrapolation of our simulator results to much larger problems indicates that many of the factors which limited our speedups in the 4 < P < 16 range will be greatly reduced in importance. This is particularly true for:

a. Overhead operations which result from partitioning the algorithm. These include process management operations and communication between processes.

b. Synchronization penalty that results from speed variation between processes.

On the other hand, conflicts between processors in obtaining access to shared resources (usually memory banks and communication paths) will continue to be important. In considering the scaling issue, the effect of cache memory may outweigh all of these factors, however, which brings us to our final point.

3. The usefulness of a traditional data cache for scientific problems with large data sets appears questionable. In the cases we have studied, performance drops rapidly with increasing problem size. Clearly this performance drop can be delayed with algorithms which optimize cache hit rates. It does not appear feasible to do this by hand, however, except for simple cases. Compilers clearly must perform this task if it is to be done at all. The problem is difficult since the optimization approach must be dependent on data set size.

In the future we plan to further increase our understanding of MIMD machines and algorithms by pursuing the comparison of our simulator results with actual measurements on the S-1 MkIIa multiprocessor, when it becomes available. Additionally, we feel that improved analytic models for MIMD performance can be constructed which will be quite useful. The work of Briggs [Bri80], Dubois [Dub82a,b], Yen et al. [Yen82], and Gilbert [Gil79], among others, provide an excellent foundation on which to build them.

References

[Axe82] Axelrod, T. S., Chase, L., Eltgroth, P. G., and Sloan, L. S., Multiprocessor Simulator User's Manual, Lawrence Livermore Laboratory, UCID-19594 (1982).

[Bri80] Briggs, F. A. and Dubois, M., "Modeling of Synchronized Iterative Algorithms for Multiprocessors," Proceedings of the 18th Annual Allerton Conference (1980).

[Cox78] Cox, L. A., Performance Prediction of Computer Architectures Operating on Linear Mathematical Models, Lawrence Livermore National Laboratory, UCRL-52582 (1978).

[Cra76] CRAY-1 Computer System Reference Manual, Cray Research Inc. Publication 2240004 (1976).

[Cro78] Crowley, W. P., Hendrikson, C. P., and Rudy, T. E., Lawrence Livermore National Laboratory, UCID-17715 (1978).

[Dub82a] Dubois, M., and Briggs, F. A., "Performance of Synchronized Iterative Processes in Multiprocessor Systems," IEEE Trans. Software Eng., Vol. SE-8, No. 4 (1982), pp.419-431.

[Dub82b] Dubois, M., and Griggs, F. A., "Effects of Cache Coherency in Multiprocessors," *IEEE Trans. Computers*, Vol. C-31 (1982), p.1083.

[Far80] Farmwald, P. M., Bryson, W., and Manferdelli, J. L., "Signal Processing Aspects of the S-1 Multiprocessor Project," *Proc. Soc. Photo-Opt. Instrum. Eng.*, Vol. 241 (1980), p.224.

[Far81] Farmwald, P. M., *On the Design of High Performance Digital Arithmetic Units*, Ph.D. thesis, Stanford University (1981).

[Far82] Farmwald, P. M., private communication, S-1 Project, Lawrence Livermore National Laboratory (1982).

[Gil79] Gilbert, E. J., "Investigation of the Partitioning of Algorithms Across an MIMD Computing System," *S-1 Project 1979 Annual Report*, Lawrence Livermore National Laboratory, UCID-18619, Vol 1 (1979).

[Kog81] Kogge, P. M., *The Architecture of Pipelined Computers*, McGraw-Hill (1981).

[Lor72] Lorin, H., *Parallelism in Hardware and Software*, Prentice-Hall, Inc. (1972).

[S-179] Wood, L. L., ed., *S-1 Project 1979 Annual Report*, Lawrence Livermore National Laboratory, UCID-18619 (3 vols), (1979).

[Smi78] Smith, B. J., "A Pipelined, Shared Resource MIMD Computer," *Proc. 1978 Int'l. Conf. Parallel Processing*, Bellaire, MI (1978), p.6.

[Yen82] Yen, D. W., Patel, J. H., and Davidson, E. S., "Memory Interference in Synchronous Multiprocessor Systems," *IEEE Trans. Computers*, Vol. C-31 (1982), p.1116.

VECTORIZATION OF DISCRETE EVENT SIMULATION

Avinash Chandak*
and
J. C. Browne
Dept. of Computer Science and the Computation Center
The University of Texas at Austin
Austin, Texas 78712

Abstract

This paper establishes a simulation model in which vectorizable discrete event models can be defined. It gives an algorithm for exposing the vectorizable structure in a given model. It applies this algorithm to a Monte Carlo particle transport problem known to be vectorizable and demonstrates why this problem is vectorizable. A few implications of vectorization for definition of models to be simulated and for implementation of simulation systems are given.

The Problem of Vectorization of Discrete Event Simulations

It is an item of folklore in computer science that discrete event simulations cannot be effectively vectorized because of the random nature of event generation. This supposed limitation is a non-trivial problem since the utility of simulations may depend on levels of accuracy which are obtainable only by very long runs on the fastest of today's scalar computers. This paper establishes the conditions under which discrete event simulation can be vectorized. The use of the framework which is given here may lead to the selection of model representations which are more vectorizable than equivalent alternative representations.

Application of the procedure is illustrated by consideration of the structure of a model for Monte Carlo particle transport process. Brown, Callahan and Martin [2] showed by analysis of the actual code that it is almost entirely vectorizable. The analysis given herein shows why this is the case.

The conditions for vectorizability are very similar to those established by Georgiadis, Papazoglou and Maritsas [3] for MIMD parallel structuring of SIMULA programs.

There are three levels of approach to vectorization. The first is to look at existing code to determine what can be vectorized. The second level is to look for algorithms which can be effectively vectorized. The third level is for structural formulations of a problem directly in terms of vectors. This study began at level 2 and, in particular, examined vectorization of time axis or event list processing. This portion of the project was successful in the sense that a vector formulation of time axis management produced a major (factor of two) improvement in run times for cases where time axis management was a major factor in total run time [1]. This factor of two was actually demonstrated in evaluations of model systems defined in the simulation system described following on a CDC CYBER 205. Further improvement at the algorithm level seemed unlikely so we turned attention to the basic structure of discrete event problems as expressed in the rather simple simulation modeling system developed for the level 2 studies. This approach leads to the results given herein.

Definition of the Simulation Modeling System

The modeling capability of this simulation system consists of the following major constructs and some system provided routines.

1. Transactions. These are entities which consist of data only. They may have some (or no) user defined parameters. Several different transaction types may be defined. The system maintains space for transactions.

2. Transaction Generators. These are user defined routines which generate transactions. A transaction generator has a transaction type associated with it.

3. Facilities. These are user defined routines which service the transactions. They manipulate the user defined transaction parameters. Each facility services some queue(s). They specify the service time needed by using the system routine SIMWAIT.

4. Queues. These are used to hold transactions which are waiting to get service at a facility. A queue is serviced by one facility, or a group of facilities.

5. Common Variables. These are variables used by transaction generators and facilities to hold data between activations and to pass information between entities in the model.

The system maintains some status on transactions by type, on facilities and on queues. These are available to the user at the completion of the simulation.

The system routines whose use affects vectorizability are described following.

1. The "SEND" routine. This is used by facilities and transaction generators to insert transactions into queues. Multiple executions will result in multiple copies of the transaction being created.

2. The "COLLECT" routine. This is used to reduce the number of copies of the transaction by one, or if there is only one copy it has the same effect as the "SEND" routine. This routine functions by searching all of the storage area for a given transaction type.

3. The "START" routine. This is used by transaction generators and facilities to enable (activate) transaction generators and facilities. If the affected routine is already active, it has no effect.

4. The "STOP" routine. This is the inverse of the "START" routine.

The following two constructs limit vectorizability when used in the definition of a model system to be simulated.

Any routine (facility or transaction generator) which "SEND"s a transaction to the front of a queue. Only those queueing disciplines which yield fixed ordering in queues lead to vectorizable code.

We also need the restriction that the "COLLECT" routine only searches the queues served by the facility doing the "COLLECT". If this restriction is not obeyed then all facilities servicing transaction types which have some facility in the simulation doing a "COLLECT" on transactions of that type lose vectorizability.

It should be pointed out that some of the operations in the simulation model are not vectorizable. At the simulation system level the non-vectorizable operations include those that maintain statistics for the various queues. This is because the statistics for the (n)th transaction are a function of the numbers for the (n-1)th transaction. The pipeline used for vector computations accepts a new set of operands before the results from the previous set are available. Therefore any operation which uses the (n-1)th result to determine the (n)th result is not vectorizable. An example is the waiting time in a queue. It is determined by the departure and service times of the previous transaction. There are other such instances and these must be handled in scalar mode.

Analysis for Vectorizability: A Graph Algorithm

The algorithm for determining the vectorizability for simulation evaluation of a given model is given following. It results in the construction of a graph. The input is the user description of the simulation.

*Avinash Chandak is currently employed at the World Bank in Washington, D.C.

0190-3918/83/0000/0359$01.00 © 1983 IEEE

1. Draw a node for each of the queues, transaction generators, facilities and common variables and mark each node "facility", "queue", "transaction generator" or "common variable" as appropriate.

2. For a set of facilities in a facility group, draw an arc from a facility which has no outgoing arc to another which has no incoming arc. This should result in a cycle connecting all the facilities in a facility group.

3. For all facilities and transaction generators, draw an arc to each queue to which it "SEND"s transactions.

4. For all queues, draw an arc to each facility that services the queue.

5. For each common variable, draw an arc to a facility or transaction generator if it is used to: (i) change a local variable, (ii) change a different common variable, (iii) change a transaction variable, (iv) affect the flow of control inside the facility or transaction generator (e.g., in an IF statement), or (v) affect the flow of a transaction inside the simulation (e.g., in a "SEND").

6. For each facility and transaction generator, draw an arc to each common variable whose value is changed at some point inside the facility or transaction generator.

7. For every "START" or "STOP" operation, create a new node. Draw an arc from the facility or transaction generator doing the "START" or "STOP" to the new node and from the new node to the facility or transaction generator being "START"ed or "STOP"ped. Mark the new node with a "START" or "STOP".

8. For every facility which specifies a service time of zero, merge the nodes for the queues serving the facility into the facility. Delete any arcs forming a self loop for such facilities.

9. For every facility which does a "COLLECT", create a new node, mark it "COLLECT", draw an arc from the facility to the new node and draw an arc from the new node to the facility.

The algorithm given above can be implemented as part of the preprocessor described in [1]. The conditions precluding vectorizability are:

1. Any node (representing a facility or transaction generator) with more than one arc entering it cannot be vectorized.

2. If there exists a cycle in the graph then the operation of any node in that cycle cannot be vectorized.

3. For any path in the graph containing a "START" or "STOP" node, any node in that path cannot be vectorized.

The arcs in the graph represent information flow in the simulation. Information from various sources can interact at the nodes representing facilities and transaction generators. This means that a node with multiple arcs entering it must synchronize the information from the different sources, and, in general, cannot operate in the vector mode. It may be possible to vectorize part of the operation at that node, but it cannot be totally vectorized. A cycle means that all the nodes in the cycle must be synchronized, and none of them can be vectorized. A path containing a "START" or "STOP" node means that both the routine doing the "START" or "STOP" and the routine being affected must be synchronized and that both must know the correct simulation time. In a vectorized simulation, various routines can be operating at different simulation times, as long as their operations do not interact.

It is possible that the graph is not connected. This means that the information and flow represented by the different pieces of the graph do not interact at all, and can be implemented as different problems.

The original model represented the simulation as a set of operations on single transactions. The default unit of information in the vectorized model is a vector of transactions. All operations are carried out on vectors. Where needed, operations can be carried out on a single entry in a vector.

Each common variable will be represented as two vectors, one containing the value and another containing the corresponding simulation time. If we have the situation shown in Figure 1, then a scalar procedure is needed to convert the vector, because the length and times of the vector produced by the facility writing the common variable need not be the same as the length and times of the vector needed by the facility reading the common variable.

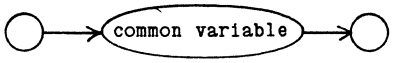


Figure 1:
A Structure Requiring Scalar Conversions

The unlabelled nodes are facilities or transaction generators.

For the situation shown in Figure 2, a scalar procedure is needed to merge the transaction vectors into one vector. The resulting vector is then passed to the facility for processing.

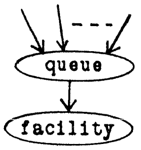


Figure 2:
Another Structure Requiring Scalar Conversions

The algorithm for vectorization actually points out situations in the simulation which cannot be vectorized. It assumes that vectorization is possible unless something prevents it.

The amount of vectorization strongly depends on the model representation of the system being simulated. An examination of the graph might suggest changes to the simulation that enhance vectorizability. In general, real time in the problem being simulated is mapped onto simulation time. If this leads to a complex model, then a simulation where real time is not mapped onto simulation time should be considered. This alternative is possible when the problem does not have queueing and competition for resources.

Monte Carlo Particle Transport: A Completely Vectorizable Problem

Monte Carlo solution of particle transport problems [5] can be cast as a discrete event simulation problem. Brown, Callahan and Martin [2] showed from examination of an actual code that it could be almost completely vectorized. This section applies the algorithm developed in the last section to this problem to illustrate analysis for vectorizability.

If real time is mapped onto simulation time for the particle transport problem, then we have a very complex simulation. We represent particles as transactions and run the simulation so that the particles undergo collisions, absorption or splitting at different times. Because simulation time and real time correspond, a queue in which insertion at an arbitrary point is possible is needed. This is done in the simulation model of [1] by means of a number of queues of different priority classes. This scheme would require a very large number of queues.

There is no competition for resources in this problem and hence there is no need for queues in the simulation. Consider a simulation model in which a transaction generator produces particles at random simulation times. In this model real time is not mapped onto simulation time but is represented as a transaction parameter. A facility processes each particle through all its events until it and all particles it creates (by splitting) are absorbed. It does this in zero simulation time. The particles are represented by x, y and z coordinates and x, y and z velocity components and time. The environment is represented by a set of constant global variables. The algorithm is applied to this model and the resulting graph is shown at each step. The resulting graphs of Figure 3 show

that this form of the particle transport problem can be vectorized.

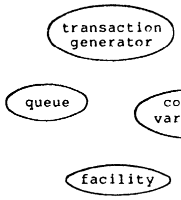


Figure 3.a: The Graph after Step 1

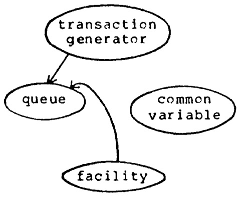


Figure 3.b: The Graph after Step 3

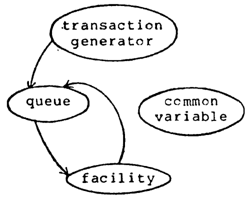


Figure 3.c: The Graph after Step 4

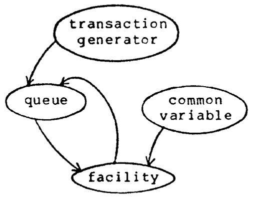


Figure 3.d: The Graph after Step 5

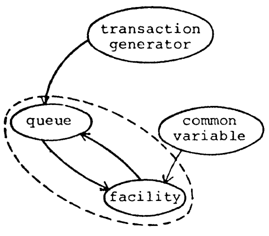


Figure 3.e: The Graph after Step 6

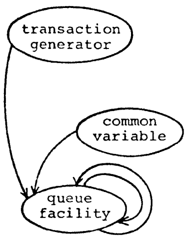


Figure 3.f: The Graph after Step 7

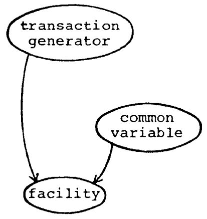


Figure 3.g: The Graph after Step 8

The resulting graph indicates that the problem is not totally vectorizable. It needs a scalar routine to convert the common variable into a vector of the appropriate length. Note, however, that the common variable node has no incoming arcs. If a common variable has no incoming arcs, it means that variable is set up during initialization and never changed after that. It can be treated as a constant.

This means that in the graph we can delete all arcs going out from common variable nodes which have no incoming arcs.

The graph resulting after that transformation is given below, and it indicates that the particle transport problem is totally vectorizable.

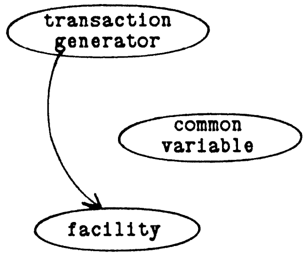


Figure 4: The Graph Structure for the Particle Transport Problem

Implementation Considerations

A simulation program structured to take advantage of vectorizability present in a given model may look very different from a conventionally structured simulation program. Each transaction generator action will produce a vector of transactions. Each activation of a facility will process a vector of transaction steps and each SEND execution will move vectors of transactions between facilities and queues or facilities and facilities. In addition to the speed-up which will derive directly from vectorization, there will also be a major saving in execution time from the deletion of the large number of subroutine calls normally required to generate and process a transaction. There will be only a single subroutine call to process an entire vector of transaction steps instead of a call per transaction step.

There will, however, be an additional storage cost. The vectors of transactions will have to have storage at each queue and facility or else sharing will have to be designed into the code structures.

Acknowledgements

This work was sponsored by grants by the Control Data Corporation and by the Department of Energy under Grant Number DE-AS05-81ER10987 and by the National Science Foundation under Grant Number MCS-8116099.

References

[1] Avinash R. Chandak, "A Study of the Application of Vector Instructions to Simulation", Master's Report, Department of Computer Sciences, The University of Texas at Austin, May 1983.

[2] Forrest B. Brown, Donald A. Callahan and William R. Martin, "Investigation of Vectorized Monte Carlo Algorithms", Working Paper, Department of Electrical and Computer Engineering, Department of Nuclear Engineering, The University of Michigan, 1981. (A Working Paper presented at the Conference on High Speed Computing, Gleneden Beach, Oregon, March 30-April 3, 1981.)

[3] P. I. Georgiadis, M. P. Papazoglou, D. G. Maritsas, "Towards a Parallel SIMULA Machine", in Conference Proceedings, the 8th Annual Symposium on Computer Architecture, pages 263-278. IEEE Computer Society and the ACM, 1981.

ANALYSIS OF BACKWARD ERROR RECOVERY FOR CONCURRENT PROCESSES WITH RECOVERY BLOCKS

Kang G. Shin and Yann-Hang Lee

Computing Research Laboratory
Department of Electrical and Computer Engineering
The University of Michigan
Ann Arbor, Michigan 48109

ABSTRACT

Although backward error recovery with recovery blocks (RB's) has received considerable attention from many researchers, no attempt has been made to structure its implementation alternatives and then to evaluate/analyze their effectiveness. In this paper we categorize three different methods of implementing RB's. These are the asynchronous, synchronous, and the pseudo recovery point (PRP) implementations. We have developed probabilistic models for estimating (i) the interval between two successive recovery lines for asynchronous RB's, (ii) mean loss in computation power for the synchronous method, and (iii) additional overhead and rollback distance in case PRP's are used.

1. INTRODUCTION

The best known technique of backward error recovery, the *recovery block* (RB), was proposed by Horning [1] and Randell [2]. It is a sequential program structure that consists of an acceptance test (AT), a recovery point (RP), and alternative processes for a given process. In case an error is detected or the AT fails, the process rolls back to an old states saved at the previous RP and executes one of the other alternatives. Unfortunately, for cooperating concurrent processes the rollback of a process may cause other processes to roll back (this phenomenon is called *rollback propagation*) because of process interactions and imperfect checking of global correctness. This rollback propagation continues until it reaches a *recovery line* [3] at which a globally consistent state does exist. In the worst case, an avalanche of rollback propagation (called the *domino effect*) can push the processes back to their beginnings. The interval between the restart point and the time point at which an error is detected, called the *rollback distance*, can be used to represent the computation loss in rollback recovery.

The domino effect and rollback propagation are the major obstacles in implementing the recovery block scheme for concurrent processes. Furthermore, decision on rollback propagation and determination of recovery lines will become more complex though they can be made in a centralized [4] or decentralized manner [5,6].

Several refinements have been proposed to overcome the drawbacks in this recovery block scheme. One approach is to put concurrent processes into a controlled scope, i.e., to synchronize the occurrence of acceptance tests. Randell [2] has suggested the *conversation scheme* which requests every cooperating concurrent process to leave its acceptance test at the same moment (called *test line*). Other mechanizations of the conversation scheme on the basis of the same concept but with more flexibility have been devised by Kim [7]. Synchronized rollback recovery schemes for transactions using a two-phase commitment protocol or transaction ordering are also studied in [8,9]. Another approach is to save additional states based on the occurrence of interactions; for example, the branch recovery points [10] and the system defined checkpoints (SDCP) [11].

In this paper we propose to employ *pseudo recovery points* [1] (PRP's) to alleviate the rollback propagation problem by allowing a process to restart at a PRP in case the process is forced to roll back by others as a result of rollback propagation. Therefore, we can classify these refinements into two categories, *synchronized recovery blocks* and *pseudo recovery points*, providing a contrast with the third category called *asynchronous recovery blocks*. To implement the rollback recovery schemes, we have to consider various trade-offs between these three categories and the characteristics of concurrent processes. It is necessary to perform quantitative analyses for estimating the mean amount of computation undone in case processes roll back, the optimal interval between two successive synchronizations, the mean size of memory space required to save states, etc.

In the following section, several assumptions are discussed and then a model for asynchronous recovery blocks is introduced. Using this model, we employ simulations to present the probability distribution of the interval between two successive recovery lines. In Sections 3 and 4, the synchronization method and the implantation of pseudo recovery points are evaluated respectively. The paper concludes with Section 5.

2. EVALUATION OF ASYNCHRONOUS RECOVERY BLOCKS

Let us consider the history diagram in Figure 1 to illustrate the activities of cooperating concurrent processes P_i, $i=1,2,...n$. Let set $A\subset\{1,...,n\}$, i.e. a subset of the indices of concurrent processes and let RP_j^i be the j-th recovery point of P_i. Then one may find a combination of RP_j^i for all $i\in A$, which forms a recovery line for set A, denoted as RL_r^A for the rth recovery line. For simplicity superscripts in representing recovery lines will be omitted in the sequel as long as that does not result in ambiguity. The interval between two successive recovery lines RL_r and RL_{r+1} in process P_i, $i\in A$ is a random variable and denoted by X_r^i. Since a recovery line provides globally consistent states to all members of process set A, it is reasonable to assume that X_r^i is stochastically identical for all $i\in A$. Thus, X_r is used to represent the interval between the r-th and $(r+1)$-th recovery lines.

The work reported here is supported in part by National Aeronautics and Space Administration Grant No. NAG 1-296. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not necessarily reflect the view of NASA.

[1] *We call it a pseudo recovery point (PRP) since there is no acceptance test before the saving of process state at a PRP. The states recorded at PRP's may have been contaminated and thus can not be used to recover a failed process.*

0190-3918/83/0000/0362$01.00 © 1983 IEEE

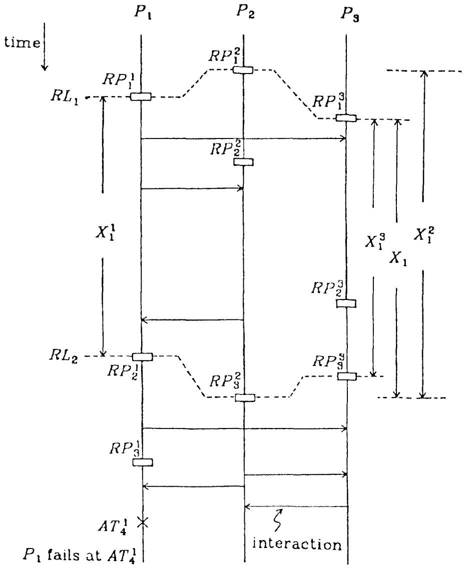


Figure 1. A History Diagram of Occurrence of Interactions and Recovery Points

2.1. Modeling Assumptions

We make the following assumptions in our subsequent analyses.

1. *Autonomous Processes:* Cooperative autonomy is regarded as the most important requirement in distributed processing. Each process should be executed according to its own program and environment, almost as if there were no processes to interfere with. Thus, processes will transmit messages or establish their recovery points independently of other processes.

2. *Perfect Acceptance Test:* Acceptance tests should detect all errors within the local process during the execution of recovery blocks and thus ensure the correctness of local execution. At least, the computation results that have passed the acceptance test should be "acceptable"[3]. However, the local acceptance test may or may not detect external errors or erroneous messages since a local process is not aware of the global system and other processes.

3. *Probability Distribution of Interactions:* Usually, process behavior is modeled as an ordered sequence which in turn is specified by the program and dependent on execution conditions. Even if the processing sequence is given, the interval between two successive interactions is variable due to conditional branches. Locking and waiting at shared resources make it even more uncertain. Nontheless, for both tractability and simplicity we have adopted here constant reference rates in the multiprocessor and exponentially distributed intervals between two successive message transmissions in the computer network. The interval for two successive interactions between P_i and P_j is thus assumed to be exponentially distributed with mean $1/\lambda_{ij}$ and $\lambda_{ij}=\lambda_{ji}$ for all $i,j=1,2,\ldots,n$ and $i\neq j$.

4. *Consistent Communications:* Let two messages m_a and m_b be sent from P_i to P_j. Consistent communications should satisfy : (i) every message sent from P_i to P_j will be received eventually by P_j, and (ii) $m_a$and m_b are received by P_j in the same logical order as that they are sent.

5. *Distribution of Recovery Points:* Because of process independence and the uncertainty of execution conditions, the appearances of recovery points are random and difficult to model. To avoid complexity, establishment of recovery points in a process is assumed to be an independent Poisson process with parameter μ_i for process P_i.

2.2. A Model for Asynchronous Recovery Blocks

Since individual recovery points by themselves may not be sufficient in rollback recovery, we consider only the formation of recovery lines for asynchronous recovery blocks. The requirements of a recovery line for processes P_i for $i=1,2,\ldots n$ can be stated as follows:

1. Each recovery line has to include one recovery point RP_j^i for every process P_i.
2. Let the moment of establishment of the jth recovery point in process P_i be $t(RP_j^i)$ and let $t_q^{ii'}$ be the moment of the qth interaction from P_i to $P_{i'}$. For every pair $(RP_j^i, RP_{j'}^{i'})$ in a recovery line, there does not exist an integer k such that $t_k^{ii'}\in[t(RP_j^i),\ t(RP_{j'}^{i'})]$ if $t(RP_j^i)\leq t(RP_{j'}^{i'})$ $(t_k^{ii'}\in[t(RP_{j'}^{i'}),\ t(RP_j^i)]$ otherwise).

The basic idea underlying the model is to trace the occurrence of both recovery points and interactions. Based on the assumptions in Section 2.1, random variable X_r can be modeled by a continuous-time Markov process starting from a recovery line (RL_r) and ending at the next recovery line (RL_{r+1}). For a set of processes, $\Omega_A=\{P_i\,|\,i=1,2,\ldots,n\}$, two types of states are defined:

(a). End states S_r and S_{r+1}: transitions start from S_r where all processes have formed the rth recovery line, and end at S_{r+1} upon establishment of the $(r+1)$th recovery line.

(b). Intermediate states $S=(x_1,x_2,\ldots,x_n)$, where $x_i=0$ if the previous action of P_i was an interaction, and $x_i=1$ if it was establishment of a recovery point.

Occurrences of interactions and recovery points in a process make the system go through these states. Note that both S_r and S_{r+1} are equivalent to state $(1,1,\ldots,1)$. We can establish the following transition rules:

R1. The system goes to state $(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$ from state $(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)$ with rate μ_i upon establishment of a recovery point in P_i.

R2. The system leaves state $(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_{j-1},1,x_{j+1},\ldots,x_n)$ and enters state

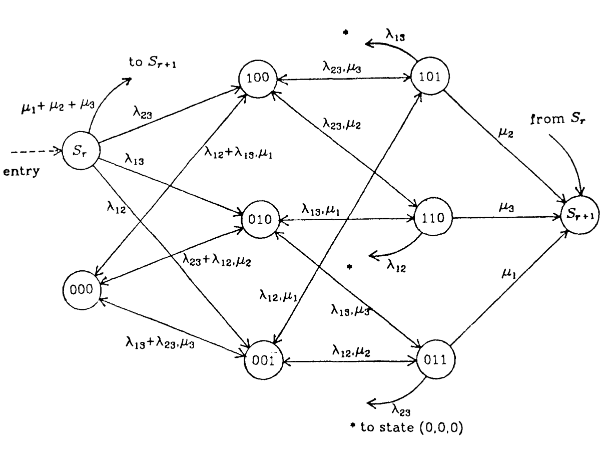


Figure 2. The Model of Asynchronous RB's for 3 Processes

$(x_1,..,x_{i-1},0,x_{i+1},..x_{j-1},0,x_{j+1},..,x_n)$ with rate λ_{ij} if there is an interaction between P_i and P_j.

R3. The system arrives at state $(x_1,..,x_{i-1},0,x_{i+1},..,x_n)$ from state $(x_1,..,x_{i-1},1,x_{i+1},..,x_n)$ with transition rate $\sum_{j \in B_i} \lambda_{ij}$ where $B_i=\{j \mid x_j=0,\ j \neq i \text{ and } j \in A\}$.

R4. The system can transfer directly from state S_r to state S_{r+1} with transition rate $\sum_{k=1}^{n} \mu_k$.

Under these transition rules a Markov model is developed for three processes P_1, P_2 and P_3, and presented in Fig. 2. The single-arrow lines are unidirectional transitions. The double-arrow lines are bidirectional transitions in which left-hand side parameters represent leftward transition rates and right-hand side parameters rightward transition rates.

When $\mu_i=\mu_j=\mu$ and $\lambda_{ij}=\lambda$ for all i, $j \in A$, the model can be simplified since all intermediate states $S=(x_1,x_2,\ldots,x_n)$ containing exactly u 1's in $(x_1,x_2,\ldots,x_n)$ can be replaced by a single state $\hat{S}_u$. A simplified model is obtained under the following transition rules and presented in Fig. 3.

R1'. For $u = 0,1,\ldots,n-1$, the system will move to state $\hat{S}_{u+1}$ from state $\hat{S}_u$ with transition rate $(n-u)\mu$ when a new recovery point is formed.

R2'. For all $u \geq 2$, the system is able to leave state $\hat{S}_u$ for state $\hat{S}_{u-2}$ with rate $\lambda u(u-1)/2$.

R3'. For all $u \geq 1$, there is a transition from state $\hat{S}_u$ to state $\hat{S}_{u-1}$ with rate $\lambda u(n-u)$.

R4'. The system can transfer directly from the entry state S_r to the terminal state S_{r+1} with transition rate $n\mu$.

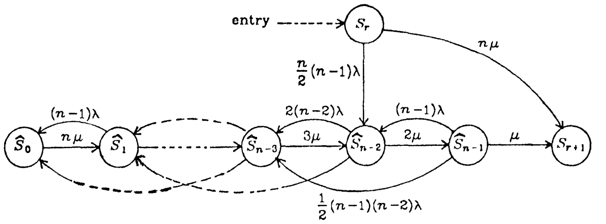


Figure 3. The Simplified Model of Asynchronous RB's for n Processes

2.3. The Analysis of Asynchronous Recovery Blocks

When the occurrences of interprocess communication and recovery point are exponentially distributed, X_r for all r becomes stochastically identical. Let X denote a random variable representing the interval between two successive recovery lines. The probability distribution of X is derived below.

Let the state space $\Psi=\{0,1,2,\ldots,m\}$ where $m=2^n$ be the set of states of the foregoing continuous-time Markov process with the following convention for numbering states:

(a). S_r--> state 0,

(b). an intermediate state $(x_1,x_2,\ldots,x_n)$ --> state $(\sum_{i=1}^{n} x_i 2^{i-1} + 1)$, and

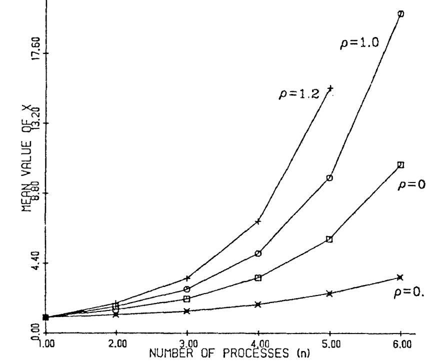
(c). S_{r+1} --> state m.

Then, the Chapman-Kolmogorov equation becomes

$$\frac{d}{dt}\pi(t) = \pi(t)\mathrm{H}$$

where H is the $(m \times m)$ transition matrix $[h(u,v)]$ in which the (u,v) element is the transition rate from state u to state v, and $\pi(t)$ is a vector whose kth element is the probability that the system is in state k at time t. The initial condition is $\pi(0)=[1,0,0\ldots,0]$. The interval between two successive recovery lines, X, is equal to the time needed for transition from state 0 to state m. Therefore, the density function of X, namely $f_x(t)$, is equal to $d(\pi_m(t))/dt$.

Suppose process P_i detects an error or fails tne acceptance test at one of its recovery points RP_j^i, where $j=1,2,\ldots,L_i$. The rollback of P_i may propagate to k processes in the process set, $\Omega_A = \{P_l \mid l \in A\}$ where $A=\{1,2,\ldots,n\}$. Let D_j^k be the rollback distance associated with the k processes and RP_j^i for $j=1,2,..,L_i$. Then, X represents the supremum of these random variables, i.e., $D_{L_i}^n$. In Figure 5, the mean values of X are plotted as a function of n. It shows that X increases drastically when there is an increase in the number of processes involved in the rollback recovery. The density function of X, $f_x(t)$, is plotted in Figure 6. For all the three cases in Fig. 6, there is a sharp pulse near $t=0$, which is due to direct transitions between S_r and S_{r+1} and a longer transition time needed once the system enters intermediate states.



$\rho=(\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} \lambda_{ij})/(\sum_{k=1}^{n} \mu_k)$

$\lambda_{ij}=\lambda$ for all i,j and $\mu_1=\mu_2=\ldots=\mu=1.0$

Figure 4. Mean value of X vs. the number of processes

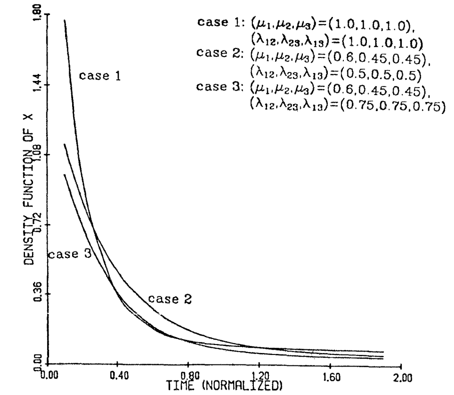


Figure 5. The Density Function of X, $f_x(t)$

3. SYNCHRONIZED RECOVERY BLOCKS

The simplest way of avoiding unbounded rollback propagations is to synchronize the establishment of recovery points during process execution. In this method, interactions are inhibited between any pair of processes during their establishment of recovery points. There are three conceivable strategies in deciding when a synchronization request is to be issued: (1) at a constant interval; (2) when the time elapsed since the previous recovery line exceeds a specified value; or (3) when the number of states saved after the previous recovery line is larger than a prespecified number. The implementation of the first strategy is simple since the synchronization request is issued without any knowledge of the state of execution. Nevertheless, this strategy may become very inefficient since it is possible to make synchronization requests immediately after the formation of a recovery line. For the second and third strategies, rollback distance and the number of saved states are prevented from becoming too large. However, in this case each process must be aware of the occurrence of a recovery line whenever it is established.

Upon receipt of a synchronization request, every process has to prepare for establishing a recovery line and also has to wait for the commitment (for establishing a recovery line) from other processes before it executes an acceptance test. Thus, all cooperating processes perform their acceptance tests at the same instant upon receiving the commitments from all other processes. Let $P_{ij}-ready$, for $j=1,2,\ldots,n$, be the flags in process P_i to indicate commitments from P_j. The steps for synchronization in each process P_i are described as follows:

```
S1. execute "its own normal process" until "acceptance
      test";
S2. set $P_{ii}-ready$ := ON and then broadcast $P_{ii}-ready$;
S3. while not (all $P_{ij}-ready$ = ON) do
        receive messages;
        if a message is $P_{jj}-ready$ then
          set $P_{ij}-ready$ := ON
        else record the message
S4. reset $P_{ij}-ready$ := OFF, for $j=1,2,\ldots,n$ and
      do "acceptance test" and record process states.
```

Establishment of recovery lines upon synchronization requests is shown in Figure 6. Synchronization causes the computation power to be diminished because processes have to wait for the commitment (as in S3). Let y_i be the interval between the receiving of a synchronization request and the moment that process P_i reaches its next acceptance test (in S1). Then, according to the assumptions in Section 2 1, y_i is an exponentially distributed random variable with parameter μ_i. Let $Z=\max\{y_1, y_2, \ldots, y_n\}$. The total loss in computation power is $CL=\sum_{i=1}^{n}(Z-y_i)$. The mean loss becomes

$$\overline{CL} = n\int_0^\infty (1-F_z(t))dt - \sum_{i=1}^{n}\frac{1}{\mu_i}$$

where $F_z(t)$ is the distribution function of Z, and equals $\prod_{i=1}^{n}(1-e^{-\mu_i t})$.

4. IMPLANTATION OF PSEUDO RECOVERY POINTS

In the construction of a recovery block, usually, an acceptance test is a number of executable assessments provided by the programmer and then followed by a state saving. Note that process states can also be recorded upon any other requests if they are considered useful in the rollback recovery. A *pseudo recovery point* (PRP) is defined as a recovery point that is established without a preceding acceptance test and is proposed here as an alternative for avoiding the domino effect in a set of cooperating concurrent processes. With a monitor as the interprocess communication means, Kim [10] and Kant and Silberschatz [11] discussed methods for implanting recovery points in a centralized manner. Similarly, we consider a method for implanting PRP's in the set of cooperating concurrent processes in a decentralized manner.

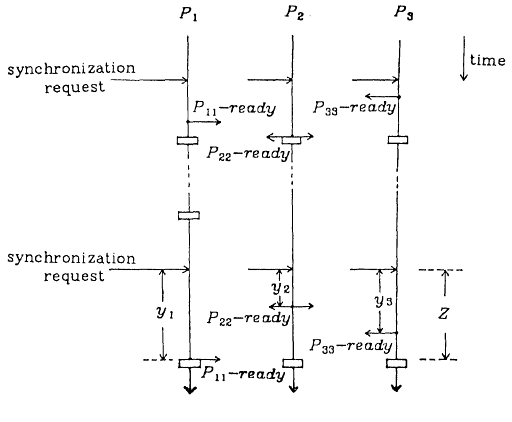


Figure 6. Establishment of Recovery Lines upon Synchronization Requests

To make every recovery point RP_j^i in P_i maximally useful for rollback error recovery, there should be corresponding recovery points in the other processes that have to roll back as a result of the rollback propagation from P_i. If such recovery points do not actually exist, a pseudo recovery point, $PRP_j^{ii'}$, has to be inserted in process $P_{i'}$ for a given RP_j^i in process P_i. Further, in order to avoid the need of tracing recovery points at that particular moment, a PRP is established in each of the other processes involved with RP_j^i. An algorithm for implanting PRP's is given below.

(1). When P_i establishes a recovery point RP_j^i, it broadcasts a PRP implantation request to other processes.

(2). If $P_{i'}$ receives the implantation request, it records its state as $PRP_j^{ii'}$ upon completion of the current instruction without an acceptance test. Then $P_{i'}$ broadcasts its commitment $C_{i'}$.

(3). Every process executes its own normal task after it establishes RP_j^i or $PRP_j^{ii'}$. However, the messages sent to other processes by $P_{i'}$ prior to $C_{i'}$ have to be retained in the state saved.

Assume that process P_i detects an error before establishing RP_{j+1}^i and that this error is local to P_i. The recovery line (called a *pseudo recovery line*, PRL_j^i) formed by RP_j^i and all $PRP_j^{ii'}$'s is able to recover these processes even if the error has already propagated to other processes. However, when the error detected in P_i is due to error propagation from another process, P_l (and therefore not local to P_i), the contents of PRP_j^{il} may have already been contaminated if this error occurred prior to establishing PRP_j^{il}. The restart from the pseudo recovery line formed by both RP_j^i and all $PRP_j^{ii'}$'s may just reproduce the same error. Therefore, rollback propagation may continue until every process involved has rolled back to a pseudo recovery line past at least one of its recovery points. Consequently, the pseudo recovery line allows the processes to have the shortest rollback distance for backward error recovery without synchronization. Note that the pseudo recovery line is now guaranteed to contain correct states of all concerned processes. An algorithm of rollback recovery with these pseudo recovery points is given by:

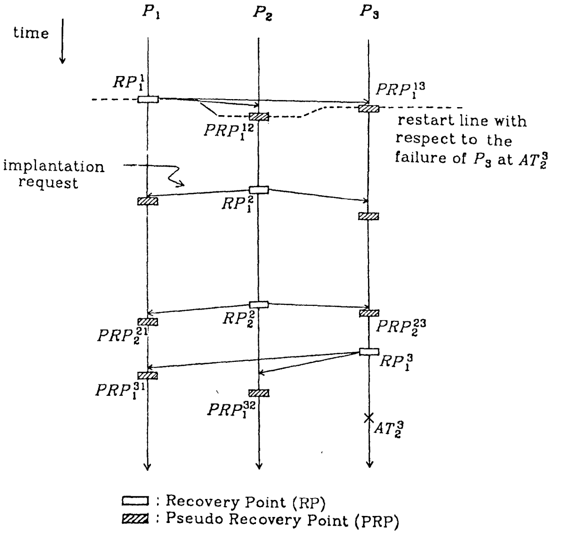
(1). If an error is found in process P_i, set $p := i$ where p is a rollback pointer.

(2). P_p rolls back to its previous recovery point RP_j^p. All processes $P_{i'}$ affected by the rollback of P_p roll back to their respective pseudo recovery points $PRP_j^{pi'}$.

(3). For every affected processes $P_{i'}$, if the rollback has not passed its most recent recovery point, then set $p := i'$ and go back to step 2.

In Figure 7, the establishment of PRP's in processes P_1, P_2, and P_3 is illustrated. When P_3 fails its acceptance test AT_2^3, all processes have to restart from the pseudo recovery line formed by $(RP_1^1, PRP_1^{12}, PRP_1^{13})$ if P_1 and P_2 are affected by the rollback of P_3.

In the above algorithm, we can find that every process needs to preserve a recovery point for restart in case it fails. Also $(n-1)$ pseudo recovery points are needed for a process to form pseudo recovery lines with other processes where n is the total number of concurrent processes. The old RP's and PRP's except those in the pseudo recovery lines $\{PRL_j^i | i=1,...,n$, and RP_j^i is the most recent RP in $P_i\}$ can be purged when a new recovery point is established, thereby reducing storage requirements for each process. Note that rollback distance is bounded by the supremum of $\{y_1, y_2, \ldots, y_n\}$ where y_i is the interval between two successive recovery points of process P_i. The additional time overhead for every recovery point is $(n-1)t_r$ where t_r is the time needed to record the process state. These overheads should be assessed against the gain of process autonomy and avoidance of unbounded rollback propagations.



Note: all occurrences of interactions are omitted

Figure 7. Establishment of Pseudo Recovery Points for Rollback Error Recovery

5. CONCLUSION

We have quantitatively evaluated three different recovery blocks employed in backward error recovery for concurrent processing and have estimated the overhead required to avoid the domino effect when recovery or pseudo recovery points are employed. For both the synchronization method and the implantation of pseudo recovery points, the overheads are largely related to the construction of synchronization, and PRP's. They would become an unacceptable burden when synchronizations and pseudo recovery points are constructed frequently but interprocess communications do rarely occur. At the other extreme, i.e. asynchronous recovery blocks, it may result in a longer rollback distance due to unlimited rollback propagations.

To select a suitable strategy or a combination of these three methods, we have to first examine the properties of concurrent processes such as the amount of interprocess communications and the distribution of recovery points. Then, we weigh the trade-off between the loss of computation power during normal operation and the increase in response time due to rollback recovery. In general, if more knowledge of the execution state in concurrent processes can be obtained, a better strategy for implementing recovery blocks can be derived.

REFERENCES

[1]. J. Horning, et al., "A program structure for error detection and recovery," *Lecture Notes in Computer Science*, Vol. 16, Springer-Verlag, 1974, pp. 171-187.

[2]. B. Randell, "System structure for software fault tolerance," *IEEE Trans. on Software Eng.*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

[3]. B. Randell, P. A. Lee and P. C. Treleaven, "Reliability issues in computing system design," *Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 123-165.

[4]. Y. H. Lee and K. G. Shin, "Rollback propagation detection and performance evaluation of $FTMR^2M$ - a fault-tolerant multiprocessor," *Proc. of Int'l Symp. on Computer Architecture*, 1982, pp. 171-180.

[5]. W. G. Wood, "A decentralized recovery control protocol," *FTCS-11*, 1981, pp. 159-164.

[6]. K. Tsuruoka, A. Kaneko and Y. Nishihara, "Dynamic recovery schemes for distributed processes," *Proc. of Reliability in Distributed Software and Database Systems*, 1981, pp. 124-130.

[7]. K. H. Kim, "Approaches to mechanizations of the conversation scheme based on monitors," *IEEE Trans. on Software Eng.*, Vol. SE-8, No.3, May 1982, pp. 189-197.

[8]. J. N. Gray, "Notes on database operating systems, " *Operating Systems: A advanced course*, edited by R. Bayer, et al., Springer-Verlag, 1979, pp.393-481.

[9]. W. H. Kohler, "A survey of techniques for synchronization and recovery in decentralized computer systems," *Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 149-183.

[10]. K. H. Kim, "An approach to programmer-transparent coordination of recovering parallel processes and its efficient implementation rules," *Proc. of Int'l Conf. on Parallel Processing*, 1978, pp. 58-68.

[11]. K. Kant and A. Silberschatz, "Error recovery in concurrent processes," *Proc. of COMPSAC*, 1980, pp. 608-614.

IMPROVED MULTIPROCESSOR GARBAGE COLLECTION ALGORITHMS

Newman I.A., Stallard R.P., Woodward M.C.
Department of Computer Studies
Loughborough University of Technology
Loughborough, Leicestershire, U.K.

Abstract -- This paper outlines the results of an investigation of existing multiprocessor garbage collection algorithms and introduces two new algorithms which significantly improve some aspects of the performance of their predecessors. The two algorithms arise from different starting assumptions. One considers the case where the algorithm will terminate successfully whatever list structure is being processed and assumes that the extra data space should be minimised. The other seeks a very fast garbage collection time for list structures that do not contain loops. Results of both theoretical and experimental investigations are given to demonstrate the efficacy of the algorithms.

Introduction

A number of previous papers have considered the problems of reclaiming space in a list processing system, dynamically, while processing (by 'mutators') continues [2,3,4,6,7]. Most papers have studied an environment in which one process, the garbage collector, works alongside a second process, the mutator, with the processes normally executing on different processors. However, some algorithms allow extensions in which several garbage collector processes can be active simultaneously. The garbage collector process generally has three phases 'set-up', 'marking' and 'collection'. In the first, all nodes are marked as'garbage', by setting a mark bit (or bits) appropriately (referred to as 'colouring' the node 'white'). In the second all nodes that can be reached from the roots of the list structure are marked as accessible (coloured 'black'). This marking may have several stages in which the mark state (colour) of the node changes which necessitates more than one mark bit. Finally, all the unmarked (white) nodes are added to the free list in the third phase. The set-up phase is typically executed as a by-product of the collection phase. A possible fourth phase, in which all accessible nodes are compacted into the minimum physical space, is not generally considered.

Background

The performance of various garbage collection algorithms has been studied as part of an ongoing investigation into applications of multiprocessor systems in which each processor has its own private memory and, in addition, memory and possibly other resources are shared. A previous paper [4] reported the results of simulation studies on two algorithms (referred to as 'Lamport' and 'Chaining'). A more detailed theoretical and experimental analysis of these algorithms together with a third ('stacking' [7]) has been carried out [5]. The collection phase is identical for all three algorithms and can be efficiently multiprogrammed by partitioning the node space. The investigation has, therefore, centered on the marking phase. Each algorithm was implemented as part of a simple list processor executing on a system comprising four TI 990/10 computers [1]. Results were taken for several types of list structures in a node space in which all nodes were of the same size and were compared with a theoretical analysis of the expected performance of each algorithm.

Analysis

Both the Lamport and Stacking algorithms will guarantee to mark any structure. The former requires a two bit field in each node to permit marking while the latter requires only one bit in the node but also a stack potentially capable of taking pointers to all the nodes in the structure except for one (the original root). The operation of these two algorithms with multiple markers is quite different. The Lamport algorithm effectively requires each marker to access a physically separate portion of the node space. Each marker examines nodes in its own node space until one finds and marks an accessible ('shaded') node and shades its successors. All markers then 'reset', restarting the search of their node space. Thus, with most list structures, only one marker does any useful work between resets. By contrast, markers under the Stacking algorithm traverse the logical structure by taking a pointer to a node from the shared stack, marking the node and placing its successors on the stack. This normally requires only one visit to each node. A highly interconnected list structure slightly modifies the behaviour of both algorithms in that several markers can stack pointers to the same node or can find grey nodes simultaneously.

Although Stacking is a fast algorithm for a single marker it does not work well for multiple markers as a marker must have exclusive access to the shared stack while adding or removing a pointer to a node and this causes substantial delays. By contract, Chaining works efficiently in this case provided the average number of successors of each node is small (either a linear list or a 'curtain' in which root nodes have several successors but subsequent nodes have only one). One controlling parameter on the performance of the Chaining algorithm is the size of the shared sub-root list. If this is large and each marker refills the list whenever it is not full then the Chaining and Stacking algorithms are closely related.

Revised Algorithms

A substantially improved version of the

0190-3918/83/0000/0367$01.00 © 1983 IEEE

Lamport algorithm is obtained simply by allowing each marker to complete its sequential pass through its section of the node space before resetting instead of resetting as soon as one marker has found and coloured a node. This has two advantages. Firstly, several markers may find 'shaded' nodes on each pass, be able to mark them and shade their successors. Secondly, the successors to a node which is marked may themselves be marked in the same pass through the node space.

The introduction of a local stack for each marker with a smaller shared stack (analogous to the subroot list of the Chaining algorithm) enables the Stacking algorithm to utilise multiple markers effectively. Each marker refills the shared stack if it is not full and if no other marker is doing so, otherwise it uses its local stack. This minimises the time spent waiting for access to the shared stack while ensuring that markers always have work available. However, the space taken by the stacks can be quite large.

Results

Type of Structure	Linear List		Curtain		Inter-Connected	
% of occupied nodes	Low		High		High	
Number of markers	1	4	1	4	1	4
Algorithm						
Lamport	2.50	1.50	54.00	18.40	53.80	12.30
Modified Lamport	0.86	0.41	2.28	0.92	2.57	0.90
Chaining	0.08	0.06	0.98	0.26	6.24	1.63
Stacking	0.09	0.10	1.22	0.88	1.62	2.40
Modified Stacking	0.07	0.05	0.70	0.23	1.60	0.42

All algorithms were run on node spaces in which the successors have a random spatial distribution and the results obtained are in seconds of elapsed time. The single marker unmodified Stacking algorithm times are high because of the overhead of entering and exiting a protected region which is provided for the multiple marker case. The speed up of more than four for the Lamport algorithm for highly interconnected structures is due to several nodes being marked 'simultaneously' reducing the number of passes required.

The times given above were with no mutators active. Both revised algorithms, however, have been shown to be reliable in conjunction with mutators, with the effect on the time taken to mark depending upon the actions being performed by the mutators.

Further Work

Two further aspects of multiprocessor garbage collection are currently being studied. The first is the efficacy of adding a compaction phase to the garbage collection process. The second is a comparison of the marking algorithms with algorithms in which the number of pointers to a node is recorded in the node itself (reference count scheme).

References

[1] R.H. Barlow et al, "A Guide to Using the Neptune Parallel Processing System", Dept. of Computer Studies, Loughborough University of Technology, Loughborough, Leics., U.K. (1981).

[2] E.W. Dijkstra et al, "On the Fly Garbage Collection: An Exercise in Cooperation", CACM, Vol. 21, No. 11 (1978), pp.966-975.

[3] L. Lamport, "Garbage Collection with Multiple Processors: An Exercise in Parallelism", Proceedings of the International Conference on Parallel Processing, Walden Woods (1976), pp.50-54.

[4] I.A. Newman, M.C. Woodward, "Alternative Approaches to Multiprocessor Garbage Collection", Proceedings of the International Conference on Parallel Processing, IEEE Computer Society (1982), pp.205-210.

[5] I.A. Newman, R.P. Stallard, Woodward M.C., "Performance of Parallel Garbage Collection Algorithms", Report No.166, Dept. of Computer Studies, Loughborough University of Technology, Loughborough, Leics., U.K.(1982).

[6] G.L. Steele, "Multiprocessing Compactifying Garbage Collection", CACM, Vol. 18, No. 9, (1975), pp.495-508.

[7] P.L. Wadler, "Analysis of an Algorithm for Real-Time Garbage Collection", CACM, Vol. 19, No. 9, (1976), pp.491-500.

EFFICIENCY OF FEATURE DEPENDENT ALGORITHMS FOR THE PARALLEL PROCESSING OF IMAGES

by

T. N. Mudge and T. Abdel-Rahman

Computing Research Laboratory
Department of Electrical and Computer Engineering
University of Michigan
Ann Arbor, MI 48109

Abstract--In this paper the concept of feature (in)dependent image processing algorithms is defined. A large class of image processing computers characterized by multiple processor-memory subsystems is efficient when dealing with feature independent algorithms but less efficient when dealing with feature dependent algorithms. Typically such machines are required to perform both types of algorithms. This paper is a preliminary attempt to provide a framework within which to model feature dependent algorithms, and to, for example, quantify the inefficiency that can occur when they are executed on the above type of parallel image processors.

Keywords--feature dependent algorithms, image processing, parallel processing.

1. Introduction

The economics of modern digital integrated circuit technology no longer restricts the designers of digital systems to the classical serial interpreter typified by the von Neumann uniprocessor architecture. This trend away from conventional machines is particularly well developed in the field of image processing where the large data sets (64K bytes to 4M bytes per image) and the high processing rates (near term predictions of 1 to 100 billion operations per second have been made in [1]) make special purpose machines an economic necessity [2]. A number of people have proposed/constructed special purpose machines for image processing. These are surveyed in [3-5].

An architectural characteristic of most of these special purpose image processors is a large number of processors working in parallel. Parallel processing is a natural strategy for dealing with the large data sets and high processing rates encountered in image processing applications; furthermore, the nature of the data and the nature of many of the algorithms make parallel processing particularly attractive. The data is usually a large two dimensional array, and many of the low level image processing algorithms can be decomposed into a large number of concurrent *neighborhood* operations. Examples include: various filtering algorithms such as smoothing to reduce high frequency noise and median filtering to reduce salt-and-pepper noise; edge detection algorithms that use operators such as the Sobel operator and the Hueckel operator; and various coding algorithms such as block truncation coding and cosine transform coding.

A natural architecture for the above class of image processing algorithms is a multiprocessor in which equal subimages are assigned to separate processors for processing. For the purpose of this discussion we will classify such processors as *multiple subimage processors* (MSP's). As might be expected, a large number of the proposed/constructed special purpose image processors can be viewed as MSP's. Figure 1 shows a block diagram of a generic MSP. Subimage *i* is handled by its own processor-memory subsystem, processing element *i* (PE_i). The PE's can communicate through some form of interconnection network (ICN). Specific examples of MSP's include: the proposed PASM architecture [6], which plans to employ multi-path routing-networks to connect a set of 1024 PE's; CLIP4 [7], a 96 × 96 array of simple bit-processors, each with a 32 bit RAM and an ICN that connects nearest neighbors in the array; the Distributed Array Processor [8], a 64 × 64 array of processors with 4K-bit storage per processor and an ICN that connects nearest neighbors in the array and provides a bus per row and column; the Massively Parallel Processor [9], a 128 × 128 array of processors with 1K-bit storage per processor and an ICN that connects nearest neighbors; and the Adaptive Array Processor [10], whose building block is a single chip 8 × 8 array with 96 bits of storage per processor.

In general, MSP's are highly efficient at performing neighborhood operations such as those listed above. These types of operations are an important subclass of what we will term *feature independent* image processing algorithms. Feature independent algorithms are characterized by equal processing per pixel. In other words, each pixel receives the same amount of processing

This work was supported in part by AFOSR grant F49620-82-C-0089.

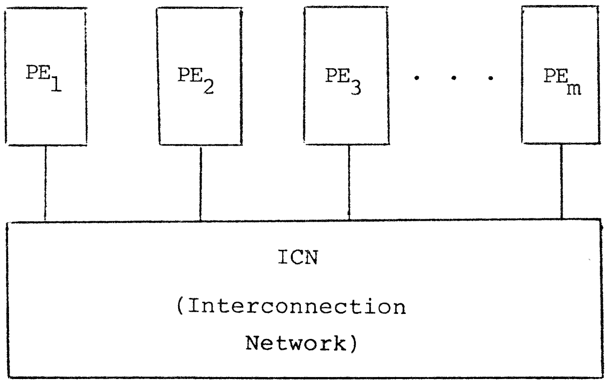


Figure 1. Generic MSP.

regardless of whether or not it is part of a feature of interest such as a line segment. As well as many neighborhood operations there are other algorithms such as histogramming and the Fourier transform which are feature independent. Unlike neighborhood operations these algorithms require significant amounts of data to be moved between processors. The effectiveness of MSP's at performing them is dependent on the bandwidth of the ICN shown in Figure 1. A multiprocessor like PASM with a high bandwidth ICN can perform such algorithms relatively easily [11-13]. Therefore, the concept of a multiprocessor in which equal subimages are assigned to separate processors for processing is also a natural way of handling the complete range of feature independent algorithms, provided the ICN is appropriate for the types of feature independent algorithms anticipated.

Although the above concept is natural for feature dependent algorithms, it becomes less attractive for *feature dependent* image processing algorithms. Feature dependent algorithms are characterized by unequal amounts of processing per pixel. This might arise when a pixel is part of a feature of interest and because of that requires separate treatment. A simple example of a feature dependent algorithm is contour tracing; only edge pixels are involved in the algorithm. In an image processing application the initial sequence of algorithms involves mostly feature independent algorithms because they are concerned with general image enhancement and potential feature location. The subsequent sequence of algorithms is much more likely to involve feature dependent algorithms because specific features are sought from the set of potential locations.

Consider processing an N-pixel image on an MSP machine having m PE's. In normal MSP operation the image is divided into N/m subimages of equal size, and each subimage is processed by a single PE. However, in the case of feature dependent algorithms the image should be divided into subimages of equal *interest*, i.e., subimages having equal numbers of pixels of interest. If, in the case of feature dependent algorithms images are divided into subimages of equal size, some PE's will receive fewer pixels of interest. This uneven distribution of work will result in some PE's being idle during part of the algorithm. Dividing the image into subimages of equal interest requires that the distribution of pixels of interest over the image can be calculated. This is not always possible. On the other hand, it may be possible, but the calculation and the redistribution on the basis of interest may involve more computation than that lost through the inefficiency of having some PE's idle during part of the algorithm.

This paper is a preliminary attempt to provide a framework within which to model feature dependent algorithms, and to, for example, quantify the above inefficiency to assist in decisions about image distribution among PE's.

The following section develops a mathematical model of feature dependent algorithms. Section 3 tests it using some real image data with edge pixels as the pixels of interest. Section 4 concludes the discussion.

2. Mathematical Model of Feature Dependent Algorithms

Consider an N-pixel image and an m-PE MSP system. Assume that the pixels of interest occur randomly in the image and that the probability of a pixel being of interest is p regardless of its position. Assume that the MSP system is executing an image processing algorithm on the image. Let the time to complete the algorithm be a function, f, of the number of pixels of interest in the image, i.e., the algorithm is a feature dependent one.

For the single PE case (m=1) the expected value of the execution time, T_1, is given by:

$$T_1 = f(Np) \qquad (1)$$

For the m-PE case assume that the image is divided among the m PE's on an equal size basis. Each PE holds an $n = N/m$ pixel subimage. Let X_i to be the random variable describing the number of pixels of interest in subimage i, i=1,2,...,m. From the above assumption that the probability of a pixel being of interest is p regardless of its position, it follows that the X_i's are identically independently distributed (i.i.d) random variables with a binomial distribution (see Figure 2).

Let $T_{\max}$ be the expected value of the maximum execution time among all PE's. Since the algorithm is not finished until all the m PE's have completed the work in their subimage, it follows that:

$$T_m = f(E[X_{\max}]) \qquad (2)$$

Where:

$$X_{\max} = max(X_1, X_2, \ldots\ldots, X_m) \qquad (3)$$

To evaluate T_m consider the following. Let p_j be the probability of exactly j pixels of interest occurring in subimage i:

$$Pr\,\{X_i = j\,\} = p_j = \binom{n}{j}\; p^j\,(1-p)^{n-j} \qquad (4)$$

Let q_j be the probability of greater than j pixels of interest occurring in subimage i:

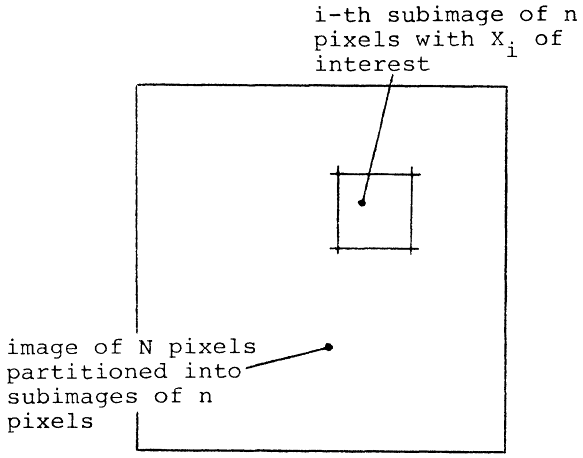

i-th subimage of n pixels with X_i of interest
image of N pixels partitioned into subimages of n pixels

Figure 2. A subimage and its associated random variable.

$$Pr\ \{X_i > j\} = q_j = \sum_{r=j+1}^{n} p_r \qquad (5)$$

Then:

$$q_j = \sum_{r=j+1}^{n} \binom{n}{r} p^r (1-p)^{n-r} \qquad (6)$$

Let $P(z)$ be the generating function for the sequence p_j, $j=0,1,\ldots,n$:

$$P(z) = p_0 + p_1 z + \ldots\ldots. + p_n z^n \qquad (7)$$

Let $Q(z)$ be the generating function for the sequence q_j, $j=0,1,\ldots,n$:

$$Q(z) = q_0 + q_1 z + \ldots\ldots. + q_n z^n \qquad (8)$$

From (7) and (8) it follows that:

$$Q(z) = \frac{1-P(z)}{1-z} \qquad (9)$$

Equation (9) can be verified by equating the z coefficients on both sides of the equation:

$$1 - P(z) = (1-z)Q(z) \qquad (q_n = 0) \qquad (10)$$

See [14].
Differentiating $P(z)$ with respect to z yields:

$$P'(z) = p_1 + 2p_2 z + \ldots\ldots. + np_n z^{n-1} \qquad (11)$$

Evaluating $P'(z)$ at $z=1$ yields:

$$P'(1) = p_1 + 2p_2 + \ldots\ldots. + np_n \qquad (12)$$

The right hand side of the above equation is simply $E[X_i]$. Thus:

$$E[X_i] = P'(1) \qquad (13)$$

Differentiating both sides of (10) yields:

$$-P'(z) = -Q(z) + (1-z)Q'(z) \qquad (14)$$

Evaluating (14) at $z=1$ yields:

$$P'(1) = Q(1) \qquad (15)$$

Comparing to (13) gives:

$$E[X_i] = Q(1) \qquad (16)$$

Next consider $Pr\{\ X_{\max} \leq j\ \}$:

$$Pr\{\ X_{\max} \leq j\ \} = Pr\{\ X_1 \leq j \text{ and } X_2 \leq 2 \cdots \text{ and } X_m \leq j\} \qquad (17)$$

Since the X_i's are i.i.d, (17) reduces to:

$$Pr\{\ X_{\max} \leq j\ \} = \left[Pr\{\ X_i \leq j\ \}\right]^m \qquad (18)$$

For any i.
Using the relation:

$$Pr\{\ X_{\max} > j\ \} = 1 - Pr\{\ X_{\max} \leq j\ \} \qquad (19)$$

Gives:

$$Pr\{\ X_{\max} > j\ \} = 1 - \left[Pr\{\ X_i \leq j\ \}\right]^m \qquad (20)$$

But from (16):

$$E[X_{\max}] = Q_{\max}(1) \qquad (21)$$

And, by definition:

$$Q_{\max}(z) = Pr\{\ X_{\max} > 1\ \} + Pr\{\ X_{\max} > 2\ \} + \ldots + Pr\{\ X_{\max} > m\ \} \qquad (22)$$

Therefore, substituting (20) into (22) gives:

$$T_m = f\left[\sum_{k=0}^{n} 1 - \left[P\{\ X_i \leq k\ \}\right]^m\right] \qquad (23)$$

Where $Pr\{\ X_i \leq k\ \}$ is given by:

$$Pr\{\ X_i \leq k\ \} = \sum_{r=0}^{k} \binom{n}{r} p^r (1-p)^{n-r} \qquad (24)$$

Notice tha the value of $T_{\max}$ is independent of i because the X_i's are i.i.d.

Following the usual arguments (see [15]) the efficiency E can be defined in terms of T_1 and T_m by:

$$E = \frac{T_1}{m\,T_m} \qquad (25)$$

Thus the efficiency of executing feature dependent algorithms can be determined from (1), (23), (24) and f, the function that describes the time to complete the algorithm.

3. Experimental Results

In an attempt to test the above results the following experiment was carried out on a set of images of industrial parts. These images were obtained from the General Motors database for the industrial bin of parts problem [17]. The names of the ones used are listed in Table 1.

The Sobel edge operator was applied to the above images. A pixel was defined to be of interest if and only if it was on an edge. The resulting image was thresholded and the number of edge points (number of pixels of interest) was computed. The threshold value was chosen to give a "good" edge image. All the images are 256x256 with 256 gray levels. The number of pixels of interest in each image and the value of p are also shown in Table 1. The value of p was estimated as the number of pixels of interest divided by the total number of pixels in the image.

The images were divided into subimages of equal sizes and the expected value of the maximum number of pixels was obtained experimentally. The experimental value obtained was compared with its theoretical value obtained from equation (23) with $f=1$, for various values of m. Those results are shown in Graph 2. It can be seen that there is a fairly good agreement between the theoretical results and the experimental results when the the features are edge pixels. The lower of the two curves is the theoretical one. This error is due the our assumption that the probability of a pixel being of interest is not related to its position. In the case of edge pixels this is clearly not so as they cluster in lines. Clustering moves the experimental line higher.

In the case of specific features better results might be obtained if a more accurate stochastic model of the features distribution can be developed. For example, more accurate models of edge pixel distributions have been developed [18], however they apply only to edges and computing $T_{\max}$ for them appears to be a problem.

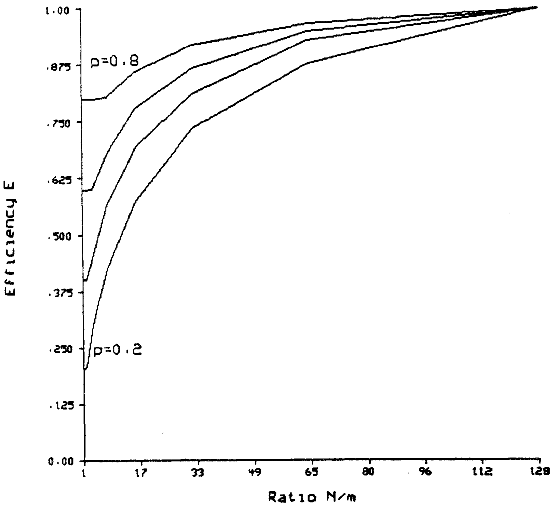
Image Name	No. of Edge Pixels	p
bin1.piv	7732	.118
bin1.piw	12205	.186
bin3.piv	9831	.150
bin5.piz	8032	.123
bin8.piv	5600	.089
yoke1.pit	4421	.064
yoke2.pit	5241	.080
yoke3.pit	8018	.122
rod1.pit	8768	.134
bin1.piw	15822	.241

Table 1.

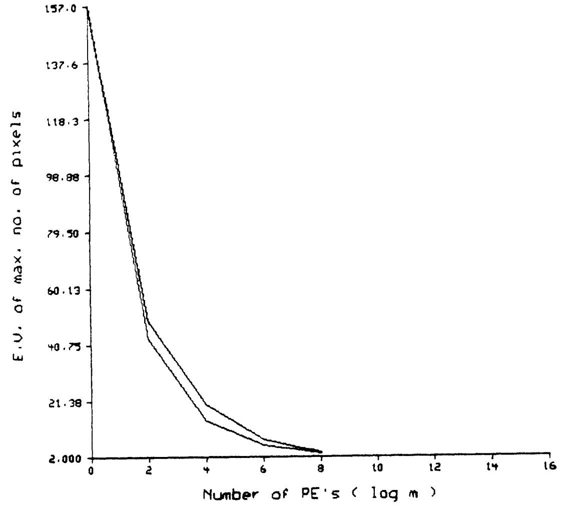
Graph 1 shows the variation of the efficiency, E, as a function of the ratio N/m for p = 0.2, 0.4, 0.6, 0.8.. The graph was plotted by assuming f to be linear. A more realistic function would depend on the specific feature dependent algorithm being considered. However, linear does appear to be a reasonable assumption for a large class of algorithms. For example, a relatively complicated feature dependent algorithm such as the Generalized Hough transform [16] is approximately linear: for each pixel of interest no more than a fixed number of accumulators have to be updated.

If care is taken T_m can be evaluated in $O(n)$ time. The term from (24) should not be evaluated from scratch for each value of k. Also, for large values of n the terms on the right hand side of (24) can be approximated by a Poisson distribution whose terms can in turn be evaluated using Stirling's formula and logarithms.

Several points can be deduced from Graph 1. The efficiency tends to p as N/m goes to 1. This agrees with intuition: if there were as many PE's as pixels, p would be the fraction likely to contain an interesting pixel, and only this fraction would have any work. For very low values of p (<<0.2) the efficiency can drop drastically for MSP's processing images that have less than an order of magnitude more pixels than they have PE's. For example, PASM with 1024 PE's will operate at less than 40% efficiency on images of 64 x 64 pixels if p=0.4. On the other hand if the images are 256 x 256 the efficiency jumps to over 80% for the same value of p. Clearly, for high efficiency the image should contain several orders of magnitude more pixels than the MSP has PE's.



Graph 1. E versus N/m



Graph 2.

4. Conclusions

This paper has presented a preliminary attempt to provide a framework within which to model feature dependent algorithms, and to, for example, quantify the inefficiency that can occur in MSP's when subimages of equal size are distributed among the PE's.

The mathematical model was simple enough to allow key terms such as $T_{\max}$ to be efficiently computed without compromising the accuracy of the result. Future work might examine how E can be determined if more complex, say Markov, models were used for the features of an image.

5. References

[1] R. Reddy and R. W. Hon, "Computer architectures for vision," in *Computer and Sensor-Based Robots*, G. G. Dodd and L. Rossol (Eds.), Plenum Press, New York, 1979, pp. 169-186.

[2] T. N. Mudge and E. J. Delp, "Special purpose architectures for computer vision," *Proc. of the 15-th Hawaii International Conf. on Systems Science*, Jan. 1982, pp. 378-387.

[3] P. E. Danielsson and S. Levialdi, "Computer architectures for pictorial information systems," *Computer*, vol. 14, no. 11, Nov. 1981, pp. 53-67.

[4] K. Preston, "Cellular logic computers for pattern recognition," *Computer*, vol. 16, no. 1, Jan. 1983, pp. 36-47.

[5] R. A. Rutenbar, T. N. Mudge and D. E. Atkins, "A class of cellular architectures to support physical design automation," *Computing Research Lab. Tech. Report CRL-TR-10-83*, Univ. Michigan, Feb. 1983.

[6] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley Jr. and S. D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. on Computers*, vol. C-30, no. 12, Dec. 1981, pp. 934-947.

[7] M. J. B. Duff, "Review of the CLIP image processing system," *Proc. National Computer Conf.*, 1978, pp. 1055-1060.

[8] J. K. Iliffe, *Advanced Computer Design*, London: Prentice Hall, Chap. 12, 1982.

[9] K. E. Batcher, "Architecture of a massively parallel processor," *Proc. 7th Annual Symp. on Computer Architecture*, 1980, pp. 168-174.

[10] M. Aoki et al., "An LSI adaptive array processor," *Proc. ISSCC*, Feb. 1982, pp. 122-123.

[11] H. J. Siegel, L. J. Siegel, R. J. McMillan, P. T. Mueller Jr., S. D. Smith, "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," *Proc. IEEE Computer Society Conf. on Pattern Recognition and Image Processing*, Aug. 1979.

[12] P. T. Mueller Jr., L. J. Siegel and H. J. Siegel, "Parallel algorithms for the two-dimensional FFT," *Proc. 5-th Int. Conf. on Pattern Recognition*, Dec. 1980.

[13] E. J. Delp, T. N. Mudge, L. J. Siegel and H. J. Siegel, "Parallel processing for computer vision," *Proc. of SPIE-The Int. Society for Optical Engineering*, vol. 336 (Robot Vision), May 1982, pp. 161-167.

[14] W. Feller, *An Introduction to Probability Theory and Its Application*, vol. 1 (3rd edition, revised printing), New York: John Wiley, 1970.

[15] D. J. Kuck, *The Structure of Computers and Computations*, vol. 1, New York: John Wiley, 1978.

[16] D. H. Ballard and C. M. Brown, *Computer Vision*, New Jersey: Prentice-Hall, 1982.

[17] M. L. Baird, "A computer database for the 'Industrial bin of parts problem'," *General Motors Research Lab. Tech. Report GMR-2502*, Aug. 1977.

[18] J. W. Modestino and R. W. Fries, "Construction and properties of a useful two-dimensional random field," *IEEE Trans. on Information Theory*, vol. IT-26, no. 1, Jan. 1980, pp. 44-50.

MATCHING PARALLEL ALGORITHM AND ARCHITECTURE*

Yetung P. Chiang
Electrical Engineering Department
Washington State University
Pullman, WA 99164-2210

and

King-Sun Fu
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract

An attributed directed graph model which is a combination of high-level Petri nets and AND/OR graphs is described. This model provides a method for matching parallel algorithm to architecture or vice versa. The analysis of parallel computation using this model is described. Examples are given to demonstrate the descriptive power of this model and how it helps us to match an algorithm and an architecture.

1. Introduction

It is well known that parallelism is an efficient method to increase the speed of computation in both hardware and software systems. Despite the achievements in designing parallel computers and parallel algorithms there has been very little attention paid to study the relationship between them. As a result, some parallel algorithms may be more effective when executed on one parallel computer than the other. Wirsching and Kishi [1] reported their five projects in investigating the efficiency of highly parallel and highly concurrent computing systems for different problems. They concluded that the efficiency of solving one particular problem varies from different parallel computers. After extensive testing experiments, Deminet [2] pointed out that when the structure of an algorithm corresponds well to the structure of the computer. a close-to-linear speedup may be achieved. Hon and Reddy [3] formulated several valuable principles which outlined the type of algorithm that is more efficient on an architecture with certain specified features. Now the question arises as based on what information should we make the decision that chooses the most efficient computer system for a particular parallel algorithm or vice versa.

Kung [4] classified parallel algorithms in terms of three dimensions: concurrency control, module granularity, and communication geometry. Jones and Schwarz [5] also pointed out three spaces for parallel computation; they are, the computation unit (granularity), communication patterns, and patterns of reference to data. Both papers [4, 5] informally classified parallel algorithms/architectures in terms of their characteristics. Cantoni and Levialdi [6] tried to match tasks in image processing to a parallel architecture. They defined "match" as the degree of exploitation of the system resources (including time) to obtain a specific solution. However, the term "degree" is not defined. Instead, they selected several coefficients for system resources and problem requirements. From these coefficients, Cantoni and Levialdi derived the equation for execution time, provided the task and architecture are specified. They claimed that execution time is useful in determining a matching value. Their work is a good start of formal analysis in matching an algorithm to an architecture. Nevertheless, we believe that system resources should not be the only parameter for measuring the degree of matching. Control flow, system layout, data movement, etc. are also responsible for the overall system performance. In this paper, we intend to study parallel computation and the relationship between parallel algorithm and architecture. We first propose a graph model which is suitable for modeling both algorithms and architectures. Then the analysis of parallel computation is presented based on this model. Finally, examples are given to show the descriptive power of this model and how it helps to make decisions on the type of architecture we should use for a particular algorithm.

2. Graph Model

A number of models for parallel computation have been proposed [7-12, 19]. Among them the directed graph model has shown its capability in modeling both parallel algorithms and parallel architectures. It is advantageous to use the same model for both hardware and software since it simplifies the study of relationships between algorithm and architecture. As Peterson claimed in his paper [7] the Petri nets, a special type of directed graph, are ideal for modeling systems of distributed control with multiple processes occurring concurrently. Another major feature of Petri nets is their asynchronous nature and the nondeterminism in Petri net execution. Figure 1 [7] shows a Petri net model for a producer-consumer problem with one producer (places P_1 and P_2) and two consumers (places P_4, P_5, and P_6, P_7). The items produced by the producer are passed to the consumers. This is modeled by place P_3 and the tokens "produced" by transition t_2 and "consumed" by transitions t_3 and t_5. Tokens are moved by the firing of the transitions in the net. A transition must be enabled (all of its input places have a token in them) in order to fire. A transition fires by removing one token from each of its input places and generating one token to all of its output places. In Figure 1, for example, the transition t_2 is enabled while t_3 and t_5 are not. If t_2 fires, the marked Petri net of Figure 2 results. In Figure 2, three transitions are enabled, t_1, t_3, and t_5. Note that the place P_3 is the same input place for both transitions t_3 and t_5; therefore the firing of either t_3 or t_5 disables the other. Consequently, these two transitions are said to be in conflict. This conflicting situation leads to a nondeterminism in Petri net execution. That is, the choice as to which transition fires is made in a nondeterministic manner which in turn is not modeled. The nondeterminism is a good feature from a model's point of view, but it should not be used when modeling a deterministic algorithm. In other words, we need a deterministic graph model.

In [12, 13] the authors introduce control nodes into the graph model in order to achieve determinism. For example, Figure 3 shows the Petri net of Figure 2 with two extra control nodes c_1, c_2. If c_1 and c_2 are mutually exclusive, i.e., one of them has a token but not both of them, they can solve the conflict between t_3 and t_5. This arrangement has two disadvantages: Firstly, the control nodes will increase the complexity of the graph model. Secondly, the implementation of control nodes consti-

*This work was supported by the NSF Grant ECS 80-16580.

0190-3918/83/0000/0374$01.00 © 1983 IEEE

tutes another step in the design process. We believe that the major cause of nondeterminism in the Petri net is that every transition and place in it cannot perform "disjoint" operation. In fact, the disjoint concept is totally ignored in the Petri net model. There is another type of graph which can explicitly distinguish "joint" and "disjoint" operations. This type of graph is called AND/OR graph [14]. Figure 4 illustrates the ability of AND/OR graph notation to solve the conflict in Figure 2. Note that in this figure, the token in P_3 can either go to t_3 or t_5 but not both. Nevertheless, in order to choose the appropriate destination a decision policy has to be imposed on P_3.

The Petri nets which carry extra information on their places and transitiions have been discussed in [11, 15, 16]. According to Genrich and Lautenbach [11] this type of high-level Petri nets adds a new dimension to the modeling power and complexity of Petri nets, namely the formal treatment of individuals and their changing properties and relations. Unfortunately, the high-level Petri nets do not cover the joint and disjoint conditions, but they do allow to associate expressions with places and transitions. In the following section, we present a new model which takes advantage of both high-level Petri nets and AND/OR graphs.

3. Attributed Directed Graph (ADG)

In the proposed attributed directed graph model there are two types of node, namely, *operation node* (0-node) and *data node* (d-node). These two types of node are equivalent to the transition and place in a Petri net, respectively. The extra information associated with the nodes is expressed in terms of attributes. This explains the nomenclature. The two basic nodes are defined as follows:

Definition 1 – An *operation node* (0-node) is defined as the expression of a subtask. The subtask, depending on the given problem, may be as simple as an ADD operation or as complicated as calculating the distance between two strings. Each 0-node has its attributes (OPR, OP, WM). OPR is the number of operands. OP is the operation or subtask assigned to this 0-node, and WM is the working memory space required by the operation.

The attributes of an 0-node explicitly reveal the characteristics of the 0-node and its relationships with others. To be more specific, OPR not only shows the number of operands required by the 0-node but also indicates that there must be connections between this 0-node and other nodes in order to obtain operands. OP represents the computation complexity of the 0-node and implies whether the operands are required by the operation simultaneously or in a sequential fashion. WM reveals the complexity of memory space.

Definition 2 – A *data node* (d-node) is defined as the place which holds the conditions of an 0-node or stores the consequences after an 0-node. In other words, it is the place where the operation stores/fetches data to/from. The attributes associated with a d-node are represented as (ID, ORD). ID represents the number of various data that reside in this node and ORD specifies the order that this d-node is referenced which may be in either parallel or sequential fashion.

The connection between two nodes is called an *edge*. Similar to SF-nets, an edge can only connect nodes of different types [9]. An edge also possesses attributes. Its attributes (V, MD) have the following meaning. V is the number of variables transmitted via this edge, and MD is the mode of transmission which may either be sequential or parallel. According to the AND/OR graph notation the joint and disjoint situations are reflected through edges as defined below.

Definition 3 – An edge is called *AND case edge* when there exists an arc connecting this edge with other edges. An *OR case edge* does not have any connecting arc. An AND case edge requires that the information transmitted through them must be in parallel. An OR case edge, on the other hand, requires mutually exclusive transmission. The exact order of transmission through an OR case edge follows the direction of connected 0-node or d-node.

Definition 4 – An attributed directed graph (ADG) is a four tuple (D,O,A,M_o)

where (1) D is a finite set of d-nodes,
(2) 0 is a finite set of 0-nodes,
(3) $A \subseteq (D \times 0) \cup (0 \times D)$ is a finite set of AND/OR case edges,
and (4) M_o is the initial marking of the ADG model. This marking is expressed by tokens (black dots). The movement of a token is governed by the firing rules which are defined below.

Definition 5 – The firing rules [9]:

(1) An 0-node is enabled if all of its input d-nodes hold at least one token and its output d-nodes are empty.

(2) Any enabled 0-node remains enable and may be fired at any time according to the operations of the 0-node.

(3) An 0-node is fired by removing one token from each member of the input d-nodes and add one token to each of the output d-nodes. At this point, the 0-node execution is complete.

4. Attributes in Parallel Computation

As mentioned earlier, many important features should be considered in a parallel computing environment, such as control interconnection, routing, memory conflict, scheduling, synchronization, etc. In order to understand the nature of parallel computing, it would be better to start from its special features. In this section we choose five features as the attributes of a parallel computation. They are described in the following.

4.1 Data Movement

Every computing system could be considered as a data manipulator. That is, it receives data and after certain operations produces resulting data. Every algorithm is a straightforward procedure which controls the computing system and hence the data movement within it. In more detail, there are many subtasks in an algorithm and each of them requires input data and produces output data. For instance, an ADD operation requires two operands and produces the sum; a string distance calculation takes in two input strings and gives the distance between them as the result. Usually, the data movement required by the subtasks is embedded in the algorithm. For instance, the previous two examples may appear in an algorithm like S = a+b and P = distance (x,y), where S and P are sum and distance, respectively. The algorithm tells what data we should use, but it never shows us where they come from. At the implementation stage different architectures have different effects on obtaining data. For example, data may be broadcasted through a central control unit or exchanged in the shared memory; it may also be delivered by data busses or by some direct data line connections. When executing an algorithm, the data movement is the other important part besides the direct calculations. We will describe data movement as regular or irregular, which refers to the connections between subtasks; and variant or invariant, which indicates whether or not the connections change with time.

4.2 Module Granularity

Every subtask (or task module) decomposed from a given task requires some execution time. We call this elapsed time the module granularity (MG). In an algorithm, the MG is not as important as the overall complexity analysis, but it does affect the analysis indirectly. In a real execution, MG is an important factor for the problem of synchronization, memory contention, etc. It sometimes forces a designer to choose a different architecture or to use a different control scheme. For instance, suppose that we have subtasks X and Y needed to be completed before we go on to subtask Z. X and Y require different execution time. A synchronizer is needed before subtask Z starts. On the other hand, if X and Y execute the same operation and they happen to fetch the same data at the same time, then the memory contention problem arises.

In the above situations, we have to take the module granularity into consideration before we decide the particular system to use or the type of hardware/software conflict resolver to implement. Here we classify module granularity as uniform or nonuniform, which indicates whether or not all the processes have the same computation. We also call a MG either "large," "small," or "constant." This is not a well-defined term; rather, it tells us the relative size of the module granularity when compared with each other.

4.3 Communication Geometry

When the task modules of a parallel algorithm are connected to represent their inter-module communication, the geometric layout of the resulting network is referred to as the communication geometry. This geometric layout does not have to be identical to the data movement. The data movement identifies the source and destination in a data transaction. The same data transaction can be achieved on different inter-module connections as long as we provide proper routing paths. With routing ability, a rather simple communication geometry can be obtained and henceforth reduces the complexity of the implementing hardware. Communication geometry is closely related to control overhead. That is, for a specific data communication, the direct connection network may require no or less control while the simple indirect connection network may need routing algorithm to manage the data movement. Since we are only interested in the relation between algorithms and architectures, routing is not discussed here. We classify communication geometry according to its geometric layout as irregular and regular. Among regular it is further divided into interconnection switch (crossbar, perfect shuffle, etc.) and array network (1-D, 2-D, etc.).

4.4 Memory Space

In algorithm analysis [17], both time and memory space are important factors in judging the efficiency of an algorithm. In terms of hardware, memory access is always slow and memory space occupies a large portion of area even with today's IC technology. Although the speed and dimension of a memory element have been improved drastically, it is still the most time-consuming and expensive part in any system. The common phenomenon with memory space is that if one prefers to put memory on the same chip with the arithmetic logic unit, he certainly can enjoy a fast memory access in the sacrifice of small memory space. On the other hand, by supplying secondary memory, one can increase memory space but suffers longer access time. Besides, in a system with large memory database, the management policy is another important factor in deciding the efficiency of a system. For the purpose of this paper, we classify memory space as local or global, which refers to its locality; and constant or nonconstant, which indicates whether or not the required memory space depends on the input data. We sometimes call the size of memory space small or large. This is not quantitatively defined; instead, it is a comparative term.

4.5 Concurrency Control

Control information is usually only implicitly shown in an algorithm, but it is essential to a hardware system. The control directs the computation sequence and assures the correctness. There are various ways to control a parallel computing system. It can be a stored segment of microcodes, system clock, special control logic, etc. Control is closely related to other attributes. In this paper, we only classify concurrency control as centralized or distributed. Detailed control techniques are not considered at this point.

The attributes mentioned above are by no means a complete description of parallel computing. The simple classifications of each attribute are only for the convenience of this study, that is, to reveal the relationship between an algorithm and an architecture. These five attributes can easily be extracted from the proposed ADG model. In other words, the ADG model is capable of describing parallel computations. Figure 5 outlines the relation between ADG model and the attributes of parallel computation. For example, the structure of an edge defines the communication geometry; memory space is determined by the WM in an 0-node and the ID in a d-node; concurrency control is covered in the ORD of d-node and the OP of 0-node; module granularity is decided based on the OP of 0-node; and data movement is directed by the OPR in an 0-node and the edge. These relations are further elaborated in the following examples.

5. Illustrative Examples

Example 1 – In a general SIMD computer all of its processing elements (PEs) execute the same instructions which are sent from the control unit (CU). These PEs may exchange their data with each other through a communication network. The ADG model for SIMD computer systems is shown in Figure 6. From this model we can determine the nature of parallel computation in a SIMD system.

The communication geometry (CG) is directly reflected by the hardware connections. It is easy to understand that in this case the CG is regular and it is a switching network type. The data movement is bounded by CG and hence is also regular. In fact, the data movement may be either variant or invariant depending on applications. This variation of data movement can be seen in the OPR of Inter-PEs' 0-node. The module granularity (MG) depends entirely on the complexity of 0-node. Speaking of the whole system, its MG is nonuniform because of the existence of CU and Inter-PE. But if we only consider PEs, they have uniform MG.

In Figure 6, the complexity of 0-node is not specified since the associated operations vary from different applications. In general the operations of CU and Inter-PE are far more complicated than those of PEs. The CU not only controls PEs and Inter-PE, but also communicates with the external world. The Inter-PE communication network takes commands from CU and exchanges information between PEs accordingly. All the PEs execute the instructions which are broadcasted from CU one at a time. Therefore the operation of PE is considered to be a straightforward single step. Judging from the number of edges pointing toward 0-node and the operations associated with 0-node, we conclude that, relatively speaking, the MG of PEs is small while the MGs of CU and Inter-PE are large.

The memory space (MS) is closely related to d-node and 0-node. From the number of incoming edges, we can decide whether the memory space is local (no greater than two incoming edges) or global (more than two incoming edges). Based on this criterion, all the memory spaces (d-nodes) are local in this case. The size of MS can be roughly determined from the number of variables on the incoming edges. In Figure 6, edge 1 conveys instruction, data and CU's control program, such as masking, routing, etc.; edge 2 transmits instruction and data; edge 3 transmits instruction, while edge 4 only transmits data. Therefore, a reasonable conclusion is that the CU memory is large and the PE memory is small. The concurrency control is centralized as clearly expressed in Figure 6. The control overhead is not severe, since all the PEs are under the same control instruction (AND case), only the Inter-PE requires additional control effort (OR case), and the ORD in the d-node of a PE is also simple.

The summary of Example 1 and the analysis results for MIMD system and VLSI systolic array are listed in Figure 7.

Example 2 – A parallel Earley's parsing algorithm [18] is recorded below.

Algorithm A.

```
for i = 1 to n do in parallel
    $t(i-1,i) = Y\ x*\ \{a_i\}$
for j = 2 to n do
    for i = 0 to n-j do in parallel
    begin
    [Scanner:]
        $t(i,i+j) = t(i,i+j-1)\ x*\ \{a_{i+j}\}$
    [Completer:]
        for k = 1 to j-1 do in parallel
            $t(i,i+j) = t(i,i+j)\ U\ t(i,i+k)\ x*\ t(k+k,i+j)$
    end
```

This algorithm constructs an upper-triangular shape parsing matrix with each element denoted as t(i,j). Algorithm A is executed in a pipeline fashion, and if we define subtask P(i,j) as calculating

$t(i,j) = t(i,j-1)\ x*\{a_j\}$
and for k=1 to j-1
$t(i,j) = t(i,j)\ U\ t(i,k)\ x*t(k,j)$,

we can represent Algorithm A by the ADG model as shown in Figure 8. It is easy to see that the data movement, as shown by the edges, is very complicated. Note that many of the edges are OR case edges, which means that the results of those subtasks do not have to be sent to all the destinations at the same time. For instance, t(0,1) is needed by subtasks P(0,2), P(0,3), and P(0,4). These subtasks are activated one at each stage, which indicates that a single data path moving t(0,1) through these subtasks is capable of doing the job. However, a special arrangement has to be made for this simple connection in order to have a correct execution. This arrangement is described in [18]. A simplification of connections changes Figure 8 to Figure 9 which describes a regular connected network. In Figure 9 each subtask simply executes

$t(i,j) = t(i,j-1)\ x*\ \{a_1\}$
and $t(i,j) = t(i,j) U\ t(i,k)\ x*\ t(k,j)$.

As described in [18], this subtask can be implemented on a special hardware and executed in a small constant time. The control of this special hardware is simple and local (distributed). From the analysis above and the characteristics of different systems shown in Figure 7 we conclude that Algorithm A is suitable for VLSI systolic array implementation.

6. Concluding Remarks

In this paper, we describe the attributed directed graph model. This model is more powerful in modeling concurrent hardware and/or software systems. Furthermore, this ADG model is a deterministic graph model which includes the control information in its attributes. We also study the features of parallel computation and express them in terms of the ADG model. Finally we use two examples to demonstrate the descriptive power of the ADG model and how this model can aid a designer to choose the proper algorithm for the proper architecture or vice versa.

References

[1] Wirsching, J. E. and T. Kishi, "Matching Machines and Problems," in *High Speed Computer and Algorithm Organization,* D. J. Kuck, D. H. Lawrie and A. H. Sameh Eds., Academic Press, 1977.

[2] Deminet, J. "Experience with Multiprocessor Algorithms," *IEEE Trans. Comp.,* Vol. C-31, No. 4, April 1982.

[3] Hon, R. W. and D. R. Reddy, "The Effect of Computer Architecture on Algorithm Decomposition and Performance," in *High Speed Computer and Algorithm Organization,* D. J. Kuck, D. H. Lawrie and A. H. Sameh Eds., Academic Press, 1977.

[4] Kung, H. T., "The Structure of Parallel Algorithms," in *Advances in Computers,* Vol. 19, M. C. Yovits Eds., Academic Press, 1980.

[5] Jones, A. K. and P. Schwarz, "Experience Using Multiprocessor Systems – A Status Report," *Computing Surveys,* Vol. 12, No. 2, June 1980.

[6] Cantoni, V. and S. Levialdi, "Matching the Task to an Image Processing Architecture," Proc. 6th Int'l Conf. Patt. Recog., Munich, Germany, Oct. 1982.

[7] Peterson, J. L., "Petri Nets," *Comp. Surveys,* Vol. 9, No. 3, Sept. 1977, 223-252.

[8] Miller, R. E., "A Comparison of Some Theoretical Models of Parallel Computation," *IEEE Trans. Comp.,* Vol. C-22, No. 8, Aug. 1973, 710-717.

[9] Foo, S. Y. and G. Musgrave, "Comparison of Graph Models for Parallel Computation and Their Extension," Proc. 1975 Int'l Symp. Comp. Hardware Description Languages and Their Applications, New

York, NY, Sept. 1975, 16-21.

[10] Mattheyses, R. M. and S. E. Conry, "Models for Specification and Analysis of Parallel Computing Systems," Conf. of Simulation, Measurement and Modeling of Computer Systems, Boulder Colorado, Aug. 1979, 215-224.

[11] Genrich, H. J. and K. Lautenbach, "System Modelling with High-Level Petri Nets," *Theoretical Computer Science,* Vol. 13, 1981, 109-136.

[12] Bradshaw, F. T., "Directed Graph Models for Hardware/Software Design," Proc. 1975 Int'l Symp. Comp. Hardware Description Languages and Their Applications, New York, NY, Sept. 1975, 7-15.

[13] Wojtkowiak, H., "Deterministic Systems Design from Functional Specifications," Proc. 18th Design Automation Conf., Nashville, Tenn., June 29 - July 1, 1981, 98-104.

[14] Nilsson, N. J., *Problem Solving Methods in Artificial Intelligence,* McGraw-Hill, New York, 1971.

[15] Jensen, K., M. Kyng and O. L. Madsen, *("A Petri Net Definition of a System Description Language," in G. Kahn, Ed., *Semantics of Concurrent Computation,* Lecture Notes in Computer Science 70, Springer, Berlin, 1979, 348-368.

[16] Jensen, K., "Coloured Petri Nets and the Invariant-Method," *Theoretical Computer Science,* Vol. 14, 1981, 317-336.

[17] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

[18] Chiang, Y. P. and K. S. Fu, "Parallel Processing and VLSI Architectures for Syntactic Pattern Recognition and Image Analysis," Tech. Rpt. TR-EE83-4, School of Electrical Engineering, Purdue University, Jan. 1983.

[19] Treleaven, P. C., "Parallel Models of Computation," in *Parallel Processing Systems,* D. J. Evans Ed., Cambridge University Press, Cambridge, 1982.

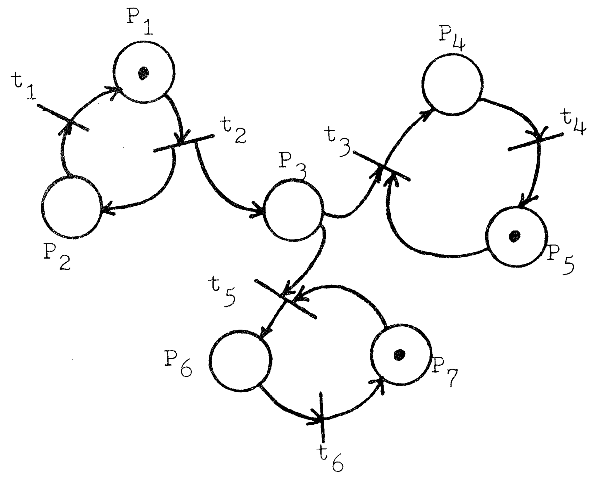


Figure 1. A marked Petri net.

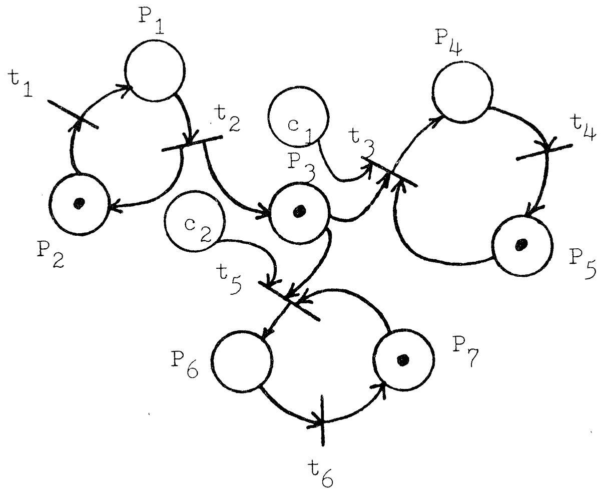


Figure 3. Petri net with control nodes.

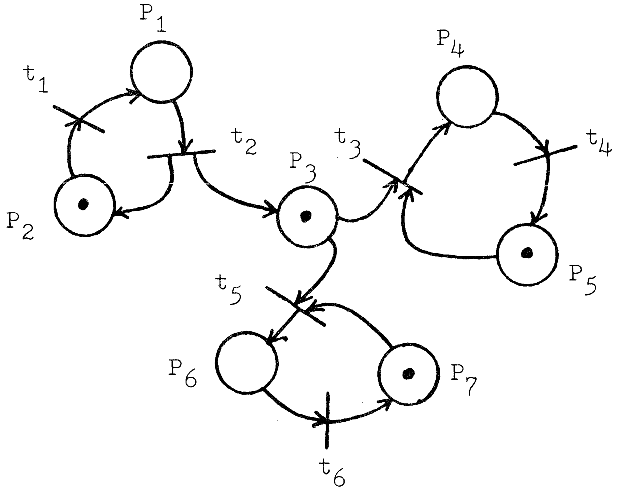


Figure 2. The resulting Petri net from Figure 1.

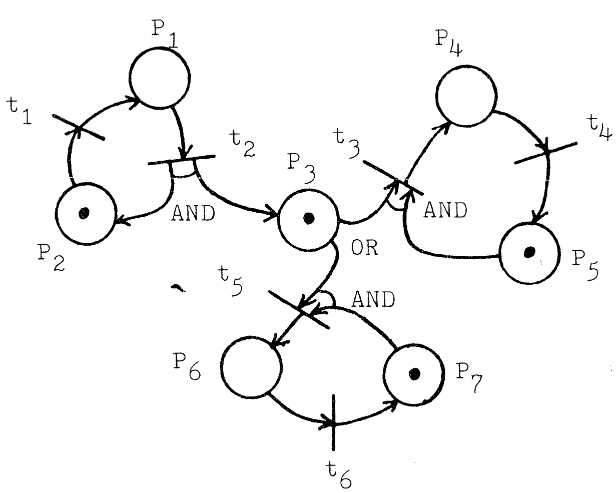
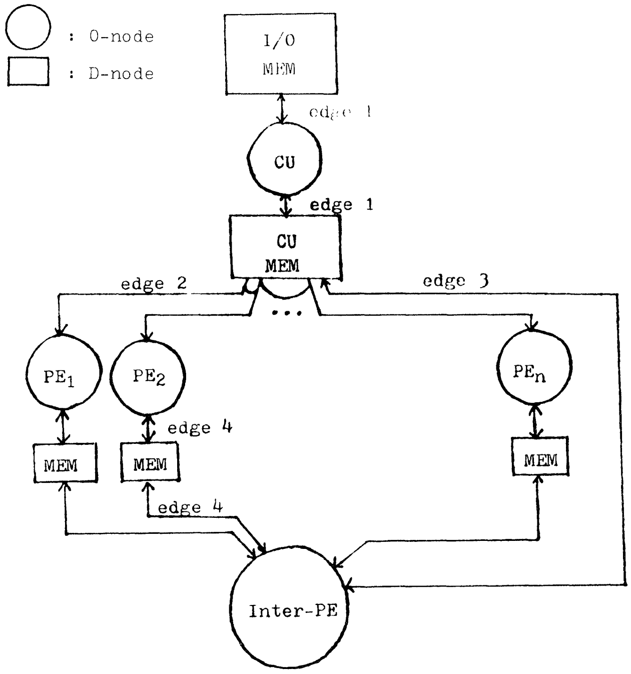


Figure 4. Petri net with AND/OR notation.

	D-NODE	O-NODE	EDGE
DATA MOVEMENT		OPR	V,MD
MODULE GRANULARITY		OP	
COMMUNICATION GEOMETRY			STRUCTURE
MEMORY SPACE	ID	OPR, WM	
CONCURRENCY CONTROL	ORD	OP	

Figure 5. The relation between ADG and the attributes of parallel computation.



CU:
O-node (2, OP, large)
D-node (N+2, ORD)
PE:
O-node (2, OP, small)
D-node (2, ORD)

Inter-PE:
O-node (OPR, OP, large)
edge 1 (3, MD)
edge 2 (2, S)
edge 3 (1, S)
edge 4 (1, S)

Figure 6. ADG representationof SIMD system.

Attributes / Architecture	Data Movement	Module Granularity	Communication Geometry	Memory Space	Concurrency Control
SIMD	regular, variant or invariant	small constant, small or large	regular, intercommunication switch,	small, local	central, simple or complex
MIMD	irregular variant	large	irregular	large global	local, complex or simple
VLSI Systolic Array	regular, invariant	small constant	regular net work	small, local	central or local, simple

Figure 7. Characteristics of parallel computing systems.

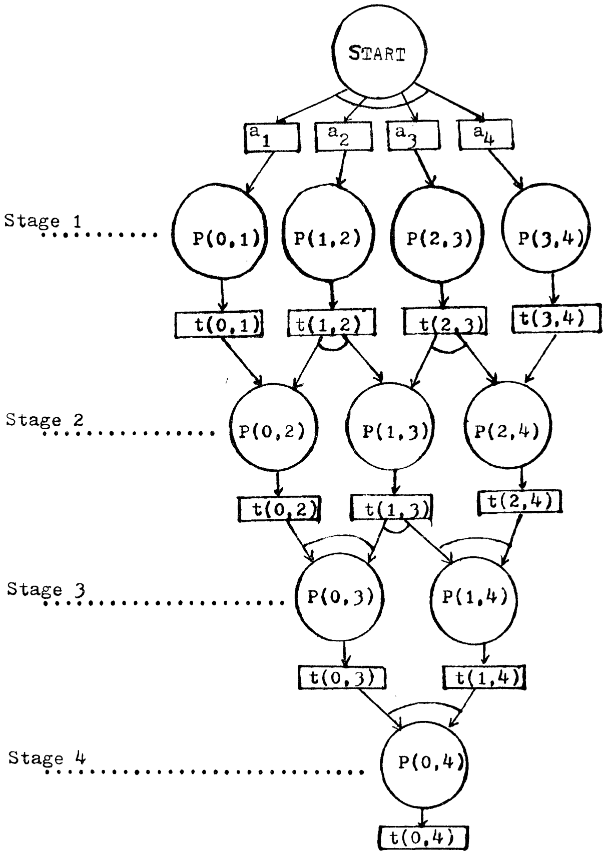


Figure 8. ADG representation of Algorithm A.

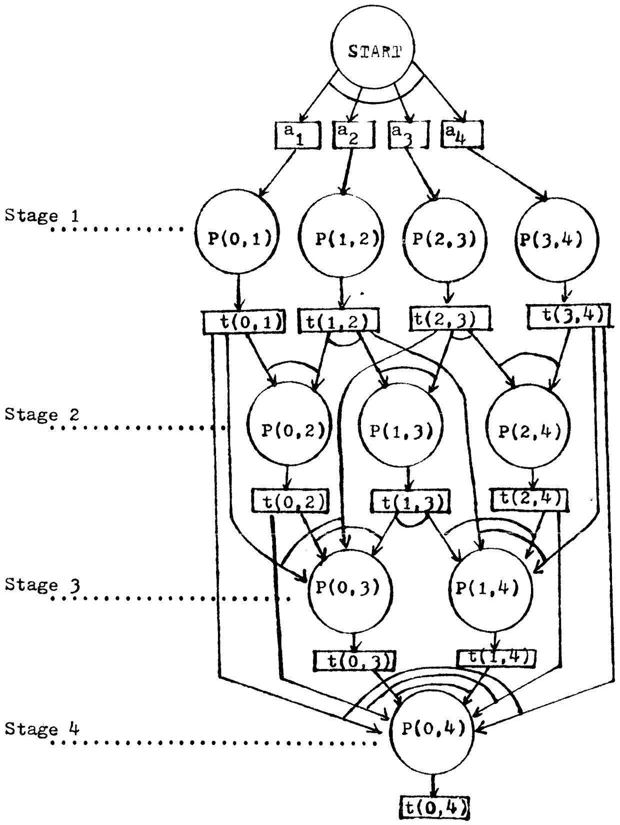


Figure 9. Reduced ADG representation of Algorithm A.

COHERENT FLOW OF INFORMATION IN PARALLEL SYSTEMS

Bruce P. Lester
Department of Computer Science
Maharishi International University
Fairfield, Iowa 52556

Abstract -- Proper functioning of any parallel system depends on balance in the flow of information. A specific flow graph model for systems is presented using linear inequalities to characterize the terminal behavior of individual components of the system. These inequalities are combined into a homogeneous system of linear equations, whose solution reveals some of the global information flow properties of the parallel system. Several theorems are stated regarding characteristics of this global information flow in deadlock-free systems.

Introduction

There is a new interdisciplinary field called the Science of Creative Intelligence, which studies universal principles of orderliness and intelligence in human beings and in natural systems (1). According to Maharishi Mahesh Yogi, the founder of the Science of Creative Intelligence, any system which expresses intelligence must have a coherent relationship between the individual parts of the system (2). Maharishi's theory states that the level of intelligence depends on the degree of coherence and integrated functioning within the system (3). In the field of computer science, the individual components of any information processing system must function in a manner which produces a global coherence and balance throughout the entire system. This is especially true of systems with a high degree of parallelism, in which there are typically a large number of individual components, each having a high degree of independence.

Flow Graphs

For the sake of brevity and readability, this paper is rather informal in presentation. The approach used in this research is similar in style to much of the work on properties of parallel control structures such as Petri-Net theory (4) and data flow schemas (5,6). For a more complete and formal analysis of the theory, the reader is referred to Lester (7). An information flow graph is defined as a directed graph, each of whose nodes contains an information processing module. The modules are of two types: Fixed modules and Union modules. Information packets flow along the arcs between modules, and each module has an internal state which determines its terminal behavior with respect to input and output of information packets.

Our focus in this paper is not so much on the detailed structure or data values contained in each information packet, but on the total number of packets sent and received via a given arc. For that reason, the packets are treated a indistinguishable, unitary entities. The total number of packets which have been sent along a give arc b of the flow graph is called the count on that arc and is denoted $|b|$. The arcs have no capacity for internal storage and serve only as channels for packets to flow between modules. The modules operate independently and interact with neighboring modules by sending or receiving packets along the connecting arcs (or terminals). Intuitively, a fixed module is one which maintains a fixed ratio for the counts on its connecting arcs. That is, if we look at the counts on the terminals of a fixed module over time, the ratios of the counts on different terminals will converge to a constant. More formally, for each pair of terminals (a,b) of a fixed module, there exists positive rational constants C,K,R such that

$$-C \le |b| - R|a| \le K$$

There are many different types of typical system components which can be modelled as fixed modules. Some examples are shown in Fig. 1.

Fixed modules are defined above as having a fixed ratio for the counts at every pair of terminals. The other type of module is the union module which has a sum property with respect to its terminals: the sum of the input counts is equal to the sum of the output counts. More formally, for any union module with input terminal set A and output terminal set B, there are positive integers C and K such that

$$-C \le \sum_{b\varepsilon B} |b| - \sum_{a\varepsilon A} |a| \le K$$

Intuitively, a union module is a storage facility information packets with some finite maximum capacity. Some examples of typical components or parallel systems, which can be modelled as union modules are shown in Fig. 2. An arc in a flow graph is defined as dead if no further information packets can flow through it. A graph is deadlock-free if there is no reachable state with a dead arc.

Current Law Equations

Union modules and fixed modules together have a broad range of modelling power, as illustrated by the examples. Now we will present a simple but useful mathematical technique for analyzing the information flow properties of any flow graph consisting of fixed modules and union modules. With each arc b of the flow graph, let us associate a current variable i_b. Now define the following current laws for the modules:

1. Union modules -
 sum of input currents = sum of output currents
2. Fixed modules - for each pair of terminals (a,b) with fixed ratio R, $i_b = Ri_a$

Intuitively, we may think of the currents as a measure of the information flow along the arcs. Current law 1 for union modules is just the familiar Kirchhoff's Current Law of electrical network theory. Knuth (8) and Deo (9) have noted the usefulness of KCL for analyzing the properties of

0190-3918/83/0000/0381$01.00 © 1983 IEEE

normal sequential flow charts. However, current law 2 is a new law that is necessary for more complex parallel systems. For any information flow graph with n terminals, the current laws define a homogeneous system of linear equations $A\underline{i} = \underline{0}$, where $\underline{i} = (i_1, i_2, \ldots, i_n)$ and A is an m by n matrix.

Theorem 1 - For any deadlock-free information flow graph, the current law equations have a positive integral solution.

The proof method for Theorem 1 is to notice that the current laws for fixed and union modules are taken directly from the linear inequalities that define the modules. From these linear inequalities, we know $-C < A\underline{x} < K$, where $\underline{x}$ is the vector of counts at the terminals. Since the flow graph is deadlock-free, the counts can grow arbitrarily large, so A must be noninvertible and $A\underline{i}=\underline{0}$ has a solution. For an example of a simple flow chart computation, whose current law equations have no solution, see Fig. 3. From Theorem 1, we know the flowchart must contain a deadlock, which is clearly seen by inspection.

Independent Currents

In the case that the current law equations do have a solution, a great deal may be learned about the global information flow in the graph from the properties of the solution. From standard techniques of linear algebra, the homogeneous system of linear equations $A\underline{i} = \underline{0}$ has a general solution $\underline{i} = B(i_1, i_2, \ldots, i_k)$, where $k = n - \text{rank } A$ and $(i_1, i_2, \ldots i_k)$ are arbitrary constants, which we call the independent currents. All of the terminal currents of the flow graph may then be expressed as a linear combination of these k independent currents. For example, the computation of Fig. 4 involves communication between processes using mailboxes and Send-Receive primitives. There are 5 union modules and 5 fixed modules. The current law equations result in a solution with two independent currents i_1 and i_2 as shown.

From the independent current solutions for Fig. 4, it is seen that the T branch of each Decider has the same current. This means that in order for the whole computation to be deadlock-free, both Deciders must make the same percentage of T choices (subject to fixed deviations depending on the initial contents and capacity of the mailboxes). If each Decider is assumed to be independently making arbitrary free choices, then a communications deadlock may result. In order to make it easier to analyze properties of porgram flow graphs, it is common in the theory of program schemas to consider Deciders as free to make independent choices. A specific assignment of choices for each Decider is usually called an interpretation.

Theorem 2 - If an information flow graph is deadlock-free (for all interpretations), then no. of independent currents < no. of Deciders.

The proof of this theorem is too complex to be included here, but can be found in Lester (7). However, the substance of the result is quite simple and intuitive. If a Decider is free to make independent choices, then there must be one degree of freedom in the overall flow of control to accommodate that Decider. The number of independent currents in the solution to the current law equations represents the total number of degrees of freedom in the global flow of information. If the system is to be deadlock-free, these degrees of freedom must exceed the number of Deciders. In Fig. 4, there are only two independent currents (i_1 and i_2) for two Deciders; thus, by Theorem 2, the overall computation will have a deadlock for some interpretation of the Deciders.

Conclusions

In this paper, we have presented a simple mathematical technique for analyzing the global information flow properties of parallel systems. The technique relies on the standard procedures of linear algebra for solving a homogeneous system of linear equations. Thus, this analysis procedure could easily be automated into a compiler for parallel programs or into any parallel system design tool. If a parallel system does not meet the conditions specified by Theorems 1 and 2, then it has a potential deadlock.

References

(1) International Symposium on the Science of Creative Intelligence, University of Mass., Amherst and Humbolt State College, Calif., MIU Press, Seelisberg, Switzerland, (1971).

(2) Maharishi Mahesh Yogi, The Science of Creative Intelligence, (A Thirty-Three Lesson Videotape course), Age of Enlightenment Press, Livingston Manor, N.Y. (1972).

(3) Education for Enlightenment, An Introduction to Maharishi International University, MIU Press, Fairfield, Iowa (1981), pp. 23-41.

(4) J.L. Peterson, Petri-Net Theory and the Modeling of Systems, Prentice-Hall, N.J.,(1981).

(5) J.B. Dennis, J.B. Fosseen, J.P. Linderman, Data Flow Schemas, in "International Symposium on Theoretical Programming", Lecture Notes in Computer Science 5, Springer-Verlag, Berlin (1974), pp. 187-216.

(6) J.M. Jaffe, "The Equivalence of R.E. Program Schemes and Data Flow Schemes,"Journal of Computer and System Sciences, 21, (1980), pp. 92-109.

(7) B.P. Lester, The Balance Property of Parallel Computations, Dept. of Electrical Engineering, Massachusetts Institute of Tech., PhD Thesis, Cambridge, Mass. (1974).

(8) D.E. Knuth, The Art of Computer Programming, Vol. 1 - Fundamental Algorithms, Addison-Wesley Reading, Mass. (1973), pp. 362-368.

(9) N. Deo, Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall, Englewood Cliffs, N.J. (1974), pp. 439-444.

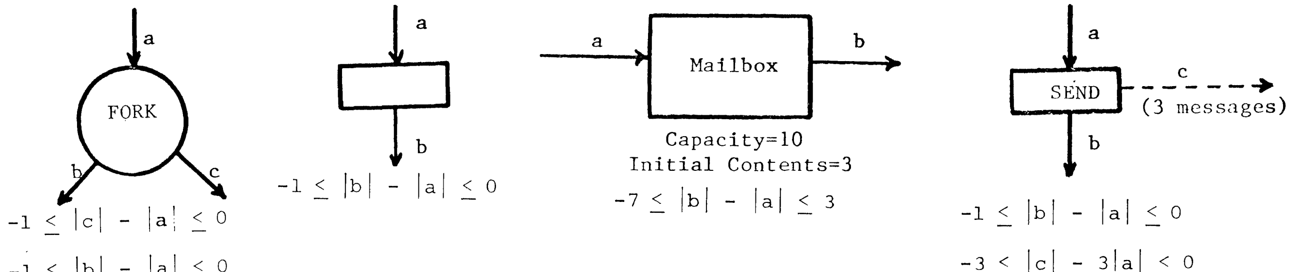

a
FORK
b
c
$-1 \leq |c| - |a| \leq 0$
$-1 \leq |b| - |a| \leq 0$
a
b
$-1 \leq |b| - |a| \leq 0$
a
Mailbox
b
Capacity=10
Initial Contents=3
$-7 \leq |b| - |a| \leq 3$
a
SEND
c
(3 messages)
b
$-1 \leq |b| - |a| \leq 0$
$-3 \leq |c| - 3|a| \leq 0$

Figure 1 - Examples of Fixed Modules

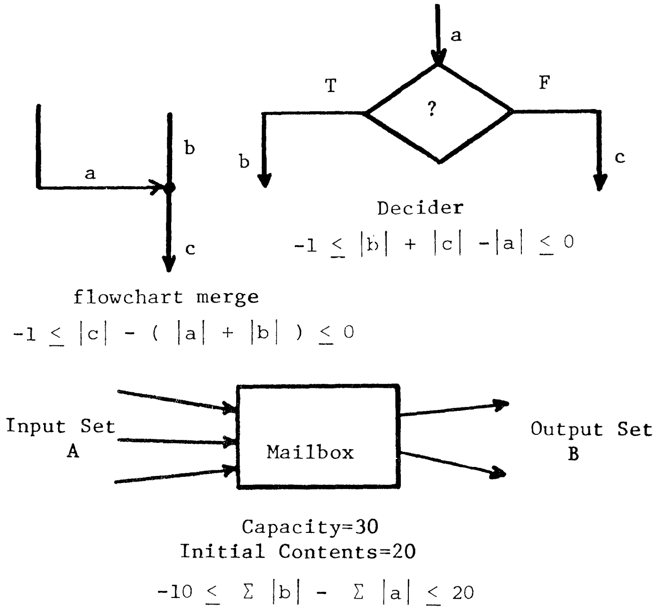

a
T
F
?
b
c
Decider
$-1 \leq |b| + |c| - |a| \leq 0$
a
b
c
flowchart merge
$-1 \leq |c| - (|a| + |b|) \leq 0$
Input Set
A
Mailbox
Output Set
B
Capacity=30
Initial Contents=20
$-10 \leq \Sigma |b| - \Sigma |a| \leq 20$

Figure 2 - Examples of Union Modules

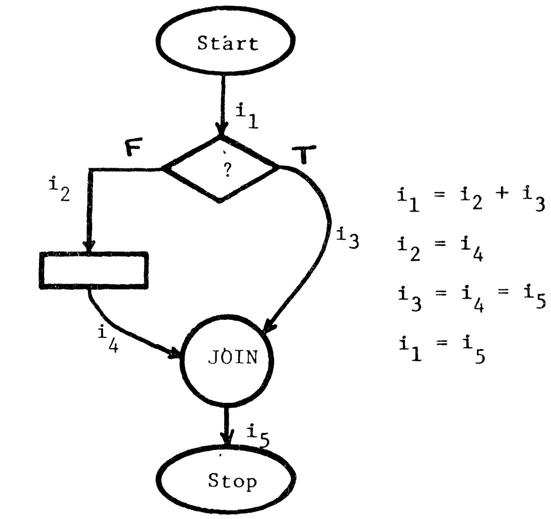


Figure 3 - No Solution to Current Equations

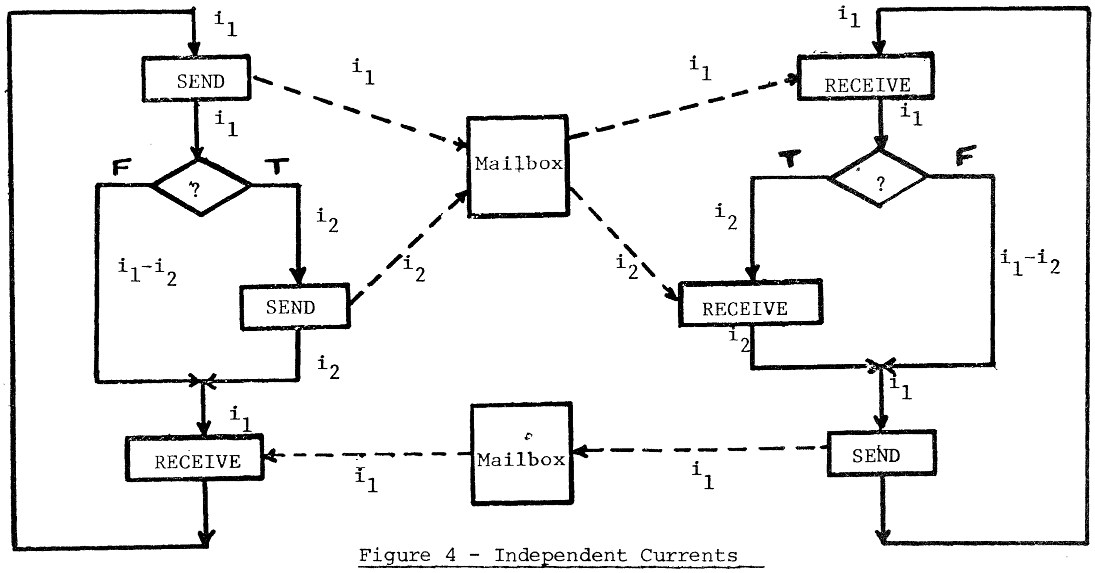


Figure 4 - Independent Currents

Virtual Time

by

David Jefferson

Department of Computer Science
University of Southern California
Los Angeles, California 90089

Abstract

Virtual time is a broad, new paradigm for organizing and synchronizing distributed systems, subsuming such heretofore distantly related problems as distributed discrete event simulation and distributed database concurrency control. It is an abstraction of real time in much the same way that virtual memory is an abstraction of real memory, and it reorganizes the concepts of concurrency and synchronization in a manner similar to the way virtual memory reorganized the subject of memory management. Virtual time systems can be implemented using the *Time Warp mechanism*, a distributed synchronization mechanism that is distinguished by its wholesale commitment to *lookahead-rollback* as its primary synchronization tool, but its implementation of rollback through *antimessages*, and by its global coordination through the concept of *global virtual time*.

1. Introduction

This paper is a short introduction to a new theoretical paradigm for distributed computation called *virtual time*, and to its implementation, the *Time Warp mechanism*. The virtual time paradigm is a method of coordinating distributed systems by imposing on them a temporal coordinate system more computationally meaningful than *real time*, and defining all notions of synchronization and timing in terms of it. The virtual time scale need not be closely related to real time, but it is still "temporal" because virtual time increases as the computation progresses, because events at the same virtual time act as parts of a single action atomic with respect to actions at other virtual times, and because one can reason correctly about vitual time relations such as "before" and "after" by using ordinary "Newtonian" intuition. The more difficult "relativistic" reasoning [Lamport 78] required to understand the real time relations "before" and "after" in distributed systems is unnecessary.

The Time Warp mechanism is a distributed process control regime that implements virtual time, in the same way that paging or segmentation mechanisms implement virtual memory. Its distinguishing feature is its wholesale commitment to process lookahead and rollback as the fundamental synchronization mechanism. Although relying on rollback may unorthodox step, it is also liberating. Many synchronization mechanism issues become much simpler once the possibility of rollback is considered. Of course at first glance rollback seems more expensive both in space and time than other synchronization mechanisms such as process blocking, so it is essential to argue for each use either that rollback will be rare or that any other synchronization mechanism would incur similar overhead anyway. Such arguments can be made on the basis of temporal locality assumptions about the dynamic behavior of programs analogous to the spatial locality assumptions underlying paging systems but these assumptions have yet to be tested.

These two notions, the virtual time paradigm and the Time Warp mechanism, offer new ways to think about distributed computation. In particular the following results derive from them, although the justifications can only be outlined in this paper.

- Discrete event simulation can be viewed as an application of the virtual time paradigm. The Time Warp mechanism provides a new method for high-concurrency discrete event simulation that is transparent to the programmer and free of deadlock and starvation [Jefferson 82],[Jefferson 83a].
- Distributed database systems can also be viewed as virtual time systems, in which case the Time Warp mechanism is a concurrency control and crash recovery mechanism, again yielding high concurrency without starvation or deadlock [Jefferson 83b].
- The Time Warp mechanism suggests a rethinking of synchronization in distributed systems. Most synchronization is based on the ability to block a process and restart it later, sometimes augmented with the ability to abort an action in progress. But the Time Warp mechanism seems to offer the first wholesale use of process rollback as the basis of synchronization, making new protocols and strategies possible.

In the next section we will describe the concept of virtual time and the semantics of virtual time systems. In Section 3 we will compare our views to those of other theorists in the field of distributed systems. Section 4 we will describe the Time Warp mechanism, both its local and its global parts. In Section 5 we will give three examples of paradigms that become unified when viewed as virtual time systems. Section 6 gives the extended comparison between virtual time and virtual memory that has been a central focus of

0190-3918/83/0000/0384$01.00 © 1983 IEEE

this research. Finally, Section 7 offers some future directions.

2. Virtual time

A *virtual time system* is a distributed system that executes in coordination with an imaginary global *virtual clock* that ticks *virtual time*. Virtual time is a temporal coordinate system used to measure computational progress and define synchronization. Viewed abstractly a virtual clock always progresses forward (or at least never backward) at a pace that may either be closely bound to real time or completely independent of it. We assume that virtual times are real values (and ∞), totally ordered as usual by the relation <.

We envision systems of hundreds or thousands of processes all executing concurrently on a network of many processors. It is useful to consider each process as occupying a "point" in *virtual space*, and its unique name as its spatial coordinate. Every primitive action executed by a system can thus be assigned both a virtual time and a virtual space coordinate, and the set of all actions that take place at the same virtual place x and virtual time t are collectively referred to as the *event* at (x,t).

Processes communicate by exchanging messages stamped with the name of the *sender*, the *virtual send time*, the *receiver*, and the *virtual receive time*. The virtual send time is the virtual time at the moment the message is sent; and likewise the virtual receive time is the virtual time when the message must be received. We can also say, equivalently, that a message is stamped with the coordinates of both the sending and receiving events. A message is simply the transfer of information from one "point" in virtual space-time to another (like a photon in physics).

The interaction of processes, messages and virtual time is subject to two semantic rules:

> Rule 1: The virtual send time of a message must be less than or equal to its virtual receive time.
>
> Rule 2: All messages directed to a particular process must be processed in nondecreasing virtual receive time order.

These restrictions, similar to Lamport's Clock Conditions [Lamport 78], embody our desire that the arrow of causality be pointed in the direction of increasing virtual time. For convenience we adopt two further rules in this paper:

> Rule 3: No two messages directed to the same process have the same virtual receive time.
>
> Rule 4: Events not involving the receipt of a message are null, i.e., no-ops.

Rule 3 has the effect of changing the work "nondecreasing" to "increasing" in Rule 2. Rule 4 removes from consideration spontaneous state changes and message sending that are not prompted by receipt of a message. The system must therefore be driven (or at least initiated) by message(s) from an "outside" process. Many interesting issues arise when these latter two rules are relaxed, as is usually necessary, but they would unduly complicate our discussion.

A non-null event at (x,t) consists of the following three actions executed sequentially:

1. Process x receives the message stamped with receiver x and virtual receive time t.
2. It updates its state accordingly.
3. It sends zero or more messages stamped with sender x and virtual send time t.

The semantics of virtual time are extremely simple.

> If an event A has a virtual time less than that of event B, then the execution of A and B must be scheduled so that A appears to be completed before B starts.

Even though A is earlier in virtual time than B, an implementation need not actually perform A before B. It may achieve better performance by scheduling A concurrently with B or even after it, as long as this fact is not detectable by any tests within the virtual time system.

A consequence of this semantic rule is that if A and B have exactly the same virtual time (even if they occur at different places) they appear to be components of a single atomic operation that is indivisible with respect to events at other virtual times, because all events at earlier virtual times must appear to have been completed before either A or B starts, and all events at later virtual times must appear not to have started until A and B are complete. Note that if A and B do have the same virtual time coordinate there are no restrictions on their scheduling.

There are several degrees of freedom in the design of virtual time systems. The virtual time scale may be discrete or continuous (although here we assume continuous). It may be partially or totally ordered (we assume totally). It may be derived from real time or be independent of it (we will give examples of both). Virtual times may be visible to programmers to be manipulated as values, or they may be hidden from them and manipulated implicitly (exactly as with their spatial counterparts, virtual addresses.) The virtual times associated with events may be explicitly calculated by user programs or assigned by fixed default rules (again, we give examples of both). And, there are many conventions that may be established to relax the restrictions of Rules 3 and 4. Each set of choices defines a different kind of virtual time systems, but all are similar enough that a unified approach to the theory and implementation is appropriate.

3. Comparison to other work involving artificial time scales

Recently there have been a number of proposals published for synchronizing distributed systems using artificial time scales. We now briefly contrast three of them with the virtual time paradigm and the Time Warp mechanism.

3.1 Lamport's work

Lamport [Lamport 78] seems to have been among the first to recognize that our understanding of real-time temporal order, simultaneity and causal relations between events in a distributed system bears a strong resemblance to our understanding of the same concepts in special relativity. In particular he showed that the temporal relationships that are operationally definable within a distributed system form only a partial order instead of a total order, and that "concurrent" events are incomparable under that partial order. He further showed that it is always possible effectively to extend this partial order to a total order by defining a system of artificial clocks, one clock for each process, that label each event with a value from a totally ordered set in a manner consistent with the partial order.

We can describe Lamport's work as starting from a particular execution of a distributed system and ending with an assignment of totally ordered clock values to the events of that execution. With virtual time we are doing the exact reverse. We *assume* that every event is labelled with a clock value from a totally ordered set (the virtual time scale) in a manner obeying Lamport's Clock Conditions. The problem addressed by the Time Warp mechanism is to find an execution that is both consistent with this labelling and that exhibits high concurrency.

3.2 Reed's work

In his study of distributed systems Reed invented the notion of *pseudotime,* which bears a strong superficial resemblance to virtual time but which has a very different motivation and implementation. Reed is primarily interested in implementing distributed atomic actions, and his work has been used as the basis for the design of multiversion timestamp order mechanisms for transaction concurrency control in distributed databases. Virtual time, by contrast, has as its goal the creation of a temporal coordinate system in which distributed computation is embedded. It was inspired by analogies to physical space and time and to virtual memory. Atomicity is not a goal *per se,* but it is a synchronization effect that can be arranged trivially within virtual time.

Two actions occuring at different pseudotimes may be committed in either order. Reed's mechanism "attempts" to manage execution so that the action with the earlier pseudotime will be committed earlier, but this is only a heuristic. With bad luck or timing the action with the earlier pseudotime may have to be aborted and retried later with a later pseudotime after the other action is complete. But with virtual time there is no abortion and no retry with a new timestamp; synchronization is done by rollback and actions *must* be executed in virtual time order. One cannot specify in which order two atomic actions with different pseudotimes are to be executed; but one is forced to specify that order for different virtual times. One last difference is worth noting. Because Reed's mechanism uses abortion and retry for synchronization, and there is no limit to the number of times an action may have to be retried, starvation is a potentially serious problem. There is no corresponding hazard with virtual time and the Time Warp mechanism.

3.3. Schneider's work

Schneider has done a more general study of synchronization [Schneider 82], in which he presents a general mechanism for implementing essentially any synchronization protocol in a distributed environment. His technique is based on broadcasting all synchronization-related messages ("phase-transition messages") to every process in the system, with every process in turn braodcasting its acknowledgement to all other processes, thus making tham all aware of every synchronization in the system. All such messages and acknowledgements are timestamped with values from a valid clock system such as Lamport's so that all processes agree both on what synchronization messages have been broadcast, and on the logical order in which they were broadcast. Broadcasting all synchronization messages and acknowledgements is apparently logically equivalent to keeping synchronization information in a globally accessible shared memory.

Under the assumption of reliable, order-preserving message delivery Schneider also shows that each process may make synchronization decisions (such as whether to proceed or stay blocked) locally, based on the set of "fully acknowledged" messages it has received. A message is considered full acknowledged at process P if P has received it as well as copies of the acknowledgements to it from every other process in the system. The importance of recognizing when a message m is fully acknowledged is that the receiver is then guaranteed that it will never again receive a message or acknowledgement with a timestamp earlier than that of m.

Schneider's mechanism can be compared to virtual time in that it does assign temporal coordinates to some of the actions in a distributed system, namely the synchronization actions. But where the Time Warp mechanism is extremely "liberal", making synchronization decisions on a provisional basis and rolling back when they turn out to be wrong, Schneider's mechanism is extremely "conservative", waiting to make such decisions until such time as it can be proved that they cannot be wrong[1]. One disadvantage of Schneider's mechanism is that it seems to be limited to systems with only a few processes; it does not scale upward smoothly to thousands of processes because of the prohibitive amount of message and acknowledgement processing inherent in mechanisms relying on broadcast. The Time Warp mechanism seems to have no such barrier to indefinite scale-up.

4. The Time Warp mechanism

The Time Warp mechanism is defined without reference to any underlying computer architecture and can run efficiently on many multiple processor systems, from a tightly coupled multiprocessor such as C.mmp[Wulf 81] or Cm*[Swam 77], to a local area network connected by Ethernet[Metcalfe 76].

[1] This observation is due to J.C. Brown.

We assume that message communication is reliable, but we do not assume that messages are delivered in the order sent; in fact such a protocol would be wasteful because messages are not generally processed exactly in sending order.

For correct implementation of virtual time in accordance with Rules 1-4, it is necessary and sufficient that *at each process* messages are accepted in virtual receive time order and events are executed in virtual time order. It is unnecessary, and generally undesirable, for an implementation to require that all processes progress through virtual time at the same rate with respect to real time, or to require that at each instant of real time all processes must be executing events of the same virtual time. In general, some processes may be ahead in virtual time and others may lag behind. It is not obvious how this criterion can be met in an efficient implementation because messages will not generally arrive at their destinations in virtual receive time order. Furthermore it is impossible for a process, on the basis of local information alone, to block and wait for the message with the "next" virtual receive time because, since we assume virtual times to be real numbers, no matter which one is presumed to be next it is always possible that another message with an earlier virtual receive time will arrive later. This is the central implementation problem that the Time Warp mechanism addresses.

The Time Warp mechanism has two major parts, the *local control mechanism*, concerned with making sure that events are executed and messages received in correct order (providing a "weakly correct" implementation of virtual time), and the *global control mechanism*, concerned with global issues such as space management, flow control, I/O, error handling and termination detection (contributing to its "strong correctness"). We discuss these in turn.

4.1. The local control mechanism

Under the Time Warp mechanism the world is viewed as a collection of processes that communicate with one another via messages. Although abstractly there is a single global standard of virtual time, there is no global virtual clock variable in the implementation that is accessible to processes; instead each process has its own *local virtual clock* variable that it may read. At any moment some local virtual clocks will be ahead of others, but this fact is invisible to the processes themselves because they can read only their own virtual clock. The virtual send time of a message is always copied from the sender's virtual clock. The virtual receive time may be assigned by any one of a variety of conventions . All interactions between processes are by message, including such things as input/output and process creation.

Because it is impossible to wait for the "next" message each process executes continuously, processing those messages that have already arrived in increasing virtual receive time order as long as it has any messages left. All of its execution is provisional, however, as it is constantly gambling that no message will every arrive with a virtual time stamp less than the one stamped on the message it is now processing. As long as it wins this bet execution proceeds smoothly. The novelty of the Time Warp mechanism is that whenever the bet is lost the process must pay by rolling back. We might describe the situation differently by saying that each process is constantly doing a "lookahead" in its input message queue, and in any kind of lookahead scheme there are certain comparatively infrequent contingencies that require the undoing of some work already accomplished. Lookahead is successful however, when the overhead of occasional undoing is outweighed by increased performance the rest of the time.[2]

In most practical situations there will be more processes than processors, so only a subset of the processes can run at any one moment. The natural scheduling rule is to always run those processes whose local virtual clocks are farthest behind. On a multiprocessor this means always running the *n* farthest behing processes, where *n* is the number of processors available. On a network it means always running, on each processor, the farthest behing process residing on that processor.

A process has a single input queue in which all arriving messages are stored in order of increasing virtual receive time. Ideally the execution of a process is simply a cycle in which it executes its events in increasing virtual time order. An event consists fo the following three actions executed sequentially.

1. A process receives the message from the input queue whose virtual receive time is the same as the time in its local virtual clock.
2. It performs the actions appropriate in response to the message,
3. It updates its local virtual clock to the time of the next message in the input queue (or to + infinity if there is none).
4. and repeats (or "terminates" if the virtual clock reads ∞).

Whenever there is no next message a process *terminates*, but its state is not destroyed because it may later roll back and *unterminate*. This ideal scenario applies as long as no message ever arrives so "late" that the receiver's virtual clock has a value greater than the virtual receive time stamped on the message, i.e. arrives with a virtual receive time in the "past". But this is bound to happen occasionally for any of several reasons, e.g. because the sender's virtual clock has not advanced as far as the receiver's or because of transmission delays in the network transport mechanism. While the incidence of late messages may be low (and can be made as low as desir-

[2] I am indebted to Tim Standish at Irvine for this characterization.

ed by introducing artificial delays), it cannot be reduced to zero because it is fundamentally dependent on the vagueries of computation speeds and transmission delays.

Whatever the reasons for the late arrival of a message, the semantics of virtual time demands that messages be received by each process strictly in virtual receive time order, and the only way to accomplish this is for the receiver to roll back to an earlier virtual time, cancelling all intermediate side-effects, and begin to execute forward again, this time receiving the late message in its proper sequence.

Rollback in a distributed environment is complicated by the fact that the process in question may have sent any number of messages to other processes, causing side effects in them and leading them to send still more messages to still more processes, and so on. Some of those messages may have requested output or some other irreversible action (dispense money, launch missile). Some of them may be physically in transit and therefore out of the system's control for finite but arbitrary durations. The paths followed by these direct and indirect messages from process to process may not form a tree, but may converge or even contain cycles, leading to worries about infinite loops or deadlock in any rollback mechanism. Nevertheless, all such messages, direct or indirect, in transit or not, causing output or not, must be effectively "unsent" and their indirect side effects, if any reversed. The Time Warp rollback mechanism is able to accomplish all this quite efficiently, and without stopping any part of the system.

4.2 Antimessages and the rollback mechanism

The name "Time Warp" derives from the fact that the virtual clocks of different processes need not agree, and the fact that they go both forward and backward in time. Over a lengthy computation each process may roll back many times while generally progressing forward. The fact that virtual clocks are sometimes set back does not violate our intention that "abstractly a virtual clock always progresses forward (or at least never backward)" because rollback is completely transparent to the process being rolled back. Programmers can write software without paying any attention to the possibility of messages arriving late, and even without any knowledge of the issue at all, just as they can write without any attention to, or knowledge of, the possibility of page faults in a virtual memory system.

To understand the rollback mechanism we must understand the nature of processes, messages and antimessages. In Fig. 4.1 we see the structure of a process named A. The blank fields in the figures are fields whose values are irrelevant in this description. A process is composed of:

1. a *process name* (virtual space coordinate), which must be unique in the system.
2. a *local virtual clock*, which in the figure reads 181, indicating that the message with receive time 181 is being processed.

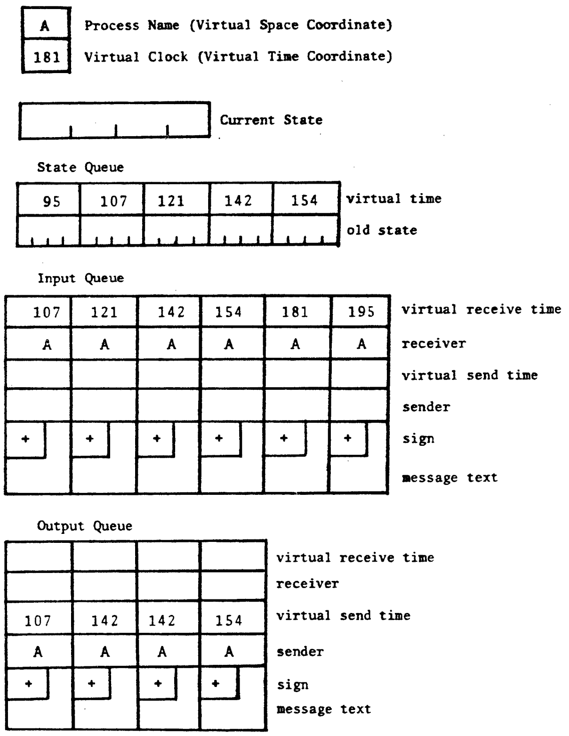


Figure 4-1: Process A with input queue, output queue and state queue.

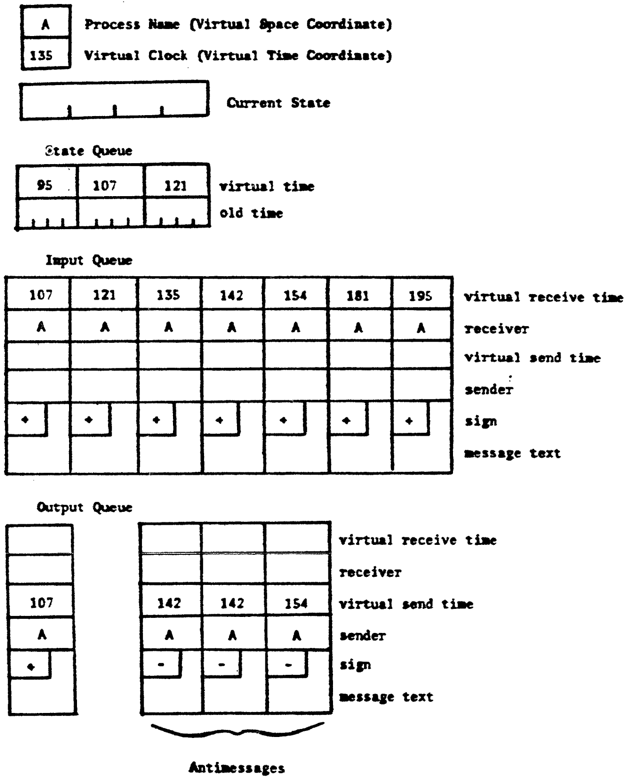


Figure 4-2: Process A rolls back to time 135 and sends antimessages.

3. a *state* which in general is the entire data space of the process, including its execution stack, its *own* variables, and its program counter. We show here only a few *own* variables to represent the whole state.
4. a *state queue*, containing saved copies of the process's recent states. As we shall see when we discuss the global control mechanism, it is not necessary to have states saved all the way back to the beginning of virtual time, but there must be at least one state saved that is older than *global virtual time (GVT)*. In this figure GVT is assumed to be 100.
5. an *input queue* containing all recent incoming messages sorted in order of virtual receive time. Some of these messages have already been received because their virtual receive times are less than 181. Nevertheless they are not deleted from the queue because it may be necessary to roll back and reprocess them. Other messages with virtual receive times greater than 181 have not yet been received, or else they have been received and "unreceived" and equal number of times. Only incoming messages whose virtual send time is greater than or equal to GVT must be saved.
6. an *output queue* containing copies of the messages it has recently sent, kept in virtual send time order. They are needed in case of rollback, for it will then be necessary to know which messages have been sent and must be "unsent". Only messages whose virtual send time is greater than or equal to GVT need be saved.

Consider now the situation that arises if a message with virtual receive time 135 arrives, as in Fig. 4.2. It is apparent that all of the work that was done by this process since virtual time 135 (in fact, since 121, the latest event earlier than 135) must be undone by rollback. The first step in the rollback mechanism is simply to search the state queue for the last state A was in before time 135 and restore it. We also restore 135 as the value in A's local virtual clock. After this we can discard from the state queue all states saved at or after time 135 and start A executing forward again. However, we still must correct for the fact that between time 135 and 181 A sent several messages to other processes that must now be "unsent". We accomplish this through an extremely simple device: *antimessages*.

Every message has an antimessage that is exactly like it in format and content except in one field, called its *sign*. Two messages identical except for opposite signs are called antimessages of one another. All messages sent by user programs have a positive (+) sign; their antimessages have a negative (-) sign and are only created during rollback. Negative messages are manipulated exactly as positive messages are. When a negative message is sent it is enqueued in the output queue of the sender according to its virtual send time, and a copy is delivered to the destination process and enqueued in its input queue according to its virtual receive time. A negative message causes a rollback at its destination if its virtual time is less than or equal to the receiver's virtual clock time, just as a positive message would.

What makes antimessages so useful is the pecular queueing discipline they follow. Whenever a message (positive or negative) and its antimessage occur in the same queue, they immediately *annihilate* one another. Thus, the result of enqueueing a message in the output or input queue of a process can be to shorten it by one rather than lengthen it. It does not matter which message, negative or positive, arrives at the queue first; if and when the second one arrives the annihilation takes place. It is perhaps unnecessary to point out that the behavior of messages and antimessages is reminiscent of the behavior of particles and antiparticles in physics (except for the fortunate lack of gamma rays).

The rollback of process A is completed simply by sending antimessages for all of those messages that it sent between times 135 and 181. In each case a copy of the negative message is first enqueued in the output queue of A, causing annihilation of the positive copy that was there. The result is that the A now has no record that those messages, positive or negative, ever existed, and it is truly in the state it would have been if the message with virtual receive time 135 had arrived in its proper order. Antimessages are also delivered to their desintations with the following possible effects. (a) if the original (positive) message has arrived but has not yet been processed, then the negative message, having the same virtual receive time, will also be in the virtual future of the destination and will not cause a rollback. It will, however, cause an annihilation, leaving the system with no record that either message ever existed, exactly as we would want if the message were to be truly "unsent". (b) Another possibility is that the original positive message has a virtual receive time that is now in the past with respect to the receiver's virtual clock, meaning that it and possibly others with later virtual receive times have already been processed, causing side effects on the receiver's state, and causing the sending of more messages to a third level of processes. In this circumstance the negative message, having the same virtual receive time as the positive one, will also arrive in the receiver's virtual past and will cause it to roll back to the virtual time when the positive one was received. It will also annihilate with the positive one, so that when the receiver starts executing forward again the situation will again be as though neither message had ever existed. As part of this secondary rollback more antimessages will be sent to the third level of processes, and the same actions we have been describing will proceed recursively. (c) There is a third case as well, namely that the negative message arrives at the destination *before the positive one*. In this case it is enqueued as usual, and will eventually be annihilated when the positive message finally arrives. If the negative message is actually received by the destination process before it is annihilated by the positive message, the receiver may take any action at all, e.g. a no-op. Any such action will eventually be rolled back when the positive message arrives.

This recursive antimessage/rollback protocol is extremely robust, and works correctly under all possible circumstances. The levels of indirection may be to any depth, and there may even be circularity in the graph of antimessage paths with no ill effects. The rollback process need not be atomic, and indeed many interacting rollbacks may be going on simultaneously with no special synchronization. There is no possibility of deadlock, and the system does not have to be stopped. The worst case is that *all* processes in the system roll back to the same virtual time as the original one did, and then proceed forward again.

Obviously the rollback mechanism can be extremely costly in the worst case, but here are a number of arguments suggesting that in a realistic system it is not nearly as costly on the average as one might imagine. First, most systems operate in a pattern where each event involves one input message and one output message. Hence, the number of antimessages *directly* sent by any one rollback is approximately the number of events rolled-back over. There is reason to believe that most programs obey the *temporal locality principle*, namely that most messages arrive in the future according to the local virtual clock at their destination, thus not causing any rollback at all, and those that arrive in the virtual past tend strongly to arrive in the *recent* past so that few events are rolled back. Even where a rollback causes several antimessages to be sent we can expect that most of them will not cause secondary rollbacks simply because they each have virtual receive times greater than or equal to that of the message that caused the original rollback, and generally the higher the virtual receive time of a message, the less likely it is to cause rollback. The extent to which the temporal locality principle applies is obviously application-dependent and can only be verified empirically. One final argument is important: the "cost" of this king of synchronization is only the cost of the rollback and antimessage overhead; the time it took to perform the computation being rolled back is not part of the cost, because the only alternative would have been to be blocked for the same length of time anyway.

It is important in virtual time systems that process creation and input/output be done by message so that they may be undone by antimessages just as any other side effects. The Time Warp mechanism is thus able to uncreate, unterminate, unerr, uninput and unoutput. We will return to these subjects in the next section.

4.3 The global control mechanism

The local control part of the Time Warp mechanism leaves a number of critical issues unresolved. How can we be sure that amidst all of the rollback activity the system makes progress globally? How can global termination be detected? How can errors and I/O be handled in the face of rollback? How can we avoid running out of memory when the local control mechanism calls for saving two copies of all messages and several copies of processes' states? The global control mechanism resolves all of these issues and several others.

The central concept of the global control mechanism is *global virtual time (GVT)*. Global virtual time is a property of an instantaneous snapshot of a virtual time system, and is defined to be the greatest lower bound of the set of all virtual times shown by all virtual clocks *in this or any possible future snapshot*. (See [Jefferson 83a] for detailed discussion of this definition, and for proof that GVT is well-defined in the case of infinite computations.)

GVT serves as a floor for the virtual times to which any process can ever again roll back. It can be proved [Jefferson 83a] that the theoretical definition of GVT for an instantaneous snapshot can be characterized operationally as the minimum of (a) all virtual times in all local virtual clocks in the snapshot, (b) all virtual send times in unreceived messages in the input queues of the snapshot, and (c) all virtual send times in messages that have been sent but not yet acknowledged (and may therefore be in transit at the moment of the snapshot). This characterization leads to a fast, distributed GVT-estimation algorithm [Jefferson 83a] that takes O(d) time, where d is the delay required for one broadcast to all processors in the system. The GVT algorithm runs concurrently with the main computation and returns a value that is between GVT at the time of the start of the algorithm and GVT at the time of its completion. It thus gives a slightly out-of-date value for GVT, which is fundamentally the best one can do without stopping the entire system.

It is a simple consequence of the definition that GVT *never decreases*, because any quantity defined to be the minimum of all future values of another quantity must be nondecreasing. In fact, we can prove that if every event terminates (and if a few other conditions hold [Jefferson 83a]) then GVT eventually increases because the scheduling rule gives priority to the farthest behind processes. This nondecreasing property makes it appropriate to consider GVT as the virtual time for the system as a whole, and to use it as the measure of system progress. It measures how much of the system's activity is final and complete. Since GVT is a global property of an instantaneous snapshot there can be no way for a process to have access to its "current" value, but it has no logical need for that information because its own progress is measured by the value in its local virtual clock.

During the execution of a virtual time system the Time Warp mechanism must calculate GVT every so often. The actual frequency is a tradeoff: high frequency produces faster response time and better space utilization (because of more frequent storage reclamation, to be discussed later) but also lower processor utilization and slower progress (measured for example in units of virtual time per second of real time). Except for the space savings this is exactly the same tradeoff we are familiar with in time-slicing operating systems when we adjust the length of the time quantum.

The next few sections describe the uses of GVT to control virtual time systems.

4.3.1. Normal termination detection

Termination detection in distributed systems has been an active field of research for some time now, with numerous papers published on the subject, e.g. [Dijkstra 80]. With the Time Warp mechanism, however, the detection of termination of virtual time systems is just one of several global problems defined by and solved in terms of GVT. Because we have assumed the processes do not spontaneously send messages we know that when a process runs out of messages it terminates normally and its local virtual clock is set to ∞ . When GVT reaches ∞ it means that all local virtual clocks read ∞ and no messages are in transit, so no process can ever again "unterminate" by rolling back to a finite virtual time. Thus, whenever the periodic GVT calculation returns ∞ , the Time Warp mechanism signals termination.

4.3.2. Memory management and flow control

One of the features of the Time Warp mechanism is that it is possible to give simple, natural algorithms for managing memory. In addition to the memory used by the code and current data of processes (which the programmer is responsible for managing) there are four kinds of memory overhead to be managed.

1. Old states in the state queues.
2. Messages stored in output queues.
3. "Past" messages (in input queues) that have already been processed.
4. "Future" messages (in input queues) that have not yet been received.

The first three classes of storage, used only to support rollback, are all managed similarly. Any message, input or output, whose virtual receive time is less than GVT can be discarded, as it is impossible to roll back to a virtual time when it might be either re-received or cancelled. Similarly, for each process all but one saved state older than GVT can be discarded. This recycling of outdated memory is called *fossil collection*.

Managing the fourth class of storage, that containing unreceived messages, is essentially the flow control problem common to all distributed systems. But because we assume every message is stamped with the virtual coordinates of the sending and receiving events, and because rollback is possible, the flow control problem has more structure than it usually does and a new approach is warranted. In most environments, where the only synchronization tool is process blocking, flow control protocols act as a valve to limit the flow of messages from sender to receiver. The receiver must be careful never to accept too many messages, because every message it accepts responsibility for it must buffer. But under the Time Warp mechanism, if a receiver's memory is full of input messages he can always make space by *sending back* an unreceived message. The original sender must then roll back to the state it was in when it sent the message, and it will resend the message as it executes forward again. The natural message to send back is the unreceived message having the latest virtual send time (regardless of where it came from).

Returning a message to the sender is the message-analog of process rollback; they are obverse and reverse of the same coin. We do not have space here to give more detailed arguments in favor of this protocol, but it seems to offer very efficient and simple flow control for virtual time systems. Of course this hypothesis also needs empirical verification.

4.3.3. Error detection

When a process commits a run-time error its state must be marked ERROR. This should not necessarily cause termination of the entire computation because the erring process might roll back and "unerr", sending an antimessage for the message containing the error indication. An error is only "permanent" if it is impossible for the process to roll back to a virtual time earlier than that of the error.

A process in an ERROR state keeps executing, but all successive states are also ERROR states. If and when an ERROR state is fossil-collected (because its virtual time is older than GVT) then no rollback will ever undo the error. It should then be reported to some error policy software and/or to the user.

4.3.4. Input and output

When a process sends a command to an output device or any other agent external to the Time Warp mechanism it is important that the physical output activity not be committed immediately because the sending process may roll back and cancel the output. The criterion for deciding when a command to an output process can actually be performed is that GVT must exceed the virtual receive time of the message containing the command. After that point no antimessage for the command can ever be generated, and the output can be physically committed.

This policy is implemented in the Time Warp mechanism by designating certain processes as *output processes*. Such processes have null bodies and act primarily as input queues. If a message to an output process is not annihilated by an antimessage it will remain there until it is fossil-collected, at which point it is known that no rollback can ever occur to cancel the output. Fossil collection then, in addition to recovering memory and detecting errors, must also physically commit all output action. Similar considerations hold for input.

4.3.5. Snapshots and crash recovery

The problem of taking a consistent, global snapshot that is useful for continuation in the event of a crash arises with all distributed systems [Russell 80]. A snapshot at a single instant of real time is, of course, theoretically impossible to implement because we cannot have perfectly synchronized real time clocks [Lamport 78]. In a virtual time system a snapshot of all processes and the relevant messages at a particular virtual time (which must be taken at different real times for different processes) forms a natural and meaningful snapshot of the system that is easily implementable. A full (nonincremental) snapshot of the system at virtual time t can be constructed by a procedure in which each process snapshots itself as it passes virtual time t in the forward direction, and "unshapshots" itself whenever it rolls back over virtual time t. Whenever GVT exceeds t the snapshot is complete and valid. A variation on this procedure can provide an incremental snapshot facility. Because of space limitations further details must be postponed to a future paper.

5. Examples of virtual time systems

A wide variety of distributed systems can be viewed as virtual time systems. In the next three subsections we give examples to illustrate the range of the concept, and in particular, we show how both the distributed discrete event simulation problem and the distributed database concurrency control problem are instances of the virtual time paradigm.

5.1. Example 1: Distributed discrete event simulation

The most extensively studied virtual time system is distributed discrete event simulation, in which every process represents an object in the simulation and virtual time is identified with simulation time. The fundamental operation in discrete event simulation is for one process to schedule an event for execution by another at a later simulation time. In a virtual time system we emulate this action simply by having the first process send an *event message* to the second process with its virtual receive time equal to the event's scheduled simulation time. The requirement that each process must receive messages in virtual receive time order is equivalent to the fundamental semantic requirement of simulation that events be executed in simulation time order. Simulation is clearly one of the most "general" applications of the virtual time paradigm because the virtual times (simulation times) of events are completely under the control of the user, and because it makes use of almost all of the degrees of freedom allowed in the definition of a virtual time system. Any mechanism for general distributed discrete event simulation can be used as an implementation of virtual time. Chandy and Misra in [Chandy 81] give a simulation method based on blocking and distributed deadlock detection. See [Jefferson 82] for a detailed comparison.

5.2. Example 2: Distributed database concurrency control

In a distributed database the fundamental synchronization problem is to make distributed transactions appear to be atomic with respect to other transactions. To accomplish this effect in a virtual time system it is only necessary to do two things. First, the entire database system, including all transaction software, management software, and even the data, must be cast as a collection of processes communicating by message. In particular, data items must be viewed as stunted processes that respond primarily <u>read</u> and <u>write</u> messages. Second, the system must <u>ensure that</u> each transaction must execute within a band of virtual time that does not overlap with the bands allocated to other processes. This can be done simply by using the real time of a transaction's initiation as the high order bits of the virtual time band allocated to it (with the place of intiation as middle-order bits to break ties). The apparent indivisibility execution of transactions follows directly. A full description of virtual time as the basis of database concurrency control can be found in [Jefferson 83b].

This assignment of virtual time bands to transactions guarantees not only that they are atomic, but that they are apparently executed (committed) strictly in virtual time order, i.e., in the order of their initiation. This is a stronger scheduling constraint than the usual serializability criterion. Of course, although the transactions appear to be executed sequentially in virtual time order, they are actually executed by the Time Warp mechanism concurrently, or in any convenient order.

Virtual time used for database concurrency control is quite different from that used for distributed simulation. First, virtual time is derived from real time in the database example, whereas it is completely independent of real time in the simulation case. Second, virtual time values in simulation are actually manipulated by user software as ordinary data. In databases the behavior of the system is time-independent and the virtual times are presumably "hidden" from most levels of DBMS software.

In many respects the Time Warp mechanism applied to database concurrency control is similar to *multiple-version concurrency control mechanisms* [Papadimitriou 82] in that it maintains several successive versions of each data item and it has the ability to achieve serializability in spite of transaction collisions because it can satisfy a *read* request that is time-stamped earlier than the "current" version of the data by simply accessing a saved earlier version. But when a multiple-version mechanism is faced with a *write* request that is time-stamped earlier than the current version of the data there is no choice but to abort the entire transaction that the write request is part of (and possibly some other transactions in progress as well), and to restart it with a later time stamp. In situations where there is a high probability of transaction collision this leads to alot of wasted computation and to the possibility of starvation. The Time Warp mechanism, however, never aborts a transaction. It may roll back parts of several transactions when a collision occurs, but the amount of the computation unwound during a collision is limited to that part that would be causally affected if requests were handled out of time stamp order.

Other parts of the colliding transactions that are not functionally dependent on the data involved in the collision are not rolled back.

5.3. Example 3: Virtual circuit communication

One of the main functions of a network communication protocol is to provide a virtual circuit facility, a buffering and synchronization mechanism that, for each sender-receiver pair, delivers messages to the receiver in the same order they were sent. This effect can be accomplished automatically in a virtual time system if the virtual receive time of a message is defined to be the real time of its sending. The requirement that messages be processed in virtual receive time order is then the same as requiring them to be processed in sending order. Implementing virtual circuit communication as a virtual time system does not confer any particular speed or concurrency benefits over other (simpler) implementations, but it does show something of the breadth of the virtual time concept. There may, however, be important practical benefits in environments where the ability to checkpoint a distributed system is required.

6. Extended analogy to virtual memory

Now that we have presented the outlines to the theory we can explain the use of the term "virtual time". I see virtual time as the natural temporal analog of virtual memory, and have been consciously guided by the lessons of virtual memory systems from the beginning of this work. There is a compelling extended analogy between the two which may lend some credibility to this otherwise unorthodox approach to distributed computation. Because of space limitations we will present the comparison as a sequence of parallel concepts in which space and time play almost symmetric roles.

- A page is analogous to an event. The virtual address of a page is its spatial coordinate; the virtual time of an event is its temporal coordinate.
- A page in memory at time t is analogous to an event in the future of process x; a page out of memory at time t is analogous to an event in the present or past of process x.
- Accessing a page in memory is comparatively inexpensive, but accessing a page out of memory causes a very expensive page fault. Analogously, sending a message into the virtual future of a process is comparatively inexpensive, while sending a message into its virtual past causes a very expensive *time fault*, i.e., rollback.
- In view of this it is only cost-effective to execute programs under a virtual memory system that obey the spatial locality principle, i.e., that most memory accesses are to pages already resident in memory so that page faults are rare. Likewise, it is only cost-effective to run programs under a virtual time system that obey the temporal locality principle, i.e., that most messages arrive in the virtual future of the destination processes so that time faults are rare.
- Memory mapping converts virtual addresses to real addresses at different times. *Time mapping* (also known as *scheduling!*)maps virtual times to real times; the same virtual time may be scheduled at different real times in different processes.
- The only acceptable memory maps are the one-to-one functions because they preserve distinctness, i.e., map distinct virtual addresses to distinct real addresses. The only acceptable time maps are the strictly increasing functions because they preserve distinctness *and order*, i.e., map distinct virtual times into distinct real times, earlier virtual times into earlier real times, and later virtual times into later real times.

The analogy between virtual time and virtual memory can be greatly extended (working set, thrashing, pointer, data structure) and it is very thought-provoking to do so. It suggests that the concept of virtual time can provide the same kind of clean, efficiently implementable abstraction of the time resource in a distributed environment that virtual memory has provided for space resources.

7. Future work

There is still a tremendous amount of work to be done to follow up on this research. For example, there is very little empirical or analytical work published on the performance of systems incorporating rollback; this is the most important immediate issue. In the longer term, a programming language for concurrent systems with virtual time as its semantic basis seem reasonable. It should also be possible to incorporate virtual time into specification languages designed for specifying and verifying the synchronization requirements of concurrent systems. Finally, it seems natural to suppose that a generalization of both virtual memory and virtual time--virtual spacetime--should be "out there", waiting to be explicated. The field seems wide open.

8. Acknowledgements

I would like to thank Henry Sowizral for many hours of discussion during our co-invention of the Time Warp mechanism, and the Rand Corporation for providing the initial impetus to study the distributed simulation problem, which eventually led to this work.

9. References

[Bernstein 81] Bernstein, Philip A. and Goodman, Nathan, "Concurrency Control in Distributed Databases Systems," *Computing Surverys* 13, June 1981.

[Bernstein 82] Bernstein, Philip A. and Goodman, Nathan, "A Sophisticate's Introduction to Distributed Database Concurrency Control," in *Proceedings of the Eight International Conference on Very Large Databases (VLDB)*, September 1982.

[Chandy 81] Chandy, K.M. and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *CACM* 24 (11), April 1981, 198-206.

[Dijkstra 80] Dijkstra, Edsger W. and Scholten, C.S., "Termination Detection for Diffusing Computations," *Information Processing Letters 11,* (1), August 1980.

[Jefferson 82] Jefferson, David R. and Sowizral, Henry A., *Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control,* The Rand Corporation, Santa Monica, Cal., Technical Report, December 1982.

[Jefferson 83a] Jefferson, David R. and Sowizral, Henry A., *Fast Concurrent Simulation Using the Time Warp Mechanism, Part II: Global Control,* The Rand Corporation, Santa Monica, California, Technical Report, August 1983.

[Jefferson 83b] Jefferson, D. and Motro,A., *The Time Warp Mechanism for Database Concurrency Control,* University of Southern California, 1983.

[Lamport 78] Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM* 21, (7), July 1978, 558-565.

[Metcalfe 76] Metcalfe, R.M. and Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM 19, (7),* July 1976, 395-404.

[Papadimitriou 82] Papadimitriou, Christos H. and Kanellakis, Paris C., "On Concurrency Control by Multiple Versions," in *Conference on Principles of Database Systems (PODS),* ACM, 1982.

[Russell 80] Russell,David L., "State Restoration in Systems of Communicating Processes," *IEEE Transactions on Software Engineering* SE-6, (2),March 1980, 183-194.

[Schneider 82] Schneider, Fred B., "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems 4, (2)* April 1982, 179-195.

[Swan 77] Swan, R., Fuller, S.H. and Siewiorek, D., "CM*--A modular, multimicrocomputer," in *Proceedings 1977 National Computer Conference*, pp. 637-644, AFIPS Press, Baltimore, MD, 1977.

[Wulf 81] Wulf, W.A., Levin, R. and Harbison, S.P., *HYDRA/C.mmp: An Experimental Computer System,* McGraw-Hill, New York, NY, 1981.

PROCESS MANAGEMENT OVERHEAD IN A SPEEDUP-ORIENTED MIMD SYSTEM

Ruknet Cezzar
The City of New York
Human Resources Administration

David Klappholz
Polytechnic Institute of New York
Computer Science Division

Abstract -- The effects of Process Management Software (PMS) on the throughput of an MIMD machine are studied. Queueing-theoretic models are used to predict the effective throughput where only the running of PMS is taken into account. Performance problems with Centralized PMS, a method of decentralization through multiple ready lists, and the required extent of decentralization are discussed and analyzed.

As a way of avoiding centralized mechanisms and the resulting performance bottleneck conditions, a policy is studied where i) multiple ready lists are used, ii) processors access the ready lists randomly and uniformly. The effective throughput for this policy is obtained by an approximate solution to a highly complex closed queueing system. This randomized-access policy is shown to achieve effective throughputs almost linear in n, for an n-processor MIMD system. The policy is also shown to achieve speed up factors very close to n, regardless of how large n is.

1. Introduction

An MIMD machine is a tightly-coupled system in which a large number of processors share primary memory, and perform parallel computations in a cooperatively-tasked manner. If contention for shared computing resources and the processing overhead associated with the management of concurrency are minimized, high computational speeds can be achieved.

This paper deals with the performance of Process Management Software (PMS) for achieving high throughput on the assumption that performance issues at the architectural level have been resolved. PMS overhead and its minimization are discussed and analyzed. Assuming that the only performance overhead is the running of PMS; parallelization of PMS to achieve nearly constant processor utilization, (i.e., speedup factors very close to n for an n-processor system) is also studied.

The architectural features of an MIMD machine which are relevant to the analyses in this paper are:

- Identical processors,
- Uniform access to common memory, where no processor is favored over the others;
- Uniform sharing of peripherals and secondary storage, in a way similar to the sharing of primary memory.

2. The Process Management Software

The PMS of an MIMD machine has the following functions:

- Creating processes,
- Putting processes to sleep,
- Waking processes up,
- Assigning processes to processors,
- Destroying processes.

The PMS will consist of executive code and the system tables referenced by that code. The system tables are: i) Ready List(s) which contain the state vectors of processes ready for processor assignment; ii) Blocked Lists which contain the state vectors of processes blocked for shared resources.

In a centralized PMS there is one global ready list, and its use by the processors requires mutual exclusion.

In such a PMS, there is a degradation of effective throughput as the number of processors becomes large. In what follows we will study the extent of this degradation as well as the performance of PMS when it is decentralized through the use of multiple ready lists.

3. The Throughput Models

In queueing-analytic models, the processors are treated as active elements sharing the ready lists as resources. The throughput is measured in number of user-level instructions per unit time. This is the "effective" throughput which indicates the work done by the processors while not engaged in ready list access. System-software overhead, other than those associated with ready list accesses, is not taken into account. The effective throughput results may be interpreted accordingly.

The queueing models yield the average (expected) number of processors in normal execution (user-state). Multiplying this by the rated speed, same for all processors, gives the effective throughput. The relationship of this to other common performance measures for multiple-processor systems is given below. Let:

s = Rated speed of each processor.

$\overline{K}_n$ = Avg. number of processors in user-state in an n-processor MIMD system.

F_n = Finishing time of a computation, when run on an n-processor MIMD system.

F_1 = Finishing time of the same computation, when run on a 1-processor system.

0190-3918/83/0000/0395$01.00 © 1983 IEEE

$\alpha = 1 - \overline{K}_1$ Percent of time a uniprocessor would spend in ready list access, with no contention from other processors.

In terms above, the performance measures for an n-processor system:

$$T_n = s\overline{K}_n \qquad \text{Effective Throughput.} \qquad (1)$$

$$S_n = \frac{T_n}{T_1} = \frac{\overline{K}_n}{1-\alpha} \quad \text{or} \quad S_n = \frac{F_1}{F_n} \quad \text{Speedup Factor.} \qquad (2)$$

$$R_n = \frac{S_n}{n} = \frac{\overline{K}_n}{(1-\alpha)n} \qquad \text{Relative Utilization Rate.} \qquad (3)$$

$$E_n = \frac{1}{R_n} \qquad \text{Relative Efficiency.} \qquad (4)$$

4. Centralized PMS Model

Suppose there is a single copy of the ready list, residing in common memory, and accessible to all processors. For this queueing model and for subsequent models, the following general assumptions are made:

- No processor is favored over the others in accessing the ready list(s);

- Processors are independent, and interact only with respect to ready list accesses.

In addition to n, and m = 1 (single ready list), we define the following input parameters:

$1/\lambda$: Actual (mean) time between ready list accesses, per processor.

$1/\mu$: Actual (mean) time inside the ready list, per ready list.

The parameters which are derived from the above and referred frequently are:

$\rho = \lambda/\mu$: Ready list access traffic intensity per processor.

$\alpha = \lambda/(\lambda+\mu)$: The PMS overhead for a 1-processor system as defined earlier.

These parameters are related as $\rho = \alpha/(1-\alpha)$ and $\alpha = \rho/(1+\rho)$.

As the target parameters, we define the following random variables, and their expectations:

K: Number of processors in normal execution, running user-level processes.

N: Number of processors engaged in ready list access, attempting to gain its lock or inside it.

Q: Number of processors attempting to gain access to the ready list, but not inside it. These processors are idled.

The results are obtained under two sets of assumptions:

1. Deterministic Case: The rate of ready list accesses and time spent inside the ready list are both constant. The model is solved for $\overline{K}$ = Average number of processors in normal execution.

2. Probabilistic Case: The rate of ready list accesses and time spent inside the ready list vary according to Poisson-exponential law. The aim is to find E(K) = Expected number of processors in normal execution.

4.1 Deterministic Case

The parameters λ and μ stand for the actual, rather than the mean, arrival and service rates. The purpose of deterministic assumptions is to uncover the best possible throughput with respect to ready list access and service patterns. Otherwise, these assumptions, especially with respect to the arrival rate are not very realistic.

This case can be analyzed as a closed queueing system where $\overline{K}$ stands for the (average) number of processors in normal execution. The transient behavior of the system depends on how initially processors access the ready list, and there are an infinite number of possible solutions for $\overline{K}$. On the other hand, because of the sequential server, even if all processors access the ready list at the same time initially, they would line up after a number of cycles. In other words, after each processor completes one cycle of normal execution and ready list access, the accesses to the ready lists must be at least $1/\mu$ time units apart. Depending on the value of ρ, this self-regulation places a limit on the maximum queue length. Keeping this self-regulation in mind, the steady state solution for K can be obtained by analyzing the following work-balance equation:

$$(1-P_0)\mu = \overline{K}\lambda, \quad P_0 = \text{Fraction of time no processors are engaged in ready list access.} \qquad (5)$$

Through a close analysis of a typical cycle of length $\lambda^{-1} + \mu^{-1}$, first P_0 is determined for the two distinct ranges $\rho < (n-1)^{-1}$ and $\rho \geq (n-1)^{-1}$. Then, the solution, as discussed in (1):

$$\overline{K} = \begin{cases} (\rho+1)^{-1}n = (1-\alpha)n & \text{for} \quad 0 < \rho < (n-1)^{-1}. \\ \rho^{-1} = (\alpha^{-1}-1) & \text{for} \quad (n-1)^{-1} \leq \rho < \infty. \end{cases} \qquad (6)$$

According to this, $\overline{Q} = 0$ in the first range, and $\overline{Q} = (\rho+1)^{-1} - \rho^{-1}$ in the second range. This means that the idled processors can grow at most linearly with n. As shown later, the growth is more rapid for stochastic systems.

4.2 The Probabilistic Case

The birth-death assumptions for this case were intended to simplify the analysis, and to find a conservative (i.e., nearly worst-case) throughput behavior with respect to ready list access and service patterns. Since one can conceive hyperexponential interarrival or service times, these assumptions do not necessarily correspond to the worst-case. However, since the closed queueing system is self-regulating, how far the worst-case can be from the birth-death case is an interesting and open question.

For this model, one way to find E(K) is to look up the solution for E(N), and then evaluate E(K) = n - E(N). The solution for E(N) corresponds to the Finite Population - Single Server Queue discussed in (10) and elsewhere:

$$E(N) = \sum_{i=0}^{n} iP_0\rho^i \frac{n!}{(n-i)!} \qquad (7)$$

$$\text{where } P_0 = \left[\sum_{i=0}^{n} \rho^i \frac{n!}{(n-i)!}\right]^{-1}$$ = Probability that no processors are engaged in ready list access.

From (7) above, the effective throughput of a 1-processor system is $(1-\alpha)s$. To find the effective throughput for $n > 1$, first P_0 and then $n - E(N)$ need to be evaluated. This could be time consuming for $n > 10$, which is likely to be the case for the MIMD system under study. As discussed in (1), if we solve directly for E(K), using the complementary birth-death formulation, simpler and more revealing results are obtained. According to this:

$$E(K) = \frac{1}{\rho}\frac{\displaystyle\sum_{k=0}^{n-1}\frac{(1-\rho)^k}{k!}}{\displaystyle\sum_{k=0}^{n}\frac{(1-\rho)^k}{k!}} \qquad (8)$$

The expression for E(K) above is free of P_0, and can further be simplified as:

$$E(K) = \frac{1}{\rho}\frac{F(n-1)}{F(n)} \qquad (8')$$ The expected number of processors in normal execution.

where F(•) is the cumulative d.f. for Poisson with mean $1/\rho$.

The expected number of idled processors are:

$$E(Q) = n - \frac{1}{\alpha}\frac{F(n-1)}{F(n)} \qquad (9)$$

The result in (9) is the same as E(idle) found in (7), except that it is in more convenient form for evaluation. Consider the numerical example discussed in (7) where $\rho = .05$ and $\alpha = .0476$. For a 14-processor system, we can look up F(13) and F(14) from Poisson tables and obtain:

$$E(Q) = 14 - \frac{1}{.0476}\frac{.0661}{.1049} = .747$$

This agrees with the results in (7) where the authors mention that "14-processor system has the effective power of 13.25 processors."

Figure-2 shows the results for $\alpha = .0476$ and $\alpha = .1$. The shaded areas provide a plausible performance range with Centralized PMS. The solutions to all hypoexponential interarrival/service times would lie in the shaded regions. For example, a highly realistic case of Poisson arrival rate and constant service times would lie in between deterministic and birth-death curves.

4.3 Conclusions for the Centralized PMS

As implied by the results of the two specific cases analyzed (deterministic, birth-death), there is an upper limit on the effective throughput. Since the average number of processors in normal execution can at most be ρ^{-1}, this upper limit is $s\rho^{-1}$ and does not depend on n. This result can be generalized to the case where no assumptions are made about ready list access and service rates, since the following work-balance equation holds:

$$(1-P_0)\mu = \overline{K}_n\lambda \qquad (10)$$

where P_0 is the fraction of time the server is idle, or the probability that all processors are in normal execution. K_n denotes K or E(K) for an n-processor system. From (10) above, it follows that:

$$\overline{K}_n = (1-P_0)\rho^{-1} \qquad (11)$$

Since $1-P_0$ is at most 1, $\overline{K}_n$ is at most ρ^{-1}, and the upper limit on the effective throughput is $s\rho^{-1}$, where s is the rated speed of each processor.

For example, if $\rho = .05$, on the average, there can at most be 20 processors in normal execution; even if the MIMD system employs 256 processors.

Notice that this result holds true even if interarrival or service times are probabilistically dependent (i.e., ready list requests are serially correlated, etc.). For the work-balance equation in (10) to hold, the only requirement is that the services are independent of the arrival process. For this MIMD system, there is no reason to the contrary. This result has further been generalized in (1), where the mean traffic intensities (ρ_i for processor i) are not necessarily equal. In that case, the upper limit on the effective throughput is $s\rho_{min}^{-1}$, where ρ_{min} is the minimum mean traffic intensity.

This result can more formally be stated as follows:

Rule #1:

For the type of n-processor MIMD system with Centralized PMS, there is an upper limit on the effective throughput. Since this upper limit does not depend on n, as n becomes large, the marginal contribution of additional processors to the effective throughput diminishes. If and when this upper limit is reached, the marginal contribution of additional processors to the effective throughput is nil.

Proof:

As was already shown, the effective throughput $T_n = s\rho^{-1}$ at most. As a restatement of the above, we must show that the relative utilization rate (i.e., relative to a 1-processor system) decreases as n increases, and tends to zero as n grows very large. From (3), by definition, the relative utilization rate is:

$$R_n = \frac{\overline{K}_n}{(1-\alpha)n} = \frac{1-P_0}{\alpha n} \quad \text{using the value for } \overline{K}_n \text{ in (11).}$$

Since $1-P_0$ can at most be 1, it is obvious that for $n > \alpha^{-1}$, R_n decreases as n increases. Furthermore:

$$\underset{n\to\infty}{\text{Limit}}\ R_n = \underset{n\to\infty}{\text{Limit}} \left[(1-P_0)(\alpha n)^{-1}\right] = 0 \, .$$

Similar arguments can be carried out for the case where ρ_i's are not the same for all processors. In that case, α will be replaced by α_{min}, where $\alpha_{min} = \rho_{min}/(1+\rho_{min})$.

5. Decentralized PMS Model

The results of the previous section point to the need for decentralization of PMS. Consider the deterministic case with a single ready list. This gives the best-case throughput for any given value of ρ, and the upper limit is reached when $n = \alpha^{-1}$. For example where $\alpha = .0476$, the upper limit is reduced when there are 21 processors in the MIMD configuration. If the MIMD system employs more than 21 processors, and if we cannot adjust the values of s and ρ by some other means, there is a need for decentralization of PMS.

The MIMD machine under study, being speedup-oriented, may be employing the fastest processors available. In that case, we cannot further increase the rated speed s, or run the PMS on a faster processor. The ready list access traffic ρ would depend on the parallel algorithm and the degree of interprocess communication, where a new process gets the processor when another processor is put to sleep. This parameter cannot always be adjusted to the desired value. Therefore, the PMS itself should be parallelized to run concurrently on multiple processors.

As the first step, we can use the decentralization scheme discussed earlier, where multiple ready lists are used. Then, there is the problem of how the processors should be accessing these ready lists, without any need for some other centralized mechanism.

The ready lists, each accessible to any one of the processors, may reside anywhere in common memory; preferably, in different memory banks to minimize memory conflict. To find the effective throughput when m (>1) ready lists are used, we can analyze the multiserver extension of the previous model. This is shown in Figure-1.

We again look at the deterministic and birth-death cases, and assume that:

1. The time spent inside a ready list does not depend on the number of ready lists used;

2. A processor always accesses a ready list which is currently available (unlocked). If none is available, it waits in idle state until one becomes available.

The first assumption above implies that the size of a ready list, in terms of the average number of activation records stored in it, does not have any effect on the time to enter or remove activation records. If the ready list is implemented as First In - First Out linked list of activation records, this assumption is highly realistic.

The second assumption implies that the processors know, without additional processing overhead, which ready lists are currently available. This additional overhead may be significant for the MIMD system considered, and is discussed in the next section. The purpose of this assumption is to find the minimum number of ready lists which, subject to the minimization or elimination of the additional overhead mentioned, will make it possible to achieve the desired throughputs.

5.1 The Multiserver Model Results

With mathematical arguments detailed in (1), the solution to the multiserver extension of the deterministic case is given as:

$$\overline{K} = \begin{cases} (\rho+1)^{-1}n = (1-\alpha)n & \text{for } \ 0 < \rho < m(n-m)^{-1}. \\ \rho^{-1}m = (\alpha^{-1}-1)m & \text{for } \ m(n-m)^{-1} \leq \rho < \infty. \end{cases} \tag{12}$$

As an interesting example, we can choose $m = \alpha n$ ready lists. For $\alpha = .1$, Figure-3(a) compares this to the centralized case (m=1) and to the fully-decentralized case (m=n). The case where $m > n$ is the same as the case m=n.

As intuitively expected, for constant ready list access and service rates, providing $m = \alpha n$ ready lists can achieve effective throughputs which grow linearly with n. As will be shown shortly, this is not necessarily the case for stochastic ready list access or service rates.

The solution to the multiserver extension of the probabilistic case, despite simplifying birth-death assumptions, is highly complex in the range $1 < m < n$. For this range, a highly concise solution for E(N) can be found in (11):

$$Pr(N=i) = \begin{cases} P_0 \binom{n}{i} \rho^i & \text{for } i = 0 \text{ to } m-1. \\ P_0 \binom{n}{i} \rho^i \dfrac{i!}{m!\,m^{i-m}} & \text{for } i = m \text{ to } n. \end{cases} \tag{17}$$

$$\text{where } P_0 = \left[\sum_{i=0}^{m-1} \binom{n}{i} \rho^i + \sum_{i=m}^{n} \binom{n}{i} \rho^i \frac{i!}{m!m^{i-m}} \right]^{-1}$$

$$E(Q) = \sum_{i=m}^{n} (i-m)\Pr(N=i) \quad \text{Expected number of idled processors.} \qquad (18)$$

$$E(N) = \sum_{i=m}^{m-1} i\Pr(N=i) + E(Q) + m\left[1 - \sum_{i=0}^{m-1} \Pr(N=i)\right] \qquad (19)$$

From $E(N)$, we can obtain $E(K) = n - E(N)$ and find the effective throughput. However, the expressions are quite complex and require evaluation of P_0. By solving the complementary birth-death formulation directly for $E(K)$, we can obtain a simpler and more revealing expression for $E(K)$. Accordingly:

$$E(K) = \frac{m}{\rho} \frac{P_0'(n)}{P_0'(n-1)} \quad \text{for} \quad 1 < m < n \; ; \qquad (13)$$

where

$$P_0'(n) = \left[\sum_{k=0}^{n-m} \frac{(m/\rho)^k}{k!} + \sum_{k=n-m+1}^{n} \binom{n}{k} (1/\rho)^k \frac{m!m^{n-m}}{n!} \right]^{-1}$$

In effect, the solution to the most difficult case of $1 < m < n$ is found in terms of:

$P_0'(n)$: Probability that no processor is in normal execution in an n-by-m system;

$P_0'(n-1)$: Probability that no processor is in normal execution in an (n-1)-by-m system.

Note that $E(K)$ still needs to be evaluated. The evaluation is simpler, since the same procedure can be used for $P_0'(n)$ and $P_0'(n-1)$. For the fully-decentralized case where $m \geq n$, the solution is simpler:

$$E(K) = (\rho+1)^{-1}n = (1-\alpha)n \quad \text{for} \quad m \geq n. \qquad (13')$$

Clearly, for the case of $m \geq n$, $E(Q) = 0$; since a customer will always find an available server.

The interesting case of $m = \alpha n$ ready lists is shown in Figure-4(b), where $m = 1$ (centralized) and $m = n$ (fully-decentralized) cases are also shown for comparison. The portion of $m = \alpha n$ curve up to 10 processors are shown as a visual aid. It corresponds to the anomaly, where presumably less than one ready list is used.

From Figure-3(b), we see that, under birth-death assumptions, employing $m = \alpha n$ ready lists does not yield effective throughputs which grow linearly with n. This counter-intuitive result will be formalized later.

5.2 Conclusions for the Decentralized PMS

About the required number of ready lists, we can assert and prove the following statements.

Rule #2:

If a fixed number m (<n) ready lists are used, the effective throughput is still bounded above by $ms\rho^{-1}$. Therefore, as n exceeds $m\alpha^{-1}$ and gets larger, the relative utilization rates diminish. As n grows very large, the contribution of additional processors to the effective throughput becomes nil.

Proof:

Since $\overline{K} \geq E(K)$, we only need to look at the deterministic case. As n becomes sufficiently large and exceeds $m\alpha^{-1}$, the second part of the solution in (12) applies, where $\overline{K} = m\rho^{-1}$ at most. The effective throughput can therefore be at most $ms\rho^{-1}$. In this case, the relative utilization is:

$$R_n = \frac{\overline{K}}{(1-\alpha)n} = \frac{m}{\alpha n} \quad \text{and} \quad \lim_{n\to\infty} R_n = 0.$$

Rule #3:

Under deterministic assumptions, if we employ $m = cn$ ready lists for some constant $c > n^{-1}$, constant utilization rate can be achieved.

Proof:

For n sufficiently large, the second part of solution for deterministic case (11) applies. If we replace m by cn in (11), we get:

$$\overline{K} = \frac{1-\alpha}{\alpha}(cn) \quad \text{and} \quad R_n = \frac{\overline{K}}{(1-\alpha)n} = \frac{c}{\alpha} \quad \text{A constant.}$$

As a corollary to this rule, since by definition, the speedup factor:

$$S_n = nR_n = \frac{c}{\alpha} n$$

The ideal speedup factor of n can be achieved if $c \geq \alpha$.

Rule #4:

Under stochastic assumptions, where there is a finite probability of having all processors engaged in ready list access (i.e., $P_n > 0$), we must employ at least $m = n$ ready lists in order to achieve constant utilization rates. This means that, if we employ $m = cn$ ready lists for some constant c, where c is in the range $\frac{1}{n} < c < \frac{n-1}{n}$, constant utilization rates are not achieved.

Proof:

First note that for most common (and non-contrived) stochastic systems $P_n > 0$, as is the case with the birth-death assumptions. For the above claim, we provide a proof by contradiction.

Suppose we do achieve constant utilization rate, where $R_n = 1 - c$, for some constant c in the range $\frac{1}{n} < c < \frac{n-1}{n}$. By definition of relative utilization:

$$\frac{E(K)}{(1-\alpha)n} = 1 - c \,. \qquad (14)$$

Also, for any stochastic system, the following work-balance equation must hold:

$$\mu[n-E(K)-E(Q)] = \lambda E(K). \qquad (15)$$

Solving (14) for E(K), and then, solving (15) for E(Q); we obtain:

$E(Q) = cn$, under the original assumption of $R_n = 1-c$. (16)

Suppose P_i is the probability that i processors are engaged in ready list access. Then, by definition:

$$E(Q) = \sum_{i=m+1}^{n} (i-m)P_i$$

If we choose the best possible value for m, namely m = n-1, from above, we obtain $E(Q) = P_n$. On the other hand, we obtained E(Q) = cn earlier. What this means is that:

$P_n = cn$ if we assume that $R_n = 1 - c$ (i.e., constant utilization). (17)

Since P_n is the finite probability mentioned earlier, the equation in (17) would make sense only for the following trivial cases:

i) $P_n = 0$. This violates the condition where $P_n > 0$, and essentially corresponds to the deterministic case. For the deterministic systems, the serialization of ready list accesses does not permit having all n processors engaged in ready list access.

ii) n = 1. In this case, m = n-1 = 0, meaning no ready lists.

iii) $c \leq n^{-1}$. This violates the condition where $c > n^{-1}$. This case corresponds to the anomaly of using less than 1 ready list. Refer to the $m = \alpha n$ curve for $n < 10$, in Figure-4(b).

As a corollary to this rule, if constant utilization cannot be achieved when m < n and $P_n > 0$; the ideal speedup factors also cannot be achieved.

The foregoing rules indicate that we should provide at least m = n ready lists to make constant utilization (or possibly the ideal speedup factor) achievable.

6. Proposed Ready List Access Policy

Even if we provide m = n ready lists, the overhead associated with processors accessing the available ready lists should be taken into account. If we provide a global index which indicates which ready lists are currently available, this global index will also be serially reusable. In much the same way as the single ready list, this global index will place an upper limit on the effective throughput. Suppose the time to check this global index is $1/\mu'$. In this case, by analysis of the deterministic case not discussed here, the upper limit for $\overline{K}$ turns out to be

$$\overline{K} = \frac{\mu'}{\lambda} \text{ at most, and for } n \geq 1 + \frac{\mu'}{(n-1)\alpha}$$

For the earlier example where $\rho = .05$, let $\lambda = .05$ and $\mu = 1$. If the time to check the global index is only half of the time inside a ready list, then u' = 2, and:

$$\overline{K} = \frac{2.0}{.05} = 40 \text{ at most, for } n \geq 43.$$

Obviously, if the time to check this global index increases with n, the result for $\overline{K}$ is worse. It is clear that, providing a centralized global index is unacceptable for the type of MIMD system. Other possible approaches and their shortcoming have been discussed in (1).

As a way of avoiding centralized mechanisms, we propose the following ready list access policy:

- Processors choose to access the ready lists randomly and uniformly (with equal probabilities). If the chosen ready list is currently locked, the processor waits in idle state until the lock is removed.

6.1 The Random-Routing Model

The analytical model for predicting the effective throughput for the proposed ready list access policy is shown in Figure-5.

These types of models with uniform and random routing of arrivals to servers received considerable attention in literature; primarily with respect to memory interference. Works done in (8,9) correspond to the deterministic case where $\lambda^{-1} = 0$ and $\mu^{-1} = 0$. A review and comparison of various approximate solutions to these types of models can be found in (12). The exact solution to these types of models are difficult or computationally intractable, even when the underlying stochastic process is Markovian. The analysis is more tractable for infinite number of customers (processors) or servers (ready lists). However, we are primarily interested in specific -and finite- values of n and m.

First note that, for any given set of parameters n, m, and ρ; the worst-case corresponds to the Centralized PMS Model results with m = 1. Similarly, the best-case E(K) corresponds to the Decentralized Model results.

For the deterministic case, $\overline{K}$ is obtained through a simulation program described in (1). The results of this simulation for $\alpha = .0476$ and $\alpha = .1$ are given in Table-1. Notice that the program was validated by running the case of m=1 for which the solution is known.

NUMBER OF PROCESSORS (N)	NUMBER OF READY LISTS (M)	EXPECTED NUMB OF PROCESSORS ENGAGED E(N)	EXPECTED NUMB OF PROCESSORS IN NORM-EXEC. E(K)
5	5	0.24	4.76
10	10	0.48	9.52
15	15	0.73	14.27
20	20	0.98	19.02
25	25	1.23	23.77
30	30	1.47	28.53
35	35	1.73	33.27
40	40	1.97	38.03

Run #1: $\rho = 1/20 \begin{cases} \text{20 time units in normal execution.} \\ \text{1 time unit inside a ready list.} \end{cases}$

5	5	0.50	4.50
10	10	1.00	9.00
15	15	1.54	13.46
20	20	2.07	17.93
25	25	2.61	22.39
30	30	3.12	26.88
35	35	3.64	31.36
40	40	4.17	35.83

Run #2: $\rho = 1/9 \begin{cases} \text{9 time units in normal execution.} \\ \text{1 time unit inside a ready list.} \end{cases}$

* Program's output when m = 1 is used as input, to compare with the analytical results of the deterministic single ready list model:

α=.10	n	$\overline{N}$	$\overline{K}$
	5	.5005	4.4995
	10	1.0047	8.9953
	15	5.9957	9.0043

Table-1: Throughput Results for the Proposed Ready List Access Policy Under Deterministic Assumptions.

Under birth-death assumptions, the exact solution exists. As outlined in (10) the solution method involves analysis of a queueing network with m+1 nodes. This solution requires enormous computational effort even for moderate values of n and m.

Instead, we can demonstrate the effectiveness of the random-access policy by obtaining an approximate solution for E(K), and making sure that it is not overestimated. This approach is based on the analyses of local queues found in front of the ready lists.

Since the overall Poisson arrival rate is randomly split into m, the arrival rate to a local queue is also Poisson, with mean λ/m per processor. The self-regulation is more complex. To simplify matters, we assume that:

- A processor accesses a particular ready list at rate λ/m, regardless whether it is queued for some other ready list.

This means that the processors engaged in ready list access rejoin local queues with probability m^{-1} and at mean intervals λ^{-1}. This does not present any difficulty with respect to the decentralization scheme, or the implementation of ready list accesses. With this assumption, the birth-death formulation for the local queue (X_j for ready list j, j = 1 to m) is as follows):

$$\lambda_x = \lambda(n-x)(1/m) \quad \text{for } x = 0 \text{ to } n. \quad \text{Discouraged Arrivals.}$$

$$\mu_x = \begin{cases} 0 & \text{for } x = 0, \\ \mu & \text{for } x = 1 \text{ to } n. \end{cases} \quad \text{Steady Service.}$$

where x denotes the number of processors engaged in accessing L_j.

This birth-death formulation does not account for all the self regulation. If i is the total number of processors queued for all the ready lists, then the overall discouraged arrival rate for this formulation is:

$$\sum_{j=1}^{m} (n-x_j)(\lambda/m) = \left(n - \frac{1}{m}\sum_{j=1}^{m} x_j\right) = \lambda(n-i/m).$$

which is greater than the "true" overall discouraged arrival rate of $\lambda(n-i)$. For this reason, the approximation based on the above birth-death formulation slightly overestimates E(N), and underestimates E(K) = n - E(N). In other words, the assumption where the processors rejoin local queues at mean intervals $1/\lambda$ guarantees that the throughput results based on this approximation are not overly optimistic.

Carrying out the single-server analysis, we first obtain the mean local queue:

$$E(X_j) = n - \frac{G(n-1)}{G(n)} \quad \text{where } G(\cdot) \text{ is d.f. for Poisson with mean } m/\rho.$$

The approximation for E(K) is then:

$$E(K) = n - \sum_{j=1}^{m} E(X_j) = \frac{m^2}{\rho}\frac{G(n-1)}{G(n)} - (m-1)n. \quad (18)$$

where $G(\cdot)$ is the cumulative d.f. for Poisson with mean m/ρ.

The results of Random-Routing Model for probabilistic case are shown in Figure-4, and compared with best-case and worst-case results for E(K). It is clear that, even for a heavy ready list access traffic (α=.1), the results of the random-access policy is remarkably close to the best-case which corresponds to the ideal speedup factor. In (1), three results have been compared to the approximation with "mean think time" suggested in (8) and found to be very close.

6.2 Concluding Comments on Proposed Policy

When we employ m = n ready lists, with random-access policy, speedup factors very close to n are achievable. If α is very high and the results are not very close to best-case, we can employ a larger number (m > n) ready lists to bring the throughput results closer to the best-case. Table-2 shows this for α = .25. It must be kept in mind that the approximation is guaranteed not to overestimate E(K), as mentioned earlier.

The random-access policy is a viable method for avoiding centralized mechanisms and the resulting performance bottleneck conditions. For the MIMD system where the number of processors may be very large, avoidance of bottleneck conditions is an important performance issue.

Number of Processors	E(K) = Expected Number of Processors in Normal Execution With Randomized-Access Policy			Best-Case
(n)	m=n	m=n+5	m=n+10	E(K)
5	3.56	3.66	3.69	3.75
10	6.93	7.13	7.23	7.50
15	10.27	10.55	10.71	11.25
20	13.61	13.94	14.14	15.00
25	16.95	17.31	17.55	18.75
30	20.28	20.67	20.94	22.50

Table-2: Throughput Results for the Proposed Policy (m > n, α = .25).

On the other hand, the Random-Routing Model takes into account only the ready list accesses. Other PMS system tables are not in the decentralization scheme. Other factors, such as what happens when a ready list is found empty, are not considered. A simulation model which takes these factors into account and uses the proposed ready list access policy is discussed in (1). The results of that simulation is also very encouraging about the proposed randomized-access policy.

References

(1) Cezzar, R. "Effects of Process Management System Software on the Throughput of a Parallel MIMD Machine," Ph.D. Thesis, Polytechnic Institute of New York, Brooklyn, New York. (June 1982).

(2) Klappholz, D. "Stochastically Conflict-Free Data-Base Memory Systems," Proceedings of 1980 International Conference on Parallel Processing.

(3) Klappholz, D. "The Symbolic, High-Level Language Programming of An MIMD Machine," Technical Manuscript, Polytechnic Institute of New York, Brooklyn, New York.

(4) Sullivan, H., Bashkow, T. R., and Klappholz, D. "The Node Kernel: Resource Management in Self-Organizing Parallel Processors," Proceedings of 1977 International Conference on Parallel Processing.

(5) Klappholz, D. "Stochastically Conflict-Free Memory/Interconnection System," Technical Report, Polytechnic Institute of New York, Brooklyn, New York.

(6) Klappholz, D. "An Improved Design for Stochastically Conflict-Free Memory/Interconnection System," Proc. 14th Asilomar Conf. on Circuits, Systems and Computers. (November 1982).

(7) Madnick, S. E., Donovan, J. J. Operating Systems, McGraw-Hill, Inc., New York, c. 1974.

(8) Baskett, F. and Smith, A. J., "Interference in Multiprocessor Computer Systems with Interleaved Memory," Comm. ACM, Vol. 19, No. 6. (June 1976).

(9) Rau, B. R. "Interleaved Memory Bandwidth in A Model of A Multiprocessor Computer System," IEEE Trans. on Computers, Vol. C-28, No. 9, (September 1979), pp. 678-681.

(10) Kleinrock, L. Queueing Systems - Volume I: Theory, John Wiley and Sons, Inc., New York, c. 1975.

(11) Hillier, F. J. and Lieberman, G. J. Introduction to Operations Research, Holden-Day, Inc., San Francisco, Calif., c. 1967.

(12) Ulema, M. and Smith, E. J. "Throughput Calculations for Multiprocessor Systems," Proc. of Computer Networking Symposium, (December 1979), Gaithersburg, Maryland.

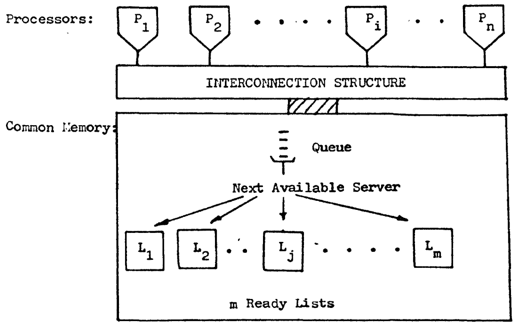


Figure-1: Decentralized PMS Model Description.

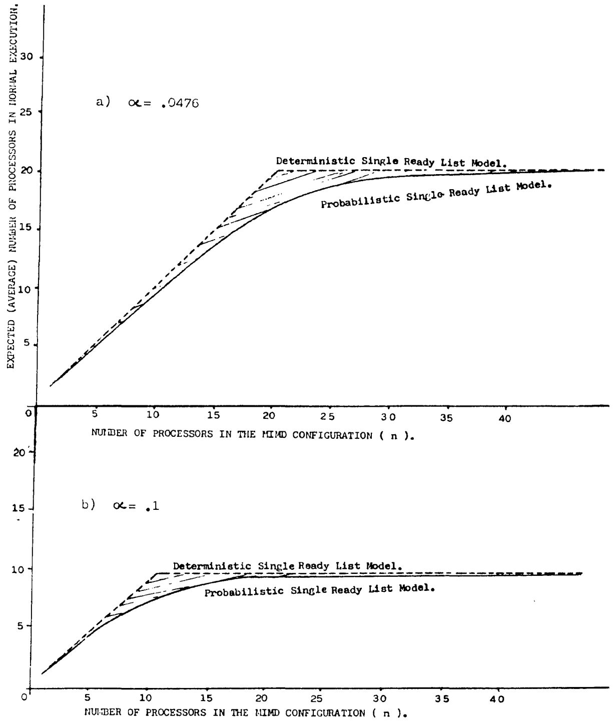


Figure-2: Effective Throughput Results for Centralized PMS Model.

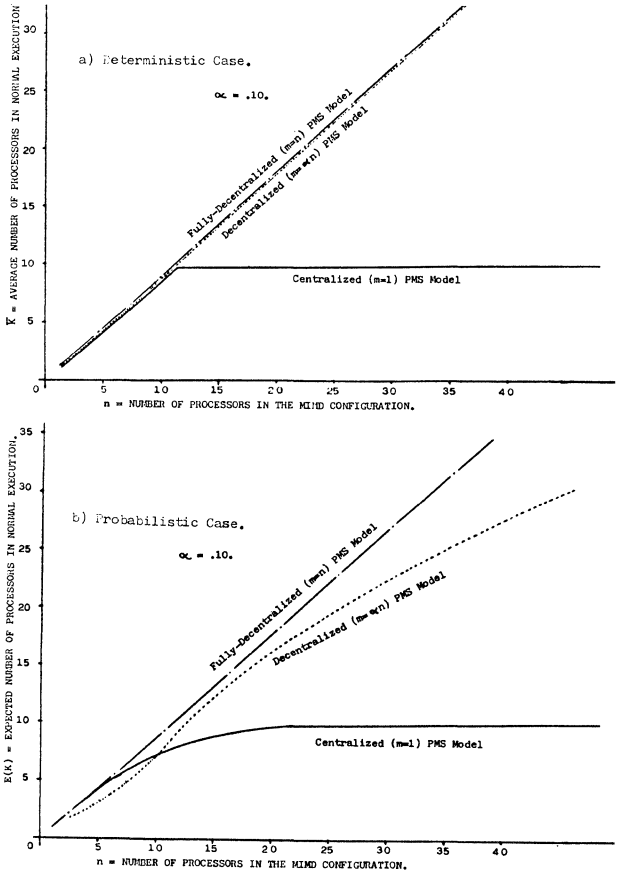


Figure-3: Effective Throughput Results for Decentralized PMS Model.

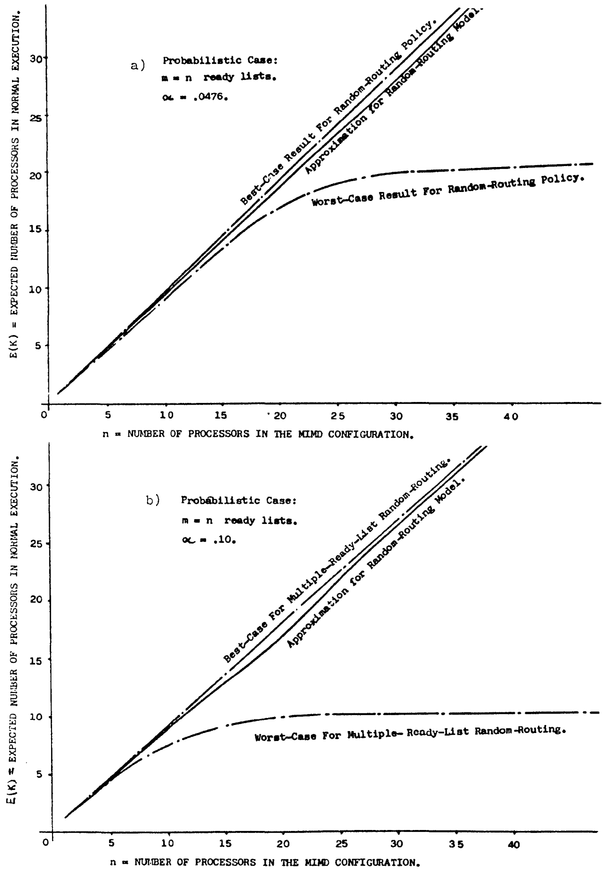


Figure-4: Effective Throughput Results of Approximate Solution to Random-Routing Model Under Birth-Death Assumptions.

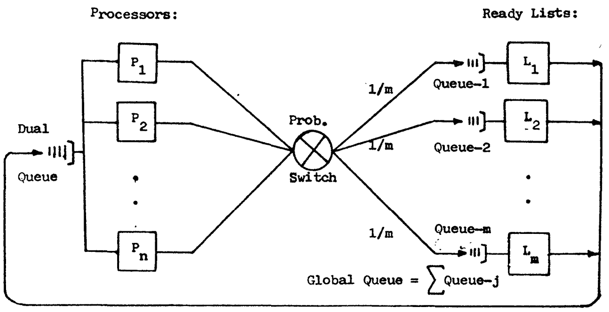


Figure-5: Finite-Population Multiple-Server Queueing Model With Random Routing of Customers to Servers.

ASSIGNING PROCESSES TO PROCESSORS IN DISTRIBUTED SYSTEMS

Elizabeth Williams
Department of Computer Science
The University of Texas at Austin
Austin, Tx 78712

Abstract – A message-based approach to interprocess communication is widely accepted for distributed computing. We present objectives necessary for assigning processes to processors in a distributed environment. Two objectives have previously been identified, neither has considered actual message delays. We give two new objectives and show how all four objectives are related to actual message delays. The importance of these objectives is illustrated by a realistic example from numerical analysis. This example was run on a testbed, consisting of a compiler and simulator used to run CSP-like programs on user specified architectures.

1. Introduction

When the processes of a distributed program are assigned to processors in a distributed environment, heuristic algorithms have traditionally only considered minimizing the communication among processors and balancing the load over the processors [2, 3, 5, 6, 8]. We have found two new objectives for the assignment problem, and shown how all four conflicting objectives are interrelated. Our studies have shown that actual message delay is an important consideration. We have developed a heuristic algorithm for assigning processes to processors which incorporates these objectives; this algorithm is presented in [9]. For this paper we motivate why these four objectives are important and present an example to illustrate the varying importance of these objectives.

We assume that the processes of a distributed program can usually execute at the same time. The objectives are for a set of processes where there is minimal ordering on the processes. Thus in a program where the processes are strongly ordered, the objectives may be applied to each subset of processes where the processes in each subset can usually execute at the same time.

2. Message Delay in a Distributed Environment

We first describe our distributed system of processors. We include the physical characteristics of the distributed architecture by considering processors with different speeds, and lines with different capacities and lengths that connect the processors. We assume that any processor can communicate with any other processor by routing messages through intermediate processors over fixed paths. We define **virtual line time** for a message between two processors connected directly by a line as a function of lower level protocols, message length in message units, number of bits per message unit, line capacity, and line length. Virtual line time does not include the time a message waits to use the communication subnet. Virtual line time for a message between two processors is the sum of the virtual line times for the lines on the route. Currently in local area networks, lower level protocols executing in the processors usually reduce the physical line capacity by at least a factor of ten for any message [1]. Virtual line time reflects this effective line capacity.

The **message delay** of a process for a blocking communication as in CSP [7] is a function of virtual line time, queueing at the port queues on the route in a store and forward network, and the processing, waiting, and queueing time of the corresponding process at its processor. Message delays can be very large compared to a process's processing time between communications.

The following limiting conditions generally hold when trying to minimize the total time a program requires.

1. As all message delays -> 0, the optimal assignment is to cluster and assign processes such that the load is balanced over all processors.
2. As all message delays -> infinity, the optimal assignment is to assign all processes to the fastest available processor.

When message delay is considered it is important to realize that it may not be advisable to use all the processors.

To illustrate we give a simple example. Consider two processors of equal speed connected by a line, and two communicating processes where each executes for time t and then sends a message to the other. We must determine for what message delay range both processes must be assigned to one processor and for what message delay range each must be assigned to separate processors. If both processes are on one processor the total time is 2*t. If there is a process on each processor then the total time is t+d, where d is the time for a message to go from one processor to the other. Thus, when t>d it is better to assign each process to a different processor; when t<d it is better to assign both processes to one processor. Note that the relationship of t to d determines the number of processors used.

3. Objectives

Minimizing the total time a program runs on a distributed architecture is very much dependent on how the processes are assigned to processors. Objectives are given for assigning processes to processors to reduce the total time a program runs.

3.1. Minimizing Interprocessor Communication

It has been established that it is important to minimize the communication among processors by effectively partitioning the processes of a distributed program and assigning the partitions to the processors [2, 3, 5, 6, 8]. There are actually two minimization problems in this statement. We can minimize the interprocessor communication (1) using a fixed number of processors or (2) letting the number of processors vary. In case (2) the minimum is obtained by assigning all the processes to a single processor.

We assume interprocess communication is independent of the assignment. To reduce interprocessor communication, processes which communicate the most are grouped (clustered) into one partition. We call each partition of processes a cluster.

0190-3918/83/0000/0404$01.00 © 1983 IEEE

3.1.1. Minimizing Communication on n Processors

We assume that if there are n processors, n clusters are formed, and each cluster is assigned to a different processor. We consider the following two functions to minimize: (a) the sum of the communication between each pair of clusters, and (b) the maximum communication between each pair of clusters. In a bus or broadcast network, interprocessor communication is reduced by minimizing (a). In a fully connected network with identical lines, interprocessor communication is reduced by minimizing (a) and (b). In a store and forward network with fixed paths through intermediate processors, interprocessor communication is reduced by considering the minimization of both (a) and (b), and the paths in the network. The fixed paths need to be considered if a line is included in the paths between several different pairs of processors; that line can become a bottleneck.

3.1.2. Minimizing Communication on Variable Number of Processors

As fewer processors are used it may be the case in a store and forward network with fixed paths that there is less total communication on all lines but more communication on some line. For large message delays, queueing delays at the corresponding ports can get large. Thus in a store and forward network, fewer processors are used only if the communication on any line does not increase.

3.2. Load Balancing

Load balancing has also been established as an objective [2, 3]. Load balancing over all the processors is only important when all message delays are small. As message delays go to zero, communicating processes can be modeled as noncommunicating processes. We know that for noncommunicating processes total time is minimized if the load is balanced over all processors. This assumes that all processors can stay busy for the entire time. At zero virtual line time the delay for a process waiting on a message depends on the processing, queueing, and waiting times of the corresponding process at its processor. For this reason it may not be possible to keep each processor busy. However, load balancing over all processors for small message delays is a reasonable goal.

3.3. Minimum Processing Requirement

As message delays get large, there must be enough processing available at a processor so that when one process blocks for a message, other processes can execute until the desired message arrives. This reduces idle periods at a processor caused when all assigned processes are blocked waiting for messages. Thus, a processor must be assigned a minimum amount of processing, and this processing requirement depends on the message delays. Only after there is the minimum amount of processing at each processor can loads be balanced. As message delays get large, fewer processors are used. This objective establishes in part how many processors will be used; this is illustrated by the example given in the last paragraph of Section 2. We are not directly minimizing interprocessor communication by reducing the number of processors. To use this objective both the message delay and available processing at a processor before it becomes idle must be estimated.

3.4. Closest Processor

In a store and forward network with fixed paths, virtual line time varies between pairs of processors. Clusters with largest intercluster communication are placed on processors connected by smallest virtual line time for a message unit. In a broadcast network or a fully connected network with identical lines, processors are equally close and thus this objective is not necessary.

4. Example

An example is now presented to illustrate the objectives given above. We solve Laplace's partial differential equation, PDE, $u_{xx}+u_{yy}=0$ on a grid with the outer edges of the grid given as boundary conditions. This example was run on a testbed consisting of a compiler and simulator. It runs a CSP-like program on a distributed architecture specified by the user and includes the characteristics described in Section 2 [9]. The testbed was validated extensively using commercial analytical and simulation packages. The testbed provides confidence interval estimates at the 90% level with relative widths less than .05 for various performance measures. We have reported only the midpoint of the confidence interval for the measure, total time.

4.1. Algorithm for Example

The iterative method used is Gauss-Seidel. The grid is partitioned into subgrids where each subgrid is some number of contiguous rows. Each subgrid is solved by a process in the same way a sequential program would solve the entire grid. A grid value is computed as the average of its four adjacent neighbors; thus, to compute a row of values, the two adjacent rows are required. Hence, a process must request the two rows contiguous to its subgrid from its two neighboring processes.

The usual way of executing the n processes is in pipeline fashion where if process n is on the k^{th} step, process 1 is on the k+n-1 step. However, in our algorithm all processes begin at the same time and assume the requested rows are zero initially. In our algorithm each process is computing in the i^{th} or $i+1^{st}$ process iteration at any given time. The algorithm converges because the Gauss-Seidel method converges for any set of starting values; the first j-1 iterations for process j compute a better set of starting values for its subgrid.

The entire grid must be calculated at each iteration as long as at least one grid value has not met the convergence criteria. The communication structure is linear; thus, convergence over the entire grid is easy to detect and termination of all processes is easy to ensure.

4.2. Analysis of Example

The above algorithm was run on the testbed. Figure 6-1 shows the communication structure of the algorithm with each process represented by a circle (the process number in the circle), the total processing time requirement per process below each circle, and the amount of communication exchanged between two communicating processes above each line. Values for communication and processing time are obtained by running the program on the testbed with any assignment and architecture; for this program these quantities are independent of the architecture and assignment.

The distributed architecture is three fully connected processors. Each processor has the same speed and each line is identical. The queueing discipline at each processor is preemptive priority with highest priority given to those processes which communicate across a line for the given assignment. We found that as message delays increased it was important to use the preemptive priority discipline and give priority to those processes which had to communicate over a physical line [9].

We studied eight different assignments. The assignments with the total processing requirement at each processor and the communication on each line are given in Table 6-1. The total processing requirement at a processor is the sum of the processing requirement of each process assigned to the processor. Assignments RR, A0, and A1 use all three processors; assignments A2, A3, A4, and A5 use two processors; and assignment A6 uses only one processor. Assignment A0 has minimum variance of the total processing requirement at the three processors, and thus better load balancing than the other assignments.

Table 6-2 gives the total time for the PDE program to complete execution for each assignment with several virtual line times for a message unit. The notation (1,2,3/4,5,6,7/x) denotes that processes 1, 2, and 3 are assigned to processor 1; processes 4 through 7 are assigned to processor 2; and no processes are on processor 3.

We now show how the objectives vary as message delays (virtual line time) increase. At virtual line time 57.9, the better load balanced assignments, A0 and A1, have minimum total time. Minimizing interprocessor communication is not important. At virtual line time 592.2, the two processor assignments A3 and A4 are better assignments than any one or three processor assignment. This illustrates that there must be a minimum amount of processing at a processor since they are preferable to any three processor assignment. At this virtual line time A2 compared to the other two processor assignments demonstrates that it is important to minimize the communication between the processors. At virtual line time 1388.6 it is better to put all processes on one processor; this is demonstrated by a performance improvement of greater than a factor of three when RR and A6 are compared. Looking at the entire Table 6-2, the trend to use fewer processors as message delays increase is shown by a * which marks the best assignment for each virtual line time.

5. Conclusion

In this paper we have presented four important objectives to consider when assigning processes to processors in a distributed environment. Two objectives are new. We have shown how all four objectives are affected by actual message delays. We have presented the results for one distributed program on architectures with various virtual line times; these results were obtained from a testbed for distributed computing.

6. Acknowledgments

The author wishes to thank Professor K. Mani Chandy for suggesting this problem and providing valuable guidance during this research, and Gael Buckley for her many editorial comments on this paper. This research was supported in part by Air Force Office of Scientific Research under grant AFOSR 81-0205.

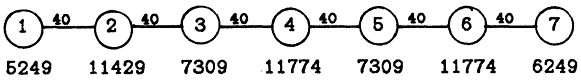


Figure 6-1: Communication Structure and Processing Time Requirement of each Process of PDE Program

assignment		processor number 1	2	3	exchanges on lines
RR:	processes	1,4,7	2,5	3,6	80,80,80
	load	23272	18738	19083	
A0:	processes	1,2,7	3,4	5,6	40,40,40
	load	22927	19083	19083	
A1:	processes	1,2,3	4,5	6,7	40,40,0
	load	23987	19083	18023	
A2:	processes	4,5,6	1,2,3,7	empty	80,0,0
	load	30857	30236	0	
A3:	processes	1,2,3	4,5,6,7	empty	40,0,0
	load	23987	37106	0	
A4:	processes	6,7	1,2,3,4,5	empty	40,0,0
	load	18023	43070	0	
A5:	processes	1	2,3,4,5,6,7	empty	40,0,0
	load	5249	55844	0	
A6:	processes	1,2,3,4 5,6,7	empty	empty	0,0,0
	load	61093	0	0	

Table 6-1: Assignments, Resulting Load at each Processor, and Message Exchanges per Line for PDE

assignment	virtual line time 57.9	592.2	871.7	1388.6
RR(1,4,7/2,5/3,6) :		99726		215993
A0(1,2,7/3,4/5,6) :	24836	61297	84156	126548
A1(1,2,3/4,5/6,7) :	24650*+	55738	72119	102185
A2(4,5,6/1,2,3,7/x) :	33271	82869	107460	155273
A3(1,2,3/4,5,6,7/x) :	38357	47990*	65405	97960
A4(6,7/1,2,3,4,5/x) :	45032	51085+	66916	98358
A5(1/2,3,4,5,6,7/x) :	59069	58951	60613*	92485
A6(all on 1 pr.) :	63636	63636	63636+	63636*+

+ assignment generated by heuristic
* assignment with minimum total time

Table 6-2: Total Execution Times for Each Assignment for PDE

References

1. E. E. Balkovich, Digital Equipment Corporation; David Wood, Mitre Corporation; Dieter Baum, Hahn-Meitner-Institute, Germany. Private Communications

2. W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe. "Task Allocation in Distributed Data Processing." *Computer* (November 1980), 57-69.

3. K. Efe. "Heuristic Models of Task Assignment Scheduling in Distributed Systems." *Computer* (June 1982), 50-56.

5. V. B. Gylys and J. A. Edwards. "Optimal Partitioning of Workload for Distributed Systems." *Compcon* (Fall 1976), 353-357.

6. K. Haessig and C. J. Jenny. "Partitioning and Allocating Computational Objects in Distributed Computing Systems." *IFIPS Congress* (1980), 593-598.

7. C.A.R. Hoare. "Communicating Sequential Processes." *Comm. ACM* (August 1978), 666-677.

8. H. S. Stone and S. H. Bokhari. "Control of Distributed Processes." *Computer* (July 1978), 97-106.

9. E. A. Williams. *Design, Analysis, and Implementation of Distributed Systems from a Performance Perspective.* Ph.D. Th., The University of Texas at Austin, 1983.

PRELOADING SCHEMES FOR THE PASM PARALLEL MEMORY SYSTEM

David Lee Tuomenoksa†
Howard Jay Siegel

Purdue University
School of Electrical Engineering
West Lafayette, Indiana 47907

Abstract -- Parallel processing systems, such as PASM, employ a large number of primary memory modules. A memory system organization using parallel secondary storage devices and double-buffered primary memories has been devised for PASM in order to prevent primary/secondary memory transfers from becoming a bottleneck. To efficiently use the memory system, it is desirable to overlap the operation of the parallel secondary storage devices with computations being performed by the processors. Due to the dynamically reconfigurable architecture of PASM, the processors which will execute a new task will not be selected until they are ready to execute the task. That is, to make effective use of double-buffering, a task must be preloaded prior to the final selection of the processors on which it will execute. Two schemes which allow for the parallel secondary storage devices to preload input data and programs into the primary memories so that system performance can be improved are presented and compared. Results show that both methods are effective techniques.

I. Introduction

In large-scale reconfigurable parallel processing systems the transfer of data and programs between the primary memories of the processors and the secondary storage can become a bottleneck. There are several types of reconfigurable parallel processing systems. *A partitionable SIMD/MIMD system* can be dynamically reconfigured to operate as one or more independent *SIMD (single instruction stream-multiple data stream)* [4] and/or *MIMD (multiple instruction stream-multiple data stream)* [4] machines (e.g., PASM [18], TRAC [8,16]). A *multiple-SIMD system* is a parallel processing system which can be dynamically reconfigured to form one or more independent SIMD machines of varying sizes (e.g., MAP [11,12]). When a partitionable SIMD/MIMD or multiple-SIMD system is forming an SIMD machine, data must be loaded into the processors' primary memories. When a partitionable SIMD/MIMD system is forming an MIMD machine, in addition to data, a program must be loaded into the primary memory of each processor which is executing the task.

PASM is a partitionable SIMD/MIMD multimicrocomputer system being designed at Purdue University for image processing and pattern recognition applications [18]. In order to prevent the primary/secondary memory transfers from becoming a bottleneck in PASM, a memory system employing parallel secondary storage devices and double-buffered primary memories has been devised [18]. To improve processor utilization by taking advantage of the double-buffering, it is necessary to overlap the operation of the parallel secondary storage devices with computations being performed by the processors. This overlap can be obtained by preloading the programs and data for the next task, while the previous task is being executed, and then by overlapping the unloading of output data with execution of the next task.

Since PASM is to be used in a research environment for parallel algorithm development (in some cases interactively), it is undesirable to require the user to specify the maximum allowable execution time of a task before it can be executed. The *first-fit multiple-queue (FFMQ)* scheduling algorithm, which has been described in [20], does not put this requirement on the user. If the FFMQ scheduling algorithm is used, due to the dynamically reconfigurable architecture of PASM, it is not known a priori which task a given group of processors will execute. Since tasks must be preloaded prior to the final selection of processors, it appears that the system would be unable to preload tasks when using the FFMQ scheduling algorithm. The problem considered is how to determine where the data and programs for tasks can be preloaded while using the FFMQ scheduling algorithm so that the performance of the memory system can be improved. Without such a preloading scheme, the full potential of the double-buffered memory modules will not be realized.

This paper presents *preloading* schemes which can be used in conjunction with the FFMQ *scheduling* algorithm. Two schemes which solve the preloading problem by determining which task's or tasks' programs and input data should be preloaded into a given set of processor memories are presented. The first scheme uses the scheduling algorithm to *preschedule* the task(s) which will follow the current task. The second scheme uses the scheduling algorithm to *predict* which task(s) may follow a given task. The performance of these preloading schemes as applied to PASM is demonstrated and contrasted through simulation studies. The preloading schemes described can be adapted to other multiple-SIMD and partitionable SIMD/MIMD systems.

II. PASM Background

PASM, a partitionable SIMD/MIMD machine, is a large-scale dynamically reconfigurable parallel processing system [18] (see Fig. 1). The *System Control Unit* is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM. The *Parallel Computation Unit (PCU)* contains $N = 2^n$ processors, N memory modules, and an interconnection network (see Fig. 2). The *PCU processors* are microprocessors that perform the actual SIMD and MIMD computations. The *PCU memory modules* are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. A pair of *memory units* is used for each PCU memory module so that data can be moved between one memory unit and secondary storage

This research was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR-78-3581, and by a Purdue University Graduate Fellowship. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

† D. L. Tuomenoksa is now with American Bell, Holmdel, New Jersey 07733.

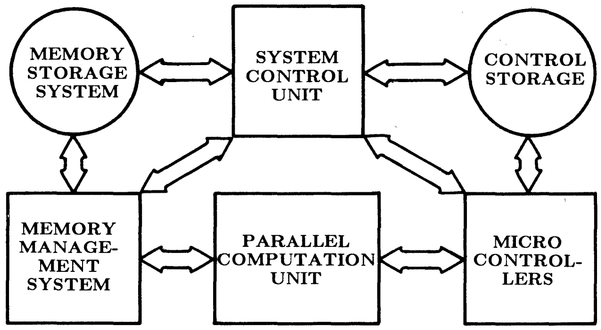


Fig. 1. Block diagram overview of PASM.

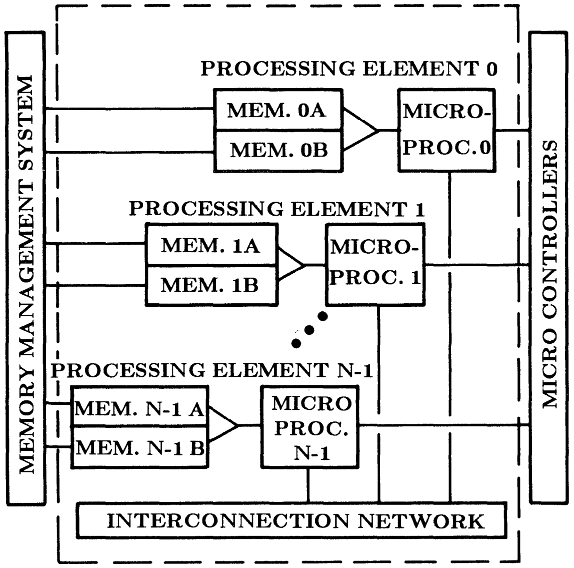


Fig. 2. PASM Parallel Computation Unit (PCU).

while the PCU processor operates on data in the other memory unit (double-buffering). A processor and its associated memory module form a *PCU processing element (PE)*. The PEs are physically addressed from 0 to N−1. The pair of memory units forming the i^{th} PCU memory module are labeled iA and iB (see Fig. 2). The *interconnection network* provides a means of communication among the PEs. PASM will use either a Cube type [1] or ADM type [10] of multistage network.

The *Micro Controllers (MCs)* are a set of microprocessors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. There are $Q = 2^q$ MCs, addressed from 0 to Q−1. Like the PEs, the MC memory modules are double-buffered. Each MC controls N/Q PEs, where possible values of N and Q are 1024 and 16, respectively. An *MC-group* is composed of an MC processor, its memory module, and the N/Q PEs which are controlled by the MC. The N/Q PEs connected to MC i are those whose addresses have the value i in their low-order q bit positions (see Fig. 3). *Control Storage* contains the programs for the MCs.

A virtual machine of size RN/Q, where $R = 2^r$ and $1 \leq r \leq q$, is obtained by combining the efforts of R MC-groups. According to the partitioning rule for PASM [18], the physical addresses of these MCs must have the same low-order q−r bits so that all of the PEs in the partition have the same low-order q−r physical address bits. For example, for Q = 16, allowable MC partitions include: (6), (14), (2,10), (0,4,8,12), and (1,3,5,7,9,11,13,15). Q is therefore the maximum number of partitions allowable, and N/Q is the size of the smallest partition. The reason for using this particular partitioning rule is because it allows multistage networks like the multistage Cube and the ADM, which are being considered for PASM, to be partitioned into independent subnetworks [17]. This rule is also valid for multistage Omega [9], shuffle-exchange [13], and indirect binary n-cube [15] networks, as well as other data manipulator [3] type networks such as the Gamma [14] network [17].

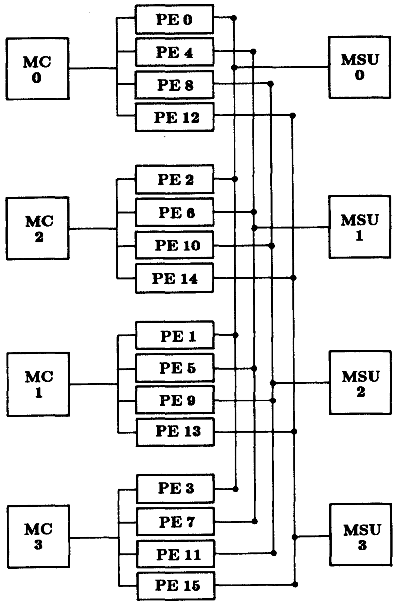


Fig. 3. Organization of the Memory Storage System for N = 16 and Q = 4.

The designator of a virtual machine composed of an allowable partition is the smallest physical address of the MCs in the virtual machine. This designator corresponds to the low-order q−r bits of the physical address of each MC in the virtual machine. For the partitions in the above example, the designators are: 6, 14, 2, 0, and 1, respectively.

The approach of permanently assigning a fixed number of PCU processors to each MC has the advantages that the operating system need only schedule Q MCs, rather than N PCU processors, and that it simplifies the MC/PE interaction, from both a hardware and software point of view, when a virtual machine is being formed. In addition, this fixed assignment scheme is exploited in the design of the Memory Storage System in order to allow the effective use of parallel secondary storage devices [18].

The FFMQ scheduling algorithm, which is being considered for use with PASM [20], makes use of q + 1 first-in first-out task queues, $TQ_0, TQ_1, \ldots, TQ_q$. A task which requires 2^k MC-groups is put into TQ_k. Whenever there are free MC-groups, the FFMQ algorithm selects the first job in TQ_k, where k is the largest integer such that a virtual machine of size 2^k MCs is available for execution. If TQ_k is empty, then the first task from TQ_{k-1} is selected. This process is continued until all available MCs have been assigned or until k = 0. The FFMQ algorithm assigns the task to the free virtual machine with the lowest designator. This is a *nonpreemptive* scheduling policy since all tasks run until completion and a multiple-queue scheduling. The FFMQ algorithm is a *centralized* scheduling algorithm [7] since the System Control Unit, which is executing the FFMQ algorithm, has complete accurate information regarding the states of all tasks in the system.

When PASM is forming a virtual machine which is to execute an SIMD task, data must be loaded into the PCU memory units and a program must be loaded into the MC memory units. When forming a virtual machine which is to execute an MIMD task, both data and programs must be loaded into each of the PCU memory units. In this paper the loading/unloading of data for SIMD tasks from the PCU memory units is considered. The loading of the SIMD program into the MC memory units is not considered since it can be overlapped with the loading of the data, following the same preloading scheme. The analysis in this paper can easily be extended to MIMD tasks; instead of loading just data, both programs and data would be loaded.

The Memory Storage System, which provides secondary storage space for the PCU memory modules, consists of N/Q independent *Memory Storage Units (MSUs)*. It is controlled by the Memory Management System. The MSUs are numbered from 0 to (N/Q)−1. Each is connected to Q PCU memory modules, as shown in Fig. 3. For $0 \leq i < N/Q$, MSU i is connected to those PCU memory modules whose physical addresses are of the form $(Q * i) + k$, $0 \leq k < Q$. For $0 \leq k < Q$, MC-group k contains those PCU processors whose physical addresses are of the form $(Q * i) + k$, $0 \leq i < N/Q$. Thus, MSU i is connected to the i^{th} PE of each MC-group.

The two main advantages of this approach for a partition of size N/Q (i.e., one MC-group) are that (1) all of the PCU memory modules can be loaded in parallel and (2) the data is directly available no matter which partition (MC-group) is chosen. This is done by storing the data for a task which is to be loaded into the i^{th} logical PE of the virtual machine in MSU i, $0 \leq i < N/Q$. Thus, no matter which MC-group is chosen, the data from the i^{th} MSU can be loaded into the i^{th} PCU memory module of the virtual machine, for all i, $0 \leq i < N/Q$, simultaneously.

Thus, for virtual machines of size N/Q, this secondary storage scheme allows all N/Q PCU memory modules to be loaded in one parallel block transfer. Consider the situation where a virtual machine of size RN/Q is desired, $1 \leq R \leq Q$. Only R parallel block loads are required if the data for the PCU memory module whose high-order n−q logical address bits equal i is loaded into MSU i. This is true no matter which partition of R MCs (which agree in the low-order q−r address bits) is chosen [18].

A *memory frame* is the amount of space used in the PCU memory units for storage of data from secondary storage for a particular task. It is possible that a task may need to process more than one memory frame. Besides being used for preloading, the double-buffered PCU memory modules can also be used to overlap task execution on one memory frame with the loading or unloading of another memory frame.

III. Memory System Model

In this section the model for the Memory System used for the analysis in this paper is described. A *memory unit set* is the set of PCU memory units within a single MC-group which have the same label (e.g., A, see Fig. 2). A *data block* consists of all the data to be loaded for one memory unit set. For a particular task which requires R MC-groups, there are R data blocks in a memory frame.

When a task is assigned to an MC-group, one of the memory units of each PCU memory module within the MC-group is used by the task. Without loss of generality for SIMD tasks, it is assumed that all of the memory units within an MC-group which are used by a given task are in the same memory unit set. Hence, all of the memory units within a memory unit set will always be assigned to the same task and will have the same status. Since all memory units within a memory unit set always have the same status, in this model it is also assumed that the loading/unloading of data for a memory unit set is done simultaneously and is considered as one action. In general, this is also true for MIMD tasks. (However, it is possible that for MIMD tasks in which the PEs have differing secondary memory system requirements that these assumption may not hold.)

All requests which are made to the Memory Management System and serviced by the Memory Storage System are for one data block. This results from the fact that the Memory Storage System can only load/unload one memory unit set at a time. There are three types of requests: *load*, *preload*, and *unload*. A *load request* is a request for input data for a task which has been assigned to its MC-group(s) (i.e., the MC-groups are ready to execute the task). A *preload request* is a request for input data for a task which has *not* been assigned to its MC-group(s) (i.e., the MC-groups are not ready to execute the task). An *unload request* is a request to unload output data for a completed task.

Load requests have the highest priority since the MC-group which is associated with the request has already been assigned to the task and is idle waiting for its input data. Unload requests have the second highest priority since they are for tasks which have already completed execution and the user is waiting for the output data. Preload requests have the lowest priority since the MC-group which is associated with the request is *not* idle waiting for the input data to be loaded. The Memory Storage System services requests from three request queues, one for each type of request.

IV. Preloading Schemes

Preloading enables the Memory Storage System to preload the data into the PCU memory units of a given virtual machine while the PCU processors of that virtual machine are still executing the previous task. In this section two task preloading schemes for use with the FFMQ scheduling algorithm are presented. These schemes make use of the double-buffered PCU memory modules. While a task is being executed using one of the memory unit sets of a given MC-group (e.g., the A memory units), the next task to be executed can be preloaded into the other memory unit set (e.g., the B memory units). Since there are only two memory units associated with each PCU processor, each processor can have at most two tasks associated with it. Hence, only single task look-ahead preloading is considered. In general, there will be more than one task preloaded into the system since different MC-groups can have different tasks preloaded.

Preloading is driven by the size of a currently executing task. When a task of size 2^k MCs, $0 \leq k \leq q$, begins to execute, the preloading of tasks of size 2^k (or smaller) is considered for that set of MCs. Thus, a task of size 2^i MCs can be preloaded only if there are tasks of size 2^i or greater currently executing. The two preloading schemes to be presented are prescheduling and prediction.

Prescheduling. When *prescheduling* is used, the task manager attempts to schedule tasks in advance of when they would normally be scheduled. Whenever a task starts executing on a virtual machine, the prescheduler determines which tasks (if any) will follow the execution of the given task. If there are any tasks, the tasks are preassigned to the appropriate MCs to be their next task executed. The prescheduling algorithm uses the FFMQ scheduling algorithm [20]; but, instead of attempting to schedule tasks for the entire machine, the prescheduling algorithm attempts only to schedule the virtual machine (or MC-groups) which is executing the given task. When a task completes execution, if no tasks have been prescheduled to follow the completing task, the regular FFMQ scheduling algorithm is called. It is noted that task prescheduling supplements task scheduling, it does not replace it.

The following example will illustrate the use of prescheduling on a PASM with four MC-groups (i.e., Q = 4). The status of the system is given as a function of time in Fig. 4. The status of the task queues are given whenever there is a change. At time 10.0 the system completes execution of tasks α, which required four MC-groups. Task β has already been preloaded, so it begins executing immediately. The prescheduling algorithm is called by the task manager to preschedule the task or tasks which will follow task β. The FFMQ algorithm determines that tasks γ and δ (which each require two MC-groups) will follow β. Tasks γ and δ are removed from TQ_1 and are preassigned to the appropriate MCs. The task manager then requests that the data for tasks γ and δ be preloaded. The Memory Storage System unloads the output data from task α and preloads the data for γ and δ. Recall that the Memory Storage System is only able to transfer the data for one MC-group at a time. Hence it takes four transfers to unload task α, two to preload γ, and two to preload δ. At time 10.8 task β is executing and tasks γ and δ are preloaded and ready to be executed. At time 12.0 task β completes executing and tasks γ and δ start executing.

As indicated in Fig. 4, task ϵ, which requires two MC-groups, was prescheduled to follow task γ. So no matter when task γ completes execution, task ϵ will follow it. As a result, even though task ϵ arrived to the system before task ς, the MC-groups did not start executing it until 5.5 seconds after task ς. As a result, the response time for task ϵ is much greater than that of task ς. This is an example of how the prescheduling scheme alters the order which task are executed.

Prediction. As with prescheduling, the prediction preloading algorithm is invoked each time a task starts executing. When ***prediction*** is used, the task manager predicts which task or tasks may follow the task which started executing. Unlike the prescheduling scheme, the task(s) are *not* removed from the task queue. The prediction scheme uses the FFMQ scheduling algorithm to predict which tasks may follow a given task. The predicted task or tasks are then preloaded by the Memory Storage System into their predicted memory units. The same enqueued task may be predicted and preloaded to follow each currently executing task whose size is equal to or greater than that of the enqueued task. The FFMQ scheduling algorithm is executed whenever a task completes execution (i.e., when MCs become free) and whenever a new task arrives to be scheduled [20]. When a task is scheduled for execution by the FFMQ scheduling algorithm, the assignment of tasks to MC-groups is made as if no preloading had taken place. When a task is assigned, the task manager sends requests to the Memory Management System indicating that the Memory Storage System should load the data. Recall that one data block is sent to each MC-group and that the task manager sends a separate request for each data block. After the Memory Management System receives the data block requests, it voids any requests for data blocks which have been preloaded. Doing data requests on a block by block basis allows the system to take advantage of and to account for partial preloading. Having the task manager requests all data blocks (regardless of whether they have been preloaded) removes the burden of keeping track of preloaded data from the task manager (which executes on the System Control Unit).

The following example will illustrate the use of prediction on a PASM with four MC-groups (i.e.,

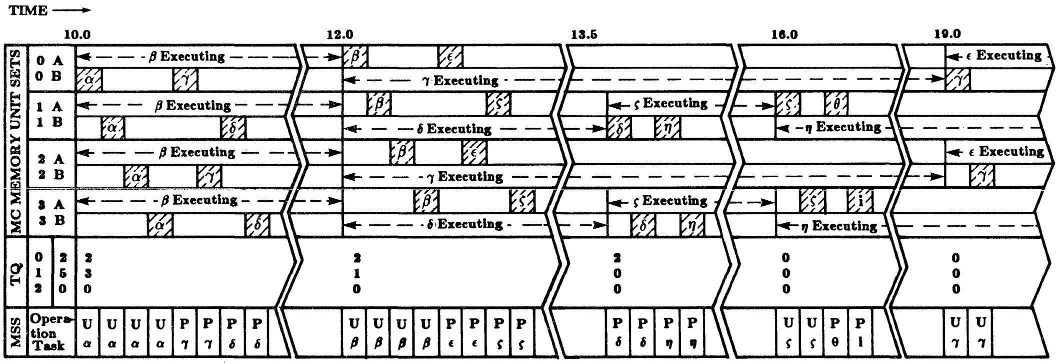


Fig. 4. An example of the use of the prescheduling scheme for determining where input data should be preloaded. The status of a PASM with four MC-groups (Q = 4) is shown. Status of the eight (2Q) memory unit sets, the three (q + 1) task queues (for scheduling), and the Memory Storage System are given. Shaded area indicates when a memory unit set is being accessed (either by loading, unloading, or preloading) by the Memory Storage System. "L," "P," and "U" indicate that the Memory Storage System is loading input data, preloading input data, and unloading output data, respectively.

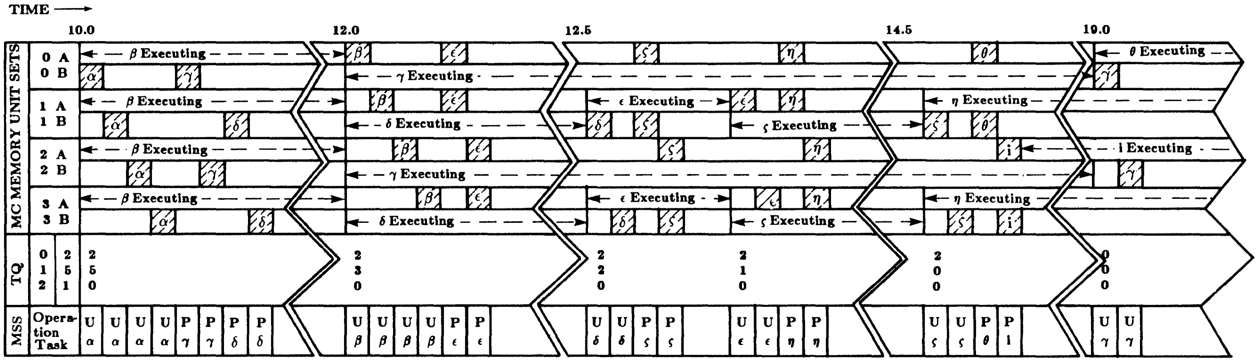


Fig. 5. An example of the use of the prediction scheme for determining where input data should be preloaded. Notation is the same as used in Fig. 4.

Q = 4). The status of the system is given as a function of time in Fig. 5. The status of the task queues are given whenever there is a change. At time 10.0 the system completes execution of tasks α, which required four MC-groups. The scheduler determines that task β will be executed next by the system. Since task β has been preloaded, execution begins immediately. The prediction algorithm is called by the task manager to predict which task or tasks may follow task β. The FFMQ algorithm determines that tasks γ and δ (which each require two MC-groups) may follow β. The task manager then requests that the data for tasks γ and δ be preloaded. Note that tasks γ and δ are not removed from the scheduler task queues as they were for the prescheduling scheme. The Memory Storage System unloads the output data from task α and preloads the input data for γ and δ. Recall that it takes four transfers to unload task α, two to preload γ, and two to preload δ. At time 10.8 task β is executing and tasks γ and δ are preloaded and ready to be executed. At time 12.0 task β completes execution. Since there are free MC-groups, the scheduler is called by the task manager. The scheduler then selects tasks γ and δ to be assigned to the free MC-groups. The tasks are then assigned and begin execution immediately since they have both been preloaded. Up to this point, the results of prescheduling and prediction are the same.

As indicated in Fig. 5, task ϵ was predicted to follow either task γ or δ. Therefore, it was preloaded into the MC-groups forming the virtual machines for both tasks. In this way, task ϵ can be executed by the virtual machine which becomes available first, preserving the FFMQ ordering. Thus, the normal scheduling policy is maintained with prediction and task ϵ does not experience the excessive delays that it did with prescheduling. Also note that in this particular example the structure of the Memory Storage System allows ϵ to be loaded into both MC-groups simultaneously. Since task δ was completed first, task ϵ was scheduled to follow it, and a new task was predicted to follow tasks γ and ϵ.

Summary. With the prediction scheme, the task manager predicts where the enqueued task might execute and preloads the data into the appropriate PCU memory units. The enqueued task may be loaded to follow more than one currently executing task. Independently of the preloading which has occurred, the scheduler selects which task will be executed next and to which MC-group(s) the task will be assigned. Thus, prediction does not alter the natural order in which tasks would have been scheduled without preloading. In contrast, the prescheduling scheme has the disadvantage that it alters the order which the tasks are executed from the natural order resulting from the use of the FFMQ scheduling algorithm. For example, when prescheduling was used, task ς was executed before task ϵ even though task ϵ was first in the task queue. As a result, prescheduling greatly increased the response time for task ϵ.

The prescheduling scheme has the advantage that it does not do any unnecessary loading of tasks which may not be used, i.e., with prescheduling a task is preloaded (or loaded) only one time. However, with prediction, a task may be preloaded one or more times. For example in Fig. 6, the two MC-group task δ was predicted and preloaded to follow task γ. Since both tasks α and β completed execution before γ, task δ did not follow task γ. Hence, unnecessary loading of task δ occurred.

To demonstrate how it is not clear which preloading algorithm will yield better performance, consider the simple examples given in Figs. 7 and 8. In the example in Fig. 7, the system variation using the prescheduling scheme completes all of the tasks first. On the other hand, in the example in Fig. 8, the system variation using the prediction scheme completes all of the tasks first. Both preloading schemes have advantages and disadvantages. In order to evaluate and quantify their relative performance, simulation studies were conducted. These studies are described in Section V.

The preloading schemes can use any scheduling algorithm, they are not limited to the FFMQ scheduling algorithm. Since the preloading schemes use the proces-

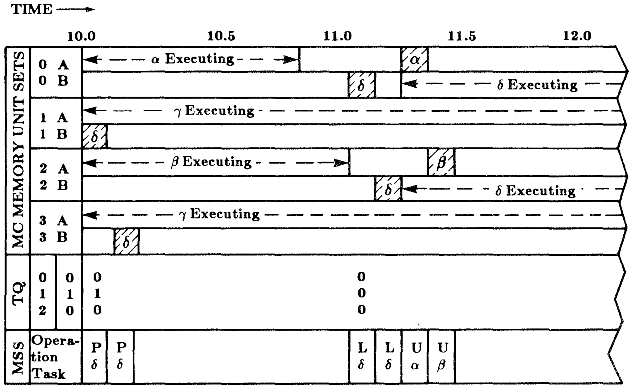


Fig. 6. Status of a PASM with four MC-groups, with notation as defined in Fig. 4. Illustrates unnecessary preloading which can occur from using prediction scheme.

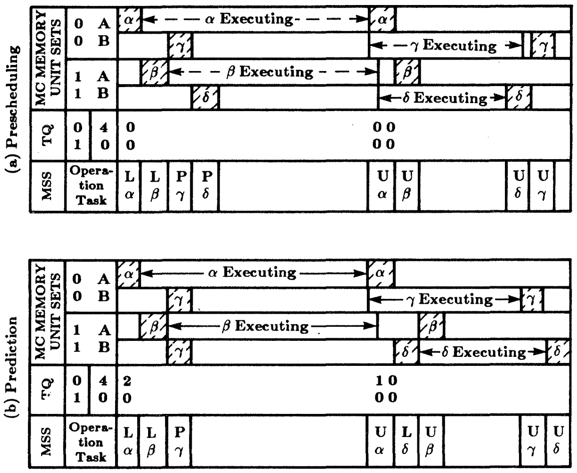


Fig. 7. Status of a PASM with two MC-groups, with notation as defined in Fig. 4. Example of case where the (a) prescheduling scheme yields better performance than the (b) prediction scheme.

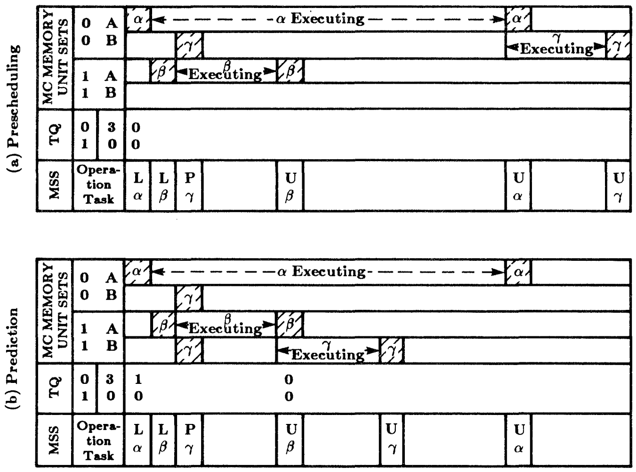


Fig. 8. Status of a PASM with two MC-groups, with notation as defined in Fig. 4. Example of case where the (b) prediction scheme yields better performance than the (a) prescheduling scheme.

sors currently assigned to a given task, they do not have to have a fixed MC-group structure. Hence, these hardware/software schemes can be adapted for use in other multiple-SIMD and partitionable SIMD/MIMD systems.

V. Performance Analysis

A PASM with 16 MCs (Q = 16) and 1024 PEs (N = 1024) was simulated using the PASM Operating System simulator, a discrete event simulator [5], under four variations in the control strategy used by the memory system (details of the simulations are given in [21]).

1. A PASM without double-buffered PCU memory modules was considered, i.e., only one memory unit per PE. This allowed for no overlapped loading or unloading of data, and is examined to demonstrate the need for the double-buffered PCU memory modules.
2. A PASM with double-buffered PCU memory modules was considered, i.e., two memory units per PE. With this variation the second memory unit was used for doing overlapped unloading of the output data from the previous task, but *no preloading* of the input data for the next task, i.e., there is no preloading scheme employed.
3. A PASM with double-buffered PCU memory modules was considered, using the prescheduling scheme for determining where to preload input data.
4. A PASM with double-buffered PCU memory modules was considered, using the prediction scheme for determining where to preload input data.

Performance measures to be considered are MC utilization, average load delay time, and average response time. The *MC utilization* is the fraction of time that the MCs are active during the simulation, specifically, the total MC active time, divided by Q and by the total simulation time. MC utilization has been selected since the utilization of the MCs reflects the utilization of the PEs.

The *average load delay time* is the average delay time to load the memory frame for a task (see Fig. 9). The load delay time for a given task is the delay between the time when the MC-group(s) are ready to execute the task and the time when the task starts executing. This is of interest since it directly shows the decrease in the time the processors are idle waiting for data to be loaded.

The *response time* for a task is the delay between the time when the task arrives at the system and the time when the task completes execution on the system (see Fig. 9). The average response time is calculated by accumulating the response time for each task executed and dividing it by the number of task completions. The response time is being considered since a decrease in response time has the greatest direct effect on the user. Response time is a significant measure since it is expected that PASM will often be used interactively. Interactive users might be experimenting with different sequences of image processing algorithms on large images. It is desirable to be able to very parameters for the algorithms and see the results in a reasonable amount of time (i.e., short response times).

The *system throughput* is the number of tasks completed per second by the system. It is not considered in detail since it is not an accurate performance measure for this type of analysis. The system throughput does not take into account the number of MC-groups the tasks required. For example, for two system variation, the throughput could be the same, but for one variation, the system could be completing all 16 MC-group tasks and for the other the system could be completing all one MC-group tasks. Hence, for the system throughput to be of use, it is necessary to weigh it with the task size, which is equivalent to looking at the MC utilization.

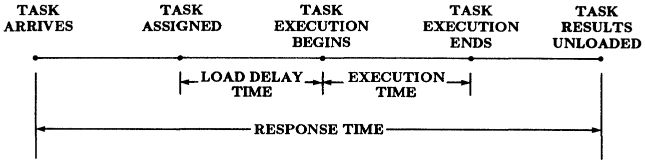


Fig. 9. Time-line which illustrates the definitions of load delay time and response time.

Since each memory unit set is preloaded independently, it is possible for a task to be partially preloaded when the previous task completes execution. Therefore, when the new task is assigned, it is only necessary to load the memory units sets which were not preloaded. The simulator is able to account for partial preloading.

The value of N (the number of PEs) is not varied since it would not effect the performance of the system. For example, if N were doubled, there would still be 16 MC-groups, but each MC-group would have twice as many PEs. Since there would also be twice as many MSUs (since there are N/Q MSUs), all of the memory units within one memory unit set could still be loaded in one parallel block transfer. Hence, the results of the simulations would not be effected.

On the other hand, if the value of Q was doubled, there would 32 MC-groups and only 32 MSUs. This change enables the system to execute tasks which require 32 PEs, in addition to the other possible task sizes. Since there are half as many MSUs, it would take twice as many parallel block transfers to load the PCU memory units for a task. Hence, doubling the value of Q would have a similar effect to doubling the time to load/unload a data block for an MC-group, as done in Experiment 2.

"In computer systems, the arrival of individuals at a card reader, the failure of circuits in a central processor, and requests from terminals in a time-sharing system are processes that are essentially Poisson in nature." [5] Since PASM serves requests from terminals (as does a time-sharing system), task arrivals are generated with a Poisson process. The mean task interarrival time was selected to be 20 simulation seconds. A uniform distribution is used for determining the number of MC-groups a task requires. Each simulation run was for 20,000 "PASM seconds" and had approximately 1000 tasks executed. The performance analysis has been divided into two experiments.

Experiment 1. In this experiment the distribution for the task execution time was chosen to be exponential. The mean execution time was varied from five to 50 simulation seconds. The time to load/unload a data block for an MC-group has been selected to be 0.090 simulation seconds. This load time is based on the time to load 64 kilobytes of data into a memory unit assuming that each MSU was a CDC BK7XX Storage Drive Module (disk) [2]. This time accounts for the seek and latency times of the disk which can be overlapped with the time to set the Memory Storage System busses. However, it does not account for any overhead from file system actions which should be insignificant when compared to the seek and latency times.

The average response time is given for the four variations in control strategy as a function of the average task execution time in Tab. 1. In [19] it has been determined both analytically and by simulation that the average number of tasks being executed by the system for a uniform distribution of task sizes is 2:58 times the MC utilization. Hence, if the system is to be 100 percent utilized, the mean task execution time must be at least 51.6 seconds (if the mean interarrival time is 20 seconds). Therefore, when the average execution time is small (less than 20 seconds), the system (or MC) utilization is low (less than 40 percent, see Tab. 2). With the utilization so low, there are usually free MC-groups and tasks can normally be scheduled immediately upon arrival. If tasks are scheduled immediately upon arrival, there is no time period in which tasks can be preloaded and as a result no preloading of input data occurs. Hence, for small average execution times (less than 20 seconds), the response time for all variations in control strategy is about the same.

Tab. 1. Average response time (in simulation seconds) is given for the four variations in control strategy as a function of the average task execution time (in simulation seconds).

	Average Response Time			
	Single	Double	Presch	Predict
5.0	6.921	6.808	6.718	6.728
10.0	14.048	13.841	13.645	13.650
15.0	23.139	22.567	22.125	22.275
20.0	35.168	34.144	33.095	33.495
25.0	53.584	51.764	53.035	49.681
30.0	82.440	78.690	78.500	76.854
40.0	324.56	277.37	235.96	233.00
50.0	1505.2	1250.0	1363.1	1105.1

Tab. 2. MC utilization is given for the four variations in control strategy as a function of the average task execution time (in simulation seconds).

	Micro Controller Utilization			
	Single	Double	Presch	Predict
5.0	0.098	0.098	0.098	0.098
10.0	0.196	0.196	0.196	0.196
15.0	0.294	0.294	0.294	0.294
20.0	0.392	0.392	0.392	0.392
25.0	0.487	0.487	0.487	0.487
30.0	0.584	0.584	0.584	0.585
40.0	0.770	0.770	0.775	0.775
50.0	0.876	0.877	0.882	0.894

The prescheduling scheme alters the order in which tasks are scheduled from the normal scheme since the tasks are scheduled in advance. The use of the prescheduling sometimes results in bad scheduling decisions (or miss-scheduling). For example, consider a PASM with two MC-groups ($Q = 2$) which is executing tasks α and β, each requiring one MC-group (see Fig. 8a). Task γ has been prescheduled to follow α. Task β completes execution before task α. Now MC-group 1 is idle and task γ is waiting to be executed on MC-group 0. Hence, task γ has been miss-scheduled resulting in increased response time for task γ. These bad decisions will have little effect when the average task execution time is small. However, when the average task execution time becomes large (i.e., greater than 25 seconds), the bad decisions have greater effect. This effect is illustrated by the average response time for the prescheduling scheme becoming greater than the average response time for the prediction scheme for large execution times (see Tab. 1).

The MC utilization for the four system variations in control strategy is given in Tab. 2. For execution times of less than 40 seconds, the system is able to service all of the arriving tasks (task arrival rate equals throughput) under each variation in control strategy. As a result, for small execution times the MC utilization is the same for all control strategies since the same set of tasks is being executed. As the average execution time increases, the throughput of tasks requiring 16 MC-groups decreases for the single and double variations. Since the system is executing fewer 16 MC-group tasks, the MC utilization is lower for the variations without preloading. When the average execution time is 50 seconds, the 16 MC-group task throughput is less for prescheduling than prediction, resulting in the difference in MC utilization. This occurs since the prescheduling scheme can only preschedule tasks which require the same number or fewer MC-groups than a currently running task. Therefore, a 16 MC-group task can only be prescheduled, if there is a 16 MC-group task running. Hence, the prescheduling scheme tends to favor tasks which do not require 16 MC-groups. As the demand on the system increases (e.g., longer average task execution times), the MC utilization becomes limited with the single and double variations since the processors cannot be utilized while they are waiting for data and programs to be loaded and unloaded. Hence, the maximum allowable system load (utilization) is higher when the preloading schemes are used.

In Tab. 3 the average task load delay time is given as a function of the average task execution time for the four variations in control strategy. This is given to show how the preloading schemes reduce the load delay times for tasks. Consider the single-buffered variation. As the average task execution time increases the system utilization approaches one (see Tab. 2). When a task is scheduled it is usually being executed by MC-groups which have just completed executing another task. As a result, the new task must wait for the output data from the previous task to be unloaded and its input data to be loaded. Hence, the load delay time increases with increased utilization for the single-buffered variation. However, with the prediction and prescheduling schemes, longer task execution times allow more time for task preloading. Therefore, for large task execution times, the average load delay time approaches zero for both preloading schemes (see Tab. 3). The average load delay time will never reach zero since there are constraints on when preloading is possible (e.g., cannot preload tasks which require more MCs than any given currently executing task) which will always prevent some tasks from being preloaded. The average load delay time for the prediction scheme does not approach zero as rapidly as it does for the prescheduling scheme since some tasks are not executed by the virtual machine in which they were preloaded (e.g., task δ of example in Fig. 6).

Tab. 3. Average load delay time (in simulation seconds) is given for the four variations in control strategy as a function of the average task execution time (in simulation seconds).

	Average Load Delay Time			
	Single	Double	Presch	Predict
5.0	0.688	0.586	0.542	0.551
10.0	0.767	0.598	0.518	0.535
15.0	0.885	0.618	0.470	0.505
20.0	0.989	0.646	0.418	0.472
25.0	1.077	0.662	0.359	0.434
30.0	1.141	0.673	0.301	0.389
40.0	1.192	0.668	0.170	0.262
50.0	1.126	0.595	0.094	0.157

In summary, for small execution times (less than 20 seconds) the system performs the same for all variations in control strategy. For large execution times (greater then 20 seconds) the prediction scheme performs best. For a given task, the load delay time is a component of the response time (see Fig. 9). As a result, load delay time does not indicate the direct effect on the user, as does the average response time. Hence, the lower average response times (for larger executing times) provided by the prediction scheme are more significant than the lower average load delay times provided by the prescheduling scheme. Therefore, this experiment indicates that the prediction scheme is the method of choice.

Tab. 4. Average response time (in simulation seconds) is given for the four variations in control strategy as a function of the time to load/unload one data block (in simulation seconds).

	Average Response Time			
	Single	Double	Presch	Predict
0.0	47.317	47.317	48.845	47.317
0.045	50.733	49.119	50.836	48.499
0.090	53.584	51.764	53.035	49.681
0.135	57.077	53.597	54.315	51.438
0.180	61.737	55.292	55.269	52.362
0.225	65.858	58.148	55.883	53.830
0.270	69.835	59.954	57.563	55.305
0.315	74.487	64.046	59.212	57.005

Experiment 2. In this experiment the distribution for the task execution time is exponential with mean task execution time of 25 simulation seconds. The time to load/unload a data block for an MC-group is varied from 0 to 0.315 simulation seconds. Varying the time to load/unload a data block could result from varying the size of the data block (which would result from varying the size of the PCU memory units) or from changing the type or speed of the secondary storage device used by the MSUs. For example, the time to load 64 kilobytes of data from a disk which employs "Winchester" technology can range from 0.2 to 0.4 seconds, depending on the particular manufacturer (e.g., for the Hewlett-Packard Model 7910, the average load time is 0.236 seconds [6]).

The average response time is given for the four variations in control strategy as a function of the time to load/unload a data block in Tab. 4. Note that in Tab. 1 the average response time was given as a function of the average task execution time, while in Tab. 4 it is given as a function of the time to load/unload a data block. If the load/unload time is zero, the average response time for the single, double, and prediction variations is the same since loading and unloading a task requires no time. The response time for the prescheduling variation is greater than the other variations when the load/unload time is zero since with the prescheduling variation, the system is still prescheduling tasks, resulting in some miss-scheduling. Hence, the zero load/unload time case directly illustrates the increase in response time resulting from the use of prescheduling.

As the load/unload time increases, the average response times for the single-buffered variation increase at a greater rate since the load and unload time for a task must be added to the execution time of every task. For all load/unload times, the prediction scheme yields the lowest response times. For load/unload times of greater than 0.045 simulation seconds, the prescheduling scheme yields lower average response times than the single-buffered variation, and for load/unload times of greater than 0.180 the prescheduling scheme yields lower average response times than the double-buffered variation without preloading. These cross-overs in the average response time occur since the benefit of the preloading (from the use of prescheduling) becomes more significant with greater load/unload times (and overcomes the increase resulting from miss-scheduling).

VI. Conclusion

Two schemes which can be used with the FFMQ scheduling algorithm for preloading input data into the PCU memory modules have been presented. The two schemes (prescheduling and prediction) make use of the double-buffered PCU memory modules. Since both schemes have advantages and disadvantages, in order to evaluate and quantify their relative performance, it was necessary to conduct simulation studies. The performance of the system has been evaluated with four variations in control strategy. It has been shown that the use of the double-buffered memory modules for overlapping the unloading of the output data from the previous task with the execution of the next task results in a significant decrease in average response time. Furthermore, it has been shown that the average response time can be decreased more significantly by using the double-buffered memories for input data preloading (along with overlapped unloading). When the system becomes heavily loaded, the system performs better with the prediction scheme than with the prescheduling scheme since the prescheduling scheme alters the natural ordering of the tasks which results from using the scheduling algorithm. However, the prescheduling scheme has the advantage that it does not do any unnecessary loading of input data which may not be used. The prediction scheme also has the advantage that in the worst case the resulting system performance will never be worse than that of the overlapped unloading case since the same scheduling order is maintained and all preloading is done with lower priority. This claim cannot be made for the prescheduling scheme since it alters the scheduling order.

In summary, the "prediction" preloading scheme makes good use of the Memory Storage System architecture and the double-buffered PCU memory modules. It overcomes the problem of how to determine where the system can preload tasks prior to final processor selection. Thus, the double-buffered primary memory - parallel secondary storage device organization can be exploited for overlapped loading of tasks as well as overlapped unloading. The preloading schemes can use any scheduling algorithm and can be adapted for use in other multiple-SIMD and partitionable SIMD/MIMD systems.

References

[1] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Comput.*, vol. C-31, pp. 443-454, May 1982.

[2] Control Data Corporation, *CDC Storage Module Drive BK7XX Hardware Reference Manual*, Control Data Corporation, Minneapolis, MN, 1979.

[3] T. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Comput.*, vol. C-23, pp. 309-318, Mar. 1974.

[4] M. J. Flynn, "Very high-speed computer systems," *Proc. IEEE*, vol. 54, pp. 1901-1909, Dec. 1966.

[5] S. H. Fuller, "Performance evaluation," in *Introduction to Computer Architecture*, 2nd edition, edited by H. S. Stone, SRA, Inc., Chicago, 1980, pp. 527-590.

[6] Hewlett-Packard, *Electronic Instruments and Systems 1982*, Hewlett-Packard, Palo Alto, CA, 1982.

[7] R. Y. Kain, A. A. Raie, and M. G. Gouda, "Multiple processor scheduling policies," *1st Int'l. Conf. Distributed Computing Systems*, Oct. 1979, pp. 660-668.

[8] R. N. Kapur, U. V. Premkumar, and G. J. Lipovski, "Organization of the TRAC processor-memory subsystem," *AFIPS 1980 Nat. Comput. Conf.*, May 1980, pp. 623-629.

[9] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145-1155, Dec. 1975.

[10] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Comput.*, vol. C-31, pp. 1202-1214, Dec. 1982.

[11] G. J. Nutt, "Microprocessor implementation of a parallel processor," *4th Symp. Comput. Architecture*, Mar. 1977, pp. 147-152.

[12] G. J. Nutt, "A parallel processor operating system comparison," *IEEE Trans. Software Engr.*, vol. SE-3, pp. 467-475, Nov. 1977.

[13] D. S. Parker, "Notes on shuffle/exchange-type switching networks," *IEEE Trans. Comput.*, vol. C-29, pp. 213-222, Mar. 1980.

[14] D. S. Parker and C. S. Raghavendra, "The gamma network: a multiprocessor interconnection network with redundant paths," *9th Symp. Comput. Architecture*, Apr. 1982, pp. 73-80.

[15] M. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458-473, May 1977.

[16] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas reconfigurable array computer," *AFIPS 1980 Nat. Comput. Conf.*, May 1980, pp. 631-641.

[17] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comput.*, vol. C-29, pp. 791-801, Sept. 1980.

[18] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934-947, Dec. 1981.

[19] D. L. Tuomenoksa and H. J. Siegel, "Analysis of the PASM control system memory hierarchy," *1982 Int'l. Conf. Parallel Processing*, Aug. 1982, pp. 363-370.

[20] D. L. Tuomenoksa and H. J. Siegel, "Analysis of multiple-queue task scheduling algorithms for multiple-SIMD machines," *3rd Int'l. Conf. Distributed Computing Systems*, Oct. 1982, pp. 114-121.

[21] D. L. Tuomenoksa and H. J. Siegel, *Design of the Operating System for the PASM Parallel Processing System*, TR-EE 83-14, School of Electrical Engineering, Purdue Univ., May 1983.

CONSTRUCTING A PARALLEL IMPLEMENTATION FROM HIGH-LEVEL SPECIFICATIONS: A CASE STUDY USING RESOURCE EXPRESSIONS

Bharadwaj Jayaraman
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

Abstract -- Resource expressions are high-level specifications of the coordination of concurrent requests to access a shared resource. This paper presents an operational semantics for resource expressions and shows how it is used to systematically construct an implementation for resource expressions. The operational semantics defines the necessary conditions to be tested and the actions to be taken when a condition holds. To simplify the definition of its semantics, resource expressions are represented in an intermediate form consisting of a set of condition-action pairs. The implementation of resource expressions represented in this intermediate form is a parallel program constructed from a set of queueing primitives and primitives for arbitration and parallel execution. This implementation is presented here by showing informally how the semantics of conditions and actions are realized by the primitives.

1. Introduction

The protection and sharing of resources are central to the construction of parallel programs. A resource is assumed here to be any data object together with a set of coordinated operations on this object. Coordination of these operations is necessary in order to maintain the consistency of the data object, or to optimize the amount of parallel execution of these operations. *Resource expressions* are a high-level language extension for specifying constraints, such as mutual exclusion, priority of operations, etc. Although their initial development was in the context of a functional language [9], they can be used in conventional languages as well. They are closely related to path expressions [3] in their basic approach to specification, but there are important semantic differences between the two languages. We use the term resource expression, rather than path expression, in recognition of the differences in their semantics. The reader is referred to [9] for a broader overview of our approach.

Resource expressions are basically regular expressions [10] extended with conditions and constructs for concurrent operations. For example, the resource expression

((write)* + [read])*

specifies a simple version of the readers-and-writers constraint of [5]. The operators "*" and "[]" specify zero or more sequential and parallel iterations of their respective bodies. Thus the subexpression (write)* specifies sequential execution of write requests, and the subexpression [read] specifies parallel execution of read requests. The operator "+" denotes selection of one of its operands, hence all read requests execute in mutual exclusion of all write requests. Since the selection of one of the operands of "+", and actual number of iterations performed by "*" and "[]" can be time-dependent, these operators are *nondeterministic* operators.

To exercise greater control over the selection of alternatives, we specify *conditions* that must hold before any alternative is selected. The conditions used are often thresholds on the number of requests present for any operation. We use $x to refer to the number of requests present for operation x. For example, the resource expression

((write)* + ($write=0 and $read>0) [read])*

specifies priority for write requests, because [read] can be selected only when there are no write requests present. However, once a read request is being executed, further read requests may be executed even if a write request should then become present.

The expression

((write)* + [($write=0 and $read>0)read])*

on the other hand, specifies stronger priority for write requests than the previous expression, because the absence of a write request is tested before every read request is executed.

Another basic primitive is *sequencing*, and is denoted by ".". For example, (f.g)* specifies that requests for operations f and g are to be executed in strict alternation. The expression f.g + h.k specifies a selection of only one of these two sequences, f.g or h.k, depending upon which sequence is ready to be selected first. There are three possible criteria for the selection of f.g (and similarly h.k):

1. Without testing for a request for either f or g;
2. As soon as a request for f is present;
3. Only when a request for both f and g are present.

We adopt the first criterion here, since the other two can be specified by means of explicit conditions before a sequence. For example, the second criterion can be stated as

($f>0)f.g

and the third criterion by

($f>0 and $g>0)f.g

This reflects the view that the conditions upon which actions are taken in a resource controller should be explicitly stated by the programmer. Although the specifications tend to become longer, the resulting programs become more self-documenting.

0190-3918/83/0000/0416$01.00 © 1983 IEEE

The syntax of resource expressions, for the purpose of discussion in this paper, is given by the following grammar. We use {} to denote zero or more repetitions of the enclosed rule.

```
RE    -->  RF   {+  RF}
RF    -->  RG   | (COND) RG
RG    -->  RH   {.  RH}
RH    -->  op   | (RE)* | [RE2]

RE2   -->  RF2  {+  RF2}
RF2   -->  RG2  | (COND) RG2
RG2   -->  op

COND  -->  REL   {and REL}
REL   -->  ($op = 0)  |  ($op > NATNO)
NATNO -->  0  | 1  | 2  | ...
```

where op is the name of an access operation.

We have omitted discussion of the operators "{}" and "#" of [9] in the presentation here due to shortage of space.

2. Operational Semantics

In order to simplify the presentation of the definition of its semantics, a resource expression is converted into an *intermediate form*, which consists of a set of *condition-action* pairs, similar to Dijkstra's guarded commands [6]. The general form of these condition-action pairs is

$$((c_1\ a_1)(c_2\ a_2) \cdots (c_n\ a_n)),$$

where

$(c_i\ a_i)$ is a condition-action pair, one for each term separated by "+", where
- c_i is of the form $(r_1$ and r_2 and ... and $r_k)$
- a_i is of the form $(x_1\ x_2\ ...\ x_j)$
- where
  - r_i is either $(\$op_i{>}n_i)$ or $(\$op_i{=}0)$, where $op_1 ... op_m$ are operations of the resource and n_i is a natural number,
  - x_i is an operation or (STAR I) or (BRACKET I) where I is $((c_{i1}\ a_{i1}) ... (c_{in}\ a_{in}))$ and STAR and BRACKET are functions for "*" and "[]" respectively (explained later).

The intermediate form may be thought of as a different syntactic form of the given resource expression, in which conditions and actions are stated in a more uniform, but perhaps less readable, manner. The translation to the intermediate form is quite straightforward and is omitted here. For example, the intermediate form of the resource expression

((\$x>0) ((\$x>1) x)*.x + (\$y>0) [(\$y>0) y])*

is of the form

(c a)

where

c = true, a = ((STAR ((c1 a1) (c2 a2))))

where

c1 = (\$x>0), a1 = ((STAR (c11 a11)) x)
where c11 = (\$x>1), a11 = (x)

c2 = (\$y>0), a2 = ((BRACKET (c21 c22)))
where c21 = (\$y>0), c22 = (y)

The operational semantics of resource expressions expressed in the above form is defined by showing the necessary conditions to be tested and the action to be taken once a condition holds. We define

eval(I, R)

which "evaluates" a resource expression I in the environment of a set of requests R, i.e. executes requests present in set R that satisfy the constraints of I. It is assumed that the content of set R can change, either because requests are added to it from outside the resource or because requests are removed from it in the course of evaluation of the resource expression.

We take some syntactic liberties in the language used in defining the semantics. For example, if the structure of the argument to a function f is (c a), we may write the definition of f as f((c a)) = ... and use c and a directly in the body of f. We may also test the structure of the argument x of a function f(x) by a predicate such as x = (c a), and then use c and a within the scope of the predicate.

We present the semantics by first presenting the definitions, followed by a brief informal explanation of them. At the top level, we have

eval(I, R) = evalaction(evalcond(I,R),R)

where *evalcond* and *evalaction* specify the semantics of conditions and actions respectively.

Semantics of conditions

```
evalcond(I,R) =
 IF I = ((c a)) THEN testuntil(R)(first(I)) ELSE
 insert(choose, applytoall(testuntil(R), I))
where
testuntil(R)( (c a) ) = WHEN present(c, R) THEN a
```

The testing of conditions is defined by *evalcond* which returns the action corresponding to the condition that is detected to be true earliest. insert(f, l) and applytoall(f, l) are primitive operations, whose meanings can be understood by considering the above definition of *evalcond* for the case when I is of the form $((c_1\ a_1)(c_2\ a_2) \cdots (c_n a_n))$, for n > 1:

```
evalcond(I,R)
= insert(choose, applytoall(testuntil(R), I)),
= choose(testuntil(R) ((c1 a1)),
        choose(testuntil(R) ((c2 a2)),
            ...
                choose(testuntil(R) ((cn-1 an-1)),
                        testuntil(R) ((cn an)),
                        ) ... ))
```

which effectively chooses the condition c_i that is detected to be true earliest. *choose* is a primitive operation which nondeterministicly selects of one of its arguments, depending upon which is evaluated earlier. (It is possible to have a more "balanced" testing of conditions than the one shown above using insert, but we do not consider it here.)

testuntil takes as input the set of requests R and produces a new operation which takes a condition-action pair (c a) as input and returns action a only when condition c is satisfied by the input requests in R. It is should be noted here that the input R can change by the arrival of new requests, hence a condition c that is not true for some set of inputs R may become true when R has more requests.

Semantics of actions

```
evalaction(a, R) =
  FOR EACH x IN a DO evaltype(x, R)
where
   evaltype(x, R) =
   IF operation(x) THEN execute(remove(x, R)) ELSE
   IF x = (STAR I) THEN evalstar(I, R) ELSE
   IF x = (BRACKET I) THEN evalbracket(I, R)
   where
      evalstar(I, R) =
      WHILE insert(or, applytoall(test(R), I))
            DO evalplus(I, R)

      evalbracket(I, R) =
      IF insert(or, applytoall(test(R), I))
         THEN {a <-- evalcond(I, R);
            req <-- remove(first(a), R);
            (execute(req) || evalbracket(I, R))}
         ELSE ;

      test(R)((c a)) = present(c, R)
```

evalaction defines sequential execution of the terms $(x_1\ x_2 \ldots x_j)$ in an action a. *evaltype* treats the different types of x: If x is a resource operation, a request for x is removed from R and executed; otherwise, x is of the form (STAR I) or (BRACKET I), whose semantics are defined by *evalstar* and *evalbracket* respectively. (STAR I) and (BRACKET I) represent "*" and "[]" respectively, whose semantics differ primarily in that "*" specifies sequential iteration, whereas "[]" specifies parallel iteration. The iterations of "*" and "[]" continue only as long as some condition in their body is true. This testing of this condition is specified in the semantic definition by the expression

insert(or, applytoall(test(R), I)),

where or is the primitive boolean operation and test(R)((c a)) returns a boolean value indicating if condition c is satisfied by the requests present in R.

The primitive operator "||" evaluates both its arguments in parallel. It should be noted that the testing of conditions and removal of requests across successive iterations of "[]" take place sequentially, whereas the execution of requests across successive iterations of "[]" take place in parallel. If the testing of conditions and removal of requests were to takc place in parallel, it is possible to detect conditions erroneously.

3. Implementation

The operational semantics of resource expressions may be viewed as defining an abstract interpreter for the language. However, our goal is to implement resource expressions by generating target code in terms of a set of synchronization primitives. We use a set of queueing primitives and primitives for arbitration and parallel execution to synchronize and schedule the execution of requests. These primitives were first defined in [8], and are given in the appendix.

The set of input requests R can be represented by a set of queues, one queue for each distinct type of operation. The addition of requests to R from outside the resource is achieved by enqueueing them to the appropriate queues. The removal of requests from R during the evaluation of a resource expression is achieved by dequeueing them from the appropriate queues.

We first show the top-level structure of the target program constructed for an intermediate form, and then show the target programs for conditions and actions separately. In each case, we first present the target program, followed by a brief explanation of the relation between the target program and the defined semantics. The entire construction is illustrated by an example.

Top-level structure

Assuming I is an intermediate form $((c_1\ a_1)(c_2\ a_2) \ldots (c_n a_n))$, then the top-level translated program is

```
LET $t_1$ = arbit(targetprogram($c_1$),$t_2$)
    $t_2$ = arbit(targetprogram($c_2$),$t_3$)
    ...
    $t_{n-1}$ = arbit(targetprogram($c_{n-1}$),
                  targetprogram($c_n$))
RESULT
    IF $t_1$ THEN targetprogram($a_1$) ELSE
    IF $t_2$ THEN targetprogram($a_2$) ELSE
    ...
    IF $t_{n-1}$ THEN targetprogram($a_{n-1}$) ELSE
                  targetprogram($a_n$)
```

where targetprogram(c_i) and targetprogram(a_i) are defined in the next two subsections.

The abbreviations t_1, t_2, etc., are equated to expressions that are evaluated only once. The above program fragment realizes the semantics of *evalcond*. The semantics of *choose* is realized by the primitive operator arbit, which evaluates both its arguments in parallel and returns a boolean value indicating which argument was evaluated earlier. The chain of arbit operations realizes the semantics of the definition of *evalcond* for the case when n>1. When n=1 no arbitration is needed, and targetprogram(c1) is used directly instead of t1.

Conditions

Assuming c is a condition of the form $(r_1 \text{ and } r_2 \cdots \text{ and } r_k)$, where each r_i can be either $(\$op_i>n_i)$ or $(\$op_i=0)$, then

```
   targetprogram(c) = spar($w_1$, $w_2$, ..., $w_k$)
where
   $w_i$ = waitq($qop_i$,1+$n_i$),  if $r_i$ is ($\$op_i>n_i$), and
   $w_i$ = waitq($qop_i$,0), if $r_i$ is ($\$op_i=0$)
   where
      $qop_i$ is the queue for operation $op_i$,
      spar and waitq are defined in the appendix
```

The conjunction of the test for equalities or inequalities is expressed by the primitive operator spar. The above program fragment realizes the semantics of *testuntil*. Unlike the semantic definition of *testuntil*, targetprogram(c) does not return (c a), but instead returns true when c becomes true; the action a is selected by the top-level program.

Actions

The semantics of actions is specified by *evalaction* and is realized as follows: Assuming a is an action of the form (x_1 x_2 ... x_j), then

```
targetprogram(a) = {targetprogram($x_1$);
                    targetprogram($x_2$);
                         ...
                    targetprogram($x_j$)}
```

where
targetprogram(x_i)
= evalq(qop$_i$), if x_i is an operation name op$_i$, and qop$_i$ is the queue for operation x_i.
= star(queues-for-I), if x_i = (STAR I)
= bracket(queues-for-I), if x_i = (BRACKET I)
where star and bracket are procedures that realize the semantics of evalstar and evalbracket

The above program fragment realizes the semantics of *evalaction* and *evaltype*. Since *evalaction* sequentially evaluates each x_i, targetprogram(a) is obtained by sequencing the program fragments for each x_i. The target program for x_i depends upon its type: If it is a simple operation name, then a request of this operation type is dequeued and executed by evalq; otherwise it must be of the form (STAR I) or (BRACE I). Since these two cases are similar, we present only the latter.

Parallel repetition

Assuming I = ((c_1 a_1)(c_2 a_2) ... (c_n a_n)), and queues-for-I denotes the set of queues for the distinct operations in I,

```
brace(queues-for-I) =
    LET $t_1$ = targetprogram($c_1$)
        $t_2$ = targetprogram($c_2$)
          ...
        $t_{n-1}$ = targetprogram($c_{n-1}$)
        $t_n$ = targetprogram($c_n$)
    RESULT
      IF or($t_1$, $t_2$, ... $t_n$)
        IF $t_1$ THEN targetprgram($a_1$) ELSE
        IF $t_2$ THEN targetprogram($a_2$) ELSE
          ...
        IF $t_{n-1}$ THEN targetprogram($a_{n-1}$)
        ELSE targetprogram($a_n$)
```

Assuming c_i is of the form (r_1 and r_2 and ... r_k) where r_i is either ($op_i>n_i$) or (op_i=0), then
targetprogram(c) = and(m_1, m_2, ... m_k),
where
m_i = nonempty(qop$_i$, 1+n_i), if r_i is ($op_i>n_i$), and

m_i = nonempty(qop$_i$,0), if r_i is (op_i=0)

Since a_i must be of the form (op)

```
    targetprogram(ai) =
      {res <-- deq(qop);
      spar(execute(res), brace(queues-for-I))}
```

The recursive call on *evalplus* from *evalbrace* in th semantic definition is realized above by a sequence o IF-THEN-ELSEs. The above program tends to bias th selection of alternatives towards t_1. This can be avoided by replacing the IF-THEN-ELSEs by a program fragment similar to that of the top-level structure. If all t_i are trivially satisfied, then the condition of the first IF statement is replaced by arbit(true, false), which nondeterministically decides the termination of brace.

xample

We illustrate our implementation by showing the ynthesized program for the resource expression

(($write>0) write + ($write=0)[($read>0) read])*

whose intermediate form is

```
true (STAR (($write>0) (write))
       (($write=0) ((BRACKET (($read>0) (read)))))
  ))
```

he complete translated program is

```
ound(writeq, readq) =
LET t1 = arbit(waitq(writeq,1),   /* ($write>0) */
               waitq(writeq,0))   /* ($write=0) */
RESULT
  WHILE true DO
  IF t1 THEN evalq(writeq)        /* write */
       ELSE bracket(readq)        /* [($read>0) read] */
}

bracket(readq) =
{LET t1 = nonempty(readq,1)          /* ($read>0) */
RESULT
  IF t1 THEN {res <-- deq(readq);         /* read */
             spar(execute(res), bracket(readq))}
}
```

4. Conclusions and related work

We have presented the systematic construction of the implementation of resource expressions starting from its operational semantics. From the example, it is easy to see that the structure of the target program preserves that of the original resource expression---there is one procedure for each of the repetitive operators, "*" and "[]", and one procedure at the topmost level, if "*" or "[]" does not occur at the topmost level. Conditions and actions are translated separately, and the resulting program fragments are assembled together.

The work on the semantics and implementation of path expressions is related. The meaning of path expressions has been defined using Petri nets [11], denotational and axiomatic methods [2], and guarded commands [1]. The implementation of path expressions has been based on semaphores [3, 4] and finite state machines [7, 1]. However, to the best of our knowledge, there has not been a systematic derivation of an implementation from the semantics of path expressions.

The semantics and implementation of resource expressions differ from those of path expressions. The semantics of our sequencing operator "." differs from that of path expressions, which base the condition for a sequence on criterion 2 described in the first section. Our implementation based on queues is radically different from the implementations for path expressions. The advantage of our approach is that the structure of the target program corresponds closely to the structure of the original expression, hence the correctness of the translation becomes more evident. Our explicit use of conditions, however, give rise to longer specifications than equivalent path expressions.

The conditions used in this paper are more restrictive than those found in [1], where conditions can test the state of the resource, the number of operations in execution, etc.

There are some differences between resource expresssions presented here and our earlier paper [9]. Here we propose the use of conditions as a more primitive concept than the commit operator "/", which restricts the selection of a sequence to be based upon the presence of requests for operations only in some prefix of it. Furthermore, it is possible to simulate "/" using conditions, although the resulting specifications tend to be longer. In this paper, we have restricted the forms of expressions inside "[]" to be simple operations, with possibly some condition before it. This simplifies the definition of the semantics as well as implementation.

We are examining extensions to resource expressions that will enhance its expressiveness. One way is to allow more general forms of conditions than merely tests on the status of the input requests. Another is to allow the actual parameters of the invoked operations to determine the conditions under which they are selected. The semantics and implementation of these features are being investigated. Also being investigated are optimizations that will enhance the efficiency of the translated programs.

References

[1] Andler, S. Predicate Path Expressions. In *Proc. Sixth ACM Symp. on Principles of Programming Languages*. pp. 226-236, 1979.

[2] Berzins, V. and Kapur, D. *Denotational and Axiomatic Definitions for Path Expressions.* Computation Structures Group Memo 153-1, Laboratory for Computer Science, M.I.T., November 1977.

[3] Campbell, R.H. and Habermann, A.N. The specification of process synchronization by path expressions. In Gelenbe and Kaiser (editors), *Operating Systems*, pages 89-102. Springer, 1974.

[4] Campbell, R.H. and Kolstad, R.B. *A practical implementation of Path Expressions*. Technical Report TR UIUCDCS-R-80-1008, University of Illinois at Urbana-Champaign, June, 1980.

[5] Courtois, P.J., Heymans, R. and Parnas, D.L. Concurrent control with readers and writers. *Communications of the ACM* 14(10):667-668, October 1971.

[6] Dijkstra, E.W. Guarded commands, nondeterminacy, and a calculus for the derivation of programs. *Communications of the ACM* 18(8):453-457, August 1975.

[7] Habermann, A.N. *Path Expressions*. Tech. Rept., Dept. of Computer Science, Carnegie-Mellon University, July, 1975.

[8] Jayaraman, B. and Keller, R.M. Resource control in a demand-driven data-flow model. In *Proc. International Conference on Parallel Processing*, pp. 118-127. 1980.

[9] Jayaraman, B. and Keller, R.M. Resource expressions for applicative languages. In *Proc. International Conference on Parallel Processing*, pp. 160-167, August 1982.

[10] Kleene, S.C. *Representation of events in nerve nets*. Princeton University Press, 1956, pages 3-40.

[11] Lauer, P.E. and Campbell, R.H. Formal semantics of a class of high-level primitives for coordinating concurrent processes. *Acta Informatica* 5:297-332, 1975.

Appendix

The primitive operators used in this paper are summarized below:

queue() creates an empty queue.

enq(q,exp) synchronizes the evaluation of exp using q by enqueueing a request for exp in q.

deq(q) removes the first request from q without executing it.

execute(req) executes a request req; the result of execution is returned to the enq operation that placed req on the queue.

evalq(q) combines deq and execute; and is same as execute(deq(q))

waitq(q,n) tests and waits until q has at least n requests and then returns true; if n=0 then waitq waits until q becomes empty.

nonempty(q,n) returns true if q has at least n requests, false otherwise; if n=0 then nonempty returns true if q is empty and false otherwise; no waiting is involved.

spar($a_1,...,a_n$) evaluates the expressions $a_1,...,a_n$ in parallel; the result is a_n, but is returned after all $a_1,...,a_n$ have been evaluated.

arbit(a_1,a_2) evaluates a_1 and a_2 in parallel; the result is false if a_2 is evaluated before a_1, otherwise true.

QUEUEING NETWORK MODELS FOR PARALLEL PROCESSING OF TASK SYSTEMS

Alexander Thomasian
Performance Modeling Center
Burroughs Corp.
Santa Ana, CA 90704

Paul Bay
T.R.W. Defense Systems Group
One Space Park
Redondo Beach, CA 90278

Abstract -- The paper deals with a procedure to determine the mean completion time and related performance measures for a task system: a set of tasks with precedence relationships in their execution sequence. The tasks, which are processed by a multiprogrammed multiprocessor system, are specified by their expected total loadings on the units of the computer system. A straightforward application of a queueing network (QN) solver to the problem is not possible due to variations in the state of the system (composition of tasks in execution). An approximate solution method is presented for this purpose based on the concept of hierarchical decomposition. At the higher level, an efficient procedure generates the Markov chain corresponding to the transitions among the system states and computes state probabilities and other parameters as each state is created. At the lower level, the transition rates among the states are computed using a QN solver, which determines the throughput of the computer system for each system state. Numerical results are presented to justify the decomposition method and validated through simulation. The approach is applicable to performance evaluation of programs with internal concurrency.

1. Introduction

An efficient procedure is developed in this paper to compute the performance measures for computations exhibiting parallelism in both centralized and distributed systems. The computation is specified by a task system (a set of tasks related by a deterministic precedence graph). Tasks are characterized by their expected total loadings at the devices of a given computer system, centralized or distributed. The potential for parallel processing occurs in numerous systems such as CPU:I/O overlap or inherent concurrency exploited by multi-tasking in centralized systems [4,5,7], real-time distributed computer systems [2], query processing in distributed databases [1], etc. The technique presented in this paper can be used to determine the key performance measure for such systems, the mean completion time.

Queueing network (QN) models have been extensively used in performance modelling of centralized/distributed computer systems [6]. However, such models assume that a program consists of a single process (task) which obtains service in a serial fashion from the devices in the QN model. Specifically, this implies that programs cannot hold more than one device at a time, and as such does not provide an accurate model for parallel processing. In addition closed QN models also assume a fixed workload, that is, when a job completes execution it is immediately replaced by an identical job. This is not the case in task systems where a completed task may be replaced by a set of tasks with different job types.

Recently, QN's were used to model programs with internal concurrency by Heidelberger and Trivedi [4,5]. These two papers also review previous work in this area, which is omitted here for the sake of brevity. In [4] a system where jobs are subdivided into two or more secondary tasks during their execution is considered. No synchronization among the tasks and the (primary) job is considered, however. In [5] a parent task spawns two or more concurrent tasks and has to wait until all such tasks are completed before it can proceed. Only very simple task systems can be handled by their approach. We consider much more complicated task systems than considered in [5], but allow only one instance of the task system at a time. The task system may correspond to the computations in a real-time system, in which case the task system is executed repeatedly, or the execution of a single instance of a program with internal concurrency.

The paper will be organized as follows. In Section 2 we first describe the task system model required for analysis. Also described is the computer system model used in our simulation for validation of results. In Section 3 we show that the model of a task system with two concurrent tasks is nearly completely decomposable and based upon this observation present a decomposition method for obtaining an approximate solution. Exact results obtained by solving the set of linear equations describing the system and results using the decomposition method are then presented. In Section 4 we derive expressions to compute the key performance measures for a task system and give a procedure for efficiently computing these measures for general task systems. A task

0190-3918/83/0000/0421$01.00 © 1983 IEEE

system is solved and validated using simulation. Lastly, we conclude the paper with a summary.

2. Task System and Computer System Models

In Section 2.1 we define a basic task system model which is required for analysis by the procedure in Section 4. In Section 2.2 a more detailed description of the task processing system is given for simulation purposes.

2.1 Task System Model

The set of tasks is to be executed on a computer system with K devices. Taking a QN modelling viewpoint only the expected total loadings (loadings for short) of each task at each device, which is the product of the mean number of requests a task makes to a device and the mean service time at that device per request, are required for computing the usual performance characteristics of the system when the QN has a product-form [6]. In summary, a task system is specified by a 3-tuple $(T,[<\bullet],[X_{ik}])$ as follows:

1. $T = (T_1,T_2,...,T_I)$ is a set of tasks to be executed. The initial or final tasks can be dummy tasks with no processing requirements (loadings equal to zero).

2. $[<\bullet]$ is a partial order defined on T specifying deterministic precedence constraints, i.e., $T_i <\bullet T_j$ means that T_i must be completed before T_j can begin. Only directed acyclic graphs (dags) are considered.

3. $[X_{ik}]$ is an I x K matrix, such that X_{ik} is the loading of task i at device k. At this point, only active system resources such as the CPU and I/O devices are considered. Passive system resources, such as memory requirements, can be incorporated easily into the model, but will be ignored in our discussions for the sake of brevity.

The concise specification of a task system as given above is required for a procedure given in Section 4 to compute the performance parameters of interest for task systems. These parameters, which are defined in Section 3.2 are the mean completion time of the task system, the mean initiation and completion time of each task, and its mean execution time. These parameters are of course dependent on the task scheduler. For the sake of simplicity, we consider a single-processor (CPU) system and assume that all tasks are executed as soon as the precedence constraints are satisfied. The procedure in Section 4 can be easily extended to take into account passive resources and to incorporate a more sophisticated task scheduling discipline.

2.2 Processing of Tasks in the Computer System

The computer system processes different combinations of tasks according to precedence constraints until all tasks are completed (in the case of a real-time system this cycle is then repeated). The tasks generally have different processing requirements at the computer system. In other words, for each task combination in progress at the computer system, we have a closed QN with multiple job types. To simplify discussion, the QN model postulated by us has a product-form solution. Under this assumption only the task loadings need be specified to compute the usual performance measures using efficient algorithms such as convolution or mean-value analysis [6]. Postulating a non-product form QN would require the use of approximate solution methods to solve the closed QN for each task combination [6]. Regardless of which method is used for solution of the closed QN, the decomposition procedure presented in this paper would remain valid.

The computer system model used is a generalization of the well known central-server model to multiple job types, called a centralized server model in. Shown in Figure 1 is the centralized model with two job types (with one task in each job type) consisting of a CPU and a disk. The queueing discipline at the CPU is Processor Sharing (PS) and the disk has a FCFS discipline. The previous assumptions assure a product-form QN. Unlike conventional closed QN's, a completed task is not immediately replaced by a new task with similar characteristics, but the precedence graph is checked to determine if any new task can be activated under precedence constraints.

3. Solving Concurrent Task Systems Using a Decomposition Approach

In this section we illustrate the decomposition method by applying it to concurrent task systems, which is defined as a set of tasks which can be executed concurrently such as the two-task system shown in Figure 2. The tasks are executed on the multiprogrammed computer system given in Figure 1. The subscripts c and d are used to denote the CPU and the disk, respectively. Loadings for task 1 and 2 are given as (120,200) and (220,400), respectively. The time units are in milliseconds. Both tasks have the same mean service times at the CPU ($1/\mu_c$ = 20 msec.) and the disk ($1/\mu_d$ = 40 msec.).

In Section 3.1 we justify the use of the decomposition method by building a Markov chain to solve the above task system exactly for its state probabilities, and also show that it is nearly completely decomposable [3]. In Section 3.2 we present the decomposition approximation method, use it to solve the same task system, and compare decomposition results to the exact solution.

3.1 Markov Chain Model

In this section we build a Markov chain for the solution of the two-task system defined in Figure 2 and executing on the computer system in Figure 1. In order to reduce the number of states in the Markov chain we assume that the queueing discipline at the disk is also PS. Therefore we do not need to specify the ordering of the tasks at the disk, which would be required in the case of a FCFS discipline. This is possible since the steady state probabilities obtained under the PS assumption at the disks will equal those obtained with the FCFS discipline (after appropriate state aggregation at the disks in the FCFS case). The state of the closed QN can be specified by the composition of jobs in execution. The number of states for the detailed state representation when a subset J of tasks is active is given by [9]: $\prod_{j\in J} \binom{n_j+K-1}{n_j}$. Needless to say, the number of states in the system increases rapidly with the number of concurrent tasks.

Figure 3 gives the transition rate matrix Q for the system at hand. Denoting the steady state probability vector by $\underline{p}$ we have $\underline{p}Q = 0$. In addition, we have the normalizing condition that state probabilities sum to one. After computing $\underline{p}$ by solving the linear equations, we can obtain the performance measures of the task system.

At this point it is interesting to note that the transition rate matrix Q can be decomposed into three sub-matrices, indicated in Figure 3 by dashed lines. For very small p_{c1c1} and p_{c2c2}, the six transition rate terms lying outside the dashed areas of the Q matrix are negligible, in which case we can say that the system is nearly completely decomposable [3]. Informally, it can be said that the system is decomposable when it reaches equilibrium between task completion instants. The latter is true when the occurrence of micro-events in the system (completions of tasks at devices) is much more frequent than macro-events (job completions). This is true in our system. The three groups of states into which the Q matrix is decomposed correspond to the following aggregated states: (i) (1,2) - task 1 and task 2 concurrently executing, (ii) (1) - task 1 only executing, and (iii) (2) - task 2 only executing. We use this observation as the motivation for solving parallel processing task systems using a decomposition method. Such an approach is particularly useful for systems with a large number of states, in which case it is not computationally feasible to solve the system exactly.

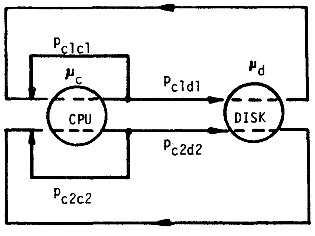


Figure 1. Detailed Centralized-Server Model of a Two-Task System

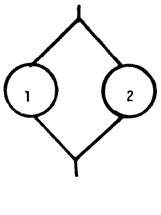


Figure 2. Task System with Two Concurrently Processing Tasks.

$$\begin{array}{c|cccc|cc|cc}
\text{AGGREGATE STATES} & \multicolumn{4}{c|}{\textcircled{1,2}} & \multicolumn{2}{c|}{\textcircled{1}} & \multicolumn{2}{c}{\textcircled{2}} \\
 & (1,1;0,0) & (0,1;1,0) & (1,0;0,1) & (0,0;1,1) & (1,0;0,0) & (0,0;1,0) & (0,1;0,0) & (0,0;0,1) \\
\hline
(1,1;0,0) & -\mu_c & \frac{\mu_c}{2} p_{c1d1} & \frac{\mu_c}{2} p_{c2d2} & 0 & \frac{\mu_c}{2} p_{c2c2} & 0 & \frac{\mu_c}{2} p_{c1c1} & 0 \\
(0,1;1,0) & \mu_d & -\mu_c-\mu_d & 0 & \mu_c p_{c2d2} & 0 & \mu_c p_{c2d2} & 0 & 0 \\
(1,0;0,1) & \mu_d & 0 & -\mu_c-\mu_d & \mu_c p_{c1d1} & 0 & 0 & 0 & \mu_c p_{c1c1} \\
(0,0;1,1) & 0 & \frac{\mu_d}{2} & \frac{\mu_d}{2} & -\mu_d & 0 & 0 & 0 & 0 \\
\hline
(1,0;0,0) & \mu_c p_{c1c1} & 0 & 0 & 0 & -\mu_c & \mu_c p_{c1d1} & 0 & 0 \\
(0,0;1,0) & 0 & 0 & 0 & 0 & \mu_d & -\mu_d & 0 & 0 \\
\hline
(0,1;0,0) & \mu_c p_{c2c2} & 0 & 0 & 0 & 0 & 0 & -\mu_c & \mu_c p_{c2d2} \\
(0,0;0,1) & 0 & 0 & 0 & 0 & 0 & 0 & \mu_d & -\mu_d
\end{array}$$

Figure 3. Transition Rate Matrix Q for Two-Task System.

3.2 Decomposition Method

Using decomposition, an approximate but highly accurate solution to our model can be obtained. The hierarchical decomposition method applicable in this case uses two modelling levels. In the higher level model the system state is specified by the combination of tasks in execution (aggregation of states of Section 3.1). The transitions among these states are governed by a Markov chain model. The transition rates among the states of the Markov chain are determined by the mean throughputs of the computer system in processing various task combinations and are computed at the lower level of the model. This computation can be carried out efficiently, since the system was assumed to have a product-form solution for each execution state (see Section 2). Otherwise, the throughputs could be computed by solving the set of linear equations specified by each sub-matrix corresponding to each aggregate state. Alternately, an approximate solution method could be used at this point to obtain the system throughput.

Figure 4 is the Markov chain respresenting the transitions among the aggregate states of the task system in Figure 2 (higher level model). Both tasks initially execute concurrently in state (1,2) until task 1 or 2 is completed, resulting in a transition to state (2) and (1) respectively. Task completions in each of these states lead to a transition to state (1,2), which indicates that the execution of the task system is complete, that is, another instance of the task system can be initiated. The states in the Markov chain are drawn at different levels, where the levels indicate the progression of the computation of the task system. Level one corresponds to the initial execution state, level two corresponds to the states which are feasible after a task is completed at the first level, and so on. The number of levels is equal to the number of tasks.

The state probabilities P(1) and P(2) can be expressed as a function of P(1,2) by solving the corresponding balance equations,

$$\begin{cases} P(1) = P(1,2) \times T_2(1,2) \;/\; T_1(1) & (3.1) \\ P(2) = P(1,2) \times T_1(1,2) \;/\; T_2(2) & \end{cases}$$

Setting P(1,2) = 1, we can obtain numerical values for P(1) and P(2). A normalization constant NC = P(1,2) + P(1) + P(2) is then used to insure that the probabilities sum to one. This simple scheme to compute the steady state probabilities is of importance when dealing with large task systems.

We define the mean completion (or cycle) time of the task system (C) as the average amount of time required to execute the task system on the given computer system. The mean execution time (E_i) of T_i is the average amount of time T_i is in execution. Also of interest is the mean initiation time (I_i) and completion time (C_i) for task T_i. Note that $E_i = C_i - I_i$.

The average rate at which the task system can be executed is given by the rate at which the initial state of the Markov chain (for the higher level model) is exited:

$$P(1,2) \times [T_1(1,2) + T_2(1,2)]$$

or the rate of exiting states (1) and (2):

$$P(1) \times T_1(1) + P(2) \times T_2(2)$$

for our example. It can be shown that the values of these two expressions are equal. The mean system completion time is the inverse of this rate,

$$C = [P(1,2) \times [T_1(1,2) + T_2(1,2)]\;]^{-1} \qquad (3.2)$$

Alternately, if we let $M(1,2) = [T_1(1,2) + T_2(1,2)]^{-1}$, where M(1,2) is the mean time spent in state (1,2), then C can be expressed as,

$$C = M(1,2) \;/\; P(1,2) \qquad (3.3)$$

The mean execution time of each task is the fraction of time a task is active in a cycle times the cycle time. It follows,

$$\begin{cases} E_1 = C \times [P(1,2) + P(1)] & (3.4) \\ E_2 = C \times [P(1,2) + P(2)] & \end{cases}$$

Finally, it should be noted that the branching probability that task i completes first in execution state (1,2), $b_i(1,2)$, is given as the ratio of the throughput resulting in the completion of the task and the sum of the throughputs [8],

$$b_i(1,2) = T_i(1,2) \;/\; [T_1(1,2) + T_2(1,2)] \qquad i=1,2 \quad (3.5)$$

The expressions obtained above are very useful in that they can be easily generalized to task systems with arbitrary precedence relationships.

3.3 Validation Results

Table 1 shows a comparison of values obtained for state probabilities, mean throughputs, mean completion time and mean execution time using the decomposition method and the exact method. In order to make the comparison meaningful, a careful interpretation of the probability vector $\underline{p}$ and the transition rate matrix Q of the Markov chain model must be made such that results equivalent to the aggregated state model of the decomposition approach are obtained. Computations for

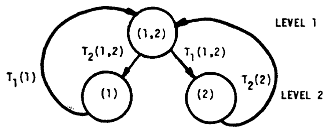


Figure 4. Markov Chain for Decomposition Method.

	Decomposition	Exact	% Error
P(1,2)	0.3851	0.3662	5.13
P(1)	0.1293	0.1404	-7.90
P(2)	0.4856	0.4934	-1.58
$T_1(1,2)$	2.034	2.094	-2.86
$T_2(1,2)$	1.050	1.118	-6.08
$T_1(1)$	3.125	2.911	7.35
$T_2(2)$	1.613	1.554	3.79
E_1	0.433	0.431	0.46
E_2	0.733	0.732	0.14
C	0.842	0.851	-1.06

Table 1. Comparison of Exact and Decomposition Method Results for Two-Task System Example. $T_i(\bullet)$ in tasks/sec.

obtaining the equivalent aggregated state values (in Table 1) from the Markov chain model are given in Appendix I.

Table 1 shows that results obtained using the decomposition method are in close agreement with exact values. It is worthwhile to mention that the mean completion time for the task system is different from the mean completion time for the larger of the two tasks.

4. Procedure for General Task Systems

In Section 4.1 we present an overview of our procedure for solving general task systems using the decomposition approach and in Section 4.2 we give a formal statement of our procedure. In Section 4.3 we solve an example and compare decomposition results against simulation results.

4.1 Procedure Overview

We are interested in obtaining the mean completion (cycle) time of a task system, as shown in Figure 5, as well as the mean initiation time, completion time, and execution time for each task in the system.

One of the key problems in dealing with task systems is the large number of states in the Markov chain at the higher level of the decomposition model. The number of states depends on the number of tasks and the precedence relationships among them. It is clear that the cost of solving a large set of linear equations ($\underline{p}Q = 0$) can be significicant for larger task systems.

Fortunately, an efficient scheme is available due to the fact that the task systems which we are interested in are directed acyclic graphs. As such, the corresponding Markov chain is also acyclic (with the exception of transitions to the initial state). Based on the observation made regarding (3.1) in Section 3.2, we can therefore express all probabilities as a function of the initial probability. Furthermore, such unnormalized probabilities can be used in computing the required performance parameters (such parameters will have to be re-adjusted by a normalization constant). Finally, to save memory space, the Markov chain can be generated one level at a time. Once an entry corresponding to a state at one level has been used to generate the entries at the following level, it can be deleted to save space. The number of states at each level is usually much smaller compared to N (the total number of states).

In the case of concurrent task systems, all tasks are initiated at time zero and the mean execution time for task i (E_i) equals the mean completion time for that task (C_i).

For a general task system, the mean execution time for task i is given by

$$E_i = C_i - I_i \qquad 1 \leq i \leq I \tag{4.1}$$

where I_i is the mean initiation time for task i. I_i is computed by considering all possible paths (in the Markov chain), which lead to the initiation of the task. Given that there are K paths leading to the initiation of task i we have,

$$I_i = \sum_{k=1}^{K} \Big(\Big[\prod_{j \in L_k} b_j \Big] \times \sum_{S \in S_k} M(S) \Big)\, b_k \tag{4.2}$$

In the above expression, L_k is the set of links (branches) along path k leading to the state which immediately precedes the initiating state of task i. S_k is the set of states along path k and b_k is the branching probability from the last state in path k to the state in which task i is initiated. This computation can also be carried out a level at a time by updating a vector of entries for task initiation times. The details appear in the procedure presented in Section 4.2.

Individual task completion times and more importantly the completion time of the task system can be computed using arguments similar to those used in deriving mean initiation times. We use this latter approach (rather than an approach based on equation (3.2)) to compute the mean system completion time.

We point out that the computational cost of solving the product-form QN models to obtain throughputs is relatively small, since there is only one task in each job type. The computational cost to compute the normalization constant for the convolution algorithm when there are N tasks in a computer system consisting of K devices is: $2^{N+1}KN$ [9]. Additional savings in computational cost can be achieved by making note of state dominance, i.e., the successor states of a state (for one or more levels) have a subset of the tasks of the parent state in their composition (unless a precedence relationship is satisified and new tasks are initiated). The throughputs of these successor states can be obtained as a by-product of the computations of the parent state.

4.2 Statement of Procedure

The notation and some of the formulas used in the procedure are as follows:

L = number of levels in Markov chain
= number of tasks in task system (I)

S_ℓ = set of states at level ℓ

S = current state (state under consideration)

$|S|$ = set of tasks in state S

R = successor state to current state

S^+ = set of all successor states to current state, $R \in S^+$

$P(S)$ = steady state probability of S

$T_i(S)$ = throughput at S due to completion of task $i \in |S|$

$T(S)$ = total throughput at $S = \sum_{i \in S} T_i(S)$

$M(S)$ = mean residence time in S, $= 1/T(S)$

$b_R(S)$ = probability of branching from S to a new state R due to completion of task i, $b_R(S) = T_i(S)/T(S)$

$p(R)$ = path probability to reach R

$= \sum_{S \in R^-} p(S)\, b_R(S)$ where R^- is the set of the predecessor states of R. Note that $\sum_{S \in S_\ell} p(S) = 1$

$D(R)$ = mean delay along path up to and including R

$= \sum_{\mathit{all\ paths}} [D(S) * b_R(S) + M(R) * p(R)]$

Each entry in the Markov chain can be represented by the following record:

$[P(S);\ p(S);\ D(S);\ T_i(S),\ i \in |S|;\ T(S);\ M(S);\ b_R(S),\ R \in S^+]$

PROCEDURE: Performance Evaluation of a Task System

Step 0. Input parameters:

- I (number of tasks)
- $[<\!\bullet]$ (precedence relationships among tasks)
- K (number of devices in computer system)
- X_{ik}, $1 \leq i \leq I$, $1 \leq k \leq K$, (task loadings)
- Initialize dummy state Q at level zero ($S_0 =$ Q): $P(Q) = 0$, $P(Q) = 1$, $D(Q) = 0$.

Step 1. Generate Markov chain:

```
for levels $\ell = 0$ to $L-1$ do
  for states $S \in S_\ell$ do
  - Determine all successor states to S: $S^+$ (taking into consideration precedence) and create new entries for these states at level $\ell+1$ and initialize them (unless previously created). $S_{\ell+1} = S_{\ell+1} \cup S^+$
  - Compute branching probabilities from S to $R \in S^+$:
      $b_R(S) = \begin{cases} 1 & \ell = 0 \\ T_i(S)/T(S) & \ell > 0 \end{cases}$
    where $i \in |S|$ and completion of $T_i$ leads from S to R.
    for states $R \in S^+$ do
    - Compute throughput $T_i(R)$ for $i \in |R|$
    - Compute total throughput $T(R) = \sum_{i \in |R|} T_i(R)$
    - Compute mean time in state, $M(R) = 1 / T(R)$
    - Compute (partial) path probability, $p'(R) = p(S)\, b_R(S)$
    - Compute mean path delay $D(R) = D(R) + D(S) * b_R(S) + M(R) * p'(R)$
    - Update total path probability to state R, $p(R) = p(R) + p'(R)$
    - Compute (partial) state probability of new state,
        $P'(R) = \begin{cases} 1 & \ell = 0 \\ T_i(S)\, P(S) / T(R) & \ell > 0 \end{cases}$
    - Update total state probability $P(R) = P(R) + P'(R)$
    - Update initiation time of tasks newly activated in R,
        $I_i = I_i + D(S) * b_R(S)$
    - Update completion time of task which executed in S but is no longer executing in R
        $C_i = C_i + D(S) * b_R(S)$
    - Update normalization constant $NC = NC + P'(R)$
    end
  end
end
```

```
Step 2. Compute final results:
- Compute task system completion time (C),
      for $S \in S_L$ do
            $C = C + D(S)$
- Compute task i execution time ($E_i$)
  for i = 1 to I do
            $E_i = C_i - I_i$
- Normalize state probabilities
  for all P(S) do
            $P(S) = P(S)/NC$
End of Procedure.
```

It should be noted that the mean completion time (C) and individual task execution time (E_i) could also be computed using the alternate formulas and techniques in Section 3.2.

4.3 Numerical Example and Validation

The above procedure was used to solve the task system of Figure 5 whose corresponding Markov chain is given in Figure 6. The inner model is a centralized server QN consisting of a CPU ($1/\mu_c$ = 20 msec.) and two identical disks ($1/\mu_d$ = 40 msec.). Task loadings are specified in Table 2.

Results obtained from the decomposition method are compared against simulation results in Table 3. The simulation was run for 8000 replications and 95% confidence intervals were obtained with interval halfwidths of less than 3% of the estimated mean value for all values presented. As can be seen, values obtained using decomposition compare very favorably to simulation results.

5. Conclusions

The paper deals with the issue of analyzing the execution of task systems to determine the mean completion time and the mean initiation time of tasks. The only parameters required for this computation are the loadings of the tasks at the various units of the computer system. An approximation approach based on hierarchical decomposition was developed to analyze the system. The computational procedure discussed in this paper can compute performance parameters while generating all possible (execution) states for the task system. The efficiency of this procedure is of essence for the feasibility of the method due to the large number of system states. Work is in progress in generalizing the procedure to handle the effect of scheduling, probabilistic task systems, etc.

(seconds)	Decomposition	Simulation	% Error
C	6.081	6.147	-1.07
E_1	1.830	1.792	2.12
E_2	1.830	1.819	1.17
E_3	2.178	2.179	-0.00
E_4	2.443	2.469	-1.07
E_5	1.698	1.679	1.15
E_6	1.698	1.721	-1.33
I_1, I_2, I_4	0	0	0
I_3	2.627	2.615	1.20
I_5, I_6	2.443	2.469	-1.07

Table 3. Comparison of Decomposition Method Results and Simulation Results for a Six-Task System.

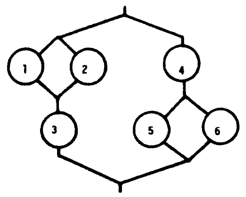


Figure 5. General Task System.

TASK	CPU	$DISK_1$	$DISK_2$
1	420	400	400
2	420	400	400
3	620	600	600
4	620	600	600
5	420	400	400
6	420	400	400

Table 2. Expected Total Loadings (milliseconds) for Figure 5 Task System.

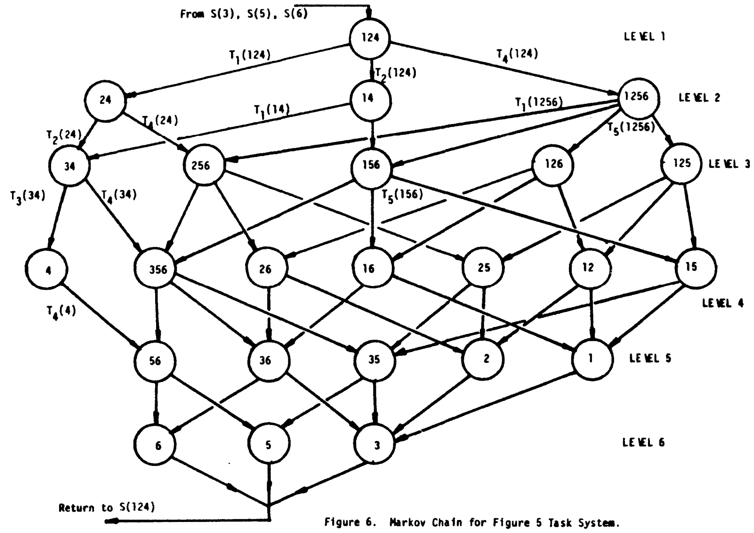


Figure 6. Markov Chain for Figure 5 Task System.
(Note: Only a representative set of throughputs shown.)

REFERENCES

1. P.A. Bernstein, et al. "Query processing in a system for distributed databases (SDD-1)," ACM Trans. Database Systems 6, 4(Dec. 1981), 602-625.

2 W.W. Chu, et al. "Task allocation in distributed data processing," IEEE Computer 13, 11(Nov. 1980), 57-69.

3. P.J. Courtois, "Decomposability: Queueing and Computer System Applications, Academic Press, 1977.

4. P. Heidelberger, and K.S. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," IEEE Trans. Computers 31, 11(Nov. 1982), 1099-1109.

5. P. Heidelberger, and K.S. Trivedi, "Analytic queueing models for programs with internal concurrency," IEEE Trans. Computers 32, 1(Jan. 1983), 73-82.

6. S.S Lavenberg (ed.) Computer Performance Modelling Handbook, Academic Press, 1983.

7. D. Towsley, K.M. Chandy, and J.C. Browne, "Models for parallel processing within programs: Application to CPU:I/O and I/O:I/O Overlap," Comm. ACM 21, 10(Sept. 1978), 821-830.

8. K.S. Trivedi, Probability and Statistics with Reliability, Queueing, and Computer Science Applications, Prentice-Hall, 1982.

9. J. Zoharjan, "The approximate solution of large queueing network models," Ph.D. thesis, Technical Report CSRG-122, Computer Systems Research Group, Univ. of Toronto, August 1980.

Appendix I

Computing aggregated state values from the Markov chain model are performed as follows. Solving $\underline{p}Q = 0$ for the steady state probabilities gives,

$$\underline{p} = [0.0764\ 0.0519\ 0.0538\ 0.1841\ 0.0491\ 0.0913\ 0.01687\ 0.3247]$$

From these state probabilities, the aggregated state probabilities can be obtained,

$$\begin{aligned} p(1,2) &= p(1,1;0,0) + p(0,1;1,0) + \\ &\quad p(1,0;1,0) + p(0,0;1,1) \\ &= 0.3662 \end{aligned}$$

$$p(1) = p(1,0;0,0) + p(0,0;1,0) = 0.1404$$

$$p(2) = p(0,1;0,0) + p(0,0;0,1) = 0.4934$$

Mean completion time of the task system is computed by first obtaining the mean rate at which the task system completes. This is given by the total mean throughput from aggregated state S(1) and S(2) to aggregated state S(1,2) (see Figure 4). Taking the inverse of the total mean throughput gives the task system mean completion time in seconds,

$$\begin{aligned} C &= [\ p(1,0;0,0) \times \mu_c \times p_{c1c1} + \\ &\qquad p(0,1;0,0) \times \mu_c \times p_{c2c2}\ ]^{-1} \\ &= 0.850 \end{aligned}$$

The throughputs between aggregated states (in jobs/second) are computed by obtaining the mean total transition rate between each source and destination aggregated state, conditioned by the probability of being in the source aggregated state. This gives the following set of equations (see Figure 4),

$$\begin{aligned} T_1[1,2] &= [p(1,1;0,0) \times 0.5 \times \mu_c \times p_{c1c1} \\ &\qquad + p(0,1;1,0) \times \mu_c \times p_{c1c1}] / p(1,2) \\ &= 2.209 \end{aligned}$$

$$\begin{aligned} T_2[1,2] &= [p(1,1;0,0) \times 0.5 \times \mu_c \times p_{c2c2} \\ &\qquad + p(1,0;0,1) \times \mu_c \times p_{c1c1}] / p(1,2) \\ &= 1.050 \end{aligned}$$

$$T_1[1] = p(1,0;0,0) \times \mu_c \times p_{c1c1} / p(1) = 3.125$$

$$T_2[2] = p(0,1;0,0) \times \mu_c \times p_{c2c2} / p(2) = 1.554$$

where p(1,2), p(1) and p(2) are computed above.

Lastly, task 1 and task 2 mean execution times, E_1 and E_2 respectively, are computed using equation (3.3) in Section 3.2 and the above values.

On the Performance of Interleaved Memories with Non-uniform Access Probabilities*

H.C. Du
Department of Computer Science
University of Minnesota
Minneapolis, Minnesota 55455

and

J.L. Baer
Department of Computer Science
University of Washington
Seattle, Washington 98195

ABSTRACT-- System structure and program behavior are two major factors that influence the performance of a tightly-coupled multiprocessor. The latter has been usually ignored in most of the previous studies. In this paper, we study the performance of a tightly-coupled multiprocessor in which a crossbar is employed to interconnect p processors to m memory modules. A set of non-uniformly distributed probabilities is also employed to illustrate the program behavior, but no distinction is made between processors. An inverse relation between the average request completion time and the effective memory bandwidth is obtained and three approximation methods are proposed. Their solutions are compared with the exact solution. Among them the Repetitive Augmenting Method which based on the idea of aggregation generates the best result.

1. Introduction

There is a well known mismatch between processor and memory speeds in computer systems. Memory speed is about one order of magnitude slower than processor speed. One technique being widely employed to solve this problem is to provide some parallelism for memory accessing by partitioning the memory into a number of modules, this is the so called interleaved memory scheme.

In a tightly-coupled multiprocessor system, an m-way interleaved memory is shared by p processors. A processor can access each memory module via some processor-memory switch (interconnection network). In the past, a crossbar switch has been widely used to interconnect various combinations of computer subsystems including processors to memories. A crossbar switch allows a processor to access any memory module as long as it is the only one trying to access the memory module. Therefore, several requests can be satisfied if they access different memory modules, but only one of these requests to a given memory module can be satisfied.

However, a crossbar switch suffers from the fact that it requires O(m.p) switching components to interconnect p processors to m memory modules. The hardware cost can be enormous for large m and p. For this reason, a whole range of ingenious interconnection networks have been proposed that include the following : Banyan network, Omega network, Delta network, Baseline network, Data Manipulator network and Augmented Data Manipulator network. By no means, is the above list exhaustive. For a good survey, see [7] and [16].

While most of the recently proposed interconnection networks require only O(n log n) switching components to interconnect n processors to n memory modules, they do not preserve the bandwidth of a crossbar switch [11]. Mudge and Makrucki [10] pointed out that, in the context of VLSI technology, reduced component complexity may be no longer an advantage within a single IC. For example, they compared the layouts of a Delta network and a bit-slice crossbar and found that the reduced component complexity does not appear to translate into more efficient space utilization in an IC layout [9].

Several investigators have studied the performance of a crossbar switch for tightly-coupled multiprocessors ([2], [3], [4], [12] [13] and [14]). The results reported by these investigators were obtained by applying either a simplified approximation mathematical model or some Markov Chain model to derive the effective memory bandwidth, that is the average number of requests that can be satisfied in one memory cycle time.

The effective memory bandwidth does not only depend on the system structure but is also highly dependent on the distribution of all requests which are outstanding at a given time. This transient distribution for requests represents the program behavior. Most studies in the past have ignored this major factor by assuming all modules have an equal probability of being accessed by any given request. In [12] a trace driven simulation technique was used to take program behavior into consideration. In the same paper Rau has also shown that the commonly used uniform access distribution is not valid. Some modules are indeed referenced more often than others.

Even though it is true, that in the long term, the probabilities of a request accessing different modules are uniformly distributed (as assumed by most models), the memory bandwidth at a given time is determined by all outstanding requests at that time. Thus, if we partition the whole observation period into several subperiods and are able to determine the program behavior in each subperiod, a more precise memory bandwidth can be obtained by averaging the effective memory bandwidths in all subperiods.

For the above reasons, in this paper we study the performance of a tightly-coupled multiprocessor using a crossbar to interconnect p processors to m memory modules and assuming a set of non-uniformly distributed probabilities to illustrate program behavior. Let P(i) denote the probability that a request is directed to the ith module. It is assumed that P(i) is not necessarily equal to P(j) for $i \neq j$. From the memory bandwidth viewpoint, the program behavior of a multiprocessing system can be characterized by a set of P(i)'s. In this study, no distinction is made between processors.

In the following sections, a simple model for a multiprocessor system is first described. Then one exact solution and three approximation solutions for finding the performance of the proposed model are presented.

0190-3918/83/0000/0429$01.00 © 1983 IEEE

An inverse relation between the average request completion time and the average memory bandwidth is also obtained.

2. System Model

In this paper, all analyses are based on a model which is similar to the one proposed by Rau [13] but which has a set of probabilities with non-uniform distributions to illustrate the program behavior. The model is shown in figure 2.1. The following assumptions are made:

1) There are p independent processors (labeled as P_1, P_2 , ..., P_p) and an m-way interleaved memory (labeled as M_1, M_2, ..., M_m) in the system, where p is not necessarily equal to m.
2) The p processors and m memory modules are connected by either a cross-bar switch (hence each processor can access any of the m modules) or a switch system which guarantees that a memory request issued by a processor can be satisfied if it is the only request directed to the desired memory module.
3) The m memory modules are synchronized, i.e., they start a memory cycle at the same time and have an identical memory cycle time.
4) At the beginning of each memory cycle, each processor generates a request. If a processor's previous request has not been satisfied, the processor will generate the same request again.
5) Each memory module can serve one and only one request during a memory cycle time even though there may be other requests directed to it.
6) There is a memory conflict resolution method which chooses one request to service by following some rule if more than one request references the same memory module.
7) Requests made by each processor have P(i) probability to access the ith memory module.

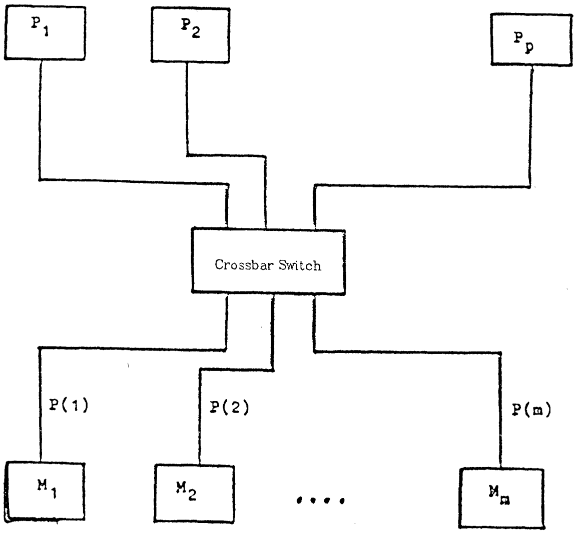


Figure 2.1 A Multiprocessor System With p Processors and m Memory Modules.

Sethi and Deo have studied the performance of a multiprocessor system with non-uniform access probabilities [15]. However, they assumed that the memory in a multiprocessor system is partitioned into modules by the higher order bits of the address and the program behavior is illustrated by a processor having probability α to access the same memory module as its previous request and probability $(1-\alpha)/(m-1)$ to access a different module. Since no distinction is made between processors and the program behavior is illustrated by a set of probabilities, our model fits the cases where several processors execute the same programs on some shared data and the memory is partitioned into memory modules by the lower order bits of the address. One such example was presented in [1] : in the batch binary search in a multiprocessing environment of p processors and m interleaved memory modules. An ordered table of n elements is stored in memory in such a way that element X_i is stored in module M_j with j = (i mod m)+1. Each processor, when free, takes a request from a queue, i.e., a key K, and performs a binary search algorithm on the table. Since some elements are referenced more often than others (for example, every query needs to access the "middle" element), the probabilities of accessing the various memory modules are different. The above example can be generalized to that of a directory of some file (or database) stored on a common interleaved memory. Each user, associated with a processor, searches the directory to respond to a given query. Since no distinction is made between the types of queries asked by all users, there is no distinction between processors.

3. Exact Solution

In this section we adapt the analysis of [2] and [3] for the uniform case to the non-uniform probability model. The complete set of states of the model described above can be defined as an m-tuple (k_1, k_2, ...,k_m), where k_i is the number of requests directed to the ith module at a given time. Since there are p processors in the system and each holds one request at a time, $k_1+k_2+\ldots+k_m$=p. The number of such states K=(k_1,k_2, . . . , k_m) is the number of ways to partition p processors into m memory modules : C(p+m-1, m-1) [6] (C(i,j) is the number of ways to choose j elements from a pool with i elements). The number (NZ(K)) of non-zero integers in k_1,k_2,... k_m is the number of requests currently being serviced at the state K=(k_1,k_2, . . . , k_m). This model results in a Markov Chain as shown in figure 3.1, since the choice of the next state is only affected by the current state. Each memory module has its own queue and each request has a probability P(i) to be directed to the ith memory module. If more than one request references the same memory module, one of them is chosen for service by some memory conflict resolution method and all others remain in the queue (regenerated at the next memory cycle) for future memory cycles.

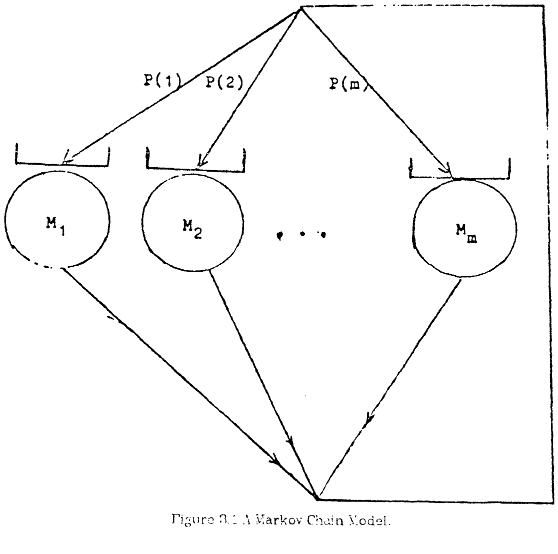


Figure 3.1 A Markov Chain Model.

It is assumed that each P(i) $\neq$ 0 for $1 \leq i \leq m$, since in the cases where P(i)=0 for some i, there is no request to reference the ith module and this can be viewed as if there were only m-1 memory modules in the system. It was pointed out in [2] that in addition to being a Markov Chain, the system is aperiodic, since a transition from any state to itself is possible in one step, and the system is irreducible, since any state can reach any other state in a finite number of steps. Let $f^i{}_{KK}$ denote the probability of taking i transitions for the system to first return to its initial state K. It is not hard to see that all states in the system are ergodic, since $\sum_{i=1}^{\infty} f^i{}_{KK}=1$ for any state K. Thus, there exists a unique stationary distribution for all states of the system [8]. This means that there exists a unique solution for all Prob(K)'s where Prob(K) is the probability of the system being in state K.

We also define a temporary state K' as (k_1', k_2', ..., k_m'), where $k_i' = k_i$-1 if $k_i \geq 1$ or $k_i' = k_i$ if k_i=0. The temporary state K' represents the state of the system at the end of the current memory cycle and before each "satisfied" processor generates a new request. By definition, it is easy to see that $k_1+k_2 + \cdots +k_m = k_1' + k_2' + \ldots + k_m'$ + NZ(K) = p and $1 \leq NZ(K) \leq m$. It is also true that NZ(K) $\geq$ NZ(K'), since $k_i' \leq k_i$ for $1 \leq i \leq m$.

Let K'= (k_1', k_2', ..., k_m') be the uniquely determined temporary state of state K=($k_1,k_2, \ldots ,k_m$). At the beginning of the very next memory cycle, there are NZ(K) free processors and each generates a new request. There are C(NZ(K)+m-1,m-1) ways to partition NZ(K) new requests into m memory modules. Let ($d_1,d_2, \ldots ,d_m$) be one such possible result, where NZ(K) = $d_1+d_2+\ldots+d_m$ and d_i is the number of new requests referencing the ith memory module. Let N= ($k_1'+d_1$, $k_2'+d_2$, ..., $k_m'+d_m$) and P(K → N) be the probability of having the system transfer from state K to state N in one step (one memory cycle). Then

$$P(K \rightarrow N)=C(NZ(K),d_1).P(1)^{d_1}.C(NZ(K)-d_1,d_2).P(2)^{d_2} \cdots P(m)^{d_m}$$

$$=(NZ(K)!/(d_1!d_2!\ldots d_m!)).P(1)^{d_1}.P(2)^{d_2}\ldots P(m)^{d_m}$$

(where n! = n.(n-1)...2.1).

Thus we can compute the probability P(K → N) for each possible state N which can be reached from state K in one step (there are C(NZ(K)+m-1, m-1) such states).

Let Prob(N) be the probability of the system being in state N. Then the following two conditions are satisfied [8] :

1) For each state N, prob(N) $=\sum_K Prob(K).P(K \rightarrow N)$.

2) $\sum_N$ Prob(N) = 1.

Since there are C(p+m-1,m-1)+1 equations which satisfy either conditions (1) or (2) and C(p+m-1,m-1) unknowns, these equations can be solved. Gauss elimination is one possible way to do so, although there are better programming methods as shown in [3].

Once all the Prob(K)'s are known, several important performance factors can be calculated as follows :

1) the average memory bandwidth B
$= \sum_K$ Prob(K).NZ(K),

2) the utilization factor of module i (U_i), i.e., the probability of the ith module being busy
$= \sum_{K \text{ with } k_i>0}$ Prob(K=($k_1,k_2, \ldots ,k_m$)),

3) the average number of requests, including the one currently being serviced, which are queued in the ith memory module (L_i)
$= \sum_K$ Prob(K=($k_1,k_2, \ldots ,k_m$)). k_i.

Although the number of possible states C(p+m-1,m-1) = p+1 in the case where m=2, the number of possible states in general is very large. For instance, C(p+m-1,m-1) = 6,435 for m=p=8 and 300,540,195 for m=p= 16. Baskett and Smith pointed out in [2] that C(p+m-1,m-1) grows as fast as 4^p for p=m. Therefore, the procedure to find out the exact solution for a system is very time consuming and this method is unrealistic for a system with medium to large p and m. Therefore we need to explore some approximation solutions.

4. Memory Bandwidth and Request Completion Time

A request generated by a processor takes one memory cycle time to be serviced and spends zero or more cycle times waiting for service. Let us call the total time that a request is staying in the system (sum of the service time and waiting time) as the request completion time. It is obvious that the request completion time is an important metric of the system performance. Let T_i denote the request completion time for a request directed to the ith memory module and L_i be the average number of requests queued in the ith memory module, including the one being serviced if any. It is easy to see that $L_1+L_2+\ldots+L_m=p$. Let T=$\sum_{i=1}^{m} P(i).T_i$ be the average request completion time.

By the definition of memory bandwidth, there are on the average B (memory bandwidth) requests being satisfied during one memory cycle. That is, on the average B new requests are issued at the beginning of each memory cycle. Therefore, the arrival rate for memory module M_i is B.P(i).

By Little's Law, the number of requests in queue
= (arrival rate) X (the average time that a customer stayed enqueued).

Applying Little's law to our model yields

$\sum_{i=1}^{m}$ B.P(i).T_i = $\sum_{i=1}^{m} L_i$ = p and

$$B=p/(\sum_{i=1}^{m} P(i).T_i)=p/T \qquad (1)$$

By equation (1), there is an inverse relation between memory bandwidth and request completion time. Two approximation methods are proposed in the next section. One of them is derived from this inverse relation.

5. Two Simple Approximation Methods

We first consider an approximation method which is a generalization of the approximation method presented in previous papers ([2], [4] and [17]). In our model, each processor with its previous request unsatisfied will regenerate the same request at the next memory cycle. Thus, at the beginning of the next memory cycle, the number of newly generated requests may be less than p. If it is assumed that each processor, independently of whether its previous request is satisfied or not, generates a new request at each memory cycle, it can be shown [18] that the average memory bandwidth B is given by :

$$B=m-\sum_{i=1}^{m}(1-P(i))^p \qquad (2)$$

Since 1-P(i) is the probability for one request not to access the i-th memory module and $(1-P(i))^p$ is the probability for all p requests not to access the i-th memory module, m- $\sum_{i=1}^{m} (1-P(i))^p$ is the expected number of busy memory modules (i.e., at least one request among p possible ones directs to it). When P(i)= 1/m for $1 \leq i \leq m$, B = m-m(1- $1/m)^p$. This is consistent with the result presented in [2], [4] and [17]. It is not hard to see that equation (2) has a maximum m-m.(1- $1/m)^p$ occurring at P(1)=P(2)= ... =P(m)= 1/m, and a minimum 1 at P(j)=1 for some j and P(i)=0 for all other $i \neq j$.

Since the queueing behavior of unsatisfied requests is ignored, the above result is optimistic, i.e., will yield larger B's than the exact solution. In the rest of this section, we derive another approximation method by utilizing the inverse relation (as shown in equation (1) of section 4) between the average memory bandwidth and the average request completion time.

In our second approximation method we isolate a processor whose request has not been satisfied. Let us assume that its request was for the ith memory module. Then at the next memory cycle it will again access M_i while the remaining (p-1) processors generate new independent requests. A memory conflict resolution method, which randomly chooses anyone of the requests directed to a given module, is also assumed. Note that different conflict resolution methods may cause little difference in the overall performance [4].

Let Q_i denote the probability for a request directed to the ith module of not being satisfied at a given memory cycle.

The probability of having j-1 ($j \geq 1$) other requests directed to the same (ith) module is :

$$C(p\text{-}1,j\text{-}1).(P(i))^{j-1}.(1-P(i))^{p-j}$$

Thus, $Q_i = \sum_{j=1}^{p}$ ((j-1)/j). $C(p\text{-}1,j\text{-}1).(P(i))^{j-1}.(1-P(i))^{p-j}$,

where the factor (j-1)/j indicates that one out of j requests will be satisfied. Thus, the probability of a request directed to the ith module being satisfied at a given memory cycle is:

$$1-Q_i = \sum_{j=1}^{p} (1/j).C(p-1,j-1).(P(i))^{j-1}.(1-(P(i))^{p-j}$$

$$=(1/(p.P(i)).\sum_{j=1}^{p} C(p,j).(P(i))^{j}.(1-P(i))^{p-j}$$

$$=(1/(p.P(i)).(1-(1-P(i))^{p})$$

The average completion time for a request directed to the ith module T_i is

$$\sum_{k=1}^{\infty} k.(1-Q_i).Q_i^{k-1}=(1-Q_i).\sum_{k=1}^{\infty} k.Q_i^{k-1}=(1-Q_i).1/(1-Q_i)^2$$

$$=1/(1-Q_i)=p.P(i)/(1-(1-P(i))^{p})$$

The average request completion time T is then :

$$\sum_{i=1}^{\infty} i.T_i = =p.\sum_{i=1}^{m} P(i)^2/(1-(1-P(i))^{p})$$

$$\textit{By equation } (1),\ B=p/T=1/(\sum_{i=1}^{m} P(i)^2/(1-(1-P(i))^{p})) \qquad (3)$$

This result is in general still optimistic but more accurate than the previous one as shown in Tables 5.1, 5.2 and 5.3 (see below). It could still be improved if we could find a more precise way to compute T_i.

For convenience, in the following discussion, the two approximation methods to compute the average memory bandwidth according to equations (2) and (3) are called Approximation Method 1 and 2 respectively (AM1 and AM2). Tables 5.1, 5.2 and 5.3 compare the exact solution of the average memory bandwidth (EXACT) and the two approximation solutions (AM1 and AM2) calculated by equations (2) and (3) when m=p=2, 4 and 6 respectively. There are 15 cases in each table and each row represents a case being considered. RD_i, for i=1 or 2, is the relative difference between EXACT and AMi and is defined to be 100.|EXACT-AMi|/EXACT. Since it takes an unbearable time to compute the exact solution for large m and p, we did the comparisons only for small m and p.

By comparing the results shown in Tables 5.1, 5.2 and 5.3, we can observe that :

1) In all cases, both AM1 and AM2 are optimistic. However, AM2 is closer to EXACT than AM1.

2) For a fixed m and p, both AM1 and AM2 are closer to EXACT (with smaller RD_i) when the probability distribution is more uniform (case (8) in Table 5.1, 5.2 and 5.3).

3) As m and p increase, in some cases RD_i may get fairly large. For instance, RD_1=50.11 and RD_2=32.50 in case (2) of Table 5.3.

Because of the disturbing feature of the last point, a new approximation method which takes polynomial time and yields better results when the exact solution is not feasible is proposed in the next section.

Table 5.1

	P(1)	P(2)	EXACT	AM1	RD1	AM2	RD2
(1)	0.89	0.11	1.1217	1.1958	6.61	1.1628	3.66
(2)	0.75	0.25	1.3000	1.3750	5.77	1.3462	3.55
(3)	0.68	0.32	1.3854	1.4352	3.60	1.4172	2.30
(4)	0.63	0.37	1.4367	1.4662	2.05	1.4560	1.34
(5)	0.59	0.41	1.4686	1.4838	1.03	1.4786	0.68
(6)	0.54	0.46	1.4936	1.4968	0.21	1.4957	0.14
(7)	0.45	0.55	1.4901	1.4950	0.33	1.4934	0.22
(8)	0.50	0.50	1.5000	1.5000	0.00	1.5000	0.00
(9)	0.43	0.57	1.4808	1.4902	0.63	1.4870	0.42
(10)	0.35	0.65	1.4174	1.4550	2.65	1.4417	1.71
(11)	0.29	0.71	1.3501	1.4118	4.57	1.3889	2.87
(12)	0.26	0.74	1.3127	1.3848	5.49	1.3574	3.41
(13)	0.17	0.83	1.1966	1.2822	7.15	1.2464	4.16
(14)	0.14	0.86	1.1586	1.2408	7.09	1.2053	4.03
(15)	0.98	0.02	1.0204	1.0392	1.84	1.0300	0.94

Memory Bandwidths Derived From Exact Solution and Two Approximation Solutions for m=p=2.

Table 5.2

	P(1)	P(2)	P(3)	P(4)	EXACT	AM1	RD1	AM2	RD2
(1)	0.88	0.04	0.04	0.04	1.1364	1.4518	27.75	1.2400	9.12
(2)	0.67	0.16	0.11	0.06	1.4921	2.0821	39.54	1.8045	20.94
(3)	0.54	0.17	0.13	0.16	1.8420	2.4099	30.83	2.2182	20.42
(4)	0.46	0.24	0.18	0.12	2.1111	2.5295	19.82	2.4224	14.75
(5)	0.33	0.41	0.06	0.20	2.2083	2.4870	12.62	2.4282	9.96
(6)	0.31	0.23	0.28	0.18	2.5416	2.7009	6.27	2.6868	5.71
(7)	0.28	0.15	0.35	0.22	2.4498	2.6606	8.60	2.6299	7.35
(8)	0.25	0.25	0.25	0.25	2.6270	2.7344	4.09	2.7344	4.09
(9)	0.22	0.19	0.13	0.46	2.1154	2.5415	20.14	2.4327	15.00
(10)	0.18	0.22	0.30	0.30	2.5345	2.6975	6.43	2.6820	5.82
(11)	0.16	0.29	0.40	0.15	2.2965	2.5764	13.06	2.5340	10.34
(12)	0.13	0.21	0.15	0.51	1.9398	2.4579	26.71	2.2994	18.54
(13)	0.08	0.38	0.16	0.38	2.2065	2.4902	12.86	2.4247	9.89
(14)	0.06	0.24	0.57	0.13	1.7457	2.2785	30.52	2.0884	19.63
(15)	0.05	0.35	0.45	0.15	2.0705	2.3938	15.61	2.3118	11.65

Memory Bandwidths Derived From Exact Solution and Two Approximation Solutions for m=p=4.

Table 5.3

	P(1)	P(2)	P(3)	P(4)	P(5)	P(6)	EXACT	AM1	RD1	AM2	RD2
(1)	0.19	0.30	0.12	0.13	0.15	0.11	3.1892	3.8278	20.02	3.7154	16.47
(2)	0.47	0.04	0.02	0.10	0.19	0.18	2.1260	3.1914	50.11	2.8170	32.50
(3)	0.16	0.21	0.15	0.11	0.09	0.28	3.2950	3.8243	16.06	3.7350	13.35
(4)	0.26	0.13	0.21	0.14	0.19	0.07	3.3874	3.8251	12.92	3.7581	10.94
(5)	0.09	0.11	0.20	0.25	0.22	0.13	3.4079	3.8362	12.57	3.7686	10.58
(6)	0.11	0.14	0.09	0.18	0.28	0.20	3.3000	3.8251	15.91	3.7373	13.25
(7)	0.21	0.07	0.11	0.15	0.32	0.14	3.0207	3.7324	23.50	3.5858	18.71
(8)	0.16	0.17	0.16	0.17	0.17	0.17	3.7781	3.9896	5.60	3.9891	5.58
(9)	0.27	0.19	0.13	0.10	0.14	0.17	3.3886	3.8697	14.20	3.7987	12.10
(10)	0.10	0.11	0.13	0.17	0.19	0.30	3.1809	3.8109	19.81	3.6971	16.23
(11)	0.21	0.14	0.23	0.11	0.25	0.06	3.3511	3.7791	12.77	3.7078	10.64
(12)	0.13	0.15	0.22	0.14	0.17	0.19	3.6552	3.9501	8.07	3.9278	7.46
(13)	0.31	0.33	0.11	0.13	0.08	0.04	2.7677	3.4819	25.80	3.2790	18.47
(14)	0.10	0.21	0.13	0.12	0.18	0.26	3.4096	3.8592	13.19	3.7913	11.19
(15)	0.16	0.12	0.15	0.12	0.13	0.32	3.0523	3.8103	24.83	3.6652	20.08

Memory Bandwidths Derived From Exact Solution and Two Approximation Solutions for m=p=6

6. Repetitive Augmenting Approximation Method (RAAM)

The next approximation method is based on the idea of aggregation. A system of i memory modules can be viewed as the union of two subsystems as shown in figure 6.1. One subsystem consists of only one memory module (say the ith module) and the other consists of all the remaining memory modules. For convenience, we denote the subsystem with the single memory module i as subsystem (a) and the other consisting of modules 1,2, ...,i-1 as subsystem (b). Assume that we know how to derive the performance behavior of the aggregated system if the performance behavior of subsystem (b) is known (the detail will be shown later). Given a system of m memory modules and p processors, the algorithm for the Repetitive Augmenting Approximation Method proceeds by starting with a subsystem of 1 memory module (say module 1) and then repetitively augmenting the number of memory modules being considered in the sequence 2,3, ...,m.

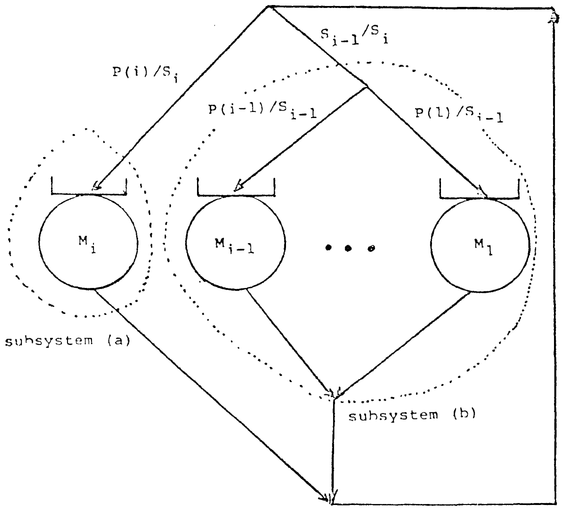


Figure 6.1 A System Is Decomposed To Two Subsystems.

In this paper, we shall assume that the behavior of a subsystem can be approximately represented by the probabilities PRO(q,n,s). PRO(q,n,s) denotes the probability for the subsystem of having s requests satisfied at one memory cycle while at the beginning of that memory cycle there were n new requests directed to it and q requests originally queued in it. Note that q+n is the total number of requests in the subsystem. In the following, we will first discuss how to initialize those PRO(q,n,s)'s for a subsystem of 1 memory module. Then show how to derive the performance behavior of a system with i memory modules if those PRO(q,n,s)'s for a subsystem of i-1 modules are known.

In a subsystem of one memory module, the probability of having one request satisfied (s=1) while at least one processor was in it (i.e., q+n ≥ 1) is 1. Thus, we initialize all PRO(q,n,s) with q+n ≥1 and s=1 to be 1 and all others to be 0.

A system of i memory modules and p processors, as shown in figure 6.1, can be viewed as the union of subsystems (a) and (b), where subsystem (a) consists of one memory module and subsystem (b) consists of i-1 memory modules. Let P(i) denote the probability for a request to access memory module i and $S_k = \sum_{j=1}^{k} P(j)$. Then for each request, the probability (P_a) of accessing subsystem (a) is P(i)/S_i and that (P_b) of subsystem (b) is S_{i-1}/S_i.

Assume that all PRO(q,n,s)'s for subsystem (b) are known, where q+n=j and 1≤s≤ min(i-1,j) (i-1 and j are the numbers of memory modules and processors in subsystem (b) and j can be an integer ranging from 1 to p- the total number of processors in the whole system). The set of possible states for the aggregated system is defined as the set of 3-tuples N=(n_0,n_a,n_b), where n_0 is the number of requests currently being serviced, and n_a and n_b are the numbers of requests queued in subsystems (a) and (b) respectively. For a possible state N=(n_0,n_a,n_b), $n_0+n_a+n_b$=j and $n_0 \leq i$ are necessary but not sufficient conditions. (For instance, N=$(1,n_a,n_b)$ with $n_a,n_b \geq 1$ and $1+n_a+n_b$=j is not a possible state but satisfies these two conditions.) Therefore, the number (NS) of possible states is bounded by C(j+2,2)= (j+2).(j+1)/2 (the number of ways to partition j elements into 3 sets).

Let Prob(N=(n_0,n_a,n_b)) be the probability of the aggregated system being in state N and let PRN(q,n,s) denote the probability of the aggregated system having s requests satisfied at one memory cycle while at the beginning of that memory cycle there are n new requests directed to it and q requests queued in it. Since the states and associated probabilities for subsystem (b) are assumed to be known (i.e., all the probabilities PRO(q,n,s) for $1 \leq s \leq i-1$ and $1 \leq q+n=j \leq p$), all probabilities Prob(N=(n_0,n_a,n_b)) for the aggregated system can be computed. Once all the probabilities Prob(N) for the aggregated system and all the probabilities PRO(q,n,s) for subsystem (b) are known, we can compute all the probabilities PRN(q,n,s) for the aggregated system. The PRN(q,n,s) then can be used for further aggregation. When all m modules have been considered the memory bandwidth B can be calculated by the equation :

$$B= \sum_N n_0 . Prob\,(N=(n_0,\, n_a, n_b).$$

In the following, we will show how to compute Prob(N)'s and PRN(q,n,s)'s. In the procedure DISTRIBUTION(i,j) we compute the probability of being in a "legal" state N =(n_0,n_a,n_b) for the current system of i memory modules and j processors. Assume that among the n_0 requests currently being serviced, d_a and d_b will be directed to subsystem (a) and (b), respectively, at the beginning of the next memory cycle ($d_a+d_b=n_0$). Let N'= $(0,n_a+d_a,n_b+d_b)$ denote the temporary state which describes the number of requests which will be in subsystems (a) and (b) for the next memory cycle. The probability of the subsystem transferring from state N to state N' (P($N \rightarrow N'$)) is $(n_0!/(d_a!d_b!))$. ${P_a}^{d_a}.{P_b}^{d_b}$ (recall that P_a and P_b are the probabilities for a request to reference subsystems (a) and (b) respectively). Assume that k requests (k must be less than or equal to the number of memory modules, i.e., i-1), among the n_b+d_b requests which stayed in subsystem (b), are satisfied. The next state will become the state $N_2=(n'_0,n'_a,\, n'_b)$, where $n'_0=k+1$, $n'_a=n_a+d_a-1$ and $n'_b=n_b+d_b-k$ if $n_a+d_a \geq 1$ or $n'_0=k$, $n'_a=0$ and $n'_b=n_b+d_b$-k if $n_a+d_a = 0$. The probability of transferring from state N' to state N_2 ,P($N' \rightarrow N_2$), is PRO($q=n_b, n=d_b$, s=k) as applied to subsystem (b) and $P(N_1 \rightarrow N_2)= \sum_{N'} P(N_1 \rightarrow N').P(N' \rightarrow N_2)$.

Once all the one-step transition probabilities between each pair of states are known, the probabilities for the system to be in state N (Prob(N)) for all possible 3-tuples N can be computed by the same procedure as the one for finding the exact solution (cf. section 3). Details of the procedure DISTRIBUTION(i,j) can be found in [5].

In the procedure FINDPRN(i,j) we compute the PRN(q,n,s)'s, i.e., the probabilities for the current system of i memory modules and j processors of having s requests satisfied while there were q requests enqueued and n new requests generated. After the PRN(q,n,s)'s have been obtained, we assign them to corresponding PRO(q,n,s)'s for future computations. Assume that the system is in state $N_1=(n_0,n_a,n_b)$ and there are d_a and d_b requests, among the n_0 currently being satisfied, being directed to subsystems (a) and (b) respectively. Let us also assume that there are k requests, among all requests in subsystem (b), being satisfied. Let $N'=(0,n_a+d_a,n_b+d_b)$ be the temporary state and $N_2=(n'_0,n'_a,n'_b)$, where n'_0=k+1, $n'_a=n_a+d_a$-1 and $n'_b=n_b+d_b$-k if $n_a+d_a \geq 1$ or $n'_0=k,n'_a=0$ and $n'_b=n_b+d_b$-k if $n_a+d_a=0$. Then according to the previous discussion $P(N_1 \rightarrow N_2)=\sum_{N'}(n_0!/(d_a!d_b!)$ $P_a^{d_a}.P_b^{d_b}.PRO(n_b,d_b,k)$. This also means that $P(N_1 \rightarrow N_2)$ is the probability for the system in state N_1 of having n'_0 requests satisfied while there were n_a+n_b requests enqueued and n_0 new requests generated. Let SP(q,n) be the sum of all Prob($N_1=(n_0,n_a,n_b)$), where q=n_a+n_b and n=n_0. Then it is not hard to see that PRN(q,n,s)= $\sum_{N_1,N_2} P(N_1=(n_0,n_a,n_b) \rightarrow$ $N_2=(n'_0,n'_a,n'_b)).Prob(N_1)/SP(q,n)$, where q=$n_a+n_b$,n=$n_0$ and s=n'_0. Since n_0 is always greater than 0 in every possible state N=(n_0,n_a,n_b), the way of deriving PRN(q,0,s) is somewhat awkward. It is basically assumed that all n_0 requests are directed to somewhere outside of the current system. Details can be found in [5].

In RAAM, we start with a subsystem of one memory module and then the number of memory modules being considered (i) will be incremented by one until i=m-1. In the system of i memory modules, the procedures DISTRIBUTION(i,j) and FINDPRN(i,j) are executed for j ranging from 1 to p because of their need in future computations.

After executing FINDPRN(m-1,p), all the probabilities PRO(q,n,s) for the system of m-1 modules are known. The probabilities Prob(N) for the whole system (with m memory modules and p processors) can be computed by DISTRIBUTION(m,p). Then the memory bandwidth B for the whole system is derived from those Prob(N).

It is obvious that the insertion sequence of the memory modules into the system will affect the derived result. One suggested criterion is to label the m memory modules in such a way that $P(1) \leq P(2) \leq \cdots \leq P(m)$, since the later a memory module is put into consideration, the more accurate and influential it will be. The memory module with the largest probability being referenced should therefore be "aggregated" last. This has been validated as shown in Table 6.1. In all cases where the probabilities were in ascending order the Repetitive Augmenting Approximation Method had a better result (closer to exact solution) than that of a descending or random order. Note that not only the average memory bandwidth for the whole system (with p processors and m memory modules) can be computed by RAAM, but also that of a sequence of subsystems.

With regard to the complexity of the algorithm, the following observations can be made :

1) In a system currently of i memory modules and j processors the number (NS) of possible states is less than C(j+2,2)=(j+2).(j+1)/2.

2) For a given state $N=(n_0,n_a,n_b)$, there are $n_0+1 \leq min(i,j)+1$ temporary states $N'=(0,n_a+d_a,n_b+d_b)$, where $d_a+d_b=n_0$. Each temporary state $N'=(0,n_a+d_a,n_b+d_b)$ can transfer to at most min(i,n_b+d_b+1)$\leq min(i,j)+1$ states in one step.

3) Once all the one step transition probabilities are known, Gauss elimination can be used to solve the NS equations. It takes O(NS^3) time to do so. Therefore, the time complexities for Procedures DISTRIBUTION(i,j) and FINDPRN(i,j) are no more than O(NS.$min(i,j)^2+NS^3$) and O($NS.min(i,j)^2$) respectively.

4) In addition to computing DISTRIBUTION(m,p), the complete algorithm necessitates computing DISTRIBUTION(i,j) and FINDPRN(i,j) for i=2 to m-1 and j=1 to p. It therefore takes only polynomial time to find the average memory bandwidth.

Tables 6.2 and 6.3 compare the exact solution with all three approximation solutions. RAAM is the approximation solution computed by the Repetitive Augmenting Approximation Method and RD_{RA} is the relative difference between EXACT and RAAM.

The following conclusions can be reached :

1) Among all three approximation methods, RAAM has the best result and AM2 is next.

2) In all cases, AM1 and AM2 are optimistic. However, RAAM is more conservative.

3) Contrary to AM1 and AM2, RAAM has its worst case when all reference probabilities are uniformly distributed.

	P(1)	P(2)	P(3)	P(4)	P(5)	P(6)	EXACT	RAAM	RD_{RA}
(R)	0.31	0.23	0.28	0.18			2.5416	2.3040	9.35
(D)	0.31	0.28	0.23	0.18			2.5416	2.2680	10.76
(A)	0.18	0.23	0.28	0.31			2.5416	2.4182	4.86
(R)	0.28	0.15	0.35	0.22			2.4498	2.3162	5.45
(D)	0.35	0.28	0.22	0.15			2.4498	2.3162	10.95
(A)	0.15	0.22	0.28	0.35			2.4498	2.3709	3.22
(R)	0.22	0.19	0.13	0.46			2.1154	2.0770	1.82
(D)	0.46	0.22	0.19	0.13			2.1154	1.9531	7.67
(A)	0.13	0.19	0.22	0.46			2.1154	2.0952	0.95
(R)	0.30	0.22	0.30	0.18			2.5345	2.3152	8.65
(D)	0.30	0.30	0.22	0.18			2.5345	2.2571	10.94
(A)	0.18	0.22	0.30	0.30			2.5345	2.4148	4.72
(R)	0.16	0.29	0.40	0.15			2.2965	2.1836	4.92
(D)	0.40	0.29	0.16	0.15			2.2965	2.0573	10.42
(A)	0.15	0.16	0.29	0.40			2.2965	2.2607	1.56
(R)	0.21	0.14	0.23	0.11	0.25	0.06	3.3511	2.8792	14.08
(D)	0.25	0.23	0.21	0.14	0.11	0.06	3.3511	2.6135	22.01
(A)	0.06	0.11	0.14	0.21	0.23	0.25	3.3511	3.2443	3.19
(R)	0.13	0.15	0.22	0.14	0.17	0.19	3.6552	3.1783	13.05
(D)	0.22	0.19	0.17	0.15	0.14	0.13	3.6552	2.8721	21.42
(A)	0.13	0.14	0.15	0.17	0.19	0.22	3.6552	3.3046	9.59
(R)	0.31	0.11	0.33	0.13	0.08	0.04	2.7677	2.4329	12.10
(D)	0.33	0.31	0.13	0.11	0.08	0.04	2.7677	2.2494	18.73
(A)	0.04	0.08	0.11	0.13	0.31	0.33	2.7677	2.7512	0.60
(R)	0.10	0.21	0.13	0.12	0.18	0.26	3.4096	3.0602	10.25
(D)	0.26	0.21	0.18	0.13	0.12	0.10	3.4096	2.6510	22.25
(A)	0.10	0.12	0.13	0.18	0.21	0.26	3.4096	3.2539	4.59
(R)	0.16	0.12	0.15	0.12	0.13	0.32	3.0523	2.9208	4.31
(D)	0.32	0.16	0.15	0.13	0.12	0.12	3.0523	2.5106	17.75
(A)	0.12	0.12	0.13	0.15	0.16	0.32	3.0523	2.9713	2.65

(R) : random order
(D) : descending order
(A) : ascending order

Table 6.1 Comparisons Between Ascending, Random and Descending Orders for m=p=4 and m=p=6.

Table 6.2

	P(1)	P(2)	P(3)	P(4)	EXACT	AM1	RD1	AM2	RD2	RAAM	RD_{RA}
(1)	0.04	0.04	0.04	0.88	1.1364	1.4518	27.75	1.24ØØ	9.12	1.1364	Ø.ØØ
(2)	0.06	0.11	0.16	0.67	1.4921	2.Ø821	39.54	1.8Ø45	2Ø.94	1.4919	Ø.Ø1
(3)	0.13	0.16	0.17	0.54	1.842Ø	2.4099	3Ø.83	2.2182	2Ø.42	1.8365	Ø.3Ø
(4)	0.12	0.18	0.24	0.46	2.1111	2.5295	19.82	2.4224	14.75	2.Ø939	Ø.81
(5)	0.06	0.20	0.33	0.41	2.2Ø83	2.487Ø	12.62	2.4282	9.96	2.19Ø8	Ø.79
(6)	0.18	0.23	0.28	0.31	2.5416	2.7ØØ9	6.27	2.6868	5.71	2.4182	4.86
(7)	0.15	0.22	0.28	0.35	2.4198	2.66Ø6	8.6Ø	2.6299	7.35	2.37Ø9	3.22
(8)	0.25	0.25	0.25	0.25	2.627Ø	2.7344	4.Ø9	2.7344	4.Ø9	2.4Ø14	8.59
(9)	0.13	0.19	0.22	0.46	2.1154	2.5415	2Ø.14	2.4327	15.ØØ	2.Ø952	Ø.95
(10)	0.18	0.22	0.30	0.3Ø	2.5345	2.6975	6.43	2.682Ø	5.82	2.4148	4.72
(11)	0.15	0.16	0.29	0.4Ø	2.2965	2.5764	13.Ø6	2.534Ø	1Ø.34	2.26Ø7	1.56
(12)	0.13	0.15	0.21	0.51	1.9398	2.4579	26.71	2.2994	18.54	1.9322	Ø.39
(13)	0.03	0.16	0.38	0.38	2.2Ø65	2.49Ø2	12.86	2.4247	9.89	2.1878	Ø.85
(14)	0.06	0.13	0.24	0.57	1.7457	2.2785	3Ø.52	2.Ø884	19.63	1.7447	Ø.Ø6
(15)	0.05	0.15	0.35	0.45	2.Ø7Ø5	2.3938	15.61	2.3118	11.65	2.Ø648	Ø.28

Memory Bandwidths Derived From Exact Solution and Three Approximation Solutions for m=p=4.

Table 6.3

	P(1)	P(2)	P(3)	P(4)	P(5)	P(6)	EXACT	AM1	RD1	AM2	RD2	RAAM	RD_{RA}
(1)	0.11	0.12	0.13	Ø.15	Ø.19	Ø.3Ø	3.1892	3.8278	2Ø.Ø2	3.7154	16.47	3.Ø862	3.23
(2)	Ø.02	0.04	0.10	Ø.18	Ø.19	Ø.47	2.126Ø	3.1914	5Ø.11	2.817Ø	32.5Ø	2.126Ø	Ø.ØØ
(3)	0.09	0.11	0.15	0.16	Ø.21	0.28	3.295Ø	3.8243	16.Ø6	3.735Ø	13.35	3.1865	3.29
(4)	0.07	0.13	Ø.14	0.19	0.21	Ø.26	3.3874	3.8251	12.92	3.7581	1Ø.94	3.2525	3.98
(5)	0.09	0.11	Ø.13	0.20	0.22	0.25	3.4Ø79	3.8362	12.57	3.7686	1Ø.58	3.2674	4.68
(6)	0.09	0.11	0.14	0.18	0.20	0.28	3.3ØØØ	3.8251	15.91	3.7373	13.25	3.1924	3.26
(7)	0.07	0.11	Ø.14	0.15	Ø.21	0.32	3.Ø2Ø7	3.7324	23.5Ø	3.5858	18.71	2.9752	1.51
(8)	0.16	0.16	0.17	Ø.17	0.17	Ø.17	3.7781	3.9896	5.6Ø	3.9891	5.58	3.19Ø3	15.56
(9)	0.10	0.13	Ø.14	0.17	0.19	0.27	3.3886	3.8697	14.2Ø	3.7937	12.1Ø	3.2246	4.84
(10)	0.10	0.11	0.13	0.17	Ø.19	0.3Ø	3.18Ø9	3.81Ø9	19.81	3.6971	16.23	3.Ø954	2.69
(11)	0.06	0.11	0.14	0.21	Ø.23	Ø.25	3.3511	3.7791	12.77	3.7Ø78	1Ø.64	3.2443	3.19
(12)	0.13	0.14	0.15	Ø.17	Ø.19	Ø.22	3.6552	3.95Ø1	8.Ø7	3.9278	7.46	3.3Ø46	9.59
(13)	0.04	Ø.08	0.11	0.13	Ø.31	Ø.33	2.7677	3.4819	25.8Ø	3.279Ø	18.47	2.7512	Ø.6Ø
(14)	0.10	0.12	0.13	0.18	Ø.21	Ø.26	3.4Ø96	3.8592	13.19	3.7913	11.19	3.2539	4.59
(15)	0.12	0.12	Ø.13	Ø.15	Ø.16	0.32	3.Ø523	3.81Ø3	24.83	3.6652	2Ø.Ø8	2.9713	2.65

Memory Bandwidths Derived From Exact Solution and Three Approximation Solutions for m=p=6

7. Conclusion

The system structure and program behavior are two major factors which influence the performance of interleaved memories. The latter has usually been ignored in most previous studies. A simple model for a multiprocessor system with p processors and m memory modules was presented. In this model, a set of non-uniformly distributed probabilities P(i) is employed to illustrate the program behavior, but no distinction is made between processors.

One exact solution and three approximation solutions were proposed in order to evaluate the performance of interleaved memory. Since it may take an unbearable time to compute the exact solution for medium to large m and p, there is a definite need to explore some approximation methods. Among all three approximation methods, RAAM gives the best result and AM2 is next. The approximation solutions computed by AM1 and AM2 are always optimistic. The results of RAAM, on the other hand, are always more conservative. An inverse relation between average memory bandwidth and average request completion time was also obtained. AM2 is derived from this inverse relation and the result of AM2 could be improved if a more precise way to calculate the average completion time could be found.

ACKNOWLEDGMENT

The authors are indebted to R.E. Ladner for several useful discussions at the early stage of this study. This work was supported in part by NSF Grants MCS76-09839 A02 and MCS- 8025616.

REFERENCES

[1] Baer, J.L., Du, H.C., and Ladner, R.E., "Binary Search in a Multiprocessing Environment," to appear in IEEE Trans. Computers.

[2] Baskett, F and Smith, A.J., "Interference in Multiprocessor Computer Systems with Interleaved Memory," Comm. ACM, vol. 19, no. 6, June 1976, pp. 327-334.

[3] Bhandarkar, D.P., "Analysis of Memory Interference in Multiprocessors," IEEE Trans. Computers, vol. C-24, no. 9, Sept. 1975, pp. 897-908.

[4] Chang, D.Y., Kuck, D.J., and Lawrie, D.H., "Effective Bandwidth of Parallel Memories," IEEE Trans. Computers, May 1977, pp. 480-490.

[5] Du, H.C. and Baer, J.L., "On the Performance of Interleaved Memories with Non-uniform Access Probabilities," Dept. of Computer Science, University of Minnesota, TR-82-6.

[6] Feller, W., *An Introduction to Probability Theory and Its Application*, vol. 1, Wiley, New York, 1968.

[7] Feng, T.Y., "A Survey of Interconnection Networks," IEEE Computer, vol. 14, no. 12, Dec. 1981, pp. 12-27.

[8] Kobayashi, H., *Modeling and Analysis : an Introduction to System Performance Evaluation Methodology*, IBM Corporation, 1978.

[9] Makrucki, B.A. and Mudge, T.N., "VLSI Design of a Crossbar Switch," SEL report no. 149, Dept. of Electrical and Computer Engineering, University of Michigan, Jan. 1981.

[10] Mudge, T.N. and Makrucki, B.A., "Probabilistic Analysis of a Crossbar Switch," Proc. of International Symposium on Computer Architecture, 1982, pp. 311-320.

[11] Patel, J.H., "Processor-Memory Interconnection for Multiprocessors," Proc. of International Symposium on Computer Architecture, 1979, pp. 166-177.

[12] Rau, B.R., "Program Behavior and the Performance of Interleaved Memories," IEEE Trans. Computers, vol. C-28, no. 3, March 1979, pp. 191-199.

[13] Rau, B.R., "Interleaved Memory Bandwidth in a Model of a Multiprocessor Computer System," IEEE Trans. Computers, vol. C-28, no. 9, Sept. 1979, pp. 678-681.

[14] Ravi, C.V., " On the Bandwidth and Interference in Interleaved Memory System," IEEE Trans. Computers, vol. C-21, Aug. 1972, pp. 899-901.

[15] Stehi, A.S. and Deo, N., "Interference in Multiprocessor Systems with Localized Memory Access Probabilities," IEEE Trans. Computers, vol. c-28, no. 2, Feb. 1979, pp. 157-163.

[16] Siegel, H.J. (Ed.), Proc. of the Workshop on Interconnection Networks, Purdue University, April 21-22, 1980.

[17] Strecker, W., "An Analysis of the Instruction Execution Rate in Certain Computer Structures," Ph.D. Diss., Carnegie-Mellon Univ., 1970.

[18] Du, H.C., "Some Design and Analysis Problems for Parallel Processing," Dept. of Computer Science, University of Washington, Tech. Report no. 81-08-03, 1981.

A MARKOVIAN QUEUEING NETWORK MODEL FOR PERFORMANCE EVALUATION OF BUS-DEFICIENT MULTIPROCESSOR SYSTEMS

IBRAHIM H. ONYUKSEL and KEKI B. IRANI

Computing Research Laboratory
The University of Michigan
Ann Arbor, Michigan 48109

ABSTRACT: A Markovian queueing network model is developed for the performance analysis of multiprocessor systems with multiple time-shared-bus interconnection networks. The effect of memory and bus contentions is investigated, and the comparative results from a unibus to a crossbar system are presented. The results show that decreasing the number of busses in a crossbar switch by factor of two produces a negligible degradation on the system performance in most cases. We have obtained exact results by devising an algorithmic method to convert the Markov chain of the queueing model to a simple birth-death process.

I. INTRODUCTION

Before starting to design and implement a real system, it is necessary for a designer to estimate the performance of a proposed system for given values of input parameters by applying analytic or simulation methods to the mathematical model of the system. Analytic models are very useful for a designer because they allow one to explore the effects of variations of system design parameters quickly and rather economically compared with simulation models.

Our study is concerned with devising an exact analytic model for the performance evaluation of a typical multiprocessor system (Fig.1), where each processor has its local memory unit and the allocation of common resources is controlled by the controller unit. The dynamic structure of the Interconnection Network(IN) enables the system to reconfigure the links between processors and the common memory. One way to realize this is to use multiple time- shared busses. When a processor requests access to the common memory, it signals the controller for a connection to the referenced module. Requests for connections are assumed to be independent from one processor to another, and more than one processor can request access simultaneously.

Since both the common memory and data paths are shared, contentions may arise, causing processors to queue for a resource which is currently in use. If every processor can be connected to a free memory module without blocking, the only cause of contention would be the common memory. If the referenced module is busy at the time of a request, then the controller puts the processor's request in a FCFS queue which is assigned to the referenced module. We call this type of systems **Bus-Sufficient**(BS) multiprocessor systems. The IN of BS system can be implemented by a full crossbar switch network. If IN is a blocking network, then a processor may have to wait for a free bus to access the common memory even if the referenced module is free at the time of the request. We call this type of systems **Bus-Deficient**(BD) multiprocessor systems.

We are concerned with determining the effects of the following two factors on degradation of the system performance:

1) Several processors may simultaneously request access to the same memory module, or a referenced module might be busy at the time of a request, so that some processors remain idle for several memory cycles. This is called **memory contention**.

2) If a blocking occurs in the IN, then some processors remain idle for several memory cycles until free busses become available for access to the common memory. This is called **bus contention**.

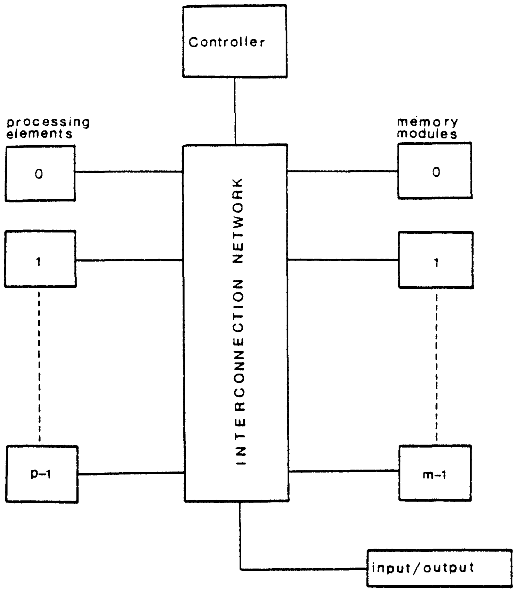


Figure 1
Basic block structure of a typical multiprocessor system

Skinner and Asher [6] were the first to use Markov Chain(MC) models to analyze multiprocessor systems. Unfortunately, their method can be applied only to small- scale systems. To analyze larger scale multiprocessor systems, some approximate and algorithmic methods have been proposed [1,2,7].

II. ASSUMPTIONS AND PERFORMANCE MEASURES

We have made the following assumptions for the mathematical model of multiprocessor systems with multiple time-shared busses.

1) When a processor requests access to the common memory, a connection is immediately established between the processor and the referenced module, provided that the referenced module is not being accessed by another processor and a bus is available for the connection.

2) A processor cannot have another memory request if its present request has not been granted yet.

0190-3918/83/0000/0437$01.00 © 1983 IEEE

3) The duration between the completion of a request and the issue of the next one to the common memory is an independent, exponentially distributed random variable with the same mean value, $1/\lambda$, for all processors.

4) The duration of an access by a processor to the common memory is an independent, exponentially distributed random variable with the same mean value, $1/\mu$, for all memory modules.

5) The request for access from a processor to the common memory is uniformly distributed for all memory modules with probability $1/m$.

If a queueing model satisfies the assumption (5), then it is called a **Uniform Reference Model**(URM). We use the traditional **Markovian queueing network theory** [4] approach for analyzing the multiprocessor systems with the assumptions stated above.

To overcome the computational complexity of the exact queueing model for the performance analysis of multiprocessor systems with multiple time-shared busses, several approximate models have been proposed by some researchers [3,5]. However, we have obtained exact results by devising an algorithmic method to convert the MC of the queueing model to a simple birth-death process, which is equivalent to the original MC.

The goal of the analysis of the queueing network model is to derive values of some performance measures of multiprocessor systems. The expected value of percentage of active processors is known as **Processing Efficiency**(PE) of a multiprocessor system. Let PE^1 and PE^2 be processing efficiencies of two different systems such that they have the same parameters as the original system except the former is a unibus system and the latter is a crossbar system. It is clear that $PE^1 \leq PE \leq PE^2$. In fact, PE^1 and PE^2 are the lower and upper bounds for the processing efficiencies of a family of multiprocessor systems such that the number of busses in the IN is the design parameter for this family. We denote the bus effect factor by ξ and define it as folow: $\xi = (PE^2 - PE)/PE^2$.

III. EVALUATION

The **configuration** of a multiple bus multiprocessor system is usually denoted by a 3-tuple (p×m×b), where p,m,b are the number of processors, number of memory modules, and the number of busses, respectively. If $b \geq \min(p,m)$ then the system is a BS system because a bus is available whenever a processor requests access to a free memory module. If $b < \min(p,m)$ then the system is a BD system because a processor may have to wait for a bus to access the referenced memory module even though the module may be free at the time of the request. If all the busses are occupied when a processor requests access to a free memory module, then the controller unit puts the processor in a wait state until a bus is available.

Each state Q of the continuous-time MC of the queueing model is represented by an m-tuple $Q=(k_1,\ldots,k_m)$, where $|k_j|$ indicates total number of processors queueing for memory module j. If $k_j>0$ then jth memory module is busy, but if $k_j<0$ then there is no available bus to access this module. If total number of active processors is p-n for a state, then we call it a level n state. Let $\bar{u}(x)$ be a binary variable such that $\bar{u}(x)=1$ for $x<0$ and $\bar{u}(x)=0$ for $x \geq 0$. If $\sum_{j=1}^{m} \bar{u}(k_j)=t$ for a state, then we call it a type-t state. A type-0 state is also called a BS state and a type-t state for $t \geq 1$ is called a BD state. If all states at level n of the MC are BS states, then we call least one BS state $Q^0=(n,0,\ldots,0)$.

To distinguish the probabilities of states at a given level of the MC, we attach weights to the states. The weight of a state Q is defined by

$$W[Q] = Pr(Q)/Pr(Q_0) \qquad (1),$$

where Q^0 is a BS state at the same level with Q. Let $\Phi(n)$ denote the set of states at level n of the MC and $L(n)=Pr\{Q|Q\in\Phi(n)\}$, then by definition of weight of a state we have shown that

$$L(n) = (\lambda_{n-1}/\mu)\,[\kappa(n)/\kappa(n-1)]\,L(n-1) \text{ or}$$

$$\bar{\mu}_n L(n) = \bar{\lambda}_{n-1} L(n-1) \text{ with} \qquad (2),$$

$$\bar{\lambda}_{n-1} = \lambda_{n-1}/\kappa(n-1) \text{ and } \bar{\mu}_n = \mu/\kappa(n) \text{ for } n=1,\ldots,p$$

where $\kappa(n) = \sum_{Q\in\Phi(n)} W[Q]$ is defined as the weight of level n and $\lambda_n=(p-n)/m$. The above equations suggest that we can replace the MC of the queueing model by a simple birth-death process with parameters $\bar{\lambda}_n$ and $\bar{\mu}_n$ for state n, which corresponds to level n of the original MC.

By analyzing the birth-death process and by definition of PE, we have obtained

$$PE = \left[\sum_{n=0}^{p-1} \kappa(n)(p-1)_n \left(\frac{\rho}{m}\right)^n\right]\left[\sum_{n=0}^{p} \kappa(n)(p)_n \left(\frac{\rho}{m}\right)^n\right]^{-1} \qquad (3),$$

where $\rho=\lambda/\mu$ (utilization factor for a single-processor single-memory system) and $(p)_n=p(p-1)\ldots(p-n+1)$ with $(p)_0=1$.

We like to reduce the size of the MC by partitioning the states into equivalence classes and generating a lumped MC for the system. If $Q=(k_1,..,k_m)$ is a state of the MC, then the states $C=\{(k_{j1},\ldots,k_{jm})|(j1,\ldots,jm)\in P_m\}$ form an equivalence class of Q, where P_m is the set of all permutations of integers $1,\ldots,m$. Let $\Psi(n)$ denote the set of equivalence classes at level n of the MC. Since the weights of states of an equivalence class all have the same value, we can define the weight of an equivalence class as

$$W[C] = N[C] \times W[Q] \text{ with } Q\in C \qquad (4),$$

where N[C] is the number of states in C. This equation implies that $\kappa(n) = \sum_{C\in\Psi(n)} W[C]$. To determine the weights of levels of the MC, we devise the following algorithm.

The Algorithm

```
1. Initialize n=0 and set the weight of level 0: $\kappa(0) \leftarrow 1$
2. $n \leftarrow n+1$
3. $\Psi(n) \leftarrow$ set of equivalence classes at level n; $\kappa(n) \leftarrow 0$
4. $C \leftarrow$ an element of $\Psi(n)$
5. $Q \leftarrow$ the representative state of C; /*$Q=(k_1,\ldots,k_m)$*/
   $N[C] \leftarrow$ the number of elements of C
6. $\Psi(n) \leftarrow \Psi(n) - \{C\}$
7. $\beta \leftarrow \sum_{j=1}^{m} (k_j \neq 0)$
8. if $\beta \leq b$ then $W[Q] \leftarrow 1$
9. else do
      a. $j \leftarrow 1$; $W[Q] \leftarrow 0$
      b. $c_j \leftarrow (|k_j|>1)$ OR $(k_j=\bar{1})$
      c. if $c_j=0$ then goto 9g
      d. $|\bar{k}_j| \leftarrow |k_j|-1$; $sign(\bar{k}_j) \leftarrow sign(k_j)$
      e. $Q^j \leftarrow (k_1,\ldots,\bar{k}_j,\ldots,k_m)$
      f. $W[Q] \leftarrow W[Q] + W[Q^j]$
      g. $j \leftarrow j+1$
```

```
    h. **if** $j \leq m$ **then goto** 9b
    i. $W[Q] \leftarrow W[Q]/b$; **end**
10. $W[C] \leftarrow N[C] \times W[Q]$
11. $\kappa(n) \leftarrow \kappa(n) + W[C]$
12. **if** $\Psi(n) \neq \phi$ **then goto** 4
13. **if** $n<p$ **then goto** 2; **end**
```

IV. NUMERICAL RESULTS

We have implemented our algorithm in PLC for the MTS(Michigan Terminal System) and the program was run for several p×m×b configurations. We have seen that execution time increases rapidly with p and by the factor of p-b for a fixed value of p. Therefore, we can say that the proposed algorithmic method is not very suitable for very large-scale multiprocessor systems.

We can compute the PE of a p×m×b multiprocessor system by applying equation (3). For p,m,b= 1,2,4,8,12,16 and $\rho=[0,1]$, PE vs ρ values are depicted in Fig.2: Fig.2a shows the effect of number of processors for m=8 and b=4. Fig.2b shows the effect of number of memory modules for p=8 and b=4. Fig.2c shows the effect of number of busses for p=m=16 on the system performance. The numerical values of ξ vs ρ are tabulated in Table 1 for 16×16×b family, where b=16 corresponds to a BS system. We see that decreasing the number of busses to b=8 makes a difference of only 0.7% in the PE(ρ=1). Thus, 16×16×16 system can be replaced by a 16×16×8 system without any significant degradation of the system performance.

ρ	b= 8	b= 4	b= 2	b= 1
0.1	-	0.08	4.35	32.22
0.2	-	1.37	25.20	61.40
0.3	-	5.43	42.77	71.33
0.4	0.02	11.68	52.26	76.13
0.5	0.06	18.16	57.78	78.89
0.6	0.13	23.65	61.33	80.66
0.7	0.24	27.96	63.78	81.89
0.8	0.37	31.28	65.55	82.78
0.9	0.53	33.86	66.89	.83.44
1.0	0.70	35.90	67.93	83.96

Table 1
Bus effect factors ξ(%) for 16×16×b family of multiprocessor systems

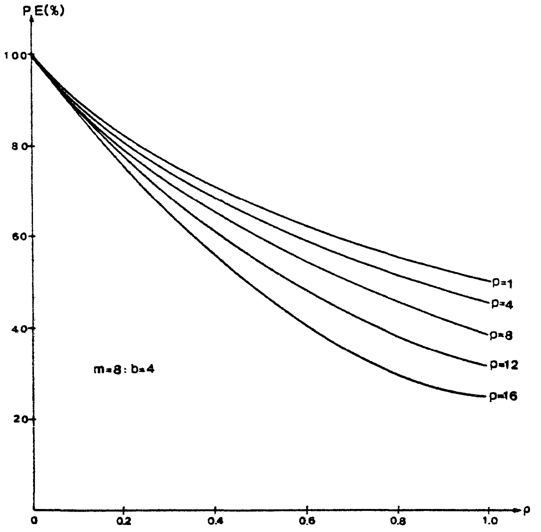
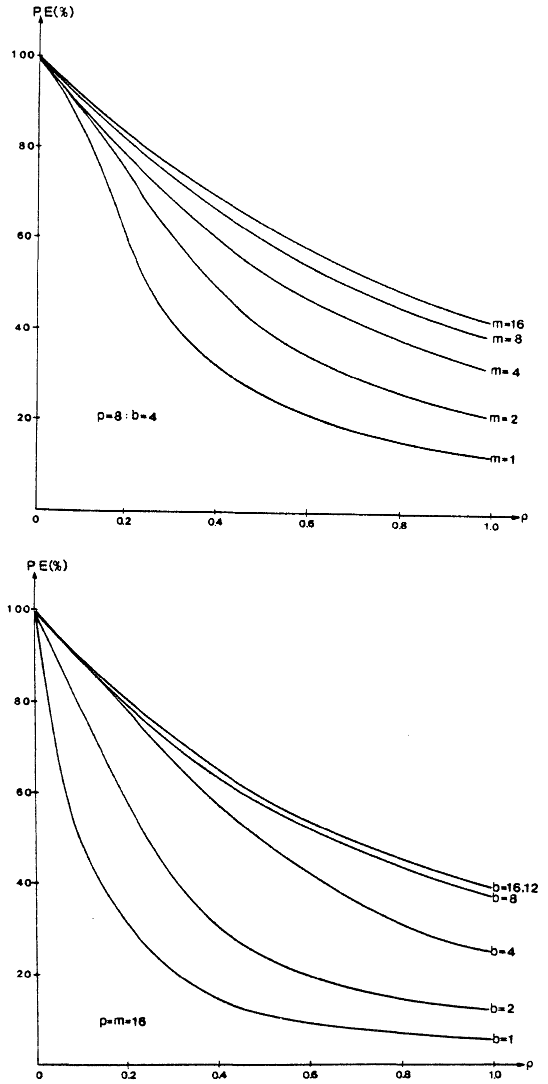


Figure 2a



Figures 2b and 2c

REFERENCES

[1] Baskett, F.S. and Smith, A.J., "Interference in Multiprocessor Computer Systems with Interleaved Memory", CACM, Vol.19, pp.327-334, Jun.1976.

[2] Bhandarkar, D.P., "Analysis of Memory Interference in Multiprocessors", IEEE T. Comp., Vol.C-24, pp.897-908, Sept. 1975.

[3] Jacobson, P.A. and Lazowska, E.D., "Analyzing Queueing Networks with Simultaneous Resource Possession", CACM, Vol.25, pp.142-151, Feb. 1982.

[4] Kleinrock, L., Queueing Systems I, John Wiley, 1975.

[5] Marsan, M.A. and Gerla, M., "Markov Models for Multiple Bus Multiprocessor Systems", IEEE T. Comp., Vol.C-31, pp.239-248, Mar. 1982.

[6] Skinner, C.E. and Asher, J.R., "Effects of Storage Contention on System Performance", IBM Syst. J., Vol.8, pp.319-333, 1969.

[7] Yen, D.W.L., Patel, J.H., and Davidson, E.S., "Memory Interference in Synchronous Multiprocessor Systems", IEEE T. Comp., Vol.C-31, pp.1116-21, Nov.82.

ON MAPPING HOMOGENEOUS GRAPHS ON A LINEAR ARRAY-PROCESSOR MODEL[a]

I.V. Ramakrishnan[b]
D.S. Fussell
A. Silberschatz

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

This paper presents a formal model of linear array processors suitable for VLSI implementation as well as graph representation of programs suitable for execution on such a model. A distinction is made between correct mapping and correct execution of such graphs on this model. A complete characterization of the structure of a class of correctly mappable graphs is obtained. The formalism developed is used to synthesize algorithms for this model.

1. Introduction

In [3, 6] specialized array processors were proposed as a means of handling compute-bound problems in a cost-effective and efficient manner. These array processors generally consist of a regular array of simple, identical processing elements which operate in synchrony. A host computer drives the array as a peripheral. The array can be of many forms, for instance a linear array, a rectangular mesh, a hexagonal mesh, etc. Simplicity and regularity of these array processors render them suitable for VLSI implementation. High performance is achieved by extensive use of pipelining and multiprocessing.

A variety of algorithms have been designed for such arrays [1, 4, 9]. An algorithm executing on such arrays is comprised of several data streams. A data stream is unidirectional, i.e., it does not change directions as it passes through processors in the array. Elements in distinct data streams move at different velocities (processors / cycle) while all elements in a given data stream move at the same velocity. Every processor in the array regularly receives data from each of the data streams, performs some short computation, and pumps the data out. The array communicates with the host through certain input/output ports designated as external input/output ports and elements in distinct data streams are pumped in through distinct external input/output ports. We will henceforth refer to such algorithms as "array algorithms".

A few methodologies have been proposed for synthesizing array algorithms from program specifications [2, 5, 12]. However in all these methodologies the synthesis problem was not studied in a formal framework. Also these methodologies shed insufficient insight into the synthesis problem for lack of a more intuitive representation of programs.

In this paper we study the synthesis of array algorithms in a more rigorous framework using a more intuitive representation of programs, namely, data-flow descriptions of programs. In particular we will be studying the synthesis of algorithms for a linear array. The array is comprised of identical processors, that is, they all execute the same set of instructions in every instruction cycle, and they are all simple, that is, they do not have any addressable local memory and cannot perform branching. The linear array is driven either by a single-phase or two-phase global clock [7]. In a two-phase clocking scheme the two phases are nonoverlapping and adjacent processors are activated by the opposite phases of the clock. Two reasons motivate our study of such a model. Firstly, this model has been used for most of the published array algorithms. Secondly, and more importantly, linear arrays require a fixed I/O bandwidth. Hence they can be attached as a peripheral to the I/O bus of any existing host without requiring any change to the host's I/O bandwidth.

We formalize this linear-array model and then define the program graphs that are appropriate for execution on them. A program graph is a directed acyclic graph representing a computation. The edges represent values and the nodes represent computation of a function whose arguments are the values represented by the incoming edges. We distinguish between correct mapping and correct execution of such program graphs on the linear array model. We provide a complete syntactic characterization for a class of program graphs (i.e., identify structural properties) that are correctly mappable and briefly mention the importance of using some semantic knowledge (i.e., some property) of the function represented by the nodes in the graph to correctly execute the graph.

This paper is organized as follows - in section 2 and section 3 we introduce the linear array and program graph models respectively. In section 4 we formalize the notion of correct execution of program graphs on linear arrays and in section 5 we examine the structural properties of correctly mappable program graphs. We illustrate the formalisms developed by synthesizing some linear-array algorithms. For brevity, the proofs of most of the Theorems in this paper have been omitted. They can however be found in [10].

2. Linear Array Model

In this section we define the linear array processor that formally captures the intuitive linear array described in the previous section. A linear array is a 3-tuple $Ar=<N,L_{Ar},\Psi_{Ar}>$ as follows.

1. N is a sequence of identical processors with indices ranging from 1 to $|N|$.
2. $L_{Ar}=\{l1, l2, .., lk\}$ is a set of labels.
3. Every processor in the array has k input ports and k output ports, with each input port and output port assigned a unique label lj from L_{Ar}. Each processor in N is connected to its neighbors in the sequence through its I/O ports. In addition the first and last processors may have input and output ports connected to the host environment.
4. The array is driven either by a single-phase or a two-phase global clock. A phase can be viewed as the instruction cycle of a processor. In a single-phase clocking scheme all processors are activated in every phase and every processor computes a k-ary function Ψ_{Ar}. In a two-phase clocking scheme adjacent processors are activated during opposite phases of the clock and every processor computes Ψ_{Ar} in the phase it is active.

[a] This research was supported in part by the National Science Foundation under Grants MCS-8104017 and ONR Contract N00014-80-k-0987

[b] Current Address: Department of Computer Science, University of Maryland, College Park, MD 20742

0190-3918/83/0000/0440$01.00 © 1983 IEEE

The function Ψ_{Ar} computed by a processor is a straight-line program. This restriction is imposed since we have assumed that a processor does not have any branching ability. We will henceforth refer to a processor in the array by its index in the sequence N. Let s be the index of a processor. Let $si_t=<si_t^1,si_t^2,..,si_t^k>$ denote the k-tuple input to processor s at time t where si_t^j is the value at the input port labelled lj of processor s at time t. Let $so_t=<so_t^1,so_t^2,..,so_t^k>$ denote the k-tuple output computed by processor s at time t, i.e., $\Psi_{Ar}(si_t)=so_t$.

For any label lj in L_{Ar}, let ρ_{lj} be the neighborhood relation imposed by label lj on processors in N. Let $<s,r>$ be any pair of processors in N.

Definition 2.1: We shall say that processor s is related to processor r by label lj denoted as s ρ r, iff the output port labelled lj of s is connected to the input port labelled lj of r.

We will refer to a path of uniform labels through the array as a data stream. The linear array has the following communication features.

1. A processor in the linear array can only communicate with up to two neighbors. All data streams are unidirectional. Hence for any label lj in L_{Ar}, if ρ_{lj} is not an empty relation, then a neighborhood constant n_{lj} is associated with lj such that the output port labelled lj of any processor s is connected to the input port labelled lj of $s+n_{lj}$ where n_{lj} is one of {1, -1, 0}.
2. The elements in a data stream move at a constant velocity, and hence a non-zero positive delay constant d_{lj} is associated with every label lj in L_{Ar} such that for any processor s, if so_t is the output computed by s at time t then so_t^j appears at the input port labelled lj of processor $s+n_{lj}$ at $t+d_{lj}$.
3. External communication takes place through certain designated input/output ports namely,
   a. if ρ_{lj} is empty then the input port and output port labelled lj of every processor communicate with the host,
   b. if $n_{lj}=1$ then the input port labelled lj of processor 1 and the output port labelled lj of processor $|N|$ communicate with the host,
   c. if $n_{lj}=-1$ then the input port labelled lj of processor $|N|$ and the output port labelled lj of processor 1 communicate with the host,
   d. if $n_{lj}=0$ then a register in every processor serves as the input/output port labelled lj. No input/output port labelled lj communicates with the host. A value is preloaded into this register before starting the computation and the result value (the preloaded value may be updated as computation progresses) is retrieved from this register after the computation terminates.

We will call the input/output ports that communicate with the host external input/output ports.

The delay d_{lj} can be implemented as a queue using a shift register of length d_{lj}-1 if single-phase clocking is used and of length $(d_{lj}-1)/2$ if two-phase clocking is used. At any time t, then, an activated processor s in the array performs the following sequence of operations:

1. Compute $\Psi_{Ar}(si_t)=so_t$ where $si_t=<si_t^1, si_t^2, ..,si_t^k>$ and $so_t=<so_t^1, so_t^2, ..,so_t^k>$.
2. For every label lj, dequeue the element at the head of the queue associated with lj and place it at the output port labelled lj of s.
3. For every label lj, place so_t^j at the tail of the queue.

Figure 2.1 illustrates a linear array with $n_{l1}=1$, $n_{l2}=-1$, $n_{l3}=0$. The neighborhood relation ρ_{l4} imposed by label $l4$ is empty. I_{l1}/O_{l1}, I_{l2}/O_{l2}, I_{l3}/O_{l3} and I_{l4}/O_{l4} are the external input/output ports associated with labels $l1$, $l2$, $l3$ and $l4$ respectively.

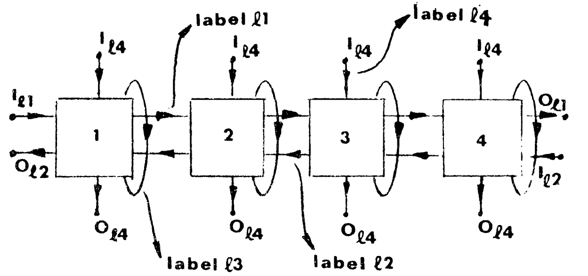


Figure 2·1

Henceforth, "linear array (arrays)" used in the rest of this paper will refer to the model defined above.

3. Homogeneous Graphs

The linear array is comprised of identical processors all of which compute the same function (or execute the same instruction) in every cycle. All the processors in the array cooperate in executing a single program. As all the processors in the array are identical, the straight-line programs they execute must also be identical. This motivates the following formalization of programs appropriate for execution on linear arrays.

A homogeneous program graph $G=<V,E,L_G>$ is a labelled DAG where:

1. $V=V_G \cup SO_G \cup SI_G$, and V_G, SO_G and SI_G are three disjoint sets of vertices with SO_G the set of source vertices, SI_G the set of sink vertices and V_G the set of remaining vertices, which we shall call computation vertices,
2. L_G is a set of labels. Let $|L_G|=k$, and
3. every vertex in V_G has k incident edges and k outgoing edges, where each incident and outgoing edge is assigned a unique label from L_G.

Input edges and output edges in G are those edges that are directed out of and into source and sink vertices respectively.

In any execution of G on a linear array, every computation vertex in G is a single instance of a function evaluation that is performed in a cycle by a processor in the array. Hence the function represented by v_x then, must be a straight-line program and we can view the k incoming edges and the k outgoing edges of a vertex v_x as representing the k-tuple input value and k-tuple output value computed by the processor that evaluates v_x. A source vertex then, represents an input value and a sink vertex represents an output value. As every computation vertex represents the same function, we refer to these program graphs as Homogeneous Graphs.

Figure 3.1 illustrates a homogeneous graph. The solid and dashed horizontal edges are labelled $l1$ and $l2$ respectively. The vertical and oblique edges are labelled $l3$ and $l4$ respectively.

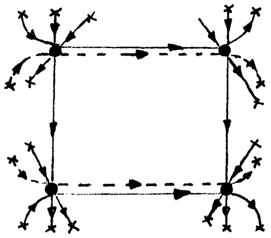


Figure 3·1

In Figure 3.1 and in all the other graphs illustrated in this paper we will be using '•' to represent computation vertices and 'x' to denote source and sink vertices.

Although homogeneous graphs are a more limited class of program graphs than, for instance, general dataflow graphs, it does allow the representation of quite a number of interesting programs which are potentially suitable for execution on the linear array model. As we shall see, not even all homogeneous graph programs can be executed on the simple computing engines we have defined.

Henceforth we will assume the following:

1. G is a homogeneous graph.
2. The label of a source (sink) vertex is the same as that of the input (output) edge directed out of the source (directed into the sink) vertex.
3. Input (output) value will always refer to the value represented by a source (sink) vertex.

4. Mapping Homogeneous Graphs

We now give a precise formulation of correct mapping and correct execution of homogeneous graphs on linear arrays. Intuitively mapping of G onto a linear array Ar assigns each computation vertex of G to a processor in Ar at a particular time step and also fixes the delay and neighborhood constant for every label in L_G. Assuming discrete time steps, let $T=\{0,1,2,..\}$ be the sequence of natural numbers representing the progress of computation from its start at time 0.

Definition 4.1: A mapping of G onto a linear array Ar is a 4-tuple <PA,TA,NA,DA> where:

1. $PA:V_G \rightarrow N$ and $TA:V_G \rightarrow T$ are many-one functions mapping computation vertices onto processors and time steps respectively.
2. Let I^+ be a set of positive non-zero integers. $NA:L_G \rightarrow \{1,-1,0\}$ and $DA:L_G \rightarrow I^+$ are many-one functions assigning neighborhood constants and delays to labels respectively.

[Note: $NA(lj)=n_{lj}$ and $DA(lj)=d_{lj}$]

We next formalize a correct mapping.

Definition 4.2: A mapping is syntactically correct iff

1. $\forall lj \in L_{Ar}$ and for any pair of computation vertices, v_x and v_y, if there is an edge labelled lj directed from v_x to v_y, then $PA(v_y)=PA(v_x)+n_{lj}$ and $TA(v_y)=TA(v_x)+d_{lj}$, and
2. no two input/output values can appear simultaneously at the same input port of a processor.

Let i be the input value represented by the source vertex of a computation vertex, say, v_x. Similarly, let o be the output value represented by the sink vertex of another computation vertex, say, v_y. Without loss of generality, let the labels of the source and sink vertices be lj. Now i is fed into the array and o is retrieved from the array through the external input port and external output port respectively associated with label lj. Let $TA(v_x)=t_1$ and $TA(v_y)=t_2$.

Definition 4.3: Entry Time for i and Exit Time for o is the time at which i is fed into and o is retrieved from the array respectively. Consumption Time of i and Production Time of o is t_1 and t_2+d_{lj} respectively.

We are now in a position introduce the notion of correct execution of homogeneous graphs.

Definition 4.4: G is correctly executed on a linear array iff

1. the mapping is syntactically correct, and
2. for every input value its value at entry and consumption times must be the same and for every output value its value at production and exit times must be the same.

Intuitively condition (2) means that we may be required to maintain a value input (outputted) to (by) the array constant as it passes through some number of processors inorder that it arrive unchanged at a processor (external output port) that will use it (from which it will be retrieved).

5. Syntactic Characterization

Our aim is to identify the structure of homogeneous graphs for which there exist syntactically correct mappings. We begin by identifying the relevant structural elements of a homogeneous graph G.

Definition 5.1: For any label lj in G, a major path labelled lj is a directed path from a source vertex v_x to a sink vertex v_y such that the label of v_x, v_y and all the edges in the path is lj.

The path label of a major path is the label of the edges in the path.

Definition 5.2: Two major paths are identical iff, ignoring the source and sink vertices in them, the two directed paths are the same.

For any label lj, let $E_{lj}=\{$major paths having the same path label $lj\}$. Not every E_{lj} is relevant for a syntactic characterization of homogeneous graphs. Consequently, we divide the labels of G into three classes:

1. $L_1=\{lj \mid$ there exists a pair of computation vertices v_x and v_y and a directed edge $e=<v_x,v_y>$ whose label is lj. Besides for any li and lj in L_1 there exists

a major path in E_{lj} that is not identical to any major path in E_{li}.} The major paths with these labels are relevant for structural characterization of correctly mappable graphs.

2. Let L_2={ lj | there exists a pair of computation vertices and a directed edge e=<v_x, v_y> whose label is lj. Besides, if lj is in L_2 then there exists an li in L_1 such that for every major path in E_{lj} there is an identical major path in E_{li}.} Given the major paths associated with the labels in L_1, the major paths associated with those in this class are redundant for structural characterization.
3. L_3={lj | there exists no pair of computation vertices v_x and v_y such that there is a directed edge e=<v_x,v_y> whose label is lj }.

Consider the homogeneous graph in Figure 3.1 again. The solid and dashed horizontal edges are labelled $l1$ and $l2$ respectively. The vertical and oblique edges are labelled $l3$ and $l4$ respectively. L_1={$l1$, $l3$}, L_2={$l2$} and L_3={$l4$}.

Henceforth, throughout the rest of this paper, labels will be assumed to be in L_1 unless explicitly mentioned otherwise.

We are are now in a position to define the class Θ of program graphs that we will be examining in this paper. If there exists a connected subgraph SG in G whose label set $L_{SG}=\{l\mu,l\nu\}\subseteq L_1$ and whose vertex set V_{SG} contains V_G, i.e., $V_G \subseteq V_{SG}$, then G is in Θ. Existence of SG signifies that there is an undirected path between any pair of computation vertices in G through edges that are labelled either $l\mu$ or $l\nu$. Θ is a large class that includes homogeneous program graphs for important computational problems like sorting, convolution, vector multiplication of band matrices, pattern matching, priority queue, etc. $l\mu$ and $l\nu$ will refer to the two labels of SG.

The structure imposed on SG by any correct mapping is elegantly formalized below.

Definition 5.3: Let I_1 and I_2 be two sequences of integers such that the sequences in I_1 and I_2 range from 0 to h_1 and 0 to h_2 respectively and let $B \subseteq I \times I$. Then, SG is a Mesh Graph iff there exists a one-one function F:V_G-->B such that the following property holds. Let $F_{l\mu}$ and $F_{l\nu}$ be the projection functions of F, i.e., for any v_x in V_G, if F(v_x)=<m,n> then $F_{l\mu}(v_x)$=m and $F_{l\nu}(v_x)$=n. For any v_x and v_y in V_G, there exists a directed path from v_x to v_y in a major path whose path label is $l\mu$ such that the distance from v_x to v_y in this directed path is d iff $F_{l\mu}(v_y)=F_{l\mu}(v_x)+d$ and $F_{l\nu}(v_y)=F_{l\nu}(v_x)$. A similar condition holds for a major path whose path label is $l\nu$.

Henceforth we will denote $F_{l\mu}(v_x)$ and $F_{l\nu}(v_x)$ as $x_{l\mu}$ and $x_{l\nu}$ respectively.

Figure 5.1 is an example of a Mesh Graph wherein the horizontal and vertical major paths are labelled $l\mu$ and $l\nu$ respectively.

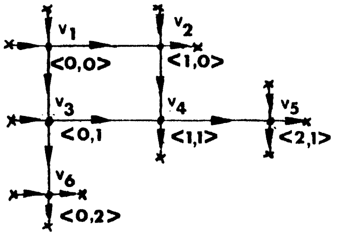


Figure 5·1

We next relate the structure of SG to the existence of a syntactically correct mapping in the following Theorem.

Theorem 5.1: If there exists a syntactically correct mapping for G then SG must be a Mesh Graph.

When G is finally mapped onto a linear array the computation vertices in G may be partitioned into sets that comprise vertices which are mapped onto the same physical processor. As we will see later on this is useful in expressing the structure of correctly mappable graphs in a simple way. To formalize this partitioning it is useful to define a Diagonalization of the Mesh Graph SG as follows.

Definition 5.4: Let w=<w_1,w_2>∈{<1,1>, <1,-1>, <1,0>, <0,1>}. A Diagonalization of SG is a pair <D,w> with the following properties.

1. D={D_1,D_2, ..,D_k} is a family of ordered sets of computation vertices and $D_1 \cup D_2 \cup .. \cup D_k = V_G$.
2. For any D_p in D, if v_x and v_y are in D_p then $w_1x_{l\mu}+w_2x_{l\nu}=w_1y_{l\mu}+w_2y_{l\nu}$.
3. Let Υ_D denote the indexing function associated with the ordered set D. For any pair of D_p and D_q in D, if v_x and v_y are in D_p and D_q respectively then $\Upsilon_D(D_p) < \Upsilon_D(D_q)$ iff $w_1x_{l\mu}+w_2x_{l\nu} < w_1y_{l\mu}+w_2y_{l\nu}$.

Henceforth, we will refer to D as the set of Main Diagonals and to w as the Main Diagonalization Factor. We will assume that the indices assigned to the diagonals in D range from 1 to |D| and if D_p is a diagonal in D then $\Upsilon_D(D_p)$=p, i.e., the index of D_p in the ordering is p. We use the ordering of the diagonals in D to define an adjacency relation imposed on them by labelled edges.

Definition 5.5: Let D_p and D_q be in D. D_p a_{lj} D_q (read 'D_p is related by a_{lj} to D_q') iff there exists a computation vertex v_x in D_p and another computation vertex v_y in D_q and a directed edge e=<v_x,v_y> whose label is lj.

Definition 5.6: a_{lj} is consistent with respect to Υ_D iff ∃ a constant m_{lj} such that $\forall D_p \in D$ and $\forall D_q \in D$, if D_p a_{lj} D_q then $\Upsilon_D(D_q)=\Upsilon_D(D_p)+m_{lj}$.

We will call m_{lj} the consistency constant of a_{lj}. Let S_D={a_{lj} | $lj \in L_1$ and a_{lj} is the adjacency relation on D imposed by edges labelled lj }.

It is useful to define the set Dc of Complementary Diagonals that is obtained by diagonalizing SG by its Complementary Diagonalization Factor w_c where $w_c \neq w$ and w_c∈{<1,1>,<1,-1>,<1,0>,<0,1>}.

Let Υ_{Dc} denote the indexing function associated with Dc and S_{Dc}={b_{lj} | $lj \in L_1$ and b_{lj} is the adjacency relation on Dc imposed by edges labelled lj }. Herein also we will assume that the index of the complementary diagonals in Dc ranges from 1 to |Dc| and if Dc_p is a complementary diagonal in Dc then its index is p. Consistency of b_{lj} with respect to Υ_{Dc} is defined similar to a_{lj}. Let c_{lj} denote the consistency constant of b_{lj}.

Consider Figure 5.1 again. Let w=<1,-1> and w_c=<0,1>. Then the set of main digonals D={D_1, D_2, D_3, D_4} is comprised of four diagonals where D_1={v_6}, D_2={v_3}, D_3={v_1, v_4} and D_4={v_2, v_5}. The set of complementary diagonals Dc={Dc_1, Dc_2, Dc_3} is comprised of three diagonals where Dc_1={v_1, v_2}, Dc_2={v_3, v_4, v_5} and Dc_3={v_6}.

Let v_x and v_y be two vertices in the main diagonals D_p and D_q respectively and complementary diagonals Dc_s and Dc_r respectively. Then we will denote the difference in indices of D_q and D_p which is q-p as $\Delta_D(v_x,v_y)$. We will also denote the difference in indices of Dc_r and Dc_s which is r-s as $\Delta_{Dc}(v_x,v_y)$.

We next define two classes of graphs $\Theta_1 \subset \Theta$ and $\Theta_2 \subset \Theta$ where:

$\Theta_1 = \{G \in \Theta \mid$ SG is a mesh graph and the main diagonalization factor w of SG is one of $\{<1,-1>, <0,1>, <1,0>\}\}$ and

$\Theta_2 = \{G \in \Theta \mid$ SG is a mesh graph and the main diagonalization factor w of SG is $<1,1>\}$.

We provide a complete syntactic characterization of program graphs in Θ_1 which have syntactically correct mappings in the following Theorem. Before doing so we introduce the notion of transitive edges which is needed in the proof sketch of the Theorem.

Definition 5.7: Let $e=<v_x,v_y>$ be a directed edge from vertex v_x to vertex v_y. Then e is a transitive edge iff there exists a vertex v_z and edges $e_m=<v_x,v_z>$ and $e_n=<v_z,v_y>$.

Theorem 5.2: Let $G \in \Theta_1$. There exists a syntactically correct mapping for G if and only if there exists a pair <D,Dc> such that each of the following conditions is satisfied:

1. Every relation $a_{\ell j} \in S_D$ is consistent with respect to Υ_D and its consistency constant $m_{\ell j}$ is one of $\{1,-1,0\}$.
2. Every relation $b_{\ell j} \in S_{Dc}$ is consistent with respect to Υ_{Dc}.
3. Let v_x and v_y be any two computation vertices. For any label ℓj if $c_{\ell j}\Delta_D(v_x,v_y)=m_{\ell j}\Delta_{Dc}(v_x,v_y)$ then there must be a major path labelled ℓj passing through v_x and v_y.

Intuitively, condition (1) ensures that a data stream is unidirectional and communication takes place only between adjacent processors while condition (2) ensures that a data stream moves at constant velocity and condition (3) ensures that no two values appear simultaneously at the input port of any processor.

We sketch only the sufficiency proof. The proof is constructive and we will be using this constructive procedure to illustrate synthesis of linear-array algorithms later on.

Proof: (Only If): See [10] for details.

(If Part): Let $D=\{D_1, D_2, .., D_n\}$ be the set of main diagonals where i denotes the index of any $D_i \in D$. Construct a linear array L_A with $|N|=n$. Now construct a mapping through the following steps.

1. Choose two-phase clocking if there exists a transitive edge labelled ℓj such that $m_{\ell j}=0$ or else choose a single-phase clocking scheme.
2. Let D_q be any diagonal in D and let v_x be any computation vertex in D_q. Then, let $PA(v_x)=q$. This assigns computation vertices to processors.
3. Next fix the neighborhood constant $n_{\ell j}$ and delay constant $d_{\ell j}$ for every label ℓj in L_1. Let $n_{\ell j}=m_{\ell j}$. Let d_a and d_b be two constants which we will be using in the construction of the delays for the labels in L_1. If the main diagonalization factor w is <1,-1> or there exists a transitive edge labelled ℓj such that $m_{\ell j}=0$ then let $d_a=2$ else let $d_a=1$. Let c_{min} be the minimum of all consistency constants among all the relations in S_{Dc}. If $c_{min}>0$ then set $d_b=1$ else set $d_b=1+|c_{min}|d_a$. Let $d_{\ell j}=m_{\ell j}d_b+c_{\ell j}d_a$.
4. Next construct the neighborhood and delay constant for the labels in L_2. By definition of L_2, if there exists a label ℓj in L_2 then there must exist some label ℓi in L_1 such that for every major path in $E_{\ell j}$ there is an identical major path in $E_{\ell i}$. Hence let $n_{\ell j}=n_{\ell i}$ and $d_{\ell j}=d_{\ell i}$.
5. For every ℓj in L_3, let the neighborhood relation imposed by label ℓj on processors in N be empty and hence no processor's output port labelled ℓj is connected to the input port labelled ℓj of any processor.
6. Construct the function TA which assigns computation vertices to time steps. Let v_s be the computation vertex which is in $D_1 \in D$ and $Dc_1 \in Dc$. Let $TA(v_s)=t_0$. Let v_x be any computation vertex in $D_p \in D$ and $Dc_q \in Dc$. Then, let $TA(v_x)=t_0+(q-1)d_a+(p-1)d_b$.

Step 1 to step 6 described above completes the construction of a correct mapping. Refer [10] to verify that the mapping is correct.

□

The three conditions of Theorem 5.2 are necessary but not sufficient for the existence of syntactically correct mappings for graphs in Θ_2. However in the next corollary we show that in certain cases it is both necessary and sufficient. Let $G \in \Theta_2$ and let $C=\{c_{\ell j}\}-\{c_{\ell\mu}, c_{\ell\nu}\}$.

Corollary 5.1: $\forall c_{\ell j} \in C$, if $c_{\ell j}>0$ or $\forall c_{\ell j} \in C$, if $c_{\ell j}<0$ then there exists a syntactically correct mapping for G if and only if the three conditions in Theorem 5.2 are satisfied.

Proof: Similar to Theorem 5.2 except in the construction of the expressions for the delays. If $c_{\ell j}>0$ then set $d_a=2$, $d_b=1$, $d_{\ell\mu}=1$ and $d_{\ell\nu}=3$. If $c_{\ell j}<0$ then set $d_a=-2$, $d_b=3$, $d_{\ell\mu}=3$ and $d_{\ell\nu}=1$. In [10] it is shown that this construction yields $d_{\ell j}>0$.

□

The sufficiency proof of Theorem 5.2 provides a methodology to synthesize linear-array algorithms for graphs in Θ. The construction used in the Theorem maps a program graph correctly. However, very often, to ensure its correct execution we need to use some property of the function represented by the computation vertices in the graph. The structure of graphs that can be executed without using such knowledge is characterized in [10].

We now apply the results described in this paper to synthesize linear-array algorithms for computing the vector multiplication of band matrices and convolution.

Example 5.1: Consider multiplication of a Band Matrix A by a Vector X as shown in Figure 5.2. Y is the result vector. The computation of this multiplication can be represented by the program graph in Figure 5.3 wherein v_{ij} denotes a computation vertex. The horizontal, vertical and oblique edges are labelled $\ell 1$, $\ell 2$ and $\ell 3$ respectively. Let Ψ denote the function represented by any computation vertex in the graph. Ψ is a 3-ary function such that for any a, b and c, Ψ<a,b,c>=<a+bc,b,c>. Let Ψ_1, Ψ_2, Ψ_3 be the three projections of Ψ, i.e., Ψ_1<a,b,c>=a+bc, Ψ_2<a,b,c>=b and Ψ_3<a,b,c>=c. If a, b and c are the input values represented by the horizontal, vertical and oblique input edges of v_{ij} then the output values represented by the outgoing horozontal, vertical and oblique edges of v_{ij} are Ψ_1<a,b,c>, Ψ_2<a,b,c> and Ψ_3<a,b,c> respectively. The input value represented by every horizontal source vertex is initialized to 0.

Figure 5.2

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \\ & & & a_{64} & a_{65} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

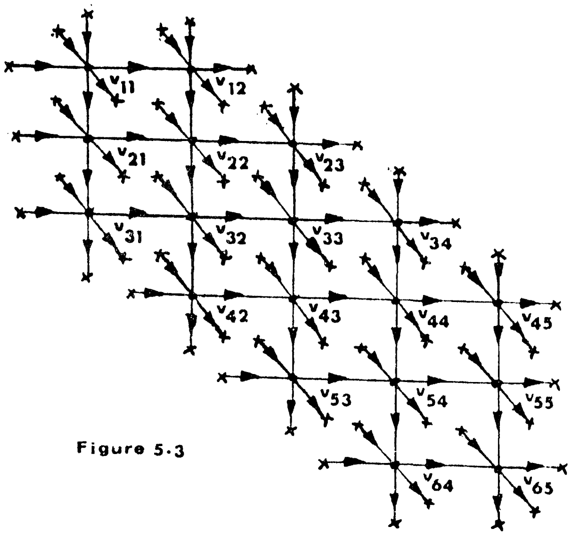


Figure 5.3

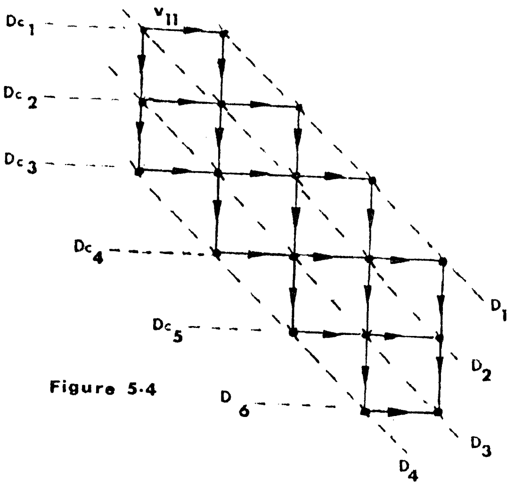


Figure 5.4

Let E_{l1}={horizontal major paths}, E_{l2}={vertical major paths} and E_{l3}={oblique major paths}. It can be seen that $L_1=\{l1,l2\}$, $L_2=\{\phi\}$ and $L_3=\{l3\}$.

Let SG be the connected subgraph shown in Figure 5.4 that is obtained by removing all the edges that are labelled $l3$. For porposes of clarity SG has been drawn without the source and sink vertices. It can be easily verified that the program graph in Figure 5.3 is in Θ. Now diagonalize SG with w=<1,-1> to form the set of main diagonals D. It can be verified that $D=\{D_1,D_2,D_3,D_4\}$ is comprised of four diagonals where $D_1=\{v_{31},v_{42},v_{53},v_{64}\}$, $D_2=\{v_{21},v_{32},v_{43},v_{54},v_{65}\}$, $D_3=\{v_{11},v_{22},v_{33},v_{44},v_{55}\}$ and $D_4=\{v_{12},v_{23},v_{34},v_{45}\}$.

Next diagonalize SG with w_c=<0,1> to form the set Dc of complementary diagonals. It can be verified that $Dc=\{Dc_1,Dc_2,Dc_3,Dc_4,Dc_5,Dc_6\}$ is comprised of six diagonals where $Dc_1=\{v_{11},v_{12}\}$, $Dc_2=\{v_{21},v_{22},v_{23}\}$, $Dc_3=\{v_{31},v_{32},v_{33},v_{34}\}$, $Dc_4=\{v_{42},v_{43},v_{44},v_{45}\}$, $Dc_5=\{v_{53},v_{54},v_{55}\}$ and $Dc_6=\{v_{64},v_{65}\}$.

In Figure 5.4 all the computation vertices belonging to the same diagonal in D lie on the same dashed line. Similarly all the computation vertices belonging to the same diagonal in Dc lie on one horizontal major path.

Now $S_D=\{a_{l1},a_{l2}\}$, $S_{Dc}=\{b_{l1},b_{l2}\}$ and $m_{l1}=1$, $m_{l2}=-1$, $c_{l1}=0$ and $c_{l2}=1$. It can be seen that this graph satisfies Theorem 5.2.

Next, using the construction in Theorem 5.2 we synthesize the linear-array algorithm in [4]. |D|=4 and hence the linear array has 4 processors indexed from 1 to 4. $m_{l1}\neq0$ and $m_{l2}\neq0$ and hence use single-phase clocking. Each processor is comprised of 3 pairs of input/output ports labelled $l1$, $l2$ and $l3$ respectively. The neighborhood relation ρ_{l3} is empty.

Let si_t^1, si_t^2 and si_t^3 denote the inputs at the input ports labelled $l1$, $l2$ and $l3$ respectively of processor s at time t and let so_t^1, so_t^2 and so_t^3 denote the outputs computed by s at time t. Then $so_t^1=si_t^1+si_t^2si_t^3$, $so_t^2=si_t^2$ and $so_t^3=si_t^3$.

The computation vertices in D_1,D_2,D_3 and D_4 are mapped onto processors 1,2,3 and 4 respectively. From the construction of Theorem 5.2, we obtain $n_{l1}=1$,$n_{l2}=-1$,$d_{l1}=1$ and $d_{l2}=1$. The resulting mapped graph is shown in Figure 5.5. The time at which a computation vertex is mapped is indicated by the side of the vertex in Figure 5.5. For instance, the computation vertex on D_3 and Dc_2 is mapped at time t+2. For correctness of execution we must ensure the invariance of the two input values ih_1 and ih_2 at their consumption and entry times and the invariance of the two output values oh_5 and oh_6 at their exit and production times. The consumption times for ih_1 and ih_2 are t and t+1 respectively. Table 5.1 gives the times at which ih_1 appears at the input port labelled $l1$ of processors 1 and 2 and ih_2 appears at the input port labelled $l1$ of processor 1. Any element pumped into I_{l1} or I_{l2} travels at the rate of 1 processor/cycle as $1/d_{l1}=1/d_{l2}=1$. Consider some row of Table 5.1, say 2. The entry in column 1 indicates that ih_2 appears at the input port labelled $l1$ of processor 1 at time t. Now Ψ_1 is such that for any b, Ψ_1<a,b,0>=a+b0=a and hence by pumping 0 into the input port labelled $l3$ of processor 1 at t invariance of ih_2 at its entry and consumption time can be maintained. Similarly by pumping 0 into the input ports labelled $l3$ of processor 1 at t-2 and processor 2 at t-1 invariance of ih_1 at its entry and consumption times can be maintained.

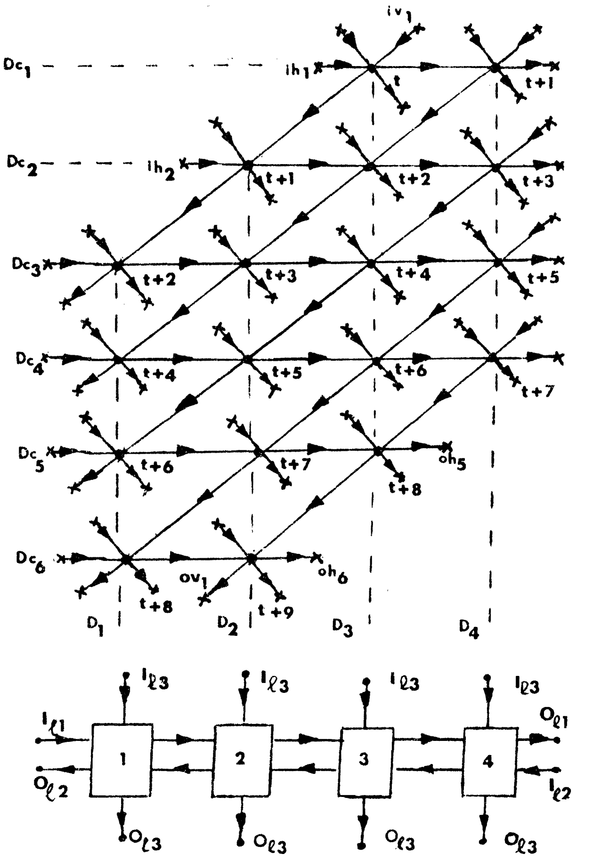


Figure 5.5

The production times for oh_5 and oh_6 are t+9 and t+10 respectively. Table 5.2 gives the times at which oh_5 appears at the input port labelled $l1$ of processor 4 and oh_4 appears at the input ports labelled $l1$ of processors 3 and 4. The entries in Table 5.2 are interpreted in the same way as the entries in Table 5.1. From Table 5.2 it is seen that by pumping 0 into the input port labelled $l3$ of processor 3 at t+10 and processor 4 at t+9 and t+11 invariance of oh_5 and oh_6 at their production and exit times can be maintained.

Lastly, as $\Psi_2<a,b,c>=b$ for any a and any c, the input value iv_1 and output value ov_1 do not change as they travel through processors in the array.

Table 5.1

	1	2	3	4
ih_1	t -2	t -1		
ih_2	t			

Table 5.2

	1	2	3	4
oh_5				t+9
oh_6			t+10	t+11

Example 5.2: Consider the convolution problem defined as follows:

Given the sequence of weights $\{w_1, w_2, .., w_k\}$ and the input sequence $\{x_1, x_2, .., x_n\}$ compute the output sequence $\{y_1, y_2, .., y_{n+1-k}\}$ defined by $y_i = \sum_{j=1}^{k} w_j x_{i+j-1}$.

We illustrate the convolution problem on n=5 and k=3. The computation of the convolution problem for n=5 and k=3 is represented by the following program graph:

In Figure 5.6, $\forall i$ and $\forall j$ $|1 \le i,j \le 3$, v_{ij} represents a computation vertex. The horizontal, vertical and oblique edges are labelled $l1$, $l2$ and $l3$ respectively.

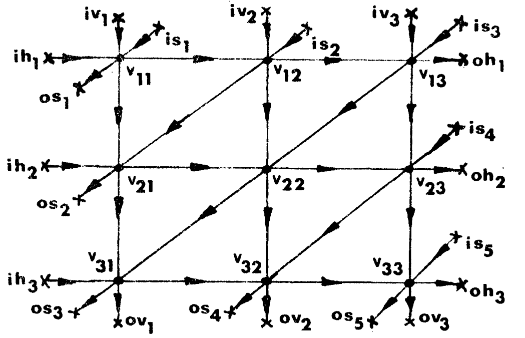


Figure 5.6

Let Ψ denote the function represented by any computation vertex in Figure 5.6. Ψ is a 3-ary function such that for any a, b and c, $\Psi<a,b,c>=<a+bc,b,c>$. Let Ψ_1, Ψ_2, Ψ_3 be the three projections of Ψ, i.e., $\Psi_1<a,b,c>=a+bc$, $\Psi_2<a,b,c>=b$ and $\Psi_3<a,b,c>=c$. If a, b and c are the input values represented by the horizontal, vertical and oblique input edges of v_{ij} then the output values represented by the outgoing horozontal, vertical and oblique edges of v_{ij} are $\Psi_1<a,b,c>$, $\Psi_2<a,b,c>$ and $\Psi_3<a,b,c>$ respectively. $\forall p \mid 1 \le p \le 5$, $\forall q \mid 1 \le q \le 3$ and $\forall r \mid 1 \le r \le 3$, let the input values represented by is_p, iv_q and ih_r be x_p, w_q and 0 respectively. It can then be verified that the output values represented by oh_r is $\sum_{q=1}^{3} w_q x_{r+q-1}$.

Let E_{l1}={horizontal major paths}, E_{l2}={vertical major paths} and E_{l3}={oblique major paths}. It can be seen that $L_1=\{l1,l2,l3\}$, $L_2=\{\phi\}$ and $L_3=\{\phi\}$.

Let SG be the connected subgraph shown in Figure 5.7 that is obtained by removing all the edges that are labelled $l3$. It can be seen that the program graph in Figure 5.6 is in Θ.

Now diagonalize SG with $w=<1,0>$ to form the set D of main diagonals. It can be verified that $D=\{D_1,D_2,D_3\}$ is comprised of three diagonals where $D_1=\{v_{11},v_{21},v_{31}\}$, $D_2=\{v_{12},v_{22},v_{32}\}$ and $D_3=\{v_{13},v_{23},v_{33}\}$.

Next diagonalize SG with $w_c=<0,1>$ to form the set Dc of complementary diagonals. It can be verified that $Dc=\{Dc_1,Dc_2,Dc_3\}$ is also comprised of three diagonals where $Dc_1=\{v_{11},v_{12},v_{13}\}$, $Dc_2=\{v_{21},v_{22},v_{23}\}$, $Dc_3=\{v_{31},v_{32},v_{33}\}$.

In Figure 5.7 all the computation vertices belonging to a single diagonal in D lie on the same vertical major path. Similarly, all the vertices belonging to a single diagonal in Dc lie on the same horizontal major path.

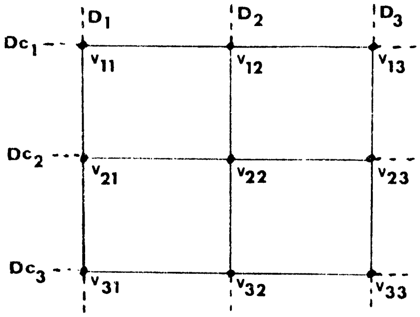


Figure 5.7

Now $S_D=\{a_{l1},a_{l2},a_{l3}\}$, $S_{Dc}=\{b_{l1},b_{l2},b_{l3}\}$ and $m_{l1}=1$, $m_{l2}=0$, $m_{l3}=-1$, $c_{l1}=0$, $c_{l2}=1$ and $c_{l3}=1$. It can be verified that Theorem 5.2 is satisfied.

We next synthesize the linear-array algorithm in [6]. $|D|=3$ and hence the linear array has 3 processors indexed from 1 to 3. $m_{l2}=0$ and there exist transitive edges labelled $l2$. Hence use two-phase clocking. Each processor is comprised of 3 pairs of input/output ports labelled $l1$,$l2$ and $l3$ respectively.

Let si_t^1, si_t^2 and si_t^3 denote the inputs at the input ports labelled $l1$, $l2$ and $l3$ respectively of processor s at time t and let so_t^1, so_t^2 and so_t^3 denote the outputs computed by s at time t. Then, $so_t^1=si_t^1+si_t^2 \times si_t^3$, $so_t^2=si_t^2$ and $so_t^3=si_t^3$.

Using the construction in Theorem 5.2, we obtain $n_{l1}=-1$, $n_{l2}=0$ and $n_{l3}=1$. We also obtain $d_{l1}=1$, $d_{l2}=2$ and $d_{l3}=1$. The computation vertices in D_1, D_2 and D_3 are mapped onto processors 1,2 and 3 respectively. The resulting mapped graph is shown in Figure 5.8.

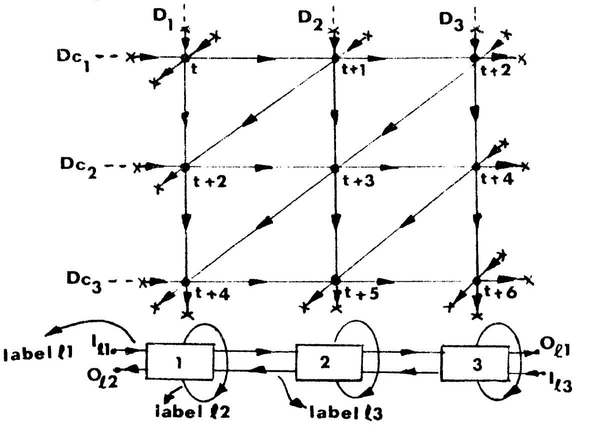


Figure 5.8

Lastly, we must some semantic properties of Ψ for correctness of execution. Ψ_2 and Ψ_3 are such that for any a,b and c, $\Psi_2<a,b,c>=b$ and $\Psi_3<a,b,c>=c$. Hence, the input/output value represented by the source/sink vertices of any vertical or oblique major paths does not change as it travels through processors in the linear array. In Figure 5.8 it is seen that the entry and consumption (production and exit) times for every input (output) value represented by every horizontal source (sink) vertex are the same.

Let t_s be the time when the computation begins. Clearly $t_s \leq t$. Since $n_{l2}=0$ a register in each processor serves as the input/output port labelled $l2$. Let r_1, r_2 and r_3 denote such a register in processors 1, 2 and 3 respectively. Then the input values of iv_1, iv_2 and iv_3 which are w_1, w_2 and w_3 respectively are preloaded into r_1, r_2 and r_3 respectively before t_s.

6. Conclusions

We presented a formal model of linear arrays, and introduced homogeneous graphs which are a natural representation of programs potentially executable on such arrays. For a large class of homogeneous graphs, a set of necessary and sufficient conditions on the structure of such graphs for the existence of a syntactically correct mapping were established. We then used our characterization to derive a systematic method for synthesizing algorithms for a class of program graphs. In [8, 11] extensions to the class of graphs examined in this paper can be found.

References

[1] L.J. Guibas, and F.M. Liang, "Systolic Stacks, Queues and Counters," Proc. MIT Conf. on Advanced Research in VLSI, (January, 1982), pp. 155-164.

[2] L. Johnsson, and D. Cohen, "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks," VLSI Systems and Computations, H.T. Kung, R.F. Sproull, and G.L. Steele, Jr., (editors), Computer Science Press, (1981), pp. 213-225.

[3] H.T. Kung, "Let's Design Algorithms for VLSI Systems," Proc. Caltech Conf. on Very Large Scale Integration: Architecture, Design, Fabrication, (January, 1979), pp. 65-90.

[4] H.T. Kung, and C.E. Leiserson, "Systolic Arrays (for VLSI)," Sparse Matrix Proceedings 1978, I.S. Duff, and G.W. Stewart, (editors), SIAM, (1979), pp. 256-282.

[5] S.Y. Kung, "VLSI Array Processor for Signal Processing," Proc. MIT Conf. on Advanced Research in Integrated Circuits, (January, 1980).

[6] H.T. Kung, "Why Systolic Architectures," IEEE Computer 15(1), (January, 1982), pp. 37-46.

[7] C. Mead, and L. Conway, Introduction to VLSI Systems, Addison-Wesley, (1980).

[8] I.V. Ramakrishnan, "A Theory of Mapping Program Graphs onto Linear Arrays," PhD Thesis, University of Texas at Austin, (May, 1983).

[9] I.V. Ramakrishnan, D.S. Fussell, and A. Silberschatz, "Systolic Matrix Multiplication on a Linear Array," Twentieth Annual Allerton Conf. on Computing, Control and Communication, (October, 1982).

[10] I.V. Ramakrishnan, Mapping homogeneous Graphs onto Linear Arrays, Department of Computer Sciences, University of Texas at Austin, TR-232, (April, 1983).

[11] I.V. Ramakrishnan, D.S. Fussell, and A. Silberschatz, "On Mapping Cube Graphs onto Linear Systolic Arrays," Seventeenth Annual Conf. on Information Sciences and Systems, The Johns Hopkins University, (March, 1983).

[12] U. Weiser, and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," VLSI Systems and Computations, H.T. Kung, R.F. Sproull, and G.L. Steele, Jr., (editors), Computer Science Press, (1981), pp. 226-234.

Unifying VLSI Array Designs with Geometric Transformations

Peter R. Cappello[†] *and Kenneth Steiglitz*[‡]

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N. J. 08544

ABSTRACT

A geometric representation of array computations is presented. Well known "systolic" designs for computing polynomial product are related to one another by affine transformations of a three-dimensional vector space. Much previous work on convolver designs is unified thus. Designs for linear transform and matrix product are unified similarly. New designs are geometrically derived for these computations that are asymptotically optimal under the VLSI complexity measure *area*×*period*^{2}, an appropriate measure for high throughput applications.

1. Introduction

There has been considerable research recently into array designs (see [Kung82] for a sampling of this work). Leiserson and Saxe [LeSa81] provide a general methodology for eliminating broadcasting from a synchronous circuit without changing its communication structure. Johnsson and Cohen [JoCo81] and Weiser and Davis [WeDa81],[JWCD81] investigate ways of formally representing computational designs. Their respective goals are similar: To be able to formally synthesize and analyze computational designs taking into account important design properties, such as correctness, area, time, communication topology, and the presence or absence of broadcasting/pipelining. Their strategies, which are also similar, center around the explicit representation of time in arithmetic expressions via a *delay* operator.

In this paper we present a geometric representation of array designs. Our goals are similar in spirit to those of Johnsson, Cohen, Weiser, and Davis. We seek a *unified* framework in which to represent array designs so that different array designs for the same computation are related in a formal way. Where they use a delay operator, we use an affine transformation. Since affine transformations are closed under composition (and in fact form a group), a single affine transformation can describe intuitively simple space/time rearrangements that may be difficult to describe as succinctly with other notations. A good representation, moreover, enhances a designer's intuition, and the geometric representation presented here may be helpful in this respect, too. It is easy to "see" for example how to transform an array design that uses broadcasting into one that does not.

Throughout this paper we use the terms *bit / word* and *serial / parallel*, *latency*, *cycle time*, *period*, and *completely –pipelined* as defined in [CaSt83]. The model of computation used in this paper is the *synchronous* model of VLSI [Thom80, BrKu81, BPP81]. We deal with classes of functions and circuits that are parameterized by a vector, π. For example, when we consider the linear transformation of a vector of *length* K, and *wordsize* B, the parameter vector is $\pi = (K, B)$. Asymptotic complexity will be measured with respect to a parameter vector, π, throughout this paper.

The remainder of the paper is organized as follows. In section 2 we introduce and apply the geometric representation to the problem of linear transform. In section 3 and 4, we apply it to convolution and matrix product. We finish with a discussion of other applications and some conclusions. The full version of this paper is [CaSt83f].

2. Linear Transform

In this section we introduce a geometric representation of array designs by way of example. Consider the computation of linear transform: $y \leftarrow A \cdot x$. It can be written as follows: $y_i \leftarrow \sum_j a_{ij} \cdot x_j$

To make the example more concrete we write out these expressions for a 3 × 3 matrix.

$$\begin{aligned} y_1 &\leftarrow a_{11} \cdot x_1 + a_{12} \cdot x_2 + a_{13} \cdot x_3 \qquad (2.1)\\ y_2 &\leftarrow a_{21} \cdot x_1 + a_{22} \cdot x_2 + a_{23} \cdot x_3 \\ y_3 &\leftarrow a_{31} \cdot x_1 + a_{32} \cdot x_2 + a_{33} \cdot x_3 \end{aligned}$$

We may think of the above set of expressions as constituting an *algorithm* for transforming x into y. That is, the expressions indicate that each output y_i can be obtained by computing certain specified products and adding them together. Considerable freedom remains as to how this algorithm can be implemented. For example, the above notation does not dictate a particular association for the additions; any will do. The following recurrence relation fixes a particular association:

$$\begin{aligned} y_{i0} &\equiv 0 \qquad (2.2)\\ y_{ij} &\leftarrow y_{i\,j-1} + a_{ij} \cdot x_j ; \\ y_i &\equiv y_{in} \end{aligned}$$

where n is the size of the vector x.

Now, to make this algorithm more specific, we adjoin an index representing *time* to y:

$$y_{ijt} \leftarrow y_{i\,j-1\,t} + a_{ij} \cdot x_j \text{ , for } t = 0. \qquad (2.3)$$

We take time to be discrete, and measure it in *cycles*[‡]. In our example, we set this time index, t, to zero. Thus as presently formulated, this whole computation occurs in one cycle: *cycle*$_0$. (The *meaning* of this time index is further explained shortly.) The algorithm is not yet completely specified; we have not indicated a particular method for performing addition and multiplication. One must have *some* primitive notions.

Definition: A computation is *primitive* if it is assumed that it can be done in constant area and with constant latency. We leave unspecified the algorithm for carrying out a *primitive* computation.

In order to give this recurrence relation a geometric interpretation we take the symbol "←" to represent the *location* of its *primitive computation*.

[†] Peter R. Cappello is now with the Department of Computer Science, University of California, Santa Barbara, CA 93106.

[‡] This work was supported in part by the National Science Foundation under Grant ECS-7916292, and in part by the U.S. Army Research Office, Durham, NC, under Grant DAAG29-79-C-0024.

[‡] This notion of time does not exclude asynchronous array computations, however.

0190-3918/83/0000/0448$01.00 © 1983 IEEE

Definition: The *primitive computation* represented by $\leftarrow$ is what occurs on its right-hand side.

In this case the computation is an inner-product-step.

Definition: The *location* represented by $\leftarrow$ is the index values of the left-hand side interpreted as coordinates.

Figure 2.a, illustrates the example. To properly interpret the figures please note that the meaning of a recurrence relation is unaffected by adding a constant to all occurrences of an index (e.g., $y_{ij} \leftarrow y_{ij-1}$ represents the same computation as $y_{i-1j-1} \leftarrow y_{i-1j-2}$). Equivalently, in the geometric representation a computation is unaffected by translation. The reader is cautioned that axes in the figures are intended merely to associate dimensions with indices: For ease of viewing, the geometric representations are translated to the nonnegative orthant. Figure 2.a is interpreted as follows. At location (1,1,0) the computation $y_{110} \leftarrow y_{100} + a_{11} \cdot x_1$ occurs. This means that the value of x_1 and a_{11} must "be" at location (1,1,0) (i.e., at spatial coordinates (1,1) at time coordinate 0). The solid lines indicate the path of a particular value of x. We refer to them as x-value contours. The dashed lines represent summation paths. They denote a particular addition association. These lines, or contours, are intended to make interpretation of the figures more intuitive. Movement of data, both input data distribution and output accumulation paths, also can be determined unambiguously from the recurrences themselves by using an ordering rule. In this paper the ordering rule is simply the lexicographic order of the spatial indices *within time*. So for example, if an output value is accumulated at more than one location with the same time index value, then it is accumulated at these locations according to the lexicographic order of their spatial coordinates. We will call Eq. 2.3 the *canonical* design of the algorithm denoted by Eq. 2.2, and denote its geometrical representation by Γ. This representation of the computation illustrates an important aspect of time as we represent it. If two computations, c_{ijt} and d_{kls} are located at distinct points in time, (i.e., $t < s$), then c occurs *before* d. Again however, if $t = s$, then c and d take place during the same cycle, but *not* necessarily "simultaneously." In fact, we have no notion of "simultaneity" in our interpretation of time.

Interpreting Figure 2.a we see that the computation occurs in nine distinct locations in space, and at one location in time.

Definition: When input information is at distinct locations in space while at the same location in time, we say it is *broadcast* to those locations. (Again, no notion of simultaneity is implied.)

Together, a geometric representation and an ordering rule indicate what data moves, where it moves to, and when it moves. The number of physically distinct computational elements is simply the number of computational locations whose spatial coordinates are distinct. The topology of these elements emerges as well when we project out the time dimension. Thus the geometric representation associates a particular schedule of computation (an algorithm) with a particular network of computational elements. We speak of the association of an algorithm with a computational structure as a *design*. In what follows we apply various geometric transformations to the canonical design. The transformations, then, relate distinct designs in a formal way.

THE KUNG AND LEISERSON DESIGN

The canonical design is not well suited to implementation because the nine processing elements of the array are idle most of the time. Leiserson and Kung [LeKu80] present an array design for linear transform that is better. Their design, denoted Λ, is illustrated in Figure 2.b. The communication structure is simply a linearly-connected array of processors. Each processor performs an inner-product-step computation. Briefly, the design works as follows. X values move to the right through the array, while output values are accumulated as they move to the left. The transform coefficients move down through the array as indicated by Figure 2.b. More detail is given in [LeKu80].

We now represent this design geometrically by obtaining it from transformations on the canonical design. We first apply R,a rotation-like transformation. We then "interchange" a space dimension with the time dimension. That is, we select a dimension that represents space and interpret it as time. Since there can be only one time dimension, we must now interpret the old one as space. A permutation transformation, T is used to effect this semantic interchange[†]. The result is depicted in Figure 2.c. It is a geometric representation of the Leiserson and Kung design. To see this, one needs to interpret the representation. Since we interchanged one of the two space dimensions with the time dimension, the resulting design has only one *non-trivial* spatial dimension: (Recall that the initial time coordinates values of the computation were all 0. Those coordinate values now represent a spatial dimension. That spatial dimension is unused: it is trivial.) Thus the communication structure is a *linearly*-connected array. The x values move right in space over time, the outputs move left in space over time, as in the Leiserson and Kung design. The transform coefficients, x values, and y values move through the linear array with the same schedule as the Leiserson and Kung design. We have derived the following formal representation of Λ:

$$\Lambda = T \cdot R(\Gamma) \ , \quad \text{where } R = \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Notice that Λ uses $2n-1$ cycles and $2n-1$ inner-product-step processing elements. We now present a new linear transform design that uses $2n-1$ cycles but uses only n inner-product-step processing elements. It is illustrated in Figure 2.d. We first apply a transformation, S, which skews the canonical representation, Γ. The time-space interchange transformation, T, is then applied as before. We interpret the result as follows. Due to the time-space interchange the array again extends in only one dimension in space. But now its image in space is a linearly-connected array of n processing elements, not $2n-1$. X vectors, whose components are skewed in time, are piped through the array while the transformed vector's components are *accumulated* in distinct processors, also over time. There is no fill-up and drainage period with this design: New x-vectors and transform coefficients can follow on the "heels" of the preceding ones. The design, denoted by Ψ, is related to the canonical design by the transformations T and S:

$$\Psi = T \cdot S(\Gamma) \ , \quad \text{where } S = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This formal derivation of a new design illustrates the utility of the geometric transform approach. Both designs Λ and Ψ have period $O(n)$, (i.e., are not completely-pipelined). We next describe a new design that has period $O(1)$ (i.e., is completely-pipelined).

[†] This is an example of what Johnsson and Cohen refer to as "mapping space into time." In this geometric representation data control information is implicit.

Furthermore, it is asymptotically optimal with respect to the complexity measure AP^2.

AN *AREA* × PERIOD ^{2} OPTIMAL DESIGN

Period rather than latency (delay) is a good measure in applications where high throughput (rather than short latency) is of interest. Before presenting our AP^2 optimal design, a few terms and facts are noted. With these we argue that linear transform is computationally a more interesting problem when the transform is *fixed* so that the input *size* of the problem is n (words), rather than $n \cdot n$ (words) where the transform is multiplication by an $n \times n$ matrix.

Vuillemin has shown [Vuil80] that 1) linear transform, convolution, and matrix product are transitive functions, and 2) any circuit computing a transitive function at data rate D must have wire area $A_w \geq a_w \cdot D^2$, for some technology-dependent constant a_w.

Vuillemin's lower bound for linear transform is not valid for *every* linear transform. The Identity transform, for example, clearly requires wire area only linear in the data rate. The bound, however, is an *existence* bound: It says there exist some linear transforms whose area is $\Omega(D^2)$. Many important transformations such as the Discrete Fourier Transform (DFT), however, are among those for which the quadratic bound holds [Vuil80]. We note that since the period $P=n/D$ where n is the number of input bits and D is the rate at which they are read in, we have that for transitive functions (such as the DFT) $AP^2(n) = \Omega(n^2)$.

We now explain how the AP^2 complexity of linear transform can be dominated by I/O.

Definition: The *aspect ratio*, σ, of a layout is W/L where W and L are its width and length, respectively.

Most families of structures, such as complete binary trees, have a set of parameters (e.g., the *number* of leaves in the tree) that characterize any member of the family. A layout for a family of structures can in general have its length, width, and hence aspect ratio be a function of those parameters. Layouts that do not have constant aspect ratios are often considered undesirable because they result in layouts that are long and thin as n gets large. Lipton and Sedgewick [LiSe81] prove (where T denotes latency) that $AT^2(n) = \Omega(n^2)$ for constant aspect ratio layouts whose n inputs are constrained to be at the boundary of the layout. We generalize this result with the following.

Theorem: Let C be a circuit that computes $f(n)$ with period P. Let L be a constant aspect ratio layout of C with area A and perimeter p. The portion of L's perimeter used for input ports is denoted by p_b. L, moreover, may have $O(1)$ convex input ports in its interior. These interior ports may accommodate more than $O(1)$ input bits per unit time: the area of an interior port may be more than $O(1)$. Then $AP^2(n) = \Omega(n^2)$. (See [CaSt83f] for proof.)

This strengthens Lipton and Sedgewick's result because the bound is retained even if a constant number of (large) interior ports exist, and because $P \leq T$.

Thus when an $n \times n$ coefficient matrix is part of a circuit's input, and the circuit has a constant aspect ratio layout, then $AP^2(n^2) = \Omega(n^4)$. However when the $n \times n$ coefficient matrix is fixed, there is a design that has $AP^2(n) = \Theta(n^2)$ as we will now show. This design achieves Vuillemin's lower bound and so is asymptotically optimal. It also shows that the previous case is I/O bound: boundary placement of I/O pads dominate the layout area.

We now describe a design that achieves the AP^2 lower bound for fixed linear transform. First the canonical design is skewed as before. But now instead of interchanging the (trivial) time dimension with a space dimension, we *pipeline* this communication structure (which is a two-dimensional mesh of processing elements). A rotation-like transform, N, accomplishes this (see Figure 2.e). We denote the resulting design by Δ, where

$$\Delta = N \cdot S(\Gamma) \ , \quad \text{where } N = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

As one can see from Figure 2.e, inputs are piped through the mesh. Input vectors have their components skewed in the time dimension. Output vector components are similarly skewed. For this design, $P = O(1)$. Since the linear transform is fixed, we can assume that the a_{ij}'s are encoded into their proper inner product step computation. (Again, the inner-product-step is taken as primitive. That this assumption is reasonable will become clearer when we give an AP^2 optimal convolution design in section 3; the inner-product-step can be viewed as a bit-level variant of convolution.) The area is $O(n^2)$. Thus $AP_\Delta^2(n) = \Theta(n^2)$, which is asymptotically optimal. For designs Γ, Λ, and Ψ, the coefficient matrix is part of the input: The input is of size n^2. By the aspect ratio theorem, Γ, Λ, and Ψ have $AP^2(n^2) = \Omega(n^4)$. That is, these designs are dominated by pin-in. When the input matrix is fixed, these designs are not optimal with respect to the measure AP^2.

3. Convolution

Consider the computation of convolution.

$$w_i \leftarrow \sum_j x_j \cdot y_{i-j} \qquad (3.1)$$

This one operation can represent an *FIR filter*, a *Discrete Fourier Transform* [RdGd75], or (when " · " is interpreted as a bit product and carry propagation is included) *multiplication*. Also, Foster and Kung [FoKu80] have noted that convolution describes string pattern matching when " · " is interpreted as ***string compare*** and "+" is interpreted as boolean ***and***. For the purposes of this paper Eq. 3.1 can represent any computation where X, Y, and W are (not necessarily distinct) sets, '·' is a map from $X \times Y$ to W, and $(W, +)$ is a monoid (i.e., an associative binary composition with identity). In this section we apply our geometric representation to this computation.

In a recent article H. T. Kung [Kung82] enumerated seven known designs for convolution. We relate six of them to one another by geometric transforms. In this way we unify much of the work on convolution designs. Then we transform these to new AP^2 asymptotically optimal designs. First we establish a geometric representation of this computation. It is very similar to linear transform. Writing out an example convolution for x and y signals of length three, we obtain:

$$\begin{aligned} w_0 &\leftarrow x_0 \cdot y_0 \qquad (3.2) \\ w_1 &\leftarrow x_0 \cdot y_1 + x_1 \cdot y_0 \\ w_2 &\leftarrow x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0 \\ w_3 &\leftarrow x_1 \cdot y_2 + x_2 \cdot y_1 \\ w_4 &\leftarrow x_2 \cdot y_2 \end{aligned}$$

We réformulate these using a recurrence relation:

$$w_{i0} \equiv 0 \qquad (3.3)$$
$$w_{ij} \equiv w_{i\ j-1} + x_j \cdot y_{i-j};$$
$$w_{in} \equiv w_i$$

where n is the signal size. We let "$\leftarrow$" represent an inner-product-step computation located in space and time as before. To do this we again adjoin a time coordinate (see Figure 3.a):

$$w_{ijt} \leftarrow w_{i\ j-1\ t} + x_j \cdot y_{i-j};\ , \text{ for } t=0 \qquad (3.4)$$

We take this to be the canonical design of the algorithm expressed by Eq 3.3, and denote its geometric representation by Γ as before. We now proceed to derive some known designs by geometrically transforming Γ to them. Table 3.a summarizes the seven designs that H. T. Kung noted in his article. Figure 3.b illustrates design B1 in a conventional way. B1 is a design especially close to the canonical design. By applying the time-space interchange transformation T to Γ, B1 is derived (illustrated in Figure 3.c): $B1 = T(\Gamma)$ We interpret that figure as follows. The same x value appears at processors that are spatially but *not* temporally distinct: x values are *broadcast* to their processors. Similarly, y values (weights) appear at processors that are temporally but *not* spatially distinct: y values "stay". And w values (results) appear at processors that are distinct in both space and time: they "move".

Seven Convolution Designs			
Name	Results	Input	Weights
B1	Move	Broadcast	Stay
B2	Stay	Broadcast	Move
F	Fan-in	Move	Stay
R1	Stay	Move in opposite directions	
R2	Stay	Move in same direction at different speeds	
W1	Move in opposite directions		Stay
W2	Move is same direction at different speeds		Stay

Table 3.a Convolution designs enumerated in [Kung82].

We summarize the other derivations in Table 3.b. The transforms used are as follows (T, R, and N are as already defined).

$$S_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad S_2 = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$S_3 = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad S_4 = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Geometric Design Definitions		
Name	Transform from Γ	AP^2 optimal counterpart
B1	T	N
B2	$T \cdot S_1$	$N \cdot S_1$
R1	$T \cdot R$	$N \cdot R$
R2	$T \cdot S_2$	$N \cdot S_2$
W1	$T \cdot S_4$	$N \cdot S_4$
W2	$T \cdot S_3$	$N \cdot S_3$

Table 3.b Geometric design definitions for the word-serial convolvers and their word-parallel AP^2 optimal counterparts

Figure 3.c illustrates the geometrical representations of the six designs. The reader is invited to compare interpretations of the geometric representations with the verbal ones in Table 3.a. Design F is not related to the others because "fan-in" is usually implemented with an add tree and the addition association of such a tree is different from that indicated by Eq. 3.3.

Nonetheless, this algorithm *does* admit other designs. We now present some AP^2 asymptotically optimal designs. Like their word-serial counterparts, these designs have different data movement characteristics. Such qualities are important in practice; a design (for, say, integer multiplication) may have data movement constraints deriving from the larger application in which it is embedded. For each of the six designs we have described geometrically, there is a corresponding AP^2 optimal design. As in *linear transform*, we transform the these six convolution designs to AP^2 optimal designs by pipelining them. In fact, the same transform is used. We replace the T transform in Table 3.b with the N transform. Consider design R2, for example. Its AP^2 optimal counterpart is illustrated in Figure 3.d. Spatially it is a hex-connected mesh of processors. X signal components, skewed in time, move along their contours (the solid lines), y signal components, skewed in time, move along their contours (the dashed lines), while output signal components are accumulated along the dotted lines. An input and output signal can be accepted every cycle of the processor assemblage. Thus the period is $O(1)$, the area is $O(n^2)$, and $AP^2_\Delta(n) = O(n^2)$. As in Section 2, Vuillemin [Vuil80] shows that convolution is a transitive function: $A(n) = \Omega(D^2)$. Thus $AP^2(n) = \Omega(n^2)$, and these new designs are asymptotically optimal. The word-serial designs displayed in the left column of Table 3.b all have $P(n) = O(n)$ and $A(n) = O(n)$: none are AP^2 optimal.

Finally, we note that *all* the designs presented have $AP(n) = O(n^2)$. Put another way, designs implementing the same algorithm all have the same *switching energy*, E_{sw} (see [MeCo80] for a definition of this quantity); this energy is just distributed differently in space and time. Clearly, one-to-one transformations (such as affine transformations) conserve E_{sw}.

4. Matrix product

In this section we examine the matrix product computation. After placing it in a geometric setting, we proceed to derive and relate the band matrix product designs of Leiserson and Kung [MeCo80] and Weiser and Davis [WeDa81]. Finally we present an AP^2 asymptotically optimal design for computing matrix product.

Given two matrices A and B, we can denote their product $C \leftarrow A \cdot B$. A more algorithmic description of matrix product is

$$c_{ij} \leftarrow \sum_k a_{ik} \cdot b_{kj}\ , \text{ for } 1 \le i,j,k \le n \qquad (4.1)$$

where A, B, and C are $n \times n$ matrices. We have written Eq. 4.1 for $n = 2$:

$$c_{11} \leftarrow a_{11} \cdot b_{11} + a_{12} \cdot b_{21} \qquad (4.2)$$
$$c_{12} \leftarrow a_{11} \cdot b_{12} + a_{12} \cdot b_{22}$$
$$c_{21} \leftarrow a_{21} \cdot b_{11} + a_{22} \cdot b_{21}$$
$$c_{22} \leftarrow a_{21} \cdot b_{12} + a_{22} \cdot b_{22}$$

Again we reformulate using a recurrence relation:

$$\begin{aligned} &c_{ij0} \equiv 0 && (4.3)\\ &c_{ijk} \leftarrow c_{i\,j\,k-1} + a_{ik} \cdot b_{kj}\ ; \\ &c_{ij} \equiv c_{ijn} \end{aligned}$$

In order to let "←" represent an inner-product-step computation located in space and time we adjoin a fourth coordinate, time (see Figure 4.a):

$$c_{ijkt} \leftarrow c_{i\,j\,k-1\,t} + a_{ik} \cdot b_{kj}\ ; \qquad (4.4)$$

We again call this the *canonical design* of the algorithm expressed by Eq. 4.3, and denote its geometric representation by Γ.

Band matrix product, an important special case of matrix product, is illustrated conventionally by the matrix expression in Figure 4.b whereas Figure 4.c illustrates a geometric representation of the computation.

$$C = \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & 0 \\ b_{21} & b_{22} & b_{23} \\ 0 & b_{32} & b_{33} \end{bmatrix}$$

Figure 4.b Band matrix product

Figure 4.d displays a summary (and conventional representation) of the systolic design to do this computation that was devised by Leiserson and Kung. The reference [MeCo80] provides more detail. Their design, which we denote by Λ, is based on the same algorithm as the canonical design: Eq. 4.3. We now present a geometric representation of the Leiserson and Kung design. To obtain it one can take the staircase-like structure of the canonical design and situate it vertically using two rotation-(like) transforms. The Λ design, illustrated in Figure 4.e, emerges when the vertical space dimension is interchanged with the (previously unshown) time dimension (i.e., interpret the vertical dimension as time). The reader is invited to verify this informally The K matrix is a transformation that *combines* the three transformations just described:

$$\Lambda = K(\,\Gamma\,)\ , \quad \text{where} \quad K = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

In Λ, approximately one third of the processing elements are active on any given cycle. Weiser and Davis present [WeDa81] a design that improves Λ in this respect. Their design, which we denote by Ψ, is depicted conventionally in Figure 4.f. Like Λ, it uses a hex-connected array. In Ψ, however, the A band matrix flows through a row at a time, and the B band matrix, a column at a time, producing the C product band matrix a column at a time. (There is, of course, a dual design producing a row at a time.) Ψ, obtained from Γ by an affine transformation, is: $\Psi = D(\,\Gamma\,)$ where D is defined as follows:

$$D = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 1 & -1 & 1 & 0 \end{bmatrix}$$

In Ψ, all processing elements are working every cycle. Its throughput rate is three times that of Λ.

As in convolution, other designs are possible. For example, Preparata and Vuillemin [PrVu80] present a matrix product design that is AP^2 asymptotically optimal; Γ, Λ, and Ψ are not. It is a recursive design: matrix product is computed by summing sub-matrix products. We now present a new matrix product design that is also AP^2 asymptotically optimal. The idea is to take the spatial restriction of Γ, the three-dimensional mesh, and project it onto two dimensions so that each processing element has a unique image. (This transformation is not affine and not one-to-one.) We skew this intermediate design in order to obtain a pipelined design (i.e., eliminate input broadcasting and skew output accumulation in time). The resulting design, denoted by Δ is illustrated geometrically in Figure 4.g.

$$\Delta = \Upsilon(\,\Gamma\,)\ , \quad \text{where} \quad \Upsilon = \begin{bmatrix} 2n-1 & 1 & 1 & 0 \\ 0 & n & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & n & 1 & 0 \end{bmatrix}$$

It has period $O(1)$ because it is pipelined and has area $O(n^4)$. Thus $AP^2_\Delta(n^2) = O(n^4)$, which matches its lower bound. As before, the lower bound derives from the fact that it is a transitive function (see section 2 and [Vuil80]).

It perhaps is worth noting that the projection of the three-dimensional mesh onto a two-dimensional space, resulting in some $\Omega(n)$ wire lengths, is necessary because VLSI is presently a two-dimensional medium. In three-dimensional VLSI, the three-dimensional mesh skewed in time produces a $VP^3(n^2) = O(n^3)$ design.

5. Other applications

This technique can be applied to a wide variety of computations formulated as recurrence relations such as the systolic designs given in [LeKu80] for LU decomposition and the solving of triangular linear systems. These designs involve arrays of *more than one kind* of processing element, an important generalization. Geometric representations can be applied in an even more general setting, however. They are suited to represent any n-dimensional cellular automata [Burk70] in an $n+1$-dimensional space. A wealth of computation designs, thus, can be explored using geometric transformations.

Indeed, viewed in this way it is easy to see that *for every computable function* f, *there is, in fact, a completely pipelined ($P_f = O(1)$) hexagonally-connected cellular automaton* (i.e., a cell simple, systolic structure) *that computes* f. (See [CaSt83f] for details.)

6. Conclusions

We have presented a technique for placing array designs in a geometric framework and we have used this formal framework to relate different array designs. New designs were given for the functions discussed, and these too were related to other designs by transformations. Design properties such as broadcasting and pipelining can be formally defined and their presence or absence in a particular design can be ascertained readily. A design's communication topology can be disclosed likewise by projecting out the time dimension of the representation.

Algorithms are traditionally designed for a random access machine (RAM). VLSI has precipitated a generalization of the algorithm designer's task: a particular algorithm may be implemented via a wealth of different communication structures, each with different properties. In the context of VLSI it has become necessary to distinguish between an algorithm and an implementing structure. We have used the noun *design* to designate a coupling of a particular algorithm with a particular communication structure. Recurrence relations such as those discussed in this paper lead to highly regular (array) designs and so are a convenient starting point for a more general investigation. On the other hand, a wide variety of functions can be formulated as such recurrence relations, so this approach has merit in its own right.

7. References

[BPP81]Bilardi, G., M. Pracchi, and F. P. Preparata, "A Critique and an Appraisal of VLSI Models of Computation," **VLSI Systems and Computations,** Edited by H. T. Kung, Bob Sproull, and Guy Steele, Computer Science Press, Rockville, Maryland, 1981.

[BrKu81]Brent, R. P. and H. T. Kung, "The Area-Time Complexity of Binary Multiplication," *JACM* Vol. 28 No. 3, July 1981.

[Burk70]Burks, A. W., Editor, **Essays on Cellular Automata,** Univ. of Illinois Press, Urbana, IL, 1970.

[CaSt83]Cappello, P. R. and K. Steiglitz, "A VLSI Layout for a Pipelined Dadda Multiplier," *ACM Trans. on Computer Systems*, Vol. 1, No. 2, May 1983, Pages 157-174.

[CaSt83f]Cappello, P. R. and K. Steiglitz, "Unifying VLSI Array Design with Geometric Transformations," University of California, Santa Barbara, Computer Science Dept. Tech. Rept. Santa Barbara, CA 93106.

[FoKu80]Foster, M. J. and H. T. Kung, "Toward a Theory of Systolic Algorithms for VLSI," (Abstract), *Proc. Advanced Research in Integrated Circuits*, M.I.T.,Cambridge, MA. Jan. 1980.

[JoCo81]Johnsson, L. and D. Cohen, "A Mathematical Approach to Modeling the Flow of Data and Control in Computational Networks," **VLSI Systems and Computations,** Edited by H. T. Kung, Bob Sproull, and Guy Steele, Computer Science Press, Rockville, Maryland, 1981.

[JWCD81]Johnsson, L. U. Weiser, D. Cohen, and A. L. Davis, "Towards a Formal Treatment of VLSI Arrays," *2nd Caltech Conf. on VLSI*, pp. 375-398, 1981.

[Kung82]Kung, H. T., "Why Systolic Architectures," Carnegie-Mellon Univ., Dept. of Computer Science, CMU-CS-81-148, Nov. 1981.

[LeKu80]Leiserson, C. E. and H. T. Kung, "Algorithms for VLSI Processor Arrays," as Section 8.3 of *Introduction to VLSI Systems*, Carver Mead and Lynn Conway, Addison-Wesley Publishing Co. Menlo Park, Ca., 1980.

[LeSa81]Leiserson, C and J. Saxe, "Optimizing Synchronous Systems," *Proc. 22nd Annual Symp. Foundations of Computer Science*, IEEE Computer Society, Oct. 1981.

[LiSe81]Lipton, R. J. and R. Sedgewick, "Lower Bounds for VLSI," *Proc. 13th Annual Sym. on the Theory of Computing*, May, 1981.

[MeCo80]Mead, C. and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Co. Menlo Park, Ca. 1980.

[PrVu80]Preparata, F. and J. Vuillemin, "Area-Time Optimal VLSI Networks for Parallel Matrix Multiplication," *Proc. of the 1980 Conf. on Information Science and Systems*, Princeton, NJ, March 1980.

[RdGd75]Rabiner, Lawrence R. and Bernard Gold, **Theory and application of digital signal processing,** Prentice-Hall, Inc., Englewood Cliffs, N.J., 1975.

[Thom80]Thomson, C. D., "Area-Time Complexity for VLSI," *Proc. 11th Annual Sym. on the Theory of Computing* April, 1979.

[Vuil80]Vuillemin, J., "A Combinatorial Limit to the Computing Power of VLSI Circuits," *Proc. IEEE 21st Annual Symposium of Foundations of Computer Science*. Syracuse, N.Y., 1980.

[WeDa81]Weiser, U. and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," **VLSI Systems and Computations,** Edited by H. T. Kung, Bob Sproull, and Guy Steele, Computer Science Press, Rockville, Maryland, 1981.

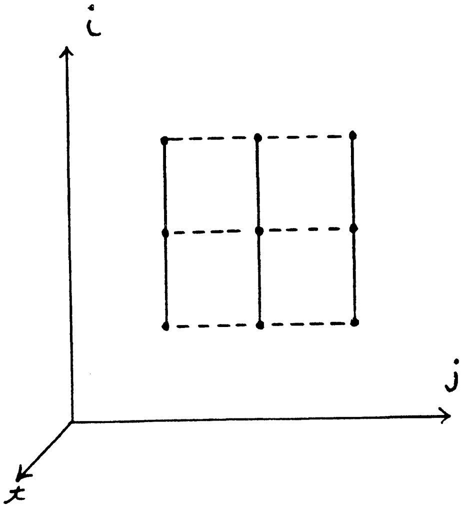


Figure 2.a Geometric representation of the canonical linear transform design.

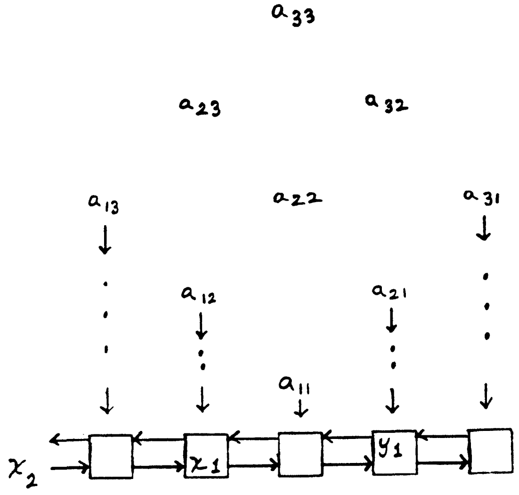


Figure 2.b Conventional representation of the Leiserson and Kung linear transform design.

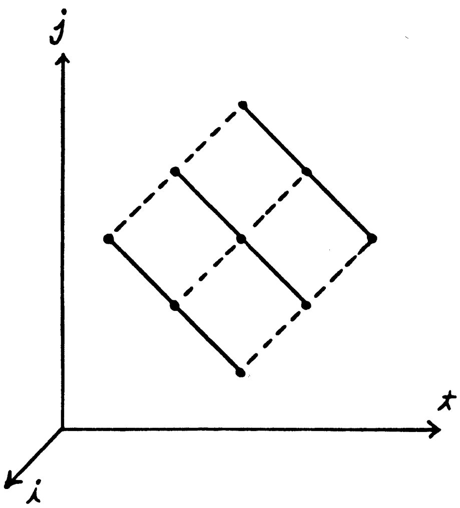


Figure 2.c Geometric representation of the Leiserson and Kung linear transform design.

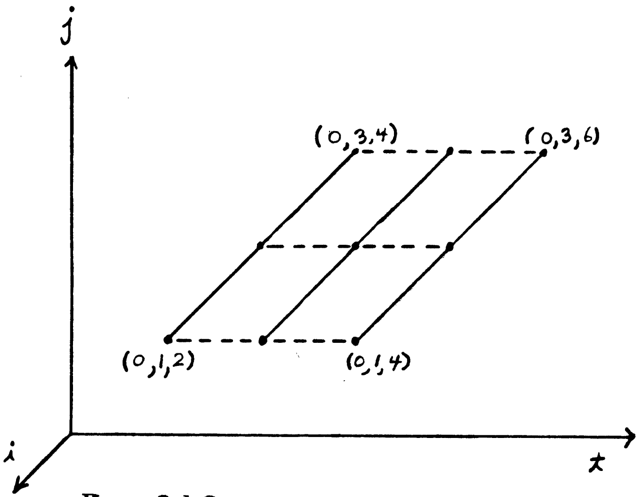


Figure 2.d Geometric representation of the design Ψ.

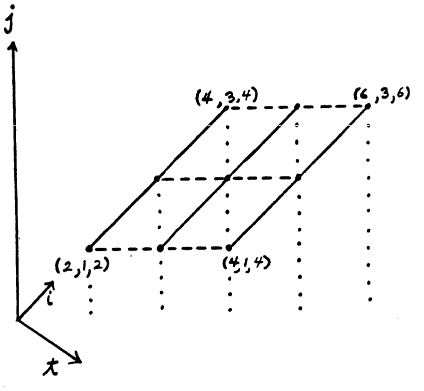


Figure 2.e Geometric representation of an AP^2 optimal design Δ. Dotted lines trace a projection of computation locations onto the *i* – *time* plane.

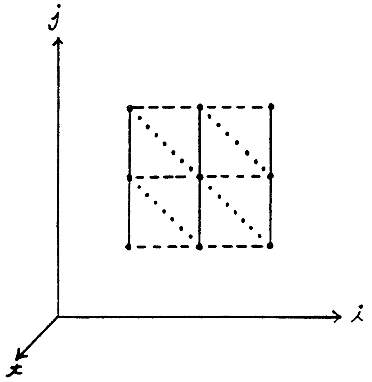


Figure 3.a Geometric representation of the canonical convolution design. In all convolution figures, *x* contours are represented by solid lines, *y* contours are represented by dashed lines, *w* accumulation paths are represented by dotted lines.

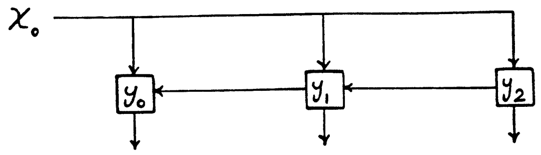
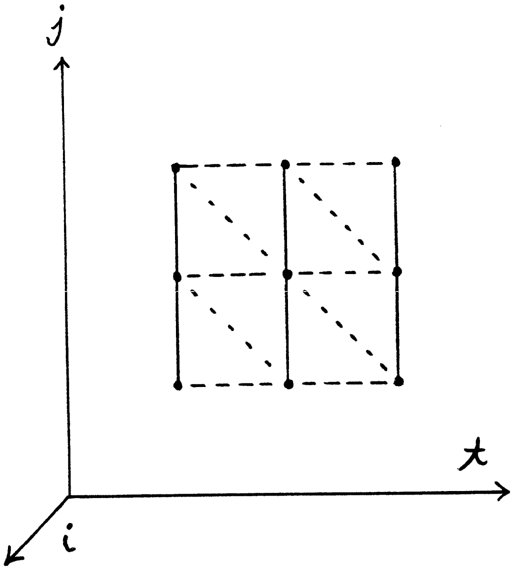
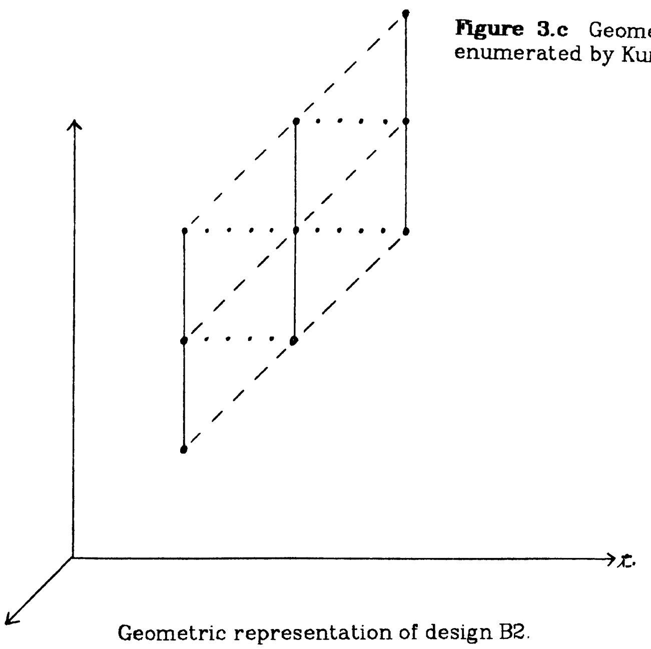


Figure 3.b Conventional representation of design B1.

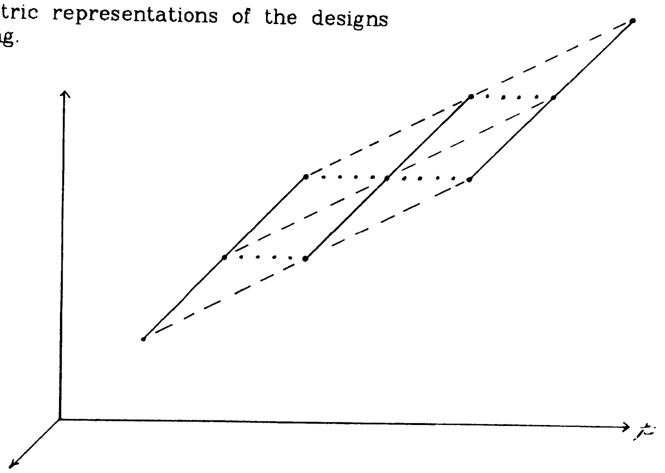


Geometric representation of design B1.

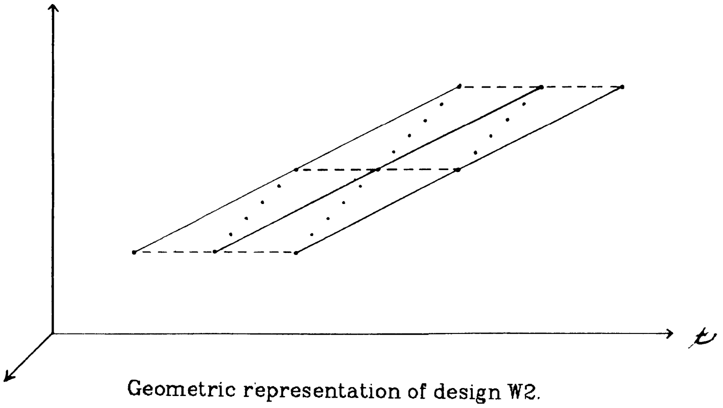
Figure 3.c Geometric representations of the designs enumerated by Kung.



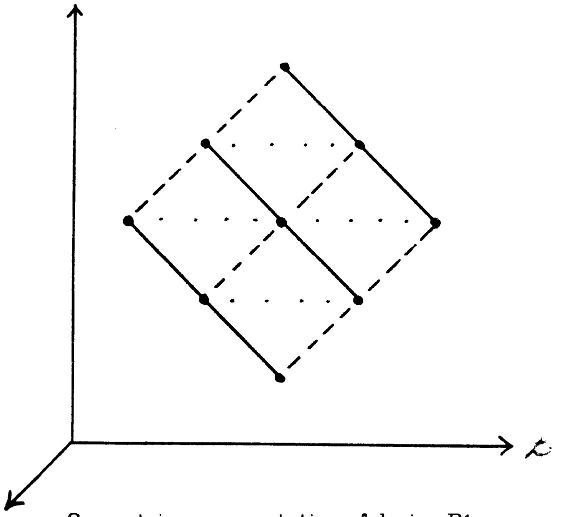
Geometric representation of design B2.



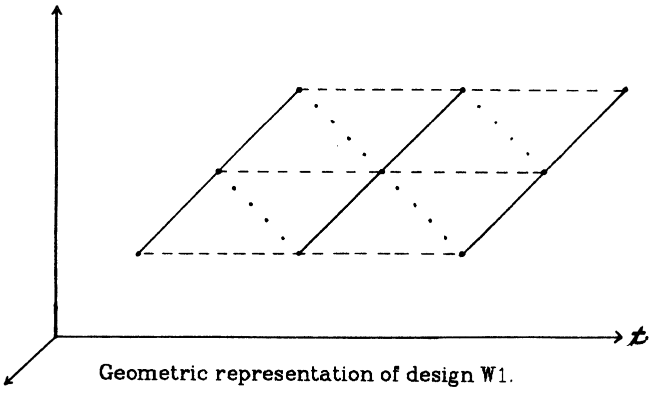
Geometric representation of design R2.



Geometric representation of design W2.



Geometric representation of design R1.



Geometric representation of design W1.

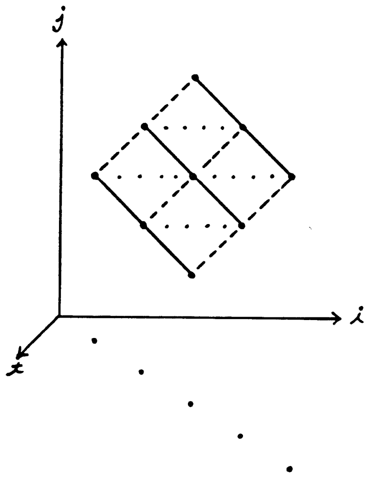
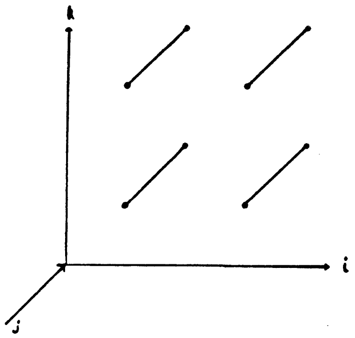
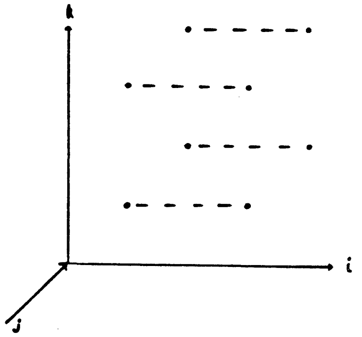


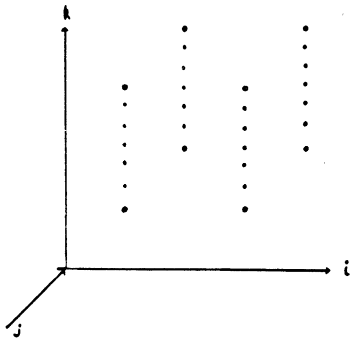
Figure 3.d Geometric representation of design R2's AP^2 optimal counterpart OR2. Dots below are the projection of the computation onto the i – *time* plane.



Geometric representation of *A* matrix contours in the canonical matrix product design.



Geometric representation of *B* matrix contours in the canonical matrix product design.



Geometric representation of *C* matrix accumulation in the canonical matrix product design.

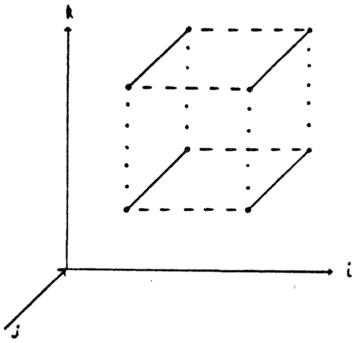


Figure 4.a Geometric representation of the canonical matrix product design. Time, the fourth dimension, is not shown in Figure 4.a.

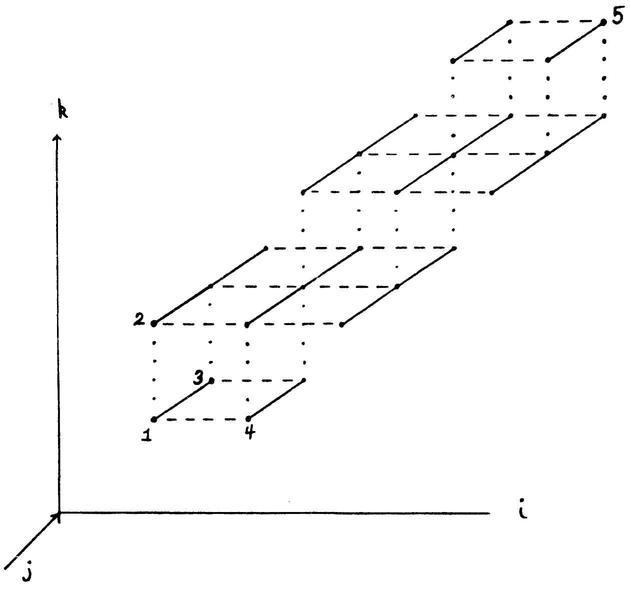


Figure 4.c Geometric representation of the band matrix product of Figure 4.b. To obtain a geometric representation of the Leiserson and Kung design 1) rotate the canonical representation such that a line connecting computations labeled 1 and 5 would be parallel to the *k* axis, and computations labeled 2, 3, and 4 all would have the same *k* coordinate value; 2) interchange the *k* and *t* dimensions (i.e., interpret the *k* axis as time). See Figure 4.e.

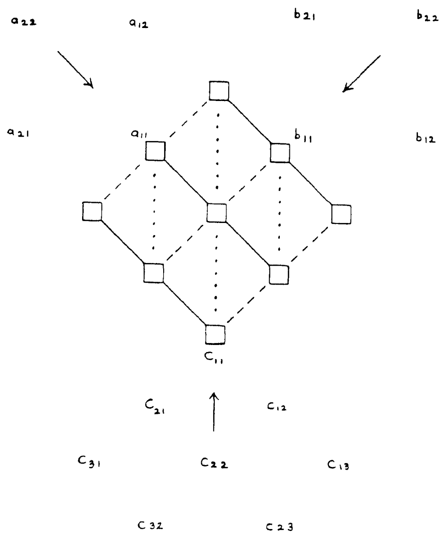


Figure 4.d Conventional representation of the Leiserson and Kung design for band matrix product.

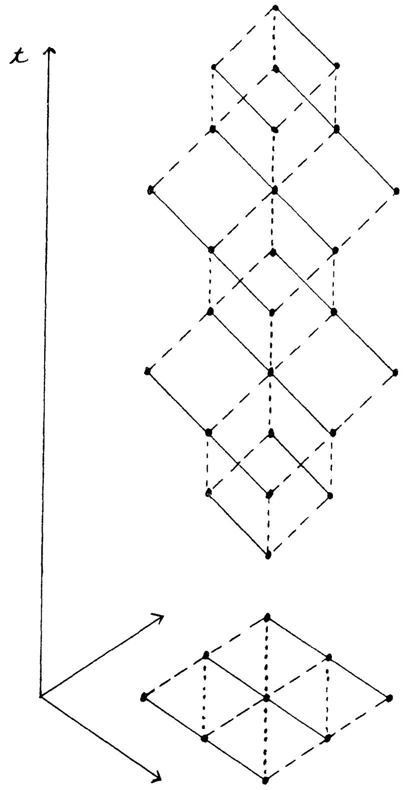


Figure 4.e Geometric representation of the Leiserson and Kung design for band matrix product.

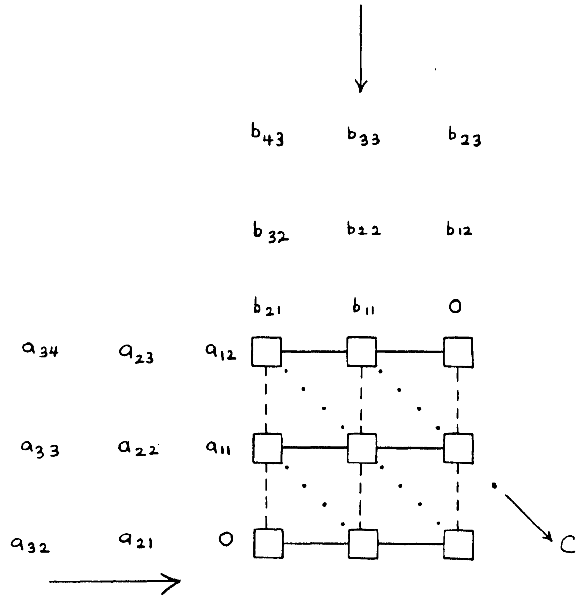


Figure 4.f Conventional representation of the Weiser and Davis design for band matrix product.

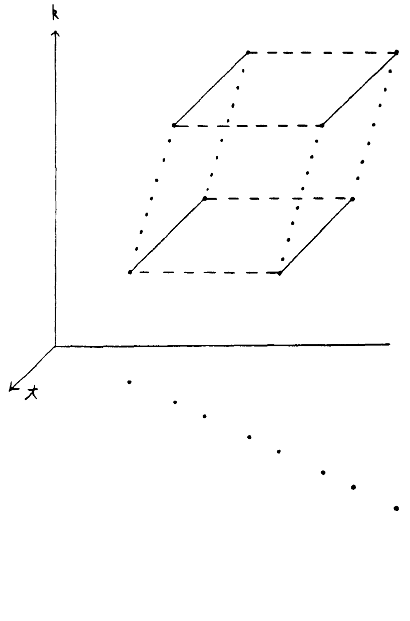


Figure 4.g Geometric representation of an AP^2 optimal matrix product design. Dots below are the projection of the computation onto the *i−time* plane.

Design of Robust Systolic Algorithms[a]

Peter J. Varman
Donald S. Fussell

The University of Texas at Austin
Austin, Texas 78712

Abstract

A primary reason for the susceptibility of systolic algorithms to faults is their strong dependence on the interconnection between the processors in a systolic array. A technique to transform any linear systolic algorithm into an equivalent pipelined algorithm that executes on arbitrary trees is presented.

1. Introduction

In this paper we present an approach to obtaining special-purpose systolic devices that are robust in the face of production flaws [1]. Due to the close coupling between communication requirements and the processor interconnection in systolic algorithms [2], any alteration in the interconnection will cause these algorithms to fail. A technique to transform the class of linear systolic algorithms [3] into equivalent pipelined algorithms that execute efficiently on an arbitrary tree of processors is described. Any connected set of fault-free processors may then be used to execute these redesigned systolic algorithms.

2. Computation Model

Let T be a rooted tree of (n+1) vertices and L a finite set of labels. Construct a graph G to serve as the tree machine as follows (Fig. 1). Consider some $j \in L$ and replace each edge in T by a pair of edges with label 'j' between the vertices joined by that edge. Repeat the construction for all labels belonging to L. Consider the subgraph G^j consisting of all the vertices in G and all edges labelled 'j'. Let P^j be an Euler circuit in G^j with the root as the initial and final vertex on the circuit. Traversal of P^j induces a direction on each edge in the path: an edge traversed from vertex u to vertex w is directed into w. Partition the 2n edges in P^j into two equally-sized sets of solid and dotted edges as follows. For every vertex u in G^j except the root, exactly one of the edges of P^j that is directed into u is solid; the rest of the edges are dotted. Index the vertex into which the i^{th} solid edge in P^j (h_i^j) is directed as v_i; index the root v_0. For every other label $k \in L$, the edges in the corresponding Euler circuit P^k are similarly partitioned so that if the i^{th} edge in P^j is a solid (dotted) edge between some two vertices in G, then the i^{th} edge in P^k is also a solid (dotted) edge between those vertices.

A token is a unit of data that passes between vertices by traversal of the edges in G. Each token has a label from the label set L and a token with label 'j' (called a j-token) traverses only edges with label 'j' in G. An execution path for a token is an ordered sequence of edges in G that the token traverses. For notational convenience we shall often drop the superscript on an edge that indicates its label, with the understanding that a token of the corresponding label would traverse that edge. We consider the case when the execution path for a token is the Euler circuit P and distinguish two types of execution paths based on the direction in which P is traversed. An execution path P is said to be of type-1 if (h_1, h_2, ..., h_n) is a subsequence of P; correspondingly, P is of type-2 if <h_n, ..., h_2, h_1) is a subsequence of P. A token that follows a type-1 (type-2) execution path will be referred to as a type-1 (type-2) token. If a token is available at some vertex then it traverses the next edge on its execution path and becomes available at the vertex at the other end of the edge traversed. A token is correctly available at vertex v_i if it is available at v_i and either (1) if it is of type-1 then the last edge traversed is the solid edge h_i or (2) if it is of type-2 then the next edge to be traversed is the solid edge h_i.

A clock cycle is the basic time unit. Each edge in G has a delay associated with it, which is the number of cycles required by a token to traverse that edge. The delay for all solid edges of label 'j' is dh^j and all dotted edges of label 'j' is dc^j. If 't' is the cycle at which a j-token is available at some vertex, it is available at the vertex at the other end of the edge at cycle $t+dh^j$ if the edge is solid, or at cycle $t+dc^j$ if the edge is dotted. All tokens begin their execution path at v_0 at the start time of the token. Note that tokens move continuously along the edges and are not delayed at any vertex.

As illustration, consider the linear array shown in Figure 2, where v_0 is the host for the array and vertices v_1 to v_4 are processors, connected by the Euler circuit P. A token is available at any vertex v_i twice during its traversal of P. A type-1 token will be correctly available at v_i after traversing (h_1,..,h_i). Similarly, P' is a type-2 execution path, and a type-2 token would be correctly available at v_i after traversal of the edges (c_1,..,c_4,h_4,..,h_{i+1}). P and P' correspond

[a] This research was supported in part by the National Science Foundation under NSF Grant Number MCS-8104017 and by ONR Contract N00014-80-k-0987

0190-3918/83/0000/0458$01.00 © 1983 IEEE

naturally to the two ways of pipelining data through a linear array.

At every cycle each vertex has a set of tokens (one of each label) that are correctly available at that vertex. A computation by a vertex consists of a transformation of the values of the tokens as they pass through the vertex onto the next edge of their respective execution paths. The set of tokens which are correctly available at a vertex at any cycle are said to meet at that vertex.

Our notion of correctness of a computation is motivated by considering again the linear array of Fig. 2. Let dh and dc be the delays along the solid and dotted edges respectively. Consider a type-1 token that starts from the host v_0 at cycle 't'. It would be correctly available (and hence used for computation) at vertex v_i at cycle t+dh*i. Whenever the token is correctly available at a vertex it meets other tokens which are also correctly available at that vertex. The simultaneous arrival of these tokens at that vertex results in their values being correctly updated. On an arbitrary tree, the execution paths would be some other permutation of solid and dotted edges. We wish to ensure that all tokens which meet at some vertex in the linear array also meet at the same vertex in an arbitrary tree.

3. Conditions for Correct Execution

In this section we present the conditions under which the correctness criterion discussed above can be met. There are two cases to be considered: (a) when all tokens are of type-1 and (b) when some tokens are of type-1 and the others are of type-2. (These correspond to the notion of one and two way pipelining as in [3].) The condition for the first case is stated in the following theorem.

Theorem 1: If two type-1 tokens with labels 'j' and 'k' respectively meet at vertex v_i in a linear array, then they meet at v_i in an arbitrary tree iff $dc^j = dc^k$.

We illustrate the application of the theorem by designing a robust version of algorithm W2 in [3] for the convolution of two vectors.

Convolution Problem: Given a sequence of weights $(w_1, w_2, \ldots, w_k)$ and an input sequence $(x_1, x_2, \ldots, x_n)$, compute the output sequence $(y_1, y_2, \ldots, y_{n-k+1})$ where $y_i = \sum_{s=1}^{k} w_s * x_{i+s+1}$, $i=1,..,n-k+1$.

Solution: Consider k=5 and let the tree machine be G in Fig. 1. The weights $w_1,..,w_5$ are preloaded into vertices $v_1,..,v_5$ respectively. Tokens of the input sequence are labelled 'x' and those for the output sequence \`y\`. All tokens are type-1 and the delays for the solid edges are $dh^x=1$ and $dh^y=2$. The start time of token x_i, is 'i' and of token y_i is 'i-1'. The algorithm above is just solution W2 put in the notation of our model. To ensure that this algorithm operates correctly on any arbitrary tree, in particular that of Fig. 1, requires $dc^x = dc^y$, and we choose the minimum possible delay equal to 1. It can be verified that required pairs of input and output tokens meeet at the appropriate processors to compute the desired values of y_i.

In a linear array of size k, if both ends of the array are accessible, then the latency of the pipeline is $k*dh^y$. By traversing a closed path around the tree back to the host, the latency is increased to $k*(dh^y + dc^y)$. However, this is a constant irrespective of the tree and the throughput is exactly the same as that of a linear systolic array.

The second case to consider is that of two-way pipelining - i.e one token is of type-1 and the other is of type-2. In this case the condition for correct computation on arbitrary trees requires that the delays along the dotted edges be zero, which is impractical. However, we can simulate the effect of two way pipelining as follows. Firstly, constrain the numbering of vertices in the tree T so that it corresponds to that obtained by some depth-first traversal [4] of T starting from the root (Fig.3). Secondly modify the type-1 execution path so that a type-1 token now follows the shortest path between v_0 and the vertex v_i, for all i=1,..,n. This is accomplished by allowing multiple copies of the token to be simultaneously present at different vertices in the tree. When a token becomes available at some vertex, it is used for computation by that vertex, and copies of it are transmitted to all adjacent vertices in the tree, except the vertex from which the token was received. This modified type-1 execution path corresponds to the token being broadcast from the root to every vertex in the tree by a series of local broadcast steps. Fig. 3 shows a modified type-1 execution path; note that all edges are solid, and the delay along any edge in the path is dh^j for a j-token. We now state the conditions for correct execution of a two way pipelined algorithm on an arbitrary tree.

Theorem 2 Let a type-1 token with label 'k' and a type-2 token with label 'j' meet at some vertex v_i in a linear array. Then a sufficient condition for the two tokens to meet at v_i in an arbitrary tree T is: (1) The vertices of T be numbered by some depth-first traversal of T starting from the root, (2) The type-1 execution path be modified as above and (3) $dc^j = dh^k$.

We illustrate this result with a robust version of algorithm W1 in [3] that employs two-way pipelining, for the Convolution problem defined previously.

Consider the tree machine of Fig. 4 where the vertices are numbered by a depth-first ordering. The weight w_i is preloaded into vertex v_{6-i}.

Output tokens are of type-2 and follow the execution path P^y shown. Input tokens follow the modified type-1 execution path - i.e they are locally broadcast from the root v_0 to all the vertices of T. As in W1, the delays dh^y and dh^x are both equal to 1. The start time of each of the tokens x_i and y_i is 2(i-1). By Theorem 2, the delay dc^y must equal the delay dh^x: hence dc^y=1.

y_1 starts from v_0 at cycle 0, and is used for computation by vertices v_5 through v_1 at cycles 3,5,6,8 and 9 respectively. Consider x_2 which starts from v_0 at cycle 2. It traverses the three edges between v_0 and v_4 in 3 cycles, and thereby meets y_1 at cycle 5 as required. Another copy of x_2 will simultaneously be present at v_5 and will be updating y_2, which started at cycle 2, and is hence correctly available at v_5 at cycle 5.

This method of simulating a two way pipelined algorithm on an arbitrary tree may not be possible if the type-1 token changes its value as it passes through the array. However, for a large class of algorithms (for example see [5]) this method is directly applicable . There is no increase in either the latency or the throughput in executing the algorithm on an arbitrary tree rather than a linear array.

4. Discussion

The techniques discussed in this paper can be used to convert any linear systolic algorithm into an equivalent one that executes on an arbitrary tree of processors. From the viewpoint of wafer scale integration this is desirable since a tree can always be constructed on any connected set of fault-free processors that are obtained by discarding the faulty ones. The method can also be seen a technique in converting algorithms designed for a particular interconnection structure into an equivalent algorithm on a different interconnection structure.

References

1. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, (1980) .
2. H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI) ," Sparse Matrix Proceedings 1978, I.S Duff and G.W. Stewart, (editors), SIAM, (1979), pp. 256-282.
3. H.T. Kung, "Why Systolic Architecture," IEEE Computer 15(1), (January, 1982), pp. 37-46.
4. A.V. Aho, J.E. Hopcroft and J.D. Ullman, Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1976, pp.176-195.
5. P.J. Varman, I.V. Ramakrishnan and D.S. Fussell, Robust Matrix- Multiplication Algorithms for VLSI, Department of Computer Sciences, University of Texas at Austin, TR-221, March 1983.

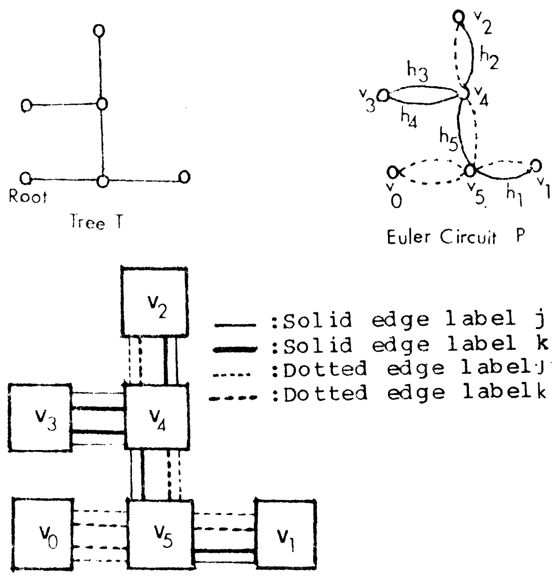


Fig.1: Construction of G for two labels.

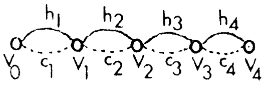


Fig. 2: Linear Array

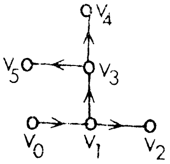


Fig. 3: Modified Type-1 Path.

P^y=(a,b,c,h_5,d, h_4,h_3,e,h_2,h_1)

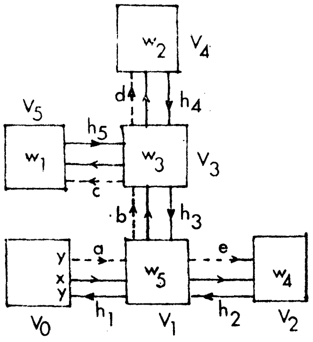


Fig.4 Convolution

STRUCTURED MEMORY ACCESS ARCHITECTURE

A. R. Pleszkun
Department of Computer Science
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706

E. S. Davidson
Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801

ABSTRACT

Conventional von Neumann architectures generate addresses for referencing memory by transferring addressing information from the memory to the CPU, by performing computations on addressing information, and by fetching and executing address bookkeeping instructions. Memory wait time is increased by computing operand addresses just before executing instructions which use those operands. These phenomena result in the von Neumann bottleneck at the CPU-memory interface. This work investigates one method of reducing the von Neumann bottleneck.

Program referencing behavior is determined by analyzing dynamic address traces. The Structured Memory Access (SMA) architecture developed in this work uses a computation processor (CP) and a decoupled memory access processor (MAP) with special hardware for efficient accessing of program and data structures and for effective branch and loop management.

Prototypical SMA machines are compared to conventional VAX-like machines. For a set of benchmark programs, the SMA machine reduces the number of memory references to between 1/5 and 2/5 of those required by a VAX. Actual performance is very sensitive to memory bandwidth and the amount of unoverlapped computation; however, SMA machines perform significantly better than conventional machines with the same parameter values. The SMA architecture reduces addressing overhead and improves system performance by (1) efficiently generating operand requests, (2) making fewer memory references, and (3) maximizing computation and access process overlap.

1. INTRODUCTION

This paper concerns the interactions between the central processing unit (CPU) and the memory of a computer system, modeled as shown in figure 1 [Hamm77]. Work performed by the CPU is partitioned into an access process and a computation process. The access process generates a stream of addresses for read and write requests to be serviced by the memory. Write data can originate from either process. The memory responds to read requests by generating a stream of data and instructions which returns to the CPU. Some portions of these data and instructions, returned to the access process, contain information for generating more references; the remaining portion is used by the computation process to produce its output data. In our view, the computation process performs the desired work of the system, while the work done by the access process represents overhead which should be reduced.

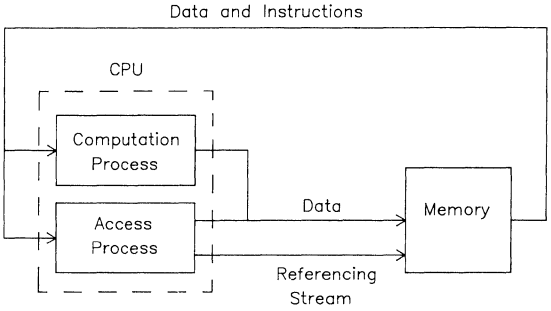


Figure 1. CPU-memory model

In conventional von Neumann architectures, the CPU interacts with only 1 memory, over 1 bus, and receives only 1 word per memory access. The computation and access processes compete for access to the memory over this single, narrow path, the so-called "von Neumann bottleneck."

A great deal of computing is performed solely for the generation of addresses. Hammerstrom [Hamm77] calculated the addressing overhead and the entropy of the stream of computation references by analyzing the address traces of several programs executed on an IBM 360. By measuring addressing overhead in bits input to the access process per computation process reference, he found that for a Gaussian elimination program and an eigenvalue-finding program, the addressing overhead was, respectively, 17.2 and 17.0 bits per computation reference. For a symbol manipulation program, the addressing overhead was 24.1 bits per computation process instruction fetch or memory data reference. These results represent a large percentage of the total number of bits input to the CPU from the memory.

The inefficiency of the conventional access process is exposed further when the addressing overhead is compared to the entropy of the stream of computation references. The entropy of the computation reference stream is likewise measured in bits per computation reference and is interpreted as the average number of bits needed to select among the possible successor references, i.e., to choose the particular next reference address given the current reference address. If the current and the possible successor reference addresses are

This work was supported by Navelex under contract N00039-80-C-0556: Reliable, High-Performance VHSIC Systems.

0190-3918/83/0000/0461$01.00 © 1983 IEEE

known, Hammerstrom found that for the programs he analyzed, between .845 and 1.86 bits of information per computation reference are needed to select among the possible successor reference addresses. These values can be treated as lower bounds on the number of bits which would be needed to specify a successor reference. Comparing these values to the addressing overhead, we find that they differ by at least an order of magnitude. Thus significantly more bits than necessary are being transferred between the memory and the access process during the execution of a program.

This access overhead and hence the von Neumann bottleneck can be reduced if the activities of the access process can be (1) modified to reduce the number of times the memory is accessed and (2) overlapped with those of the computation process. This paper introduces a Structured Memory Access (SMA) machine organization which includes mechanisms to take explicit advantage of a program's structure and of the regular patterns in which data structures are referenced. A more detailed presentation of the SMA architecture is found in [Ples82]. This architecture shares some goals and implementation characteristics with Smith's decoupled access/execute (DAE) architecture [Smit82], notably the decoupling of and separate processors for access and execution. The SMA, however, has explicit mechanisms for reducing the bookkeeping overhead for data and program structure references. SMA executes a single instruction stream, whereas DAE requires two.

The SMA "access mechanisms" eliminate most of the accessing overhead for well-structured computations. Previous attempts with conventional architectures, i.e., adding new address modes and including vector data types, have less effect. Use of cache memory actually increases accessing overhead both in time and in costly cache management hardware. Cache memory must be fast enough, and hence expensive, to overcome this overhead and provide improvement. By adding just one additional VLSI processor chip to run the access process, SMA achieves high performance with conventional slow memory. Effective access prediction tolerates slow memory if adequate interleaving is provided; eliminating overhead references can allow SMA to outperform a conventional processor with fast cache memory.

The SMA also includes a "loop mode" which eliminates the need to refetch instructions for each iteration of short loops. Finally, by physically separating the two processes into a memory access processor (MAP) and a computation processor (CP) and allowing them to be loosely coupled, memory wait time is reduced and significant process overlap is achieved. Compilation of SMA programs is straightforward and well suited to the capabilities of conventional compilers.

Little use is made of program structure to reduce addressing overhead in conventional machines. Our preliminary studies [Ples81] have indicated that a substantial amount of structure can be ascertained directly from a mechanical analysis of a program's address stream. Knowing, or at least accurately predicting, possible successor memory references is very important in achieving an efficient access process and can significantly reduce the addressing overhead of a program. Additionally, exploiting this predictability leads to a more nearly autonomous operation of the access process and the computation process, thus permitting an overlapped execution of the two processes and a reduction of memory wait time.

2. STRUCTURED MEMORY ACCESS MACHINE (SMA) ARCHITECTURE

From the analysis of program address traces, it is possible to determine the control and data structures of a program and the mechanisms by which data structures are accessed. Instructions can be divided into blocks such that blocks are entered only at the first instruction and execution always proceeds sequentially to the last instruction. A block may have one, two, or more successor blocks. The control structure of a program is well identified by a graph in which nodes are blocks and arcs point to successor blocks. This control structure can be found by mechanically analyzing the dynamic address trace of a program. Furthermore, if an instruction in a static listing always references the same memory location for its operand, e.g., with a direct addressing mode, that reference is called a scalar data reference; if more than one location is referenced by that operand reference as it recurs in a dynamic address trace, e.g., by using an index mode where the index value varies, the reference is called a data structure reference. This behavioral definition of scalar and structure tends to correspond in practice to common definitions. A sequential pattern of accessing through a data structure, as required for successive executions of a data structure reference, e.g., a row major scan of the upper triangle of a matrix, is called an access mechanism. Scalar data, data structures, and access mechanisms can likewise be found mechanically from trace analysis.

The Structured Memory Access (SMA) architecture uses this structural information to reduce access process overhead. Access process overhead exists in two forms. Address specification overhead refers to the increasing number of address bits needed to address a memory location as the address space becomes large. Most of these bits are redundant, given knowledge about possible address sequences. The second and more costly form of overhead, address calculation overhead, refers to address calculations explicitly performed by the CPU. Address calculation overhead involves some combination of extra instructions, parts of instructions, registers, memory accesses, and computation time. Both types of overhead are greatly reduced by the SMA architecture. Consider, for example, branch target addresses and operand references.

In the SMA machine, the complete branch target address is specified in a branch instruction. However, since the SMA machine provides instruction buffers to capture repeatedly executed instruction blocks, the number of times the branch instruction and its target address are accessed is reduced. The instruction buffer effectively limits the number of bits fetched from memory to specify branch target addresses.

To reference scalars, the SMA machine provides a base register. A scalar reference specification is simply an offset to be summed with the contents of the base register to form an entire scalar address. Traces we have analyzed reference few distinct scalars for the computation process and a few bits are sufficient for the offset.

The referencing of data structures is the prime cause of address calculation overhead. Conventional machines use bookkeeping scalars and instructions to manage iterative sequencing through data structures. To reduce this overhead, the SMA architecture uses special hardware to generate data structure references with minimal input to the access process from the memory.

The SMA machine implements the function of index registers by using a hardware stack. This stack tracks the active indices of inner loops during program execution, and all data structure references are made by using a subset of these index values. To reduce access process input, tables in the SMA processor are used to store the base address of each data structure and other information necessary to generate an entire address from indices. These tables must be loaded before any instruction which uses them is executed. Depending on the amount of hardware allocated for the tables, the number of data structures, and their access mechanisms, the tables may only have to be loaded once at the beginning of program execution. The tables may also be loaded during the execution of the program. A data structure reference specification is thus a set of pointers to table entries. Such a scheme provides the necessary flexibility for generating access mechanisms while maximizing the rate of address generation through the use of pipelining techniques.

Generally, the value of an index only needs to be associated with the access process. Thus, the stack containing indices, the tables for generating data structure references, and the address generation portion of the CPU may be separated from the computation-oriented portions of the CPU. This partition divides the computer system into two processors: a computation processor (CP) and a memory access processor (MAP). The CP is used strictly for the computation process, i.e., the useful computations of the system; while the MAP is responsible for the access process, i.e., generating all addresses for data and instructions. The index stack and the associated access tables mentioned above are kept in the MAP. Since only the MAP generates addresses, it controls all transactions with the memory.

There is no address bus between the CP and the memory since all memory requests are generated and controlled by the MAP. Also, since the CP is not responsible for addressing, the instructions sent to the CP contain no addressing information. Thus, the instructions are short and contain little more than opcodes and register tags. The CP is strictly devoted to performing computations and contains the ALU of the system; instructions and data are streamed into the CP by the MAP. The CP may receive entire blocks of instructions which it then holds in an internal instruction buffer. The CP may execute in loop mode by iterating over one or more blocks of instructions in its buffer without refetching. The CP also has a set of registers for holding the scalars used by an instruction block. The internal instruction buffer and the registers are provided to eliminate some repeated memory accessing and its associated time and load on the MAP.

The MAP, as shown in figure 2, has an internal Operand-Instruction Buffer (OIB) to hold its instructions and the operand specifications of CP instructions. The MAP can also operate in a loop mode fashion. Operation of the MAP is, to a great extent, independent of the CP. When the MAP begins receiving instructions, it forwards the MAP instructions and the operand specification portions of CP instructions to its OIB. The opcode and register tag portions of CP instructions are forwarded to the CP instruction buffer. The MAP immediately begins generation of operand addresses. The operand addresses are then placed on the read or write queue of outstanding memory requests. Write data is produced in the same order as corresponding write addresses. Thus when write data is produced, it is paired with the appropriate queued write address. As soon as a read request is serviced, the operand returned by that request is forwarded to the CP or to the MAP tables. With such a scheme, reads are performed early, writes are done late, the CP concentrates on the useful calculations of a program, and the MAP is left with the important, but overhead-related, generation of addresses.

The MAP accesses instruction blocks, scalars, and data structures with special hardware. Special instructions initialize and control the access mechanisms used for address generation. An SMA program thus contains a mixture of MAP and CP instructions. The data type of each operand is explicitly specified in each instruction. This extra information found in SMA instructions requires that the compiler must be capable of distinguishing loop control (index) branching from data dependent branching, and scalars from data structures.

Super computers and vector machines also contain special hardware for array referencing; however, the programming of these machines quite often requires rearranging of an algorithm to suit the hardware, and program compilation is difficult. Furthermore, their structured data access mechanisms are usually limited to a single vector of the structure at a time, i.e., "a constant stride" or constant step-size access mechanism with one index. Also, the same operation must be executed on each element of the vector, few vectors can be active at once, and access mechanisms are not easily suspended and resumed when more complex program loops are executed. The TI-ASC offers somewhat more flexibility by providing both inner and outer loop control for stepping though a matrix, i.e., two active indices.

The SMA machine provides more flexibility in the accessing of matrices since it offers more index levels by providing a stack on which to store indices. In our observations, even a 2-dimensional structure can require 3 levels of loop nesting for controlled rescan. Extra levels of nesting are also useful for providing nonconstant strides.

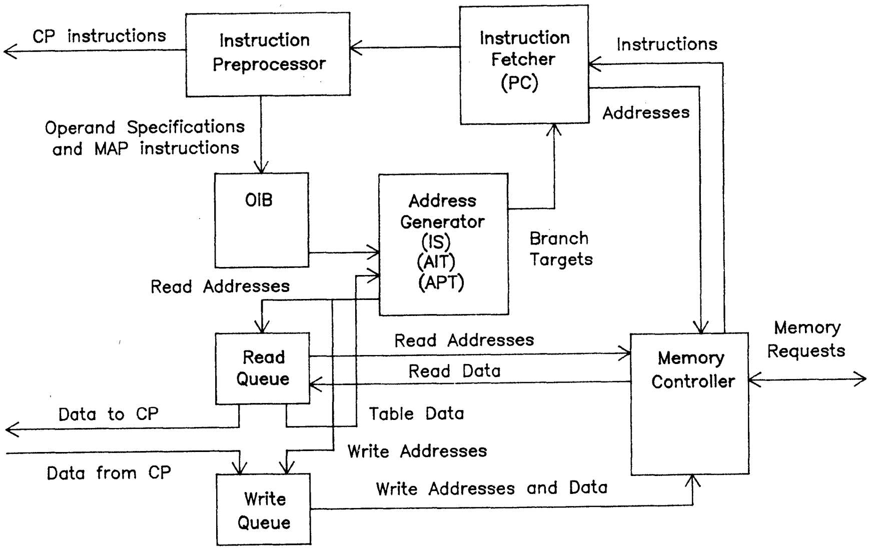


Figure 2. MAP internal organization

In vector machines, the vector access mechanisms are explicitly coded into instructions and then recognized and set up during execution time. The SMA architecture is designed so that data structure access mechanisms are recognized as early as possible. Some accesses mechanisms can be set up as early as compile time or load time. This early recognition can lead to reduced run-time overhead.

In conventional systems, the ALU makes branch decisions. In the SMA, two types of branch decisions are distinguished: decisions based on program data, which are made by the CP, and those based on indices used for referencing data, which are made by the MAP.

The SMA thus reduces the serial dependence which exists between the access process and the computation process. Since the MAP makes branch decisions based on index values during the execution of a loop, the MAP can generate memory requests for operands before the CP is ready to execute the instructions requiring those operands. In fact, the MAP should normally stay ahead of the CP so as to minimize the amount of time that the CP waits for data from memory. The MAP must wait for the CP when the MAP's read data queue is full or when the CP must resolve a computation-dependent branch. The CP must wait for the MAP when the MAP's write data queue is full, when the read data queue is empty, or when the CP instruction buffer does not contain the next instruction.

The SMA organization described above is used to reduce the addressing overhead primarily by improving the accessing of data structures through efficient access mechanisms and prefetching. The process of accessing instructions can likewise be improved if information concerning the instruction block structure of a program, which is apparent in high-level source code, is kept with the program as it is translated down to machine level. Retaining the block structure of a program can be used advantageously to cause the CP to enter and leave loops.

In conventional machines, loop mode control is generated dynamically during execution. Upon recognizing a short backwards branch, it is assumed that the second iteration of a loop is about to begin. The instructions of the loop are refetched and trapped in the loop buffer where they remain for repeated execution until the loop ending branch is unsuccessful.

The loop buffers in the CP and the MAP also trap loop instructions. However, loop mode control is set up at compile time. Loop structures are quite explicit and obvious in the high-level language source code available at compile time. If the instruction blocks which form the body of a loop are sufficiently short, they may all be stored in the instruction buffer at the same time. The processors thus are able to trap the body of a loop the first time the loop is executed. The loop buffers eliminate the need for repeated memory accesses for the same instructions during the execution of a loop. In any case, repetition requires no data-dependent branch and no wait time. Execution continuation after the loop is also efficient when the successor block is known, since it can be prefetched by the MAP during loop execution.

This structured prefetching by the MAP from a conventional slow memory with small processor buffers and without overhead references is felt to be superior to the unstructured use of a costly cache memory with attendant miss penalties, superfluity problems, and access overhead reference cycles.

3. AN SMA IMPLEMENTATION

3.1. Data Types

In this implementation, the SMA distinguishes among four types of operands: (1) immediate operands, (2) scalar operands, (3) data structure operands, and (4) index operands. Immediate operands are data whose values are embedded in an instruction. Scalar and data structure operands are defined as above. An index operand is one of the current indices found on the index stack. The index operand is used only to read a current index value from the index stack and transfer its value to the CP. An index operand differs from a scalar operand primarily in that the index operand originates from the MAP while the scalar operand originates from the memory. The operand type may be specified in a subfield of an instruction's operand field or it may be implicitly associated with a particular instruction. Additionally, an indirect addressing mode is provided specifically for use in the calling of subroutines and in the accessing of data items from structures such as linked lists. As with operand type specification, indirect addressing may be specified in a variety of ways.

At some time it may be desirable to distinguish explicitly among several types of data structures. Instead of having a data structure operand, one may wish to have an array operand, a linked list operand, a binary tree operand, etc. For each operand type, some special accessing mechanisms would be provided to improve the speed with which an operand address is generated. Accessing mechanisms as implemented allow the instruction code to reference structured data simply by pointing to the mechanism, which then references the next data in the established pattern.

3.2. Scalar Data

Scalar data is treated in the manner of a vector rather than as a set of disassociated items. The specification for a scalar operand includes a specification of a MAP base register and a displacement into a scalar data area in memory. The MAP can have more than one scalar base register to aid in the accessing of local and global variables, such as during subroutine calls. Such a base register can be used as the argument pointer set during a subroutine call. For the programs we studied, the number of simultaneously active scalars is relatively small, particularly in an SMA program.

3.3. Index Operations

The SMA's memory access processor has special mechanisms to track indices for data structure computations. These indices are used to generate the addresses for specific items of the data structure to be referenced. An index is specified by its current value, final value, step-size and indexing level. When the index is first established, the current value is equal to its initial value. At any time, several indices may be active; and the level, or nesting, of these indices is dictated by the time at which they were instantiated, or set up. In the SMA, the current value, final value, and step-size of an index are kept on a LIFO stack structure known as the _index stack_ (IS). Each stack position is numbered sequentially, with the bottom of the stack numbered level 1. This convention provides a convenient way of referring to the current value of any active index because the bottom of stack entry corresponds to the outermost level of nesting, i.e., level 1. Stack continuation in memory can be provided for overflow.

When a "setup index" instruction is executed by the MAP, the initial value, final value, and step-size are pushed onto the stack. To change the current value of an index, an "increment index" instruction is used. This instruction must specify three items: the level of the index which is being incremented, and two initial addresses of blocks which are the targets of a branch outcome. If the current value of the index is less than the final value, control is transferred to the first block which is specified. If, on the other hand, the current value equals or exceeds the final value, control is transferred to the second specified block.

By checking the index value early, during each increment index instruction, and by having the branching information available, the next instruction can start being accessed while the CP is still performing the final computations of a loop. Furthermore, no guess is made about which direction an index-based branch will take, thus no time is wasted in fetching potentially unnecessary blocks of instructions from the main memory.

When the current value of the index has reached its final value, that index should be at the top-of-stack and it is removed (popped) from the stack. Two other methods of removing indices from the stack are (1) the "remove index" instruction which removes the highest level current index from the top of the stack and (2) a "clear all indices" instruction which removes all indices from the stack.

To save loading of index instruction parameters from memory, The MAP is loaded with a set of templates for these values at the start of program execution. A template is a specification of the values needed to initialize an index on the index stack. Templates are loaded into an _index template table_. When an index is set up in the IS, the IS is loaded directly from the index template table. For a particular program, the number of distinct templates could be fairly small. For example, analysis of a Gaussian elimination program shows that 995 dynamic index setups are required, representing 16 static index setups, but only 8 templates are needed. Each index activated with a particular initial specification can use the same entry from the index template table. Even if the number of templates exceeds the table size, judicious reloading limits overhead.

3.4. Data Structure Accessing

To access data structures in the SMA, one must combine index values to form a data address. In the SMA, information for forming proper combinations is stored in two data tables within the MAP. As with some of the other repeatedly used information, the contents of the tables may be loaded when the program begins execution. The two tables are the _access pattern table_ (APT), which indicates the index levels to be used, and the _access information table_ (AIT), which contains information about data structures.

Each line of the APT is divided into several dimension fields. Each dimension field is divided into 2 subfields. The index level subfield indicates which level of the index stack (IS) is associated with that dimension field. The offset subfield contains the value of a small positive or negative offset to be added to the index before the index is used. This feature is useful since quite often the index of a data structure access is an existing presently active index, plus or minus a small constant. An entry in the APT may be used by more than one data structure since the information is not altered during execution and does not depend on accessing a specific data structure.

For each data structure currently being used by the program, there is an entry in the AIT. If the number of data structures in a program is sufficiently small, the AIT need only be loaded at the beginning of program execution. Each entry of the AIT is composed of three types of values: (1) the base address of the data structure, (2) a displacement for each dimension of the data structure, and (3) an upper bound for each dimension of the data structure.

A data structure reference may be made by specifying an entry in the APT and an entry in the AIT. A data structure address is generated by summing the base address in the AIT entry and the index terms for each dimension. Each term is formed by adding the offset in the APT entry to the index value identified by the level in the APT entry and multiplying by the displacement in the AIT entry. Bounds checking can be performed by comparing the index terms with the bound in the AIT entry.

While this computation may be tedious to perform for each data structure access, hardware must be present in the MAP to perform this computation at least occasionally. Once the hardware is present, it can be pipelined at little additional cost to allow the straightforward solution of performing this computation for every data structure reference. Pipelining allows a high rate of address generation; effective prediction overcomes the pipeline delay and allows the MAP to remain ahead of the CP.

3.5. Control Issues

The instruction fetcher in the MAP is responsible for generating instruction requests. The instruction fetcher sends the instructions it receives from the memory to the instruction preprocessor. The instruction preprocessor forwards portions of instructions to the CP with operand specifications replaced by buffer tags. MAP instructions and operand specifications are placed in the Operand-Instruction Buffer (OIB). The address generation unit steps through the MAP instructions and operand specifications found in the OIB, executes MAP instructions, generates operand addresses, and forwards each operand address to the read or write queue. When the memory returns the data associated with addresses in the read queue, that data is sent to a FIFO buffer in the CP. The CP sends data to the MAP for the write queue. Addresses in the write queue which have received their associated data are serviced by the memory.

The CP has an instruction buffer to hold the instructions it receives from the MAP. An execution unit in the CP steps through the instruction buffer, executing instructions one by one. If an instruction needs a data item from memory, that data is found at the head of the FIFO buffer. If the buffer is empty, execution is suspended until a data item is received from the MAP. Along with each data item, the CP receives an additional bit from the MAP which is used as an end-of-data signal. Assertion of the end-of-data signal in loop mode indicates that execution of the current instruction loop is to terminate and that the CP should begin execution of another block found in its instruction buffer, or wait until a new instruction block arrives from the MAP. The CP generates write data and signals the MAP regarding success or failure of CP tests for data-dependent branches performed in the MAP.

A program begins execution by having the monitor or operating system jump to the beginning of the program; that is, the operating system sets the program counter (PC) to the starting address of the program. In the SMA machine, the PC is located in the instruction fetcher of the MAP. When the PC is set to the beginning of the program, the instruction fetcher generates requests for instructions from the memory. Instruction requests are generated until the end of a block is encountered. If the instruction at the end of a block is a branch instruction, the instruction fetcher suspends operation until the branch is resolved. An end-of-block bit, attached to each instruction, indicates the last instruction of each block. The starting address of each block, as found in the PC, is saved to be used later when checking whether a block loops upon itself or branches to some other block in the OIB (and in the CP instruction buffer).

With the information stored in the OIB, the address generation unit can generate all the data requests required by a program. As the OIB is loaded, the address generator can begin executing MAP instructions and generating operand addresses by stepping through the entries of the OIB with its own internal program counter.

The addresses in the read and write queues are kept in the order that they were generated so that the CP receives read operands in the expected order and the MAP receives write data in the proper order. If write data is soon read back from memory, it is possible that the address of that data item will appear in both queues at the same time. Each time a read address is placed on the read queue, the write queue must be checked for an outstanding write to that address in order to prevent the reading of invalid data. If a match occurs, the read must not be permitted to occur before the write; otherwise, reads have priority over writes.

3.6. Branching

When the instruction fetcher of the MAP reaches the end of a block and the instruction is a branch, the instruction fetcher suspends further sequential instruction requests. If the branch depends on a condition in the CP, a signal must be received from the CP before the instruction fetcher and the address generator can resume operation. This signal indicates the success or failure of the branch. If the branch, however, depends on the value of an index in the index stack, the branch is resolved in the MAP. Thus, if the result of an index-dependent branch requires executing a new block of instructions, the instruction fetcher can begin fetching the instructions of the new block while the CP is performing calculations on the data for a previous block. The address generator can even begin making data requests for the new block while the previous block is still executing in the CP.

At any one time, the CP's instruction buffer may contain the CP instructions for more than one instruction block. The OIB in the MAP must, at the same time, be capable of holding the accessing information and MAP instructions corresponding to the instruction blocks in the CP buffer. The CP's instruction buffer and the MAP's OIB, while they hold information for the same number of blocks, are not necessarily the same size since corresponding CP and MAP blocks themselves differ in size. Monitoring the amount of information held by both the buffers is the responsibility of the instruction preprocessor since the instruction preprocessor fills the OIB and forwards CP instructions to the CP.

When a branch is resolved, there is a chance that the target block of the branch is already resident in the OIB and the CP's instruction buffer. The address generator checks for this situation by comparing the branch target address against the saved first address of each block currently found in the OIB. If there is a match, the address generator can immediately begin generation of data addresses for the new block. If, on the other hand, the information for the block is not in the OIB, the instruction fetcher is signaled by the address generator that a new block must be fetched. In such a case, the address generator must wait until new MAP instructions arrive in the OIB. When a branch is resolved, the MAP must signal the CP which one of the following three branch options the CP should take: (1) continue repeated execution of the currently executing block, (2) execute some other block found in the CP's instruction buffer, or (3) expect to receive a new instruction block from the MAP. When an entire block does not fit in the instruction buffer, it may be streamed through, but loop mode is not possible.

Normally, the CP is in a loop mode type of operation and expects a stream of data from the MAP. That is, if the end of the currently executing block is not a branch which depends on data in the CP, the execution unit of the CP will re-execute the currently executing block as long as the CP receives data from the MAP and the end-of-data signal is not set. This mode of operation is especially well-suited for executing an instruction block which operates on an array. Since the number of times such a loop is executed depends on the size of the array and the value of indices in the IS, branches will occur in the MAP based on values in the IS. The only effect these branches have on the CP is that data continues to be supplied to the CP until the loop terminates.

If the MAP determines that branch options 2 or 3 are to be followed, an active end-of-data flag is sent to the CP on the read data queue after the last data item associated with the currently executing block. The value of the data word sent with an active end-of-data signal informs the CP whether option 2 or option 3 is followed. One data value is reserved to indicate that the CP should expect a new block from the MAP (option 3). Any other data value is a pointer to a block in the CP's instruction buffer (option 2). Thus, program execution in the CP is controlled through the read data stream and the CP checks for the end-of-data signal on each read data queue access.

When the CP performs the test for a data-dependent branch, the MAP ceases prefetching data until the branch is resolved. This wait time incurred by the MAP is undesirable when such a test is executed frequently and a particular outcome is expected. Instead, the wait time could be used to prefetch the data for the likely branch target. The end-of-data signal provides a convenient way of disposing of data wrongly prefetched by the MAP. A reserved data value, sent with the end-of-data signal, could signal the CP to purge all buffered and incoming data until the next end-of-data signal. Such a reserved value would be written by the MAP into its read buffer whenever the MAP continued prefetching data and received a wrong-way branch indication from the CP. This signaling capability would be allowed only by special CP branch instructions whose opcodes would instruct the CP to purge data upon a wrong-way branch. All data in the CP read buffer is then purged up to the "purge" end-of-data signal and all following data is purged up to the next end-of-data signal. Prefetching instructions in such a case has no purge problem since the next end-of-data signal after the "purge" end-of-data signal indicates which instruction block to execute next.

The methods for communication between the MAP and the CP are designed to limit the number of interruptions in execution due to branching. Branches which depend on data in the MAP may occur many times without interrupting the operation of the CP; therefore, once the CP has a block of instructions in its buffer, the MAP can keep a stream of data flowing into the CP.

3.7. Subroutine Calls

The SMA uses a control stack for handling subroutine calls. A stack pointer, frame pointer, and an argument pointer are used as in the Digital Equipment VAX system. These pointers are maintained in the MAP, and MAP instructions are provided to access the pointers and to push and pop the SP.

4. SMA EVALUATION

The effectiveness of the SMA machine in reducing addressing overhead has been evaluated by comparing an SMA machine's performance to that of a VAX-like machine, primarily with respect to the execution of a Gaussian elimination algorithm (GAUSS). Some other evaluations are mentioned briefly. GAUSS, written in FORTRAN, is taken from [SSPP68]. From the high-level program source, the program is compiled into assembly language for a VAX running the UNIX operating system and for the example SMA machine. To compile the program into SMA assembly language, the VAX assembly listing is modified only with respect to the way data referencing occurs. That is, when a matrix is being accessed, SMA instructions are added to setup the indices for the matrix and to increment these indices. These SMA instructions, however, eliminate the need for some of the variables used and calculations performed by the VAX. Care is taken not to give either machine any special advantages. Thus, the code produced for the SMA by this transformation of VAX machine code is not hand optimized to any extent.

4.1. Number of Memory References Generated

A program's instruction blocks can be identified from the high-level source. Figure 3 is a diagram of the control flow for GAUSS in terms of instruction blocks. For GAUSS, only two of the branches are probabilistic in the sense that they are truly data dependent. Each of the other branches in the program are determined by the value of an index. These and the unconditional branches are handled very well by the MAP of the SMA machine.

The results of a static analysis of GAUSS are shown in Table 1. In the SMA version, GAUSS requires fewer than half the instructions needed in the VAX version. When counting the SMA instructions, MAP instructions are also included in the total number of instructions. The difference in the number of data references is as dramatic as the difference in the number of instructions. Since the VAX and the SMA versions of GAUSS make the same number of data structure references, the difference in data referencing is due to the scalar references. The SMA programs have fewer distinct scalars than the VAX programs due to overhead reduction; thus, the VAX program not only has more scalars but also performs overhead instructions to operate on these scalars.

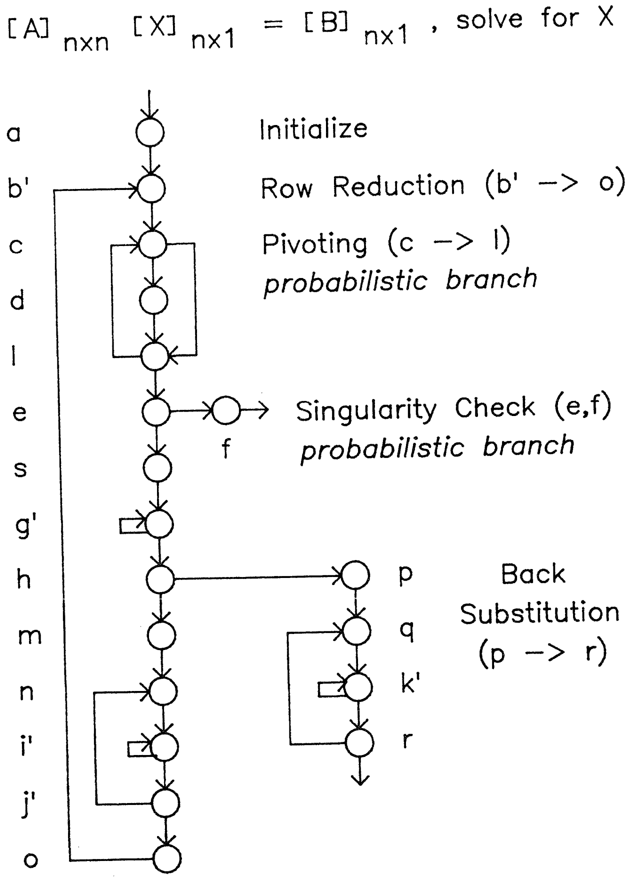


Figure 3. Instruction blocks for Gaussian elimination

Table 1. Statistics from a static analysis of GAUSS, EIGEN, and QSORT.

Number of	GAUSS		EIGEN		QSORT	
	VAX	SMA	VAX	SMA	VAX	SMA
instruction blocks	19	19	61	61	14	18
distinct scalars	16	6	36	23	8	8
distinct data structures	2	2	3	3	1	1
access patterns	11	11	19	19	1	1
instructions	123	50	534	251	68	59
data references	84	40	446	319	61	62
scalar	62	13	386	251	54	53
data structure	22	22	60	60	7	7
index	0	3	0	8	0	2

The static differences between the VAX and the SMA versions of GAUSS translate directly into substantial differences in the dynamic count of the number of memory references for each program. To obtain this dynamic count for GAUSS, the number of memory references generated by each block is calculated as a function of n, the matrix size. For data dependent branches, successors are chosen to produce a path with the largest number of instructions and data references. Thus in GAUSS, pivoting is always done and a singular matrix is not encountered. Therefore, this is a worst case dynamic memory reference analysis.

In this analysis, it is also desirable to see what effects loop mode has on the number of data references. Thus for each machine there are two cases: one with loop mode and one without loop mode. For GAUSS, blocks (n, i', j'), blocks (q, k', r), and blocks (c, d, l) are considered inner loops for loop mode execution. To hold each set of these blocks in a loop buffer, a hypothetical VAX with loop mode added would need to provide a buffer of 24 instructions, while the SMA needs a buffer of only 8 instructions.

Figure 4 shows a plot of the dynamic count of the total number of memory references required by the GAUSS program for a VAX and an SMA machine with and without loop mode as a function of n. The SMA machine always makes fewer memory references than the VAX, even if the VAX has a loop mode. The number of memory references needed by an SMA machine running the GAUSS program on a 100 x 100 matrix is only 20% of the number of memory references made by the VAX without loop mode. Thus the SMA with slow memory can easily outperform a VAX with a 1 clock cache cycle.

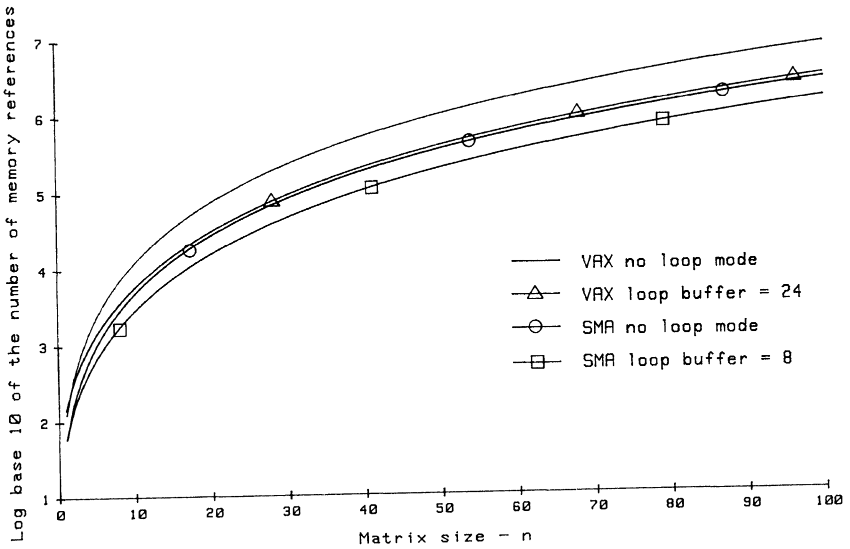


Figure 4. Log of the number of memory references for GAUSS for an nxn matrix

A similar analysis was performed on an eigenvalue-finding algorithm (EIGEN), and a quicksort algorithm (QSORT). EIGEN is written in FORTRAN and is the HQR routine from the Eispack subroutine package [Smit74]. QSORT is a recursive program written in PASCAL and based on an algorithm from Horowitz and Sahni [Horo76].

In a static count, the SMA version of EIGEN requires fewer than half the instructions and only approximately 75% of the data references of the VAX version. A dynamic analysis indicates the SMA machine without loop mode generates approximately the same number of memory references as the VAX with loop mode. For the EIGEN program operating on a 100 x 100 matrix, the SMA with and without loop mode makes only 30.5% and 47.2%, respectively, of the references made by the VAX without loop mode. In this analysis an "instruction" represents an opcode or a memory reference. Thus a VAX instruction with two memory reference operands would count as three of these simple instructions.

A static analysis of QSORT reveals little difference between the VAX and SMA versions. Nevertheless, the SMA machine reduces dynamic memory references for QSORT approximately as much as for EIGEN.

4.2. An Estimate of Relative Performance

A program's execution time can be partitioned into 1) time spent accessing memory and 2) time spent computing. The execution time is reduced by overlapping these two quantities. Generally, it is not possible to overlap all of the computation time with memory referencing activity. We call unoverlapped computation time the computational overhead. By allocating a portion of this computational overhead to each memory reference, the execution time of a program can be expressed as:

$$T = M\,(1/v + c)$$

where M is the number of memory references, c is computational overhead, and the term 1/v is the effective amount of time needed per memory access. The variable v is the memory bandwidth and is included as a parameter so that comparisons can be made between machines whose memory speeds differ. A larger v represents a faster memory and, therefore, a reduced memory access time. If the memory is interleaved, v takes the interleaving factor into account. The same algorithm executed on different machines will yield a different execution time because the term M will vary from machine to machine, as will the term c.

The computational overhead, c, is difficult to measure. It varies from one program to another and also from one machine to another. Due to machine dependencies, different models of even the same machine will have different values of c. The value of c is also a function of the memory bandwidth v. As memory access time decreases (increasing v), less computation time can be overlapped with memory accesses, causing c to increase. Due to these dependencies, c is treated as a parameter in our comparison of performance.

The performance of a machine is given by the inverse of the execution time. We calculated the performance of conventional machines and an SMA machine for c ranging from 0 to 2 and for v taking on values of 1, 2, 4, and 8. The computational overhead is in units of standard memory cycle times per memory reference, as is the term 1/v. The factor M is taken from the dynamic analysis of GAUSS run on a 100 x 100 matrix. To aid in comparing one machine with another, performance is normalized to the performance of a conventional machine with no computational overhead (c=0) and a memory bandwidth of one (v=1).

The normalized performance for GAUSS is shown in figure 5. Machines with and without loop mode are treated separately because the presence of loop mode affects the number of memory accesses required

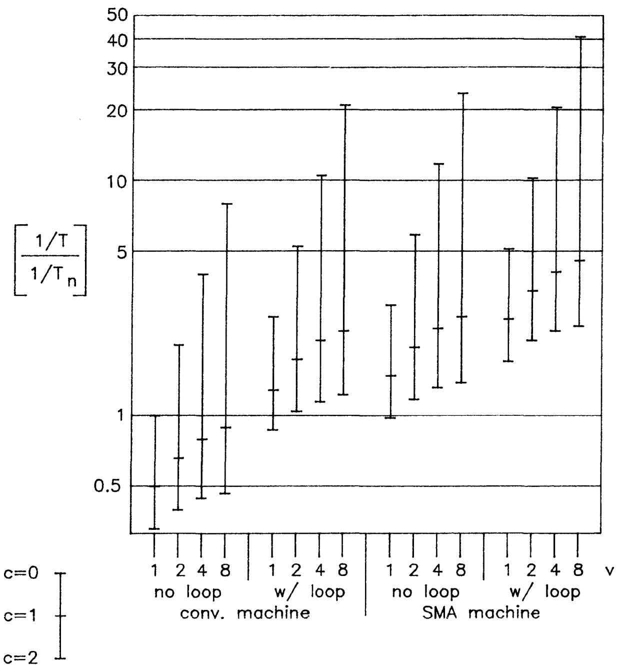


Figure 5. Normalized performance for GAUSS (v = memory bandwidth, c = computational overhead, T = run time)

by the program. Once loop mode is established, memory requests for the instructions of the loop are not needed.

Each vertical line of the graph represents the relative performance of a machine with a particular memory bandwidth and with the computational overhead ranging from 0 to 2. A conventional machine with no loop mode, v=1, and c=1 would require approximately twice as much time to run a Gaussian elimination program on a 100 x 100 matrix as the machine with c=0. At the other extreme, a c=0 SMA machine with loop mode and a memory bandwidth of 8 would perform approximately 42 times better than the base machine: conventional, no loop, v=1, and c=0.

The performance of the base machine was also compared to an SMA machine running EIGEN and QSORT. In this comparison, there is a great improvement in performance when an SMA machine is used; however, the improvement is not as dramatic as for GAUSS.

For all of the three programs and a given memory bandwidth, a conventional machine with loop mode and an SMA machine without loop mode perform almost equally. Furthermore, performance is sensitive to changes in computational overhead, especially when c varies from 0 to 1. Different machines should not simply be compared with the same value for c.

5. CONCLUSIONS

5.1. Summary of Results

Due to the von Neumann bottleneck, inefficiencies exist in the way address generation is performed in most conventional machines. The research presented here has studied the access process to discover where the access inefficiencies lie and how they can be reduced.

From a detailed analysis of program address traces [Ples82] we determined the types of features that should be included in a machine designed to generate memory references efficiently. The proposed Structured Memory Access (SMA) machine contains these features. The SMA machine is divided into a computation processor (CP) and a memory access processor (MAP). As their names imply, the CP is responsible for the desired computations of the system, while the MAP generates all the memory references for a program. The SMA machine reduces addressing overhead by providing special access mechanisms in the MAP to generate references efficiently for blocks of instructions and several data types. The storing of bounds information permits bounds checking to occur automatically in hardware when data structures are accessed. Because of the system's organization, the CP and MAP can operate relatively independently of one another. In particular, prefetching of instructions and data, and index-dependent loop control are inherent features of the SMA machine.

The operation of the MAP and its interactions with the CP were discussed as were the types of access mechanisms which reside in the MAP. The machine's ability to reduce addressing overhead was then evaluated. A comparison was made between a hypothetical SMA machine and a VAX-like machine with respect to the number of memory references generated by a set of programs. Depending on the program, the SMA machine reduces the number of memory references to between 1/5 and 2/5 of those required by a conventional VAX.

The performance of the SMA machine was then evaluated. A machine's performance was parameterized by the memory bandwidth and the computational overhead. It was found that performance is very sensitive to these parameters; however, an SMA machine performs significantly better than a conventional machine with the same parameters. Now that the SMA concept has been justified, more detailed performance evaluation and design modifications are being carried out in our continuing research.

REFERENCES

[Hamm77a] D. W. Hammerstrom and E. S. Davidson, "Information Content of CPU Memory Referencing Behavior," Fourth Annual Symposium on Computer Architecture, March 1977, pp. 184-192.

[Hamm77b] D. W. Hammerstrom, "Analysis of Memory Addressing Architecture," Tech. Report R-777, Coordinated Science Lab., Univ. of Illinois, Urbana, IL July 1977.

[Horo76] E. Horowitz and S. Sahni, The Fundamentals of Data Structures, Computer Science Press, Inc., 1976, p. 347.

[Ples81] A. R. Pleszkun, B. R. Rau, and E. S. Davidson, "An Address Prediction Mechanism for Reducing Processor-Memory Address Bandwidth," *Proc. 1981 IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Nov. 11, 1981, pp. 141-148.

[Ples82] A. R. Pleszkun, "A Structured Memory Access Architecture", Computer Systems Group Report CSG-10, Coordinated Science Lab., Univ. of Illinois, Urbana, IL Oct. 1982.

[Smit74] B. J. Smith, J. M. Boyle, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Lecture Notes in Computer Science, Volume 6: Matrix Eigensystem Routines - EISPACK Guide*, Springer-Verlag, 1974.

[Smit82] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *Ninth Annual Symp. on Computer Architecture*, April 1982, pp. 112-119.

[SSPP68] *1130 Scientific Subroutine Package Programmer's Manual*, International Business Machines Corp., 1968, p. 115.

A SIMPLE ARCHITECTURE FOR LOW LEVEL PARALLELISM

Charles E. McDowell

Electrical Engineering and Computer Science
University of California at San Diego
La Jolla, California 92Ø93

Abstract -- This report describes SIMAC, a new SImple Multiple Alu Computer for exploiting low level parallelism. The main feature of SIMAC is that all scheduling of operations to be executed in parallel is done at compile time. This results in much simpler hardware. Performance is increased because the overhead of scheduling is done only once and not repeated each time the program is executed. Preliminary results indicate that significant speedups are possible on both conventional sequential programs and on numerical array processing problems.

Introduction

Given a sequential program for a particular algorithm on a computer, there are three general approaches that can be taken to speed up the execution of the program. The execution speed of individual operations can be increased by using faster hardware, the program can be broken into two or more independent tasks which are run simultaneously on more than one processor (high level parallelism), or more than one simple operation can be performed simultaneously (low level parallelism).

In recent years there have been several studies investigating how much low level parallelism exists in algorithms expressed as computer programs. The results of the studies have varied considerably. Tjaden and Flynn[14] among others have measured the parallelism available within basic blocks. (A basic block is a sequence of instructions without any conditional jumps.) These studies have found a speedup of 2 to 3, where speedup is defined to be the sequential execution time divided by the parallel execution time. In a more recent study Nicolau and Fisher[9] report that by exploiting global parallelism, speedups approaching 1ØØØ are possible, and factors of 1Ø or more were found in most programs tested. The speedups reported by Fisher assumed unlimited hardware, but they still point out that there is a substantial amount of parallelism in typical programs that is currently not being exploited.

Pipelining is one method that is used widely today to exploit a small amount of the potential low level parallelism within a program. Except on large machines like the CDC 66ØØ or CRAY I, or special purpose vector processors like the IBM 3838, pipelined computers only exploit parallelism by overlapping the separate phases of the instruction fetch-decode-execute cycle. The pipelines of these large or special purpose machines are capable of pipelining the execution of various numerical computations. In general, pipelining is only used for the overlap of repeated executions of the same operation on different data. There is much more low level parallelism that is not handled by traditional pipelining techniques, and is only touched on slightly by some very large machines. SIMAC allows exploitation of more of this low level parallelism on a simple computer.

Another common form of low level parallelism found today is the parallelism found in horizontally microprogrammed computers. Unfortunately this cannot be easily used since it is in general below the level accessible by the user. Some work is being done in compiling high level languages directly to microcode[12] and in migrating common high level operations into microcode[6,1Ø] to attain speed improvements.

Data flow computers are another attempt to exploit low level parallelism. In general, data flow computers require very complex hardware to control the parallel operation of the multiple processing elements[15]. This is an active area of research.

0190-3918/83/0000/0472$01.00 © 1983 IEEE

There is a group of researchers today advocating simple or reduced instruction set computers[11,5]. This is in contrast to the more traditional trend toward more complex instruction set computers as can be seen in the DEC VAX11 or INTEL's iapx432. The development time for a new computer is directly proportional to the complexity of the architecture. In order to make maximum use of today's rapidly changing hardware technology, it is important to keep the development time short. RISC architectures are simpler and can therefore be implemented very rapidly using the latest in hardware technology. Another important factor in favor of RISC architectures is the large difference in communication speeds between VLSI chips versus the speeds within VLSI chips. With a RISC architecture it is possible to put one or more entire processors on a single chip greatly reducing the amount of communication off of the chip.

Multiprocessor systems are being developed that exploit high level parallelism by running separate tasks in parallel, each on a different uniprocessor. This still leaves open the opportunity for exploiting more low level parallelism in the uniprocessors that make up the multiprocessor systems.

SIMAC is a computer system which uses simple hardware and sophisticated compiler techniques to exploit the low level parallelism which exists in a wide range of programs. SIMAC is a single instruction stream multiple data stream computer. The processing elements (PEs) are identical and communicate through a shared memory and a set of shared registers. Each PE contains a simple ALU plus logic to issue reads and writes to memory. The PEs are controlled by a master control processor (CP) which fetches instructions from memory and issues instructions to the PEs. SIMAC differs from existing machines with low level parallelism (eg. CDC 6600/7600) in that all scheduling is done at compile time.

SIMAC Architecture

Low level operations are statically scheduled for each PE by software at compile time. These statically scheduled parallel instructions allow for parallel execution of low level operations on each processor. By moving the scheduling task into the software the hardware has been kept simple. Although hardware costs may be dropping, this does not necessarily imply that the hardware should be more complex, only that there should be more hardware. By keeping the architecture of SIMAC simple, the cost of the processor will be low and therefore many SIMAC processors could be combined into a larger multiprocessing system.

System Architecture

The SIMAC architecture is independent of the number of PEs. The number of PEs is limited only by the use of the registers and the bandwidth of the memory. There is nothing in any of the instructions that depends on the number of PEs. This is not the same as the ability to simply add PEs to the system at any time. For any particular implementation the number of PEs will be fixed. The implementation of the memory controller, register switch and CP all depend upon the number of PEs. A block diagram of the processor is shown in figure 1.

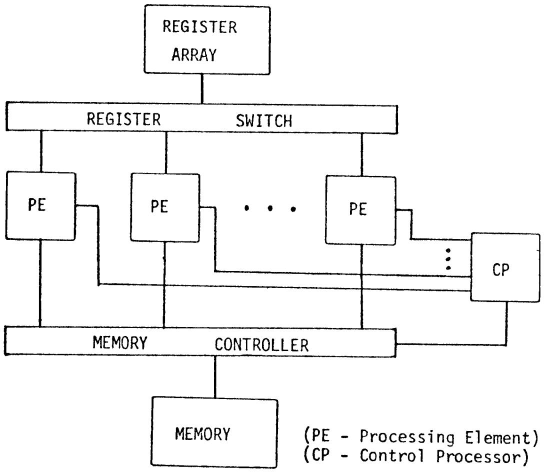


Figure 1. Block Diagram of SIMAC.

Each PE in SIMAC is a load/store RISC[11] style processor. Each PE is capable of executing a small set of simple operations (arithmetic, logical and shift, and test). All branching operations are processed by the CP. All MOPs (machine operations) are encoded in 32 bits and operate only on registers with the exception of load and store.

SIMAC has only one program counter (PC) maintained by the CP hence a single instruction stream (istream). However each instruction (PI or parallel instruction) may consist of up to n (the number of PEs) unrelated machine operations. Another way of viewing the multiple instruction capability is to think of this as a microprogrammed machine which is capable of accepting varying width horizontal instructions. The instruction width can be any multiple of 32 bits up to the maximum size determined by the number of PEs. The instruction width is controlled by a bit in each MOP.

The PEs have no local registers. There are 32 general purpose 32-bit registers that are shared by all PEs. Each register may be read by any or all PE's each cycle. Also each PE may write one register per cycle. It is the responsibility of the software to insure that there are no resource conflicts when more than one PE is active.

SIMAC is similar to a variable width, horizontally microprogrammed machine. With 1 PE active, SIMAC simply processes a single 32 bit MOP every cycle. With n>1 PEs active, SIMAC executes a single 32 bit MOP on each of the n PEs. Each PE gets a different 32 bit MOP. PEØ gets its MOP from M(PC), PE1 gets its MOP from M(PC+1), PE2 gets its MOP from M(PC+2) etc., and the PC is incremented by n. In this way n distinct parallel istreams are being executed with the non-trivial restriction that all n istreams must branch together. This is insured by the software which only allows one of the n MOPs for a single PI to contain a branch type MOP.

Each PE in SIMAC has two bits, T (test) and V (valid) that are used for branching. Whenever a PE executes a test instruction (eg. tlss, test less than and teq, test equal), the V bit in that PE is set. In addition the T bit is set if the test is true and cleared if false. The actual conditional branch is then executed by the CP when it encounters a branch instruction. There are two types of conditional branch instructions, BAND (branch anding) and BOR (branch oring). BAND and BOR instructions clear all V bits whether or not the branch is taken. A BAND branch is taken if the logical "and" of all test instructions executed since the last branch is true. A BOR branch is taken if the logical "or" of all test instructions executed since the last branch is true. More formally a BAND branch is taken if the following is true:

```
    (T1 or ~V1) and (T2 or ~V2) and ...
```

A BOR branch is taken if the following is true:

```
    (T1 and V1) or (T2 and V2) or ...
```

This allows for rapid, parallel evaluation of high level branches such as:

```
    IF ( A > B and C <= D ) then ...
```

Assuming A, B, C, and D are registers, then the above could be executed by the following two parallel instructions:

```
    PI1:  tgtr A,B  /  tleq C,D  / ...
    PI2:  band label / ...
```

By using boolean algebra, tests can be transformed into a sequence of comparisons "ored" together or a sequence of comparisons "anded" together.

Control Processor Architecture

The Control Processor (CP) contains the program counter and executes all branching type operations. The CP is also responsible for requesting the instructions from the memory controller and dispatching the MOPs to the PEs. The Control Processor (CP) is responsible for the following tasks:

- Maintain the program counter.

- Perform branch operations using the T and V bits from each PE.

- Fetch instructions from memory.

- Issue non-branch MOPs to the PEs.

Processing Element Architecture

Each PE is an extended ALU capable of doing register to register arithmetic and doing simple memory accesses. The instructions are sent to the PE from the CP.

The instructions are broken into five groups, simple register to register instructions (eg. add, shift), complex register to register instructions (ie. multiply, divide), load/store instructions, test, and miscellaneous. The miscellaneous, test, and simple register to register instructions all execute in one cycle, the number of cycles for the others will vary.

Bit 31 of each MOP, called the P bit, is used to pack MOPs into parallel instructions (PIs). All MOPs within a single PI are executed in parallel. The assignment of MOPs to PEs is determined by the position in the PI. The leftmost

MOP is executed by PEØ the next by PE1 etc. The last MOP of each PI may be a branch operation which is executed by the control processor. The P bit of the last MOP in each PI will be clear and the rest will be set. For example a 128-bit PI would look something like:

```
127 --- 95 --- 63 --- 31 --- Ø
  1 ...  1 ...  1 ...  Ø  ...
```

Register switch

The register switch provides simultaneous read access to all registers by all PEs and allows more than one PE to reference the same register. This can be implemented as a large selector switch. For a fixed number of registers the number of transistors for the switch grows linearly with the number of PEs. This would require approximately 12,ØØØ transistors per PE. Assuming VLSI chips with 25Ø,ØØØ transistors, the selector circuit for 4 PEs would utilize less than 2Ø% of the chip area.

Memory controller

The memory controller accepts data read/write requests from all PEs and instruction read requests from the CP. This is a potentially critical section of the overall system design due to the possibility that the memory bandwidth will be a bottleneck in the system. For this study we will assume that this is not a problem, and that with sufficient hardware a memory system can be built to supply the necessary bandwidth (eg. interleaved memory).

Compaction and Scheduling

Code generation for SIMAC can be divided into 3 components (not necessarily done in 3 separate phases), generation of sequential MOPs, compaction of MOPs into PIs, and ordering of MOPs within a PI. The need for the first 2 components should be clear. The third component is required to allow the execution of two PIs containing different length MOPs (eg. addition and multiplication) to overlap using only simple hardware controls (figure 2).

All MOPs within a single parallel instruction (PI) must begin execution in the same cycle, but they may end at different times. A simple mechanism is provided to allow for overlapping the execution of 2 PIs assuming both PIs do not use all PEs. A new PI will be issued by the CP as soon as all PEs for which the new PI contains an MOP are ready. This makes it possible for the software to schedule long MOPs on high numbered PEs and continue using the lower numbered PEs for short MOPs. As shown in the following example, by properly scheduling the MOPs into PEs, the two additions can overlap the execution of the multiply.

program segment

```
x=a*b
y=a+b+c
```

sequential code

```
R1 <- R2 * R3  ; MOP1
R4 <- R2 + R3  ; MOP2
R4 <- R4 + R5  ; MOP3
```

compacted code

```
PI1:  R1 <- R2 * R3 / R4 <- R2 + R3
PI2:  R4 <- R4 + R5
```

PE0	MOP2	MOP3			
PE1	MOP1	------	------	------	

Figure 2. Scheduling example.

Software Scheduling of MOPs into PIs is an instance of the processor scheduling problem. The problem can be described as follows: given a set of tasks t1,t2,...,tn and a partial order on those tasks which specifies that if task ti<tj, then ti must complete before tj may begin. Each task takes some length of time Ti to be processed and there are m identical processors, p1,p2,...,pm on which to process these tasks. A schedule is an assignment of the tasks to discrete time units such that:

1. No more than m tasks are assigned to any time unit (one for each processor).

2. If ti<tj, then ti is assigned to a time unit at least Ti time units before tj.

The problem is to find the schedule which completes all tasks in the shortest number of time units. This problem is NP-complete. The question then becomes, can a good approximation be found? Fortunately much work has been done in this area. Various "list scheduling" algorithms were compared in a report by Adam, Chandy and Dickson[1]. They found that HLFET (highest levels first, estimated times) was the best of those tested, and in many cases actually achieved the best known lower bound, that of Fernandez-Bussel[3]. Most cases tested came within 0.2 percent of the lower bound. In this method tasks are assigned priorities equal to the length of the longest chain from the task to the end. Tasks are then scheduled into time units from a list of data ready tasks with the highest priority data ready task being scheduled first.

Preliminary results

The portable C compiler[7] has been modified to generate sequential code for SIMAC. The output of the compiler is then compacted (scheduled) into parallel instructions. The optimization and compaction is done in a post pass and is currently 3000 lines of C. The dynamic performance of the sequential code is then compared with the parallel code using a simulator. The simulator is currently 1600 lines of C.

Tables 1 and 2 present the results of simulations run on several benchmarks. All times are reported in terms of speedup relative to the optimized sequential output of the portable C compiler.

Table 1 lists the speedups achieved using 4 processing elements. For these tests, it is assumed that memory reads and writes require 2 cycles, multiply and divide require 4 cycles and all other MOPs require only 1 cycle. The first two columns show speedups for local compaction and local plus global compaction. Column three shows speedups assuming that PIs with different length MOPs (figure 2) are not allowed to overlap. The last column shows the speedups assuming that all MOPs take only one cycle to execute.

Table 2 lists the speedups achieved using various numbers of processing elements. For all of the speedups in table 2, both global and local compaction were performed as well as overlapping PIs with different length MOPs.

Using 4 Processing Elements.				
Program	**Local Motion**	**Global Motion**	**No PI Overlap**	**Equal MOPs**
quicksort	1.6	1.9	1.6	1.9
bubblesort	1.8	2.4	1.9	2.4
testio	1.7	1.7	1.3	1.8
factorial	1.9	1.9	1.6	2.2
prime	1.3	1.4	1.4	1.7
puzzle	1.4	1.7	1.4	2.2
search	1.4	1.5	1.4	1.8
acker	2.3	2.4	2.0	2.6
LLL	2.2	2.2	1.4	2.6
matrix	2.6	2.6	1.8	3.0
array	2.1	2.2	2.4	1.9
MEAN	1.8	2.0	1.6	2.2

Table 1. Speedup with 4 PEs.

Program	2 PEs	3 PEs	4 PEs	6 PEs	8 PEs
quicksort	1.5	1.7	1.9	2.0	2.1
bubblesort	1.7	2.2	2.4	2.4	2.4
testio	1.6	1.7	1.7	1.8	1.8
factorial	1.6	1.7	1.9	2.0	2.0
prime	1.4	1.4	1.4	1.5	1.5
puzzle	1.5	1.7	1.7	1.7	1.7
search	1.4	1.4	1.5	1.5	1.5
acker	1.9	2.4	2.4	2.9	2.9
LLL	1.9	2.2	2.2	2.2	2.2
matrix	1.9	2.4	2.6	2.8	2.8
array	1.8	2.1	2.2	2.2	2.3
MEAN	1.6	1.9	2.0	2.0	2.1

Speedup with different numbers of PEs.
Table 2.

Conclusion

A new architecture for exploiting low level parallelism has been presented. The principle of parallel or horizontal control has proven itself to be very effective in the microprogrammed control of hardware. SIMAC is an innovative and useful extension of this principle to a higher level. SIMAC has increased the amount of parallelism that can be exploited in general-purpose programs using relatively simple hardware.

SIMAC moves the burden of scheduling from the hardware at execute time, to the software at compile time, which has resulted in greatly simplified hardware design, and reduced execution time, when compared to other machines with low level parallelism.

Due to the limited number of PEs, SIMAC will never be able to fully exploit the massive parallelism of vector operations, as is being done by some highly parallel array processors[2,4]. However SIMAC is capable of exploiting the parallelism in non-vector calculations, which has previously been done only on data flow machines and to a limited degree in the pipelines of some machines[13,8].

Acknowledgment

I would like to thank my advisor at UCSD, Professor Bill Appelbe, for his constant help. I would also like to thank Professor Dave Patterson of UC Berkeley for his help and comments.

References

1. Adam, T. L., Chandy, K. M., and Dickson, J. R., "A comparison of list schedules for parallel processing systems," *Comm. ACM*, Vol. 17, (12) pp. 685-690 (December 1974).

2. Batcher, K. E., "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, pp. 1-9 (September 1980).

3. Fernandez, E. B. and Bussel, B., "Bounds on the number of processors and time for multiprocessor optimal schedule," *IEEE Trans. Comp.* , Vol. c22, (8) pp. 745-751 (August 1973).

4. Haynes, L. S., Lau, R. L., Siewiorek, D. P., and Mizell, D. W., "A Survey of Highly Parallel Computing," *Computer*, Vol. 15, (1) pp. 9-24 (January 1982).

5. Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., and Gill, J., "MIPS: A Microprocessor Architecture," *Micro 15*, pp. 17-22 (October 1982).

6. Holtkamp, B. and Kaestner, H., "A Firmware Monitor to Support Vertical Migration Decisions in the UNIX Operating System," *15th Annual Microprogramming Workshop*, pp. 153-162 (October 1982).

7. Johnson, S. C., "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104 (January 1978).

8. Kogge, P. M., *The Architecture of Pipelined Computers*, McGraw-Hill (1981).

9. Nicolau, A. and Fisher, J. A., "Using an Oracle to Measure Potential Parallelism in Single Instruction Stream Programs," *14th Annual Microprogramming Workshop*, pp. 171-182 (October 1981).

10. Olbert, A. G., "Crossing the Machine Interface," *15th Annual Microprogramming Workshop*, pp. 163-170 (October 1982).

11. Patterson, D. A. and Sequin, C. H., "RISC I: A Reduced Instruction Set VLSI Computer," *Eighth Annual Symposium on Computer Architecture*, pp. 443-457 (May 1981).

12. Sint, M., "A Survey of High Level Microprogramming Languages," *Micro14*, pp. 141-153 (November 1980).

13. Thornton, J. E., *Design of a Computer - The Control Data 6600*, Scott, Foresman and Co., Glenview, Ill. (1970).

14. Tjaden, G. S. and Flynn, M. J., "Detection and Parallel Execution of Independent Instructions," *IEEE Trans. Comp.*, Vol. 19, (10) pp. 889-895 (October 1970).

15. Watson, I. and Gurd, J., "A Practical Data Flow Computer," *Computer*, pp. 51-57 (February 1982).

HIERARCHICAL MICRO-ARCHITECTURES OF A TWO-LEVEL MICROPROGRAMMED MULTIPROCESSOR COMPUTER

Takanobu Baba*, Katsuhiro Yamazaki, Nobuyuki Hashimoto,
Hiroyuki Kanai, Kenzo Okuda, and Kazuhiko Hashimoto**

Department of Information Science
Utsunomiya University
Utsunomiya 321, Japan

Abstract -- Hierarchical micro-architectures have been designed and developed for a two-level microprogrammed multiprocessor computer, MUNAP. In the MUNAP, a 28-bit microinstruction simultaneously drives several nanoprogram streams of 40-bit nanoinstructions in four 16-bit processor units.

On the basic microinstruction and nanoinstruction sets, the micro-assembly language level (Level 1) architecture has been defined to allow the user to describe frequently used micro-nano combinations and microprogram level SIMD operations in one statement. Experimental results show that the description of parallel processing is divided into (i) 33.3 % micro-nano combined statements, (ii) 41.9 % parallel nanostatements for the SIMD operations, and (iii) 24.8 % different nanostatements for the MIMD operations.

The system description language level (Level 2) architecture is defined on the Level 1 primitives. This level has many features common to other system description languages on such items as data types, operators, and functions, and some MUNAP dependent features. The experimental results show that the average processor utilization for each machine cycle varies from 1.1 to 3.8 within 4 processor units depending up on the content of processing.

1. Introduction

A significant trend in computing system design is the implementation of a computer that can operate on a wide variety of problems with high efficiency [5, 9]. We have developed a research-oriented multiprocessor computer, MUNAP (MUlti-NAnoProgram machine), as a vehicle for solving a wide range of nonnumeric and associated problems [1, 4]. The design objectives of the MUNAP system are: 1) the system should support basic functions for a wide range of nonnumeric processing through firmware and hardware; and 2) as a research vehicle, the system should provide a powerful, yet flexible, architecture. In order to attain these objectives, we have designed a new, multiprocessor computer architecture, controlled via a two-level microprogramming scheme. Architecture comparisons of MUNAP with other possible architectures, such as single-level versus two-level controls, and multiprocessor versus uniprocessor, verified the advantages of MUNAP for such parameters as control storage requirements, execution steps, and multiprocessor unit parallelism [4].

The key for utilizing such an innovative machine efficiently is to provide the users with good architectural views [9]. The difficulty is that we must define an architecture which not only aids the programming process but also utilizes the basic hardware features, such as a parallelism among multiple processors, a two-level microprogrammed control, and nonnumeric processing units. The higher we define the architecture, the more difficult it is for the user to use the hardware features efficiently. The lower we define it, the more difficult the programming process is because of its innovative hardware organization. In ordinary machines, the machine language is defined as an interface between software and firmware (or hardware). This makes it difficult to utilize the micro-architecture level parallelism, because the level of machine language architecture is too low and too rigid to utilize such parallelism. To solve the problem, we have designed and developed hierarchical micro-architectures; the micro-assembly language level architecture [2] and the system description language level architecture. In the former architecture, the user can describe concurrent operations in one statement. The operations include the micro-nano combined operation and the micro-level SIMD operation. In the latter architecture, the macro functions are directly implemented in the micro-assembly language to provide a relatively high-level architecture that implicitly contains parallelism of multiple processors and nonnumeric processing functions. These hierarchical architectures have been provided in MUNAP for users who have various system requirements. Several experiments have been made to clarify the effect of the hierarchical micro-architectures.

In this paper, we will describe each architecture on such items as design objectives, language features, processing, and experimental results. Based on the results, we consider the effect of multiple micro-architectures for providing the user "easy to use" and "efficient" interfaces, based on a multiprocessor computer with innovative hardware organization. The experimental results also show how many processors are active on average when we apply multiple processors to ordinary (not parallel) problems.

*Visiting at the College of Business and Management, University of Maryland, College Park, MD 20742
**Hitachi Software Engineering, Japan

0190-3918/83/0000/0478$01.00 © 1983 IEEE

2. Basic Concepts of Hierarchical Micro-architectures

2.1 Hierarchical micro-architectures

At the initial stage of hardware development for MUNAP, we were required to do hand-coding for checking the hardware by running test microprograms. The experience taught us that it is very difficult to describe a microprogram with multiple nanoprograms. The nanoprogram level MIMD feature makes the process more complicated. At times, we had to write 28-bit microinstruction with 4 40-bit (i.e. 160-bit) nanoinstructions to specify the control for one machine cycle. Based on the experience, we decided to develop an architecture that is not only easy-to-use but efficient. To satisfy these contradictory requirements, the hierarchical micro-architectures have been developed.

Figure 1 shows the basic idea of hierarchical micro-architectures. At the lowest level, the multiple nanoinstruction sets are defined for multiple processors. The meaning of a microinstruction is partially determined by the nanoprograms. The microinstruction set is defined on the nanolevel architecture. This micro-level instruction set, combined with the nanoinstruction sets, represents the visible micro-architecture of the machine (Level 0). On the Level 0 architecture, we define a higher level, enhanced view of the machine to make it easier to develop microprograms by utilizing symbolic expressions for arithmetic operations and sequencing, and providing the user a facility for describing the combined micro-nano operations in one statement (Level 1). The key for defining the Level 1 architecture is that it should not lose the flexibility for specifying micro-nano combinations and the parallelism among multiple processors. When we met a decision point for designing the language, that is, easy-to-use or efficient, we chose efficiency or prepared two types of expressions, one for efficiency and the other for ease of use for the same operation (examples are in later section 3.2). At the next level, a higher view is defined as a system description language level (Level 2). This level provides problem solving capabilities by including several data types, operators, and functions, as in usual system description languages [6]. Further, the tagged architecture is defined to aid the user′s debugging process and realizing dynamic data type transformation. An additional layer to be considered is derived from the varieties of user specialties that may surround the system (Level 3). By using the Level 2 facilities, we can describe a high level language processor, such as the PASCAL compiler for Level 3.

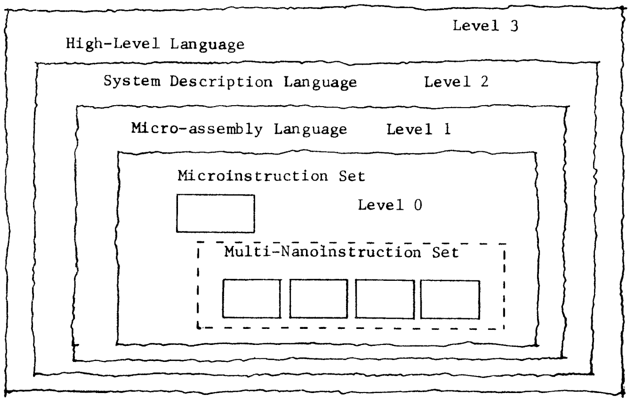


Fig. 1 Basic concepts of hierarchical micro-architectures.

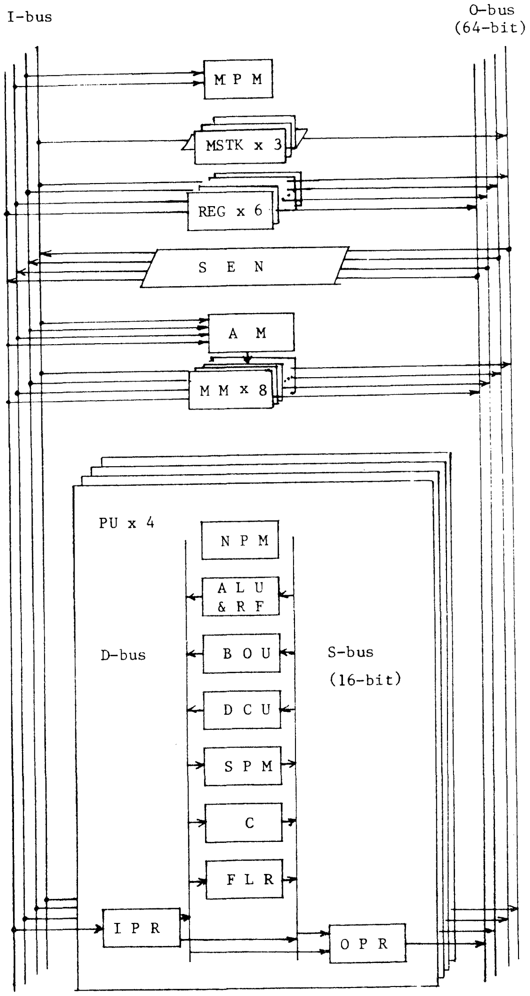


Fig. 2 MUNAP hardware organization.

2.2 Hardware organization

The basic hardware organization of MUNAP, that supports the above architecture, is outlined here along with the basic micro-nano interaction mechanism, and some software and hardware support systems, developed on the console processor of MUNAP. Figure 2 outlines the data flow of MUNAP. There are one microprogram memory (MPM) and four nanoprogram memories (NPM) within four identically constructed processor units (PU). A 28-bit microinstruction (μI) simultaneously drives several nanoprogram streams in the four 16-bit processor units. A 40-bit nanoinstruction (nI) has a 1-bit field to specify the end of a nanoprogram at each execution step. When all the nanoprograms end, the next microinstruction is activated. Since the nanoprogram memory is also reloadable as the microprogram memory, this allows the user to specify any combination of nanoinstructions in multiprocessor units.

Several hardware units are distributed among the microprogram level and nanoprogram level. The microprogram-controlled units are the four levels of 16-number, 4-bit segment shuffle exchange network (SEN) with exchange and broadcast cells for interconnection between processors and main memories and for data permutation [7, 8], and the 8 banks of main memory modules (MM) with address modifier (AM) for variable length word access and two dimensional table access. The nanoprogram controlled units are arithmetic and logic unit (ALU), bit operation unit (BOU) for bit count, bit test and priority encode, and field division and concatenation unit (DCU). The other units include the micro stacks (MSTK) and general registers (REG) at the micro level, and the register file (RF), scratchpad memory (SPM), counter (C), flag register (FLR), and port registers (IPR and OPR) at the multi-nano level.

The ECLIPSE S/130 minicomputer is attached to MUNAP as a console processor. It allows the user to read/write the content of MUNAP facilities and start/stop the microprograms. The loader of two-level microprograms and the evaluater for getting the run-time data are also available on the ECLIPSE. MUNAP has been designed and constructed with about 2500 IC´s at our laboratory. For the detail of MUNAP organization, see [4].

In the following sections, we will concentrate our discussion on the design objectives and the language processing for Levels 1 and 2. The experimental results are also shown to verify the effectiveness of the design.

3. Micro-assembly Language Level Architecture

3.1 Design objectives

In order to obtain an efficient microprogram, allowing the user to utilize the hardware features, the register-transfer-level language has been designed and implemented [3]. (The later Figure 5(b) shows a sample description.) The language features are summarized in the following two types:

(1) Description of two-level microprograms: Basically, the user can describe any combination of a microinstruction and multinanoprogram, activated by the microinstruction, in order to efficiently make use of the flexibility of two-level microprograms. The microprogram and nanoprograms are described in a sequence and distinguished by indentation. At this level, the hardware functions are uniform and frequently used functions may be described in one statement without being aware of the operations of elementary micro- and nano-instructions. The following Examples 1 and 2 illustrate the uniformity and one statement description, respectively.

(Example 1) *IF POS2 = 1 THEN GOTO L0;

At the Level 0, there is no flag that indicates the data is positive. This is to avoid the redundancy of hardware that has test functions for zero and negative data. However, this causes inconvenience for the user. The virtual flag, named POS, is defined at the Level 1 to make the test functions uniform.

(Example 2) SPM 0,1(5) := RF 2,3(6);

The contents of register file RF(6) are read in parallel from processor units 2 and 3, and sent to SPM in processors 0 and 1 through the shuffle exchange network (SEN). This statement is decomposed into 1 microinstruction and 4 nanoinstructions as described later.

(2) Description of parallel processing: The parallel processing of MUNAP is divided into three categories: (i) microinstruction and multiple nanoinstructions are tightly coupled to perform a single task; (ii) SIMD operation, in which multiple PU´s do the same operation; and (iii) MIMD operation. Example 2 is an example of (i). Although the MIMD operation should be described in several statements, the SIMD operation may be described in one statement. Examples 3 and 4 show the examples for MIMD and SIMD operations, respectively.

```
(Example 3)  SPM0(A) := RF0(B) <+> 1;
             SPM1(A) := RF1(B) <+> 2;
             SPM2(A) := RF2(B) <+> 3;
```

The statements represent the different (i.e., MIMD) operations in PU 0, 1, and 2.

(Example 4) SPM 0,1,2(A) := RF 0,1,2(B) <+> 4;

This statement implies parallel additions in the PU 0, 1, and 2 by three nanoinstructions.

3.2 Decomposition into two-level microprograms

The language processor has been developed in PL/I consisting of 6200 statements. The major features of the translation are the following.

1) divide a micro-nano combined statement into a microinstruction and nanoinstructions, and assign the nanoinstructions to appropriate PU´s;

2) extend a statement for multiple PU´s to several statements and assign them to appropriate PU´s; and

3) optimize the two-level microprograms.

We will illustrate about 1) and 2) by using the sample translations. The optimization conditions and its implementation issue are found in [3].

(Example 5) Translation of the Example 1 statement as an example of feature 1).

To explain the translation process, the mechanism for reflecting the parallel test results in multiple processors to the micro-level sequencing is shown in Figure 3. In each PU, a nanoinstruction selects 2 flags from the 32-bit flag register (FLR), does logical operations (f0) on the 2 flags, and sets the result to the TEST flag for each PU. Another microinstruction does logical operations (fl) on the 4-bit TEST flag, and the result is used for a branch condition at the microprogram level. Thus, the IF statement of Example 1 is decomposed as follows:

```
m: IF TEST2 = 1 THEN GOTO L0;
n2: IF NEG2 = 1 NOR DZ2 = 1 THEN SET TEST2;
```

Notice that m represents microinstruction and ni represents nanoinstruction of PUi. We show the contents of each object micro- or nano-instruction not in a bit pattern format but in the statement format to aid the readability. Notice also that n2 is executed before m according to the timing constraints [4]. Thus, the "negative or" operation (NOR) for the negative flag (NEG) and zero flag (DZ) of PU2 realizes the virtual positive flag (POS).

(Example 6) Translation of the Example 2 statement as an example of features 1) and 2).

The statement is decomposed as follows.

```
m: OPR -> <32-bit shift by the SEN> -> IPR

n2: RF2(6) -> OPR2       n0: IPRO -> SPMO(5)
n3: RF3(6) -> OPR3       nl: IPRl -> SPMl(5)
```

The IPRi, OPRj represent the port registers for input to PUi and output from PUj, respectively (see Figure 2). The nanoinstructions n2 and n3 read out data from the register file to the OPR of PU´s 2 and 3. The microinstruction m controls transfer from the OPR 2,3 to the IPR 0,1, through the SEN. The n0 and nl write the data on the IPR 0,1 into the scratchpad memory.

Notice that the system allows the user to use not only the simple, one statement description, such as Example 1, but also the direct description of Level 0 operations, such as Example 5. These options are the answer to the contradictory requirements of "ease of use" and "efficiency" for language design, as described in 2.1.

3.3 Experimental results

We performed an experiment in order to evaluate the effectiveness of the Level 1 architecture on such items as (1) the way in which the two-level microprograms are described by using micro-nano combined statements, (2) the way in which the parallel operations are described by using micro-nano combined statements and parallel nanostatements, and what the degree of parallelism is, and (3) the effectiveness of translation process for supporting high-level description. Ten problems were given to the members of our laboratory who were knowledgable about the micro-assembly language and support system on the console processor. The results are shown in Table 1. Notice that the capital letter in paranthesis in the following description corresponds to an item in Table 1.

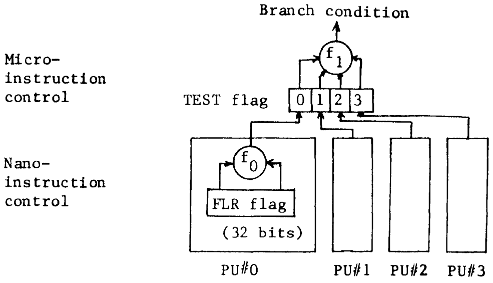


Fig.3 Parallel test by multiple processors.

(1) Description of two-level microprograms: Microinstructions are described in micro-statements(B), which do not include nano-level operations, and micro-nano statements(D). The micro-nano statements(D) account for 33.1 % of the total microinstructions(B+D) used in the microprograms. This significant usage demonstrates the effectiveness of the micro-nano combined statements for describing tightly coupled micro-nano operations.

(2) Description of parallel operations: The description of parallel processing is 27.0 % of total descriptions. (This percentage came from the careful reading of all the object microprograms and not expricitly shown in Table 1.)

Table 1 Experiment results at micro-assembly language level

Microprogram Number			1	2	3	4	5	6	7	8	9	10
Number of Statements		(A)	106	138	179	134	144	61	145	187	159	95
Micro	1 μ - 0 n	(B)	35	35	76	38	51	20	35	64	33	23
Nano	0 μ - 1 n		33	79	61	55	32	30	37	78	102	59
	0 μ - 2 n	(C)	4	12	0	0	6	3	6	0	0	0
	0 μ - 3 n		0	0	0	0	3	1	5	2	0	0
	0 μ - 4 n		18	0	4	7	30	0	37	21	6	3
Micro-Nano	1 μ - 1 n		9	5	11	19	13	1	6	9	2	2
	1 μ - 2 n	(D)	0	4	0	11	0	0	6	1	0	0
	1 μ - 3 n		0	0	0	0	0	2	1	0	0	0
	1 μ - 4 n		7	3	27	4	9	4	11	12	16	8
Number of Nanoinstructions after Decomposition												
	PU0		63	7	93	87	54	35	88	75	53	30
	PU1	(E)	31	77	34	18	58	7	68	60	56	31
	PU2		28	41	34	17	55	7	69	47	43	23
	PU3		28	3	35	20	54	10	51	45	40	21
Total		(F)	150	128	196	142	221	59	276	227	192	105

μ: microinstruction
n: nanoinstruction

According to the classification of 3.1 (2), we can classify them into three categories: (i) micro-nano combined statements (33.3 %), (ii) parallel nano statements (41.9 %), and (iii) different nano statements (24.8 %). The item (i) corresponds to (D) and items (ii) and (iii) are included in (C). These results show the effectiveness of our approach for describing parallel processing by the micro-nano combined statements and the parallel nano statements for the SIMD operations. The items (i) and (ii) are further divided according to the number of PUs used. In item (i), the ratio between 1 micro - 2 nano (i.e., 1 microinstruction activates 2 PUs at the same time), 1 micro - 3 nano, and 1 micro - 4 nano is 7:1:34 (see (D)). In item (ii), the ratio between 2 nano, 3 nano, and 4 nano is 3:1:10. These results show that 4 PUs are effectively utilized for various problems.

(3) Translation of two-level microprograms: The number of nanoinstructions(F) after decomposition is 2.3 times that of explicitly described nanoinstructions(C). This is due to (i) the implicit description in the micro-nano combined statements, and (ii) the parallel nano statements that allow the user to describe parallel operations in one statement. These results show the effect of high-level description at Level 1. The item (E), the number of nanoinstructions for each PU after decomposition, shows the locality of multiple processor usage. In some problems, for example, microprograms 2, 3, 4, and 6, the locality is evident.

4. System Description Language Level Architecture

4.1 Design objectives and language features

The micro-assembly language has provided the relatively high-level micro-architecture to the user. However, it is still difficult to describe large, utility programs or application programs in such a language.

The following objectives are defined for designing the MUNAP System Description Language (MSDL):

(1) Definition of high-level architecture: On the Level 1 architecture, we defined a rich set of data types, operators, and functions. Included are, in particular, data types of two dimensional array and structure, control statements of IF, WHILE, FOR, and SWITCH, opeartors of data exchange and concatenate, and functions for bit and string operations.

(2) Utilization of hardware features: To utilize the hardware features of MUNAP, we represent some of them in the language. Examples are string functions for nonnumeric function units such as BOU and DCU, and shift and exchange operators for the SEN. Some of the data types, such as flag, are also represented. This is a compromise between the high-level, problem-oriented architecture and low level hardware organizations.

(3) Tagged architecture: The goals of an effective computer architecture are not only the efficient processing of large amounts of data but also the enhancement of the debugging process and the enhancement of reliability of the computing system [5, 9]. Higher processing capability, obtained by the parallel processing, should be applied not only for processing large amounts of data at high speed but also for improving the user interface by semantic checking during program execution. To implement these concepts at the system description level, we designed the tagged architecture. This architecture is expected to provide the facilities for (i) detecting several kinds of errors at run time such as refering to an unassigned data value, and (ii) automatically transforming the data types of operands. These two items aid the development process of programs.

4.2 Parallel processing of MUNAP System Description Language (MSDL)

Basically, the source program written in MSDL is translated into an intermediate form by the host processor ECLIPSE, and then interpretively executed by MUNAP. In order to make use of the MUNAP micro-architecture features, the intermediate language has one to one correspondence with the MSDL source statement. The translator and the interpreter are described in the ECLIPSE assembly language and the MUNAP micro-assembly language (i.e.,the Level 1 language), respectively. The interpreter consists of 3.2 K microinstructions and 7.6 K (1.9 K for each PU) nanoinstructions.

We will concentrate our discussions on the processing features supported by the parallelism and nonnumeric functions of MUNAP. For the detail of intermediate language formats and the processing, see [10].

(1) Parallel processing for operators: The arithmetic and logical operations are executed in parallel in the four PUs for each operator of MSDL. The type check for operands and, if necessary, the translation to an appropriate type are also done dynamically. The information about the result is stored in the tag field, as described later.

As an example of parallel processing, we show the outline of the shift operator microroutine that does a N-bit circular shift on the 64-bit data. As 4 bits are the smallest unit of the SEN operation, the SEN shifts the data 4 x D4 (D4 = N div 4) bits in one machine cycle. Then, the ALU shifts it M4 (M4 = N mod 4) bits, one bit by one bit. The use of the SEN reduces the number of machine cycles from 31.5 to 2.5 on the average. This example shows not only the enhancement of processing by parallelism but also the provision of a uniform function (in this case, shift) to the user. If we use the function at the micro-assembly language level, we must directly control the SEN and the ALU shifter to get appropriate results.

(2) Parallel processing of string functions: The string functions are executed by using the bit count and priority encode functions of the bit operation unit (BOU), field extraction and embed-

ding functions of the divide and concatenate unit (DCU) in the four processor units, and the shift and broadcast functions of the SEN. For example, a bit string extraction function BSUBST(BIT, POS, N) extracts N-bit data from the bit position of POS-bit of 64-bit variable BIT. To do this operation, this routine first gets the data from 8 banks of MM to register file (RF) in 4 PUs in parallel. Then, the data is shifted (POS - 1) bits by using the SEN and the ALU shifter. After computing i (= N div 16) and j (= N mod 16), the data is concatenated with 0 at the (j+1)-bit at PUi, and is stored into 8 banks of MM in parallel.

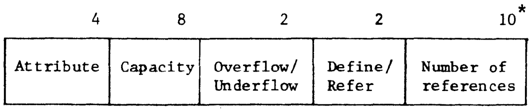
This example shows the difficulty and tediousness of handling the multiple processing units, especially if they have some specific features, such as nonnumeric processing functions. These functions provide the user a high-level but efficient interface by doing tedious and, sometimes, tricky operations for utilizing the parallelism of the micro-architectures instead of the user.

(3) Implementation of tagged architecture: Each variable in MSDL has 26-bit tag field as shown in Figure 4. The check points are divided into two major parts: (a) checks at the fetch and operand access phase, and (b) checks in the execution phase. The checks for item (a) include: (i) system variable error (check the range of system variables such as intermediate language instruction counter), (ii) parameter error (check the number, attribute, and order of formal parameters for procedure call instruction), and (iii) access error (check if the MM and SPM addresses point to the user variable area, the operand value is defined, and the index of array is within the correct range). The checks for item (b) depend on the content of processing. In the arithmetic operations, for example, overflow, underflow, and division by 0 are checked. In both (a) and (b), the BOU and DCU functions ease the reference to the tag, which is divided into several fields.

These checks may seem to be redundant. However, it not only enhances the user interface but also checks erroneous actions caused by incorrect input data. Further, the tagged architecture is made feasible by fast parallel processing of multiple processors.

4.3 Experimental results

To evaluate the effectiveness of the system description language level architecture, we performed an experiment. Table 2 shows the static information about the MSDL interpreter. That is, for each function, the number of microstatements (M), the number of nanostatements (N), the number of micro-nano combined statements (MN) in micro-assembly language, and the average PU numbers activated in each machine cycle are given. The following features are observed.



4	8	2	2	10*
Attribute	Capacity	Overflow/ Underflow	Define/ Refer	Number of references

* bit length

Fig. 4 Tagged data structure.

Table 2 Static data from interpreter

Function of Module	Micro (M)	Nano (N)	Micro-Nano (MN)	Active PUs
Control				
Main	5	2	3	1.80
Initialize	1	20	7	3.77
Fetch Instr.	2	13	21	2.66
New Code	10	19	41	1.79
Operand Access	15.25	21	22.25	1.64
Decode	24.75	1.50	2.50	1.13
Operators				
Arithmetic	6.07	8.93	15.66	2.09
Logical	2	5	4	2.56
Shift	13	20.16	45.23	1.92
Exchange	10	11	15	2.23
Expression	32	6.33	18.67	1.95
Statements				
Procedure	16	14.50	25.50	1.69
IF	2	1.30	3.33	1.50
GOTO	6	16	6	1.58
FOR	6.25	12	10.6Q	2.19
WHILE	4	7.67	10.33	1.86
String Func.				
Bit	12	10	43.75	1.90
Character	9.75	10	37.25	2.21
Subroutines	5.54	4.67	9.29	2.12
Average	9.05	8.71	16.66	2.00

Table 3 Processing time ratio for tag processing

	Type Check	Type Transform	Tag Create	Result Set
Integer	28.24	46.56	14.50	10.69
Real	31.71	40.65	16.26	11.38

unit: %

(1) The numbers M, N, and MN represent the numbers of statements required for realizing the system description language level (Level 2) by using the assembly level language (Level 1).

(2) From 1.13 to 3.77 PUs are used for each machine cycle in microprogram modules. The average number of active PUs for all the micro-routines in the interpreter is 2.00. Further, as a result of optimization, we have improved it to 2.19. The numbers for the control part varies greatly. However, in the routines for operators and functions, the number of active PUs are around 2.

Further, the results of dynamic data are summarized as follows: (detailed data is found in [10].)

(3) The ratio of micro (M), nano (N), and micro-nano combined (MN) are 19:23:58. This shows the effective use of Level 1 micro-nano combined statements at Level 2.

(4) The average number of dynamically active PUs is 2.14.

(5) The overhead caused by error checks and related operations is classified into 4 categories as shown in Table 3. Type check and type transformation are major parts.

Items (2) and (4) provide a guideline of how many processors are active on average when we apply multiple processors to ordinary problems (not special parallel problems, such as array processing).

5. Effects of Architecture Hierarchies

The effects of architecture hierarchies are illustrated in Figure 5, where a sample bit count function is described at three levels. At the lowest level, we must describe the microprograms in bit pattern format for multiple processors (Figure 5(c)). At the highest, system description language level, it may be written as a single function call(Figure 5(a)). At the middle level, the microprogram is written in register transfer language (Figure 5(b)).

To make the difference clear, we summarize it in Table 4. The following items are observed from the table.

(1) The higher the level of language is, the richer the facility is. This can be generalized to all the aspects of language, such as the data structure, arithmetic and logical operators, and control functions.

(2) At the lowest level of Level 0, the user must take care of the parallelism and two-level control scheme of the bare machine. At Level 1, the frequently used micro-nano combinations, and the instructions with the SIMD feature, may be described in one statement. But these features do not completely hide the hardware features, such as parallelism among multiple processors and two-levels of control, from the user. At Level 2, such hardware features are almost hidden from the user, and problem-oriented functions are provided.

(3) The utilization of multiple processor parallelism does not change between Levels 0 and 1, because they have the same description capability. However, it slightly decreases from Level 1 to Level 2, in exchange for independence of parallelism recognition by the user.

(4) The error check function is only provided at Level 2 to aid the programming process and enhance the system reliability.

(5) The extensibility of the language differs from Levels 1 and 2. The extensibility at Level 1 corresponds to the extensibility of hardware such as the addition of a new microinstruction field or micro-order. The Level 2 extension is the addition of new functional modules.

```
BCT(BIT,1)
```

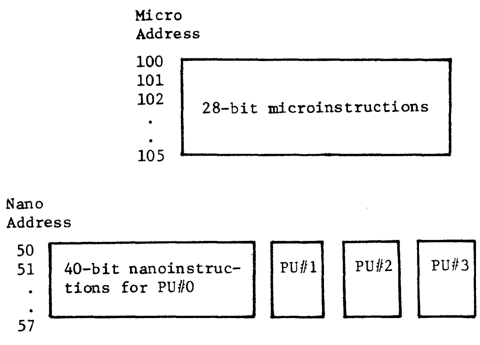
(a) Level 2

```
ST-NO.          STATEMENT

  1  MICRO MAIN BITCOUNT (100);
  2     EXT NEXT (2);
  3        *;
  4           RF(2) := 0;
  5           CXO := 0;
  6  L1:   AM MODE M8 (X,H) PU(3-0);
  7           OPRO := RFO(1) <+> CXO;
  8        SPM(10) := MM;
  9           RF(3) := <BCT,1> SPM(10);
 10           RF(2) := RF(2) <+> RF(3)  CXO+1;
 11        *IF CXO MOD4<>0 THEN GOTO L1;
 12        IPR := <SRL16> OPR;
 13           OPRO := RFO(2);
 14           OPR1 := RF1(2) <+> IPR1;
 15           OPR2 := RF2(2) <+> IPR2;
 16           RF3(4) := RF3(2) <+> IPR3;
 17        GOTO NEXT;
 18  END;
```

(b) Level 1*

* see [4] for comments



(c) Level 0

Fig. 5 Sample bit count programs at Levels 2, 1, and 0.

The above comparison shows that we have defined a reasonable interface as a compromise between the user´s requirements and the multiprocessor system throughput.

6. Concluding Remarks

We have developed hierarchical microarchitectures for a two-level microprogrammed multiprocessor computer. The results from the development of language processors and some experiments show the effectiveness of such architectures. These results will be especially useful for defining a "easy to use" but "efficient to implement" architecture on a machine with innovative hardware organization. As the hardware cost decreases it becomes feasible to construct such machines in many application areas. Thus, the need for defining a good architecture on the machine will increase.

The experimental results also show the important guideline that about 2 of 4 multiple proces-

Table 4 Comparison among three microarchitecture levels

	Level 0	Level 1	Level 2
Language Features			
Data Structure	Integer (16), Character, Boolean	Integer (16), Character Boolean	Integer (16,32,64), Real (32,64), Character, Boolean
Arithmetic Ops.* and Test	Functions of bare hardware	Level 0 plus combined ops. for transfer and test	Problem-oriented operators and functions
Control	Branch, Branch on condition	IF, GOTO, CASE that correspond to the L0 functions.	IF, WHILE, FOR, SWITCH in problem-oriented format
Extensibility	Equals that of hardware	New microinstr., Microinstr. field, Micro-order	New microprogram module
Architectural Features			
Parallelism	Direct descrip. by users	Single statement for the SIMD ops.	User independent feature
Two-Level Microprograms	Direct descrip. by users	Single st. for the tightly coupled micro-nano ops.	User independent feature
Uniformity	Limited by avoidance of hardware redundancy	Uniformity for some test ops.	Uniformity for all the functions
Facilities for Program Test	Debugger to run and monitor microprograms	Debugger to run and monitor microprograms	Tagged architecture

*ops.: opeartions

sor units are activated on an average, even if the system is applied to ordinary (not parallel) problems. This means that in most machine cycles multiple (i.e., from 2 to 4) processors are activated in parallel. The experienced microprogrammer efforts for exploiting inherent parallelism within the problems yield such results.

Our future problem is to develop the application programs, such as a database system, in the system description language and verify the effectiveness of the architecture from the viewpoints of 1) parallelism utilization of multiple processors and nonnumeric units distributed under two-levels of control and 2) effectiveness of the tagged architecture for software development.

Acknowledgment: We would like to express our appreciation to Alan R. Hevner and the reviewers for their helpful comments.

REFERENCES

[1] Baba, T, Ishikawa, K., Okuda, K., and Kobayashi, H.: "MUNAP - A Two-Level Microprogrammed Multiprocessor Architecture for Nonnumeric Processing," Proc. IFIP Congress 80, (Oct. 1980), pp. 169-174.

[2] Baba, T., and Hagiwara, H.: "The MPG System: A Machine-Independent Efficient Microprogram Generator," IEEE Trans. Comput., (June 1981), pp.373-395.

[3] Baba, T., Hashimoto, N., Yamazaki, K, and Okuda, K.: "Microprogramming Support System for a Two-Level Microprogrammed Computer MUNAP," Trans. IECE Japan, (Oct. 1982), pp. 1265-1272.

[4] Baba, T., Ishikawa, K., and Okuda, K.: "A Two-Level Microprogrammed Multiprocessor Computer with Nonnumeric Functions," IEEE Trans. Comput., (Dec. 1982), pp. 1142-1156.

[5] Denning, P. J.: "Why Not Innovations in Computer Architecture?," Computer Architecture News, (April 1980), pp. 4-7.

[6] Kernigham, B. W. and Ritchie, D. M.: The C Programming Language, Prentice-Hall, Inc., (1978).

[7] Lang, T. and Stone, H. S.: "A Shuffle-Exchange Network with Simplified Control," IEEE Trans. Comput., (Jan. 1976), pp. 55-65.

[8] Lawrie, D. H.: "Access and Alignment of Data in an Array Processor, " IEEE Trans, Comput., (Dec. 1975), pp. 1145-1155.

[9] Myers, G. J.: Advances in Computer Architecture, New York: Wiley, (1978).

[10] Yamazaki, K., Hashimoto, N., Kanai, H., Baba, T., Okuda, K., and Hashimoto, K.: A System Description Language for a Two-Level Microprogrmmed Computer MUNAP: MSDL, Tech. Rep. of IECE Japan, EC82-60, (Dec. 1982).

ALTERNATIVE DATA STRUCTURES FOR LISTS IN ASSOCIATIVE DEVICES

J. L. Potter
Department of Mathematical Sciences
Kent State University
Kent, Ohio 44242

Abstract -- If the full power of parallel associative devices are to be utilized, alternative implementations of standard logical data structures must be developed. This paper presents three methods for implementing lists suitable for associative memories and processors.

The associative tree technique is based on the logical tree structure implementation of a list. Its main advantage is that storage allocation and release are straight forward and garbage collection avoided. The CDAR encoding technique stores the tokens of the list with a code that encodes their position. This technique supports sublist matching since it allows any CDAR code range to be searched for in constant time. The EPS method is similar to the CDAR code method. The main difference is that the position information is more efficiently encoded allowing more compact storage of long lists but requiring slightly more complex searching. All three of these techniques appear well suited for implementing list based languages such as LISP and PROLOG on associative processors with the potential of substantially improving program execution speeds.

Introduction

The Symbolic or S-expression is a parenthesised list of atomic symbols which forms the basic data structure for many AI oriented languages such as PROLOG and LISP. In conventional computers, the lists are logically implemented as tree structures which in turn are normally physically implemented with linked lists. The use of lists in AI programming research is popular because it supports symbolic manipulation which allows complex algorithms to be programmed relatively easily. Unfortunately, programs written using lists for data and program storage tend to result in slow program execution when processing data bases of non-trivial sizes. If LISP, PROLOG and other high level languages using lists are to be used in practical applications with large data bases, new faster alternative data structures must be developed.

Several approaches are possible to address the problem of slow execution of list based languages such as LISP and PROLOG. One approach is to optimize the code of a conventional computer for list functions [2],[5]. Another is to optimize the linked list structure [3]. Both of these approaches speed up execution but still suffer from the inherent drawback of linked list memory allocation - garbage collection. The time for garbage collection can be dramatically reduced by using 5th generation type devices such as associative processors and memories to implement conventional linked list storage [4]. However, this approach does not fully take advantage of the parallel search capability of these devices.

This paper discusses three alternative approaches to conventional linked list structures suitable for list implementation in parallel associative memories and computers.

Traditional Storage

The binary tree representation of the list (A (B C) D) is shown in Figure 2-1. Figure 2-2 shows

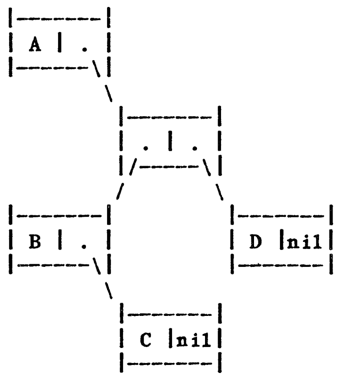


Figure 2-1 Tree Representation of (A (B C) D)

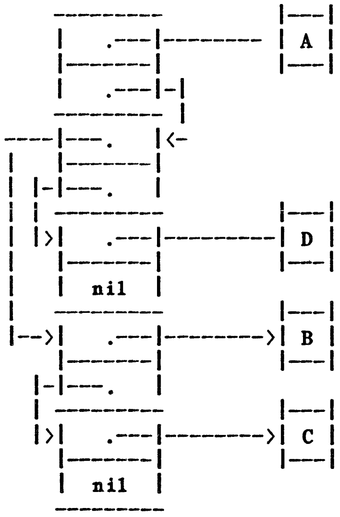


Figure 2-2 Link List Implementation of (A (B C) D)

0190-3918/83/0000/0486$01.00 © 1983 IEEE

a typical linked list implementation. Many texts have been written explaining LISP and its S-expression implementations [7]. This section will simply review concepts important to the following discussion. Readers acquainted with LISP and list processing concepts may wish to skip this section.

In LISP the two basic functions CAR and CDR allow the programmer to traverse binary trees or equivalently to generate sublists from lists. Basicly a CAR is an instruction to traverse to the left subtree from a node while a CDR causes a traverse to the right subtree. The equivalent actions in list notation for a CAR is the sublist obtained by extracting the left most element of the list. The CDR is the sublist composed of the remainder of the list after the first element has been extracted. Thus the CAR of the list (A (B C) D) is A. While the CDR is ((B C) D). CAR and CDR functions are frequently chained together. The chaining is abbreviated by writing simply A or D in sequence to represent CAR or CDR and then adding a single C and R as in CADDAR. The order of application is from right to left (inner most function call to outer most).

CONS is another list manipulation function (construction) and is used to build lists. It inserts a new element at the beginning of a list. Thus the CONS of A and ((B C) D) is (A (B C) D).

Associative Program Design Language (APDL)

In the following sections, APDL will be used for algorithm descriptions implemented in associative memories. APDL reflects the fact that loop information is vital in a sequential computer program, but in an associative memory, it is redundant and consequently not needed. Thus the statement:

```
FIELDj(*) := FIELDj(*)+CONSTANT
```

is used in place of a loop. The * indicates that the statement is executed in parallel on every enabled word in the associative memory [6].

The responses to a parallel search can be processed sequentially by using the NEXT (get next responder index) and EOR (end of responders) functions and a ~ index variable notation. If there are no responders, i.e. all elements of the variable are false, EOR(RESPOND(*)) will be true and RESPOND(*)~ will be undefined. If EOR(RESPOND(*)) is false, then NEXT(RESPOND(*)) will assign the internal memory index of the next true word recorded by RESPOND(*) to RESPOND(*)~ and sets the value for that word to false. RESPOND(*)~ can be used as an 'index' variable in place of * in any associative field reference.

In general, any field can be restricted to the responders of a parallel search within the scope of an IF statement. For example,

```
IF FIELDi(*) = 'NAME' THEN FIELDj(*) := 'value'
```

modifies only those elements of FIELDj whose corresponding elements of FIELDi equal NAME.

In an associative memory, storage allocation is easily handled by defining one of the fields to be a STATUS field. Thus if a new word is needed, its index is obtained by the code:

```
AVAIL(*) := STATUS(*) = 'IDLE'
NEXT(AVAIL(*))
INDEX := AVAIL(*)~
STATUS(INDEX) := 'BUSY'
```

Conversely, when a word is no longer needed, it is returned to the available storage category simply by writing IDLE in its STATUS field.

In the following sections, associative code will be used for algorithm descriptions implemented in associative memories. All associative statements are easily understood if it is kept in mind that a * index is equivalent in a sequential computer to a loop through all words in the memory.

Associative Structures

Associative Trees

Figure 4-1 illustrates a straight forward implementation of the traditional tree representation of a list in an associative memory. The primary difference in this implementation is that pointers are replaced by a node name or ID. Each link record contains its own ID and as such is 'relocatable.' That is, any node record can be stored in any memory location (word). In figure 4-1, the data is sorted on the ID field for ease of presentation and understanding. In an actual associative memory, the 9 records shown could be in any order in any of the memory's words. The actual configuration would be a function of the values in the Status Field at the time the entries were made.

In this associative memory implementation of a tree data structure, a list or sublist is designated by the ID of the root node of the logical tree structure. Thus the list (A (B C) D) is designated by ID #1.

The implementation shown stores a pointer to the atom in the left cell instead of the atom itself, thus the CAR function is simply implemented by chaining on the left child node ID. The CAR of ID #1 is ID #6 which is the storage node for atom 'A'. The CDR of ID #1 is ID #2 which is the node ID that designates the list ((B C) D). List construction (CONS) is accomplished by generating a new entry in any available word of memory (i.e. idle status) with the appropriate node IDs. Thus CONS of A, node ID #6, and ((B C) D), node ID #2 would generate the following and return node ID #m.

B	m	N	6	2

	STATUS FIELD	ID FIELD	ENTRY TYPE FIELD	NAME FIELD	
			ATOM NODE	LEFT CHILD	RIGHT CHILD
WORD n+0	B	1	N	6	2
WORD n+1	B	2	N	3	4
WORD n+2	B	3	N	7	5
WORD n+3	B	4	N	8	nil
WORD n+4	B	5	N	9	nil
WORD n+5	B	6	A	A	
WORD n+6	B	7	A	B	
WORD n+7	B	8	A	D	
WORD n+8	B	9	A	C	

Figure 4-1 Associative Linked List

Note that with this type of memory organization, the CAR, CDR and CONS functions are as easy to implement as with conventional linked lists. The primary difference is that this organization is best suited for associative memories since the data is accessed via a key (node ID in this case) and not by a memory address. Since associative memories can access the entire memory in the same amount of time as one word, the data does not have to be organized (i.e. sorted) to achieve retrieval efficiency. When records (i.e. node entries) are no longer needed, the status field of the word is simply marked idle and is thus available for reuse by the next memory access seeking an idle word. Complex garbage collection is not needed.

CDAR Codes

The associative tree organization uses associative memories but still requires that chains of CAR and CDR functions (henceforth abbreviated CDAR functions) be executed sequentially. Another data representation shown in Figure 5-1 allows any sublist which can be defined by a CDAR function to be searched for directly and in parallel. Thus with this representation, all CDAR functions can be executed in a constant amount of time.

This storage technique uses a CDAR code illustrated in Figure 5-1 designed so that numeric range searches can be used to search for sublists. Thus if the list ((A B (C D) (((E) F))) G) is to be processed by the function CDDAR, the function string is first converted into the CDAR code 011. Then, the lower bound of the search is obtained by adding zero fill, the upper bound by adding one fill. Thus in this example, the CDDAR of the list shown in Figure 5-1 is obtained by selecting all elements greater than or equal to 011000000000000 and less than or equal to 0111111111111111. These elements, C, D, E, and F, form the sublist ((C D) (((E) F))).

Let 0=CAR, 1=CDR, left justify with 1 fill (order of application is left to right) then LIST = ((A B (C D) (((E) F))) G) is

LIST NAME	CDAR CODE	ATOM

LIST	0011111111111111	A
LIST	0101111111111111	B
LIST	0110011111111111	C
LIST	0110101111111111	D
LIST	0111000011111111	E
LIST	0111001011111111	F
LIST	1011111111111111	G

Figure 5-1 CDAR Encoding

Figure 5-2 gives the algorithm for generating the CDAR code for a list from list input. For simplicity, it is assumed that the input string has been scanned and the items have been broken out and stored in the associative memory field ITEM in a manner such that index I will reference them in the proper order.

Basically, the algorithm contains a scan and a generate procedure. The scan procedure identifies the next item in the list. If the item is a left parenthesis, the level of the tree is incremented by one and the number of nodes on that level is initialized to zero. If the item is an atom, the appropriate CDAR code is generated and associated with the atom. After the code and atom have been associated, the count of nodes for the current level is incremented. If the item is a right parenthesis, the appropriate CADR code is generated and associated with a 'nil' symbol (This marks the end of a substring). The level of the tree is decremented and the number of nodes on the lower level is incremented.

The CDAR code generation function simply generates a string of code for each level. The code consists of 1 one for each node on a level terminated on the right with a zero. The codes from the levels are concatenated from right to left (highest level to lowest) with one fill on the right.

The CAR and CDR functions are of course, special cases of the more general CDAR function. The CONS function is equally easy to implement with the CDAR code approach. If the list L1 = (C (D) E), stored in memory as shown in Figure 5-3a, is to be CONSed to the list (A B), Figure 5-3b, the process is simply one of appending a zero to the front (left) of the CDAR code for L2 = (A B), appending a 1 to the front of the codes for L1, and changing the list name of L1 to L2 (See Figure 5-3c). If a new list is being generated, the elements of L1 and L2 would be copied before modification and both list names would be changed.

```
    PROCEDURE SCAN
    TYPE
         CDARRECORD = RECORD
              CDARCODE: CODETYPE
              ATOM: ATOMTYPE
              LISTNAME: NAMETYPE
              END
    VAR
    CDARMEMORY: ARRAY[1..m] OF CDARRECORD
    NODECT: ARRAY[1..n] OF INTEGER
    FUNCTION GENERATE
    CONST
         ACODE = 0
         DCODE = 1
    BEGIN (* GENERATE *)
    GENERATE := -1 (* SET TO ALL ONES *)
    FOR K := LEVEL TO 1 DO
      BEGIN
      GENERATE := RIGHTSHIFT(ACODE,GENERATE)
(* RIGHTSHIFT SHIFTS THE SECOND ARGUMENT RIGHT
   ONE BIT AND SHIFTS THE FIRST ARGUMENT INTO
   THE LEFT MOST BIT *)
      FOR L := NODECT(K) TO 1 DO
        GENERATE := RIGHTSHIFT(DCODE,GENERATE)
      END
    END (* GENERATE *)
    BEGIN  (* SCAN *)
    LEVEL := 0
    J := 1
    FOR I = 1 TO ENDI DO
      CASE ITEM(I) OF
      LEFTP:
         BEGIN
         LEVEL := LEVEL + 1
         NODECT(LEVEL) := 0
         END
      ATOM:
         BEGIN
         CDARCODE(J) := GENERATE
         ATOM(J) := ITEM(I)
         LISTNAME(J) := NAME
         J := J + 1
         NODECT(LEVEL) := NODECT(LEVEL) + 1
         END
      RIGHTP:
         BEGIN
         CDARCODE(J) := GENERATE
         ATOM(J) := NIL
         J := J + 1
         LEVEL := LEVEL - 1
         NODECT(LEVEL) := NODECT(LEVEL) + 1
         END
      END
    END.
```

Figure 5-2 CDAR Code Generation Algorithm

NAME	CODE	ATOM	NAME	CODE	ATOM	NAME	CODE	ATOM
L1	01111	C	L2	01111	A	L2	00111	A
L1	10011	D	L2	10111	B	L2	01011	B
L1	11011	E				L2	10111	C
						L2	11001	D
						L2	11101	E

a - List (C (D) E) b - List (A B) c - List ((A B) C (D) E)

Figure 5-3 List Concatenation

Explicit Parenthesis Storage

The Explicit Parenthesis Storage (EPS) technique associates the list structure with the atoms by explicitly saving the left and right parenthesis. Figure 6-1 shows a typical EPS associative record. The record contains the name of the list, the name of the atom, the number of left parenthesis in the list preceding the atom, the number of right parenthesis preceding and immediately following the atom and the position of the atom in this list.

```
                  NUM   NUM
LIST              LEFT  RIGHT POSITION
NAME  ATOM  PARAN PARAN NUMBER
FIELD FIELD FIELD FIELD FIELD
|-----|-----|-----|-----|--------|
|     |     |     |     |        |
|-----|-----|-----|-----|--------|
```

Figure 6-1 An EPS Record Format

The algorithm for generating an EPS representation of a list is shown in Figure 6-2. The algorithm is quite similar to the CDAR code generation algorithm (Figure 5-2) except that the left and right parenthesis counts (NLP and NRP respectively) are saved as part of the data directly. The algorithm is straight forward and simply counts left and right parentheses as they are encountered. However, since the right parentheses count of an atom in position n includes the right parentheses up to the atom in position n+1, the calculation of the NRP values lag behind one iteration. Thus the NRP value for the last atom is stored after the entire list has been processed. Figure 6-3 shows a list and its EPS representation.

In this representation of a list, the list or sublist of interest is delineated by specifying the lowest and highest position number in the list. Figure 6-4 and 6-5 give the algorithms for the CAR and CDR functions respectively. These algorithms assume that the global variables LOWEST and HIGHEST delineate the list position numbers on entry and they update the variables and adjust the NLP and NRP values accordingly. The CAR is found by setting the HIGHEST variable to the left most atom (i.e. lowest position number) with sufficient right parenthesis to balance the first atom in the list. The CAR is obtained by finding the same atom, but setting the LOWEST variable to the next highest value. The remaining statements adjust the NLP and NRP counters accordingly.

The CONS function is accomplished by adding one to the left parentheses count of the elements of the first argument and then adding the total number of NLP, NRP and POSN of the first argument to the NLP, NRP and POSN of the elements of the second argument. Figure 6-6a, b and c give an example. Figure 6-7 gives the algorithm.

```
LPCNT := 0
RPCNT := 0
J := 1
K := 1
FOR I := 1 TO ENDI DO
CASE ITEM(I) OF
  LEFTP: LPCNT := LPCNT + 1
  ATOM:
     BEGIN
     LISTNAME(J) := NAMEOFLIST
     ATOM(J) := ITEM(I)
     NLP(J) := LPCNT
     (*DEFINE RANGE OF RP TO INCLUDE 0 *)
     NRP(J-1) := RPCNT
     POSN(J) := K
     K := K + 1
     J := J + 1
     END
  RIGHTP: RPCNT := RPCNT +1
  END
END
NRP(J-1) := RPCNT
```

Figure 6-2 List to EPS Transformation Algorithm

		NUM	NUM	
LIST		LEFT	RIGHT	POSITION
NAME	ATOM	PARAN	PARAN	NUMBER
FIELD	FIELD	FIELD	FIELD	FIELD
		(NLP)	(NRP)	(POSN)
LISTC	C	2	0	1
LISTC	D	2	1	2
LISTC	A	2	1	3
LISTC	B	3	2	4
LISTC	E	3	3	5

Figure 6-3 EPS Representation of ((C D) A (B) E)

```
BEGIN (* CAR *)
IF LOWEST <= POSN(*) AND POSN(*) <= HIGHEST THEN
   HIGHEST := MINIMUM(POSN(*),NLP(*)-1<=NRP(*))
IF LOWEST = HIGHEST THEN
   NRP(HIGHEST) :=NRP(HIGHEST) - 1
IF LOWEST <= POSN(*) AND POSN(*) <= HIGHEST THEN
   NLP(*) := NLP(*) - 1
END (* CAR *)
```

Figure 6-4 CAR Algorithm for EPS

```
BEGIN (* CDR *)
LPCNT := NLP(LOWEST) - 1
IF LOWEST <= POSN(*) AND POSN(*) <= HIGHEST THEN
   LOWEST := MINIMUM(POSN(*),NLP(*)-1<=NRP(*))
RPCNT := NRP(LOWEST)
LOWEST := LOWEST + 1
IF LOWEST <= POSN(*) AND POSN(*) >= HIGHEST THEN
   BEGIN
   NLP(*) := NLP(*) - LPCNT
   NRP(*) := NRP(*) - RPCNT
   END
END. (* CDR *)
```

Figure 6-5 CDR Algorithm for EPS

		NUM	NUM	
LIST		LEFT	RIGHT	POSITION
NAME	ATOM	PARAN	PARAN	NUMBER
FIELD	FIELD	FIELD	FIELD	FIELD
		(NLP)	(NRP)	(POSN)
LISTB	C	1	0	1
LISTB	D	1	1	2

Figure 6-6a The EPS Representation of LISTB

		NUM	NUM	
LIST		LEFT	RIGHT	POSITION
NAME	ATOM	PARAN	PARAN	NUMBER
FIELD	FIELD	FIELD	FIELD	FIELD
		(NLP)	(NRP)	(POSN)
LISTA	A	1	0	1
LISTA	B	2	1	2
LISTA	E	2	2	3

Figure 6-6b The EPS Representation of LISTA

		NUM	NUM	
LIST		LEFT	RIGHT	POSITION
NAME	ATOM	PARAN	PARAN	NUMBER
FIELD	FIELD	FIELD	FIELD	FIELD
		(NLP)	(NRP)	(POSN)
LISTC	C	2	0	1
LISTC	D	2	1	2
LISTC	A	2	1	3
LISTC	B	3	2	4
LISTC	E	3	3	5

Figure 6-6c The EPS Representation
of (CONS LISTB LISTA)

```
PROCEDURE CONS(LOWESTB,HIGHESTB,LOWESTA,HIGHESTA)
BEGIN (* CONS *)
IF LOWESTB <= POSN(*) AND POSN(*)<=HIGHESTB THEN
  NLP(*) := NLP(*)+1
IF LOWESTA <= POSN(*) AND POSN(*) <= HIGHESTA THEN
  BEGIN
  NLP(*) :=  NLP(*) + NLP(HIGHESTB)
  NRP(*) :=  NRP(*) + NRP(HIGHESTB)
  POSN(*):= POSN(*) + POSN(HIGHESTB)
  END
END (* CONS *)
```

Figure 6-7 CONS Algorithm

Conclusion

Three different techniques of storing logical list structures in associative devices for efficient processing are described. Preliminary analysis indicates considerable speed up in pattern matching can be achieved if conventional LISP and PROLOG implementations were to be implemented. While this avenue should be explored, it would appear that even greater speeds are possible by implementing the searching functions inherent in these languages at a more direct level.

References

[1]. Aho, A. V. and M. J. Corasick, 'Efficient String Matching: An Aid to Bibliographic Search,' COMMUNICATIONS OF THE ACM, 18, 1975, pp. 333-340.

[2]. Baker, Henry G.,'List Processing in Real Time on a Serial Computer,' COMMUNICATIONS OF THE ACM, April, 1978, pp. 280-294.

[3]. Bobrow, Daniel G. and Douglas W. Clark, 'Compact Encodings of List Structure,' ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS, October, 1979, pp. 266-703.

[4]. Bonar, Jeffrey G. and Steven P. Levitan, Real-time LISP Using Content Addressable Memory,' IEEE, 1981, pp. 112-117.

[5]. Greenblat, R., LISP MACHINE PROGRESS REPORT, AI Lab., M.I.T., Cambridge, Mass. memo 444, August, 1977.

[6]. Potter, J. L., 'MPP Architecture and Programming' in MULTI-COMPUTERS AND IMAGE PROCESSING, K. Preston and L. Uhr, eds., Academic Press, 1982, pp.275-290.

[7]. Weissman, C., LISP 1.5 PRIMER, Dickenson Publishing Company, Inc., Belmont, California

DETERMINATION OF THE ROTATIONAL AND TRANSLATIONAL COMPONENTS OF A FLOW FIELD USING A CONTENT ADDRESSABLE PARALLEL PROCESSOR

M. E. Steenstrup, D. T. Lawton, C. Weems

Department of Computer and Information Science[1]
University of Massachusetts at Amherst

Abstract

The realization of motion perception in artificial systems will require highly parallel architectures. Here we demonstrate the use of a Content Addressable Parallel Processor (CAPP) [1,2] as an effective means of quickly and accurately decomposing a flow field into its rotational and translational components [3] to recover the parameters of sensor motion.

Organization of the CAPP

The CAPP is a VLSI-based Single Instruction Multiple Data (SIMD) machine designed at the University of Massachusetts [4]. It consists of a parallel processor containing 512x512 cells and a central controller. The central controller issues instructions to the parallel processor, controls loading and unloading of data in the parallel processor, and serves as an interface to the host computer and to secondary storage devices. It broadcasts data to the parallel processor bit serially, and the entire memory may be bulk-loaded in one video frame time (1/30 second). The central controller contains a set of micro-coded subroutines in ROM for performing high-level CAPP routines and a writeable control store for adding microcode.

The parallel processor consists of an 8x8 array of processing circuit boards and a set of boards which control CAPP edge treatment. Each processor board, in turn, consists of an 8x8 array of special purpose CAPP IC chips plus random buffer logic. Each chip then contains 64 cells, an instruction decoder, and some miscellaneous logic. There are eight basic instruction types recognized by the chip, each performed in parallel by the constituent cells. Most instructions take one minor cycle time (100 nanoseconds) to execute. Inter-cell communication is bit serial and is accomplished by a four-way (N, S, E, W) cell interconnect network, allowing for three types of edge treatments: dead-edging, circular wrap, and zig-zag wrap.

Each unit cell consists of 64 bits of fully static memory, four one-bit static "tag" registers A, B, X, and Y, a static carry bit register Z, and an ALU which continuously generates X NAND Y, X NOR Y, and X + Y + Z. Also, each cell contains logic for selecting a data source (a register (excluding Z), memory, an ALU function, broadcast data, or a neighboring cell (N, S, E, or W)), possibly inverting the selected signal, and storing it in a destination (a register or memory). The X register is the main tag register. Its output is connected to Some/None logic, indicating cell response, and to the neighbor communication network. The A register controls whether or not a cell is active. An inactive cell ignores all but a small set of instructions broadcast by the central controller. The Y register provides a secondary store for tag bits, while the B register provides a secondary store for activity bits.

Flow Field Decomposition Procedure

Our algorithm is an exhaustive search procedure which uses a set of rotational and translational flow field templates to find a component pair which can account for the motion depicted in a given flow field. Currently, 1000 rotational templates and 200 translational templates are used. These are generated from 100 axes which are uniformly distributed with respect to a unit hemisphere, and all pass through the origin of the sensor coordinate system. Each flow field consists of 16x16 vectors and is stored on a 2x2 square of chips consisting of 256 cells. The 2x2 chip arrangement facilitates flow field addressing. Each cell contains the horizontal and vertical components of a flow vector, each specified with 10 bits of precision.

The algorithm consists of four basic steps.

(0) The rotational templates are loaded into the CAPP, one template for each flow field location. Each flow field location corresponds to one of the squares in the CAPP diagrams shown in Figures 2a, 2b, and 2c. The rotational templates need only be loaded once since they are used in determining any flow field decomposition.

(1) A copy of the input flow field is loaded into each flow field location in the CAPP. Figure 1a and 1b show two sample input fields, both produced by the same motion and environment, except that Figure 1b was produced by adding random spike noise to Figure 1a.

[1] This research was supported by DARPA under Grant N00014-82-K-0464.

(2) A set of *difference fields* is formed by subtracting each rotational template from the copy of the input flow field stored with it. For each resulting difference field, the slope of each difference vector is computed by dividing the vertical component by the horizontal component. These subtraction and division procedures are performed in parallel across all flow fields represented in the CAPP.

(3) The similarity between the difference fields and each of the translational templates is evaluated, proceeding sequentially through all the translational templates. For a given translational template, this comparison is done in parallel with all difference fields stored in the CAPP and consists of the following steps:

(3a) The slope of each component vector of the translational template is loaded into the corresponding vector location of each difference field. The sign of the slope of each difference vector is compared with the sign of the slope of the corresponding translational template vector. If the signs agree, the absolute value of the difference between the slope of the difference vector and the slope of the translational template vector is computed, and then scaled according to the absolute value of both slopes. If the scaled slope difference does not exceed a predetermined maximum error value, then a vector match is designated at that position. The quantity of error permitted here allows the algorithm to be resistant to uniformly distributed Gaussian noise of low variance present in the original flow field.

(3b) For each difference field the number of vector slope matches is counted. If this sum exceeds a predetermined minimum number of matches (in our implementation, 75% of the field size), then the associated rotational and translational templates become a candidate pair for the flow field decomposition. Utilization of a minimum number of required matches ensures that only templates which are reasonably close to the actual motion will be chosen and permits some resistance to random spike noise. Figure 2a shows, for difference fields resulting from the input field in Figure 1a, the CAPP response to the translational template which is closest to the actual translational motion. Each black dot within a square represents a position in a difference field at which the slope of the difference vector matches the slope of the translational template. Figure 2b shows, for difference fields resulting from the input field in Figure 1b, the CAPP response to the translational template which is closest to the actual translational motion. Figure 2c shows the CAPP response to a translational template which is not close to the actual translational motion. This translational template is shown in Figure 3.

(3c) For all difference fields yielding at least the required minimum number of matches, the variance of the scaled slope difference is computed, and the difference field with the minimum variance is determined. This value is compared to the minimum variance found from processing the preceding translational templates. If this value is less than the preceding minimum, it becomes the new global minimum, and the rotational template associated with the difference field together with the current translational template become the current best candidate pair for the flow field decomposition.

Steps 3a, 3b, and 3c are performed for each translational template.

(4) The flow field decomposition considered to be the best is the rotational and translational template pair resulting in the difference field yielding at least the required minimum number of matches and the least slope difference variance. Utilizing minimum variance instead of the maximum number of matches, the algorithm has achieved better results, particularly for motions whose component parts lie between sets of templates. Figures 4a and 4b show the rotational and translational templates selected by the algorithm in the presence of and in the absence of noise, for the input fields in Figures 1a and 1b. These templates are the closest ones to the actual motions. Figures 5a and 5b show the difference fields resulting from subtracting the rotational motion in 4a from the original fields in Figures 1a and 1b respectively.

Experiments

Experiments have been performed with a CAPP simulator on a VAX 11/780 using a wide variety of motions and simulated environments. In all cases examined, the translational template closest to the actual translational motion was selected. The rotational template was always close to the actual rotational motion, but was sometimes not the closest template. The procedure proved to be resistant to limited Gaussian noise as well as to limited random spike noise in the original flow field. Applying motion to points at random depths produced results similar to those obtained in the noise experiments. The algorithm's performance degraded slightly if each flow vector component was specified by eight bits of precision instead of by ten.

The CAPP timing calculations revealed that the algorithm could perform the rotational-translational decomposition in slightly more than 1/4 second. If two CAPPs are used in parallel, then the time can be reduced to less than 1/5 second, since only half of the translational templates need be tested on each CAPP. Given fabrication techniques available in the immediate future, we expect execution times to be significantly improved. We also suspect that performance will improve by increasing both the number and size of the rotational and translational templates. This amounts to utilizing more CAPPs in parallel.

References

[1] Foster, Caxton C. *Content Addressable Parallel Processors*. Van Nostrand Reinhold, New York, 1976.

[2] Lawton, D.T., Steenstrup, M.E., Weems, C. "Determination of Rotational and Translational Components of a Flow Field using a Content Addressable Parallel Processor", COINS Technical Report, Computer and Information Science Department, University of Massachusetts, February, 1983.

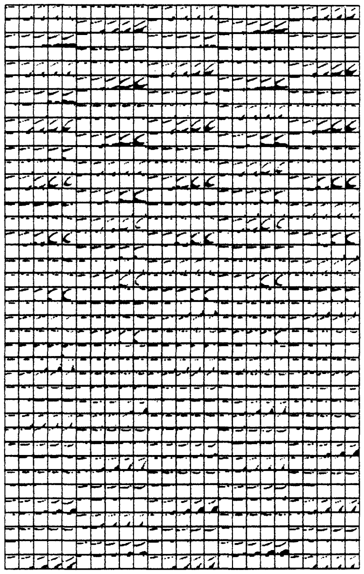


Figure 2c

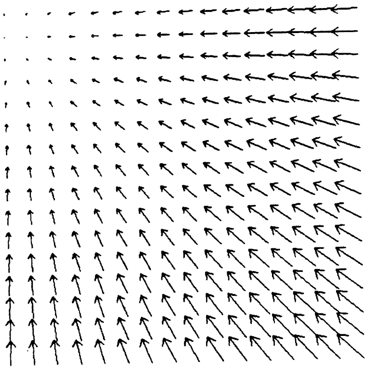


Figure 3

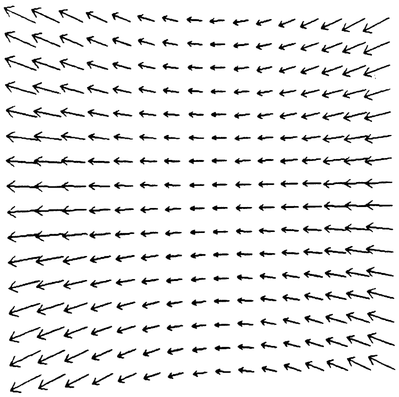


Figure 4a

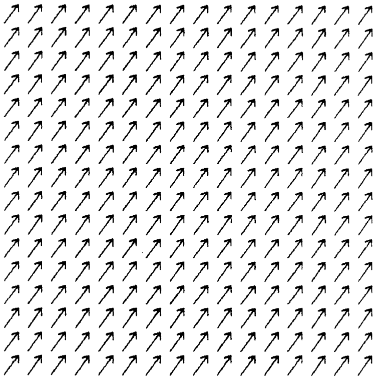


Figure 4b

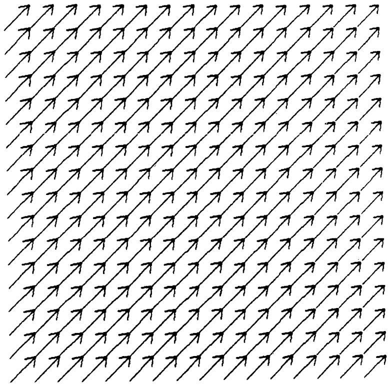


Figure 5a

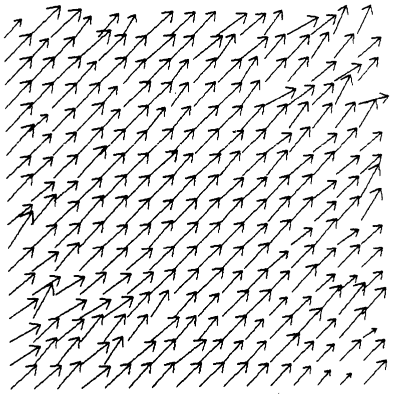


Figure 5b

[3] Prazdny, K. "Determining the Instantaneous Direction of Motion from Optical Flow Generated by a Curvilinearly Moving Observer." Proc. of the Pattern Recognition and Image Processing Conference. Dallas, Texas, August 1981, pp. 109-114.

[4] Weems, C., Levitan, S., and Foster, C. "Titanic: A Content Addressable Parallel Array Processor for Image Processing." IEEE International Conference on Circuits and Computers. New York, September 1982.

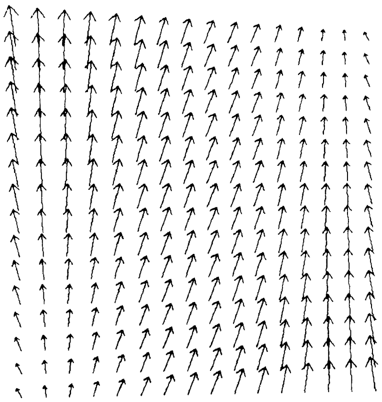


Figure 1a

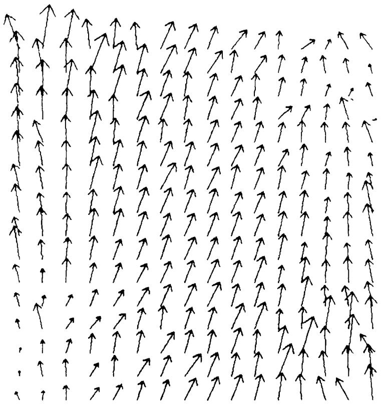


Figure 1b

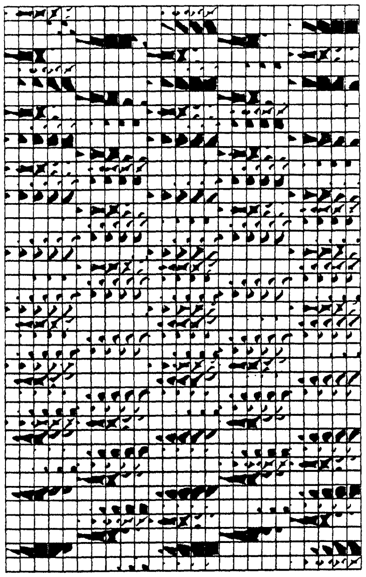


Figure 2a

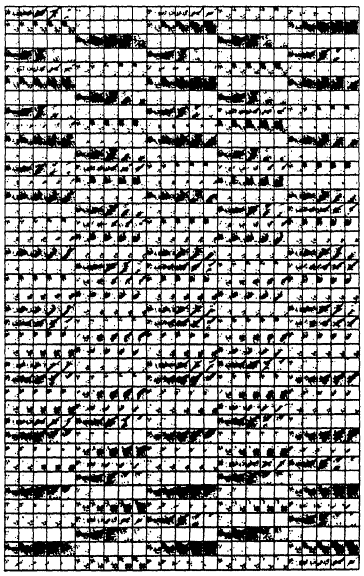


Figure 2b

DYNAMIC RELIABILITY MODELING AND ANALYSIS OF COMPUTER NETWORKS*

Srinivas V. Makam

UCLA Computer Science Department
University of California
Los Angeles, CA 90024

C. S. Raghavendra

Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089

ABSTRACT

The reliability analysis of computer communication networks is generally based on Boolean algebra and elementary probability theory. Several interesting reliability problems of computer networks include terminal-pair connectivity, tree connectivity, and multi-terminal connectivity. Traditionally, attempts were made to compute only the point and average reliabilities for networks because of the computational complexity involved. In this paper, an attempt is made to perform the dynamic analysis of reliability problems of computer networks. Time-dependent expressions for reliability measures are derived assuming Markovian behavior for failures and repairs. The advantages of the proposed methods are: the provision for incorporation of different distributions for failure and recovery times, computation of task and mission related measures such as Mean Time to First Failure (MTFF) and Mean Time Between Failures (MTBF), and network design based on the dynamic behavior. The advantages of dynamic reliability analysis is illustrated by a detailed study of the bridge network.

1. INTRODUCTION

With the increased interest in resource sharing, computer networking and distributed processing is becoming increasingly popular [ROBE 70]. Computer networks are being employed in many different applications such as distributed process control, electronic banking, defense systems, etc. Since such applications usually demand very reliable operation, redundant computers and communication links are incorporated in the design of the computer networks. Reliability modeling and analysis of such computer communication networks has drawn the attention of many researchers for a number of years [HANS 72, FRAT 73, GRNA 79].

The reliability models used for computer communication networks were based on only one of the following techniques: the discrete-state Markov chains, Boolean algebra or simple probability theory [FRAT 73, GRNA 80a]. Most of the studies were concerned with computation of only the point and steady state reliabilities for networks because of the computational complexity involved [BALL 80] There seems to be no analytic methods reported in the literature for studying the dynamic behavior of reliability problems of computer networks.

Consider the computer communication network shown in Figure 1. In such a computer network, there are several reliability problems. The probabilistic events of interest are:

- Terminal-pair connectivity
- Tree (broadcast) connectivity
- Multi-terminal connectivity

These reliability problems depend on the network topology, distribution of resources, operating environment, and the probability of failures of computing nodes and communication links. The computation of the reliability measures for these events require the enumeration of all the paths between the chosen set of nodes. The complexity of these problems, therefore, increases very rapidly with network size and topological connectivity.

Terminal-pair connectivity is useful because most applications of computer networks require connection between two nodes over a period of time; for example, in remote interactive computing. Several methods have been proposed in the literature to analyze reliability of terminal-pair connectivity [FRAT 73, FRAT 76, FRAT 78, GRNA 79, GRNA 80a, HANS 72, HANS 74]. The tree connectivity problem has not been dealt with in the literature, and is useful in studying the reliability of successful broadcasting of information by a central controller to a set of nodes in the network.

* This research was supported by the ONR Contract No. N00014-79-C-0866 (Research in Distributed Processing).

0190-3918/83/0000/0496$01.00 © 1983 IEEE

In a distributed processing system where resources such as files and programs are distributed among many computers, the successful completion of a task generally requires that several sites should be up, and communicating with each other. Execution of a task may require access to several files residing at different sites and communication paths between several node pairs. The probability of successful execution of a task is therefore more complex and useful than the terminal-pair connectivity in a distributed communications network. To handle this problem, we are interested in the event **multi-terminal connectivity** which was introduced in [GRNA 81]. The multi-terminal connectivity reflects fairly accurately the survivability of distributed systems with redundant processor, data base and communication resources [HILB 80, MERW 80, GRNA 81].

In this paper, an attempt is made to study dynamic or time-dependent analysis of the various connectivity problems of computer networks. We consider two different operating environments for computer networks, namely, closed or non-repairable, and repairable. The advantages of dynamic reliability analysis are: the provision for incorporation of different distributions for failure and recovery times, computation of task and mission related measures such as mean time to first failure (MTFF) and mean time between failures (MTBF), and network design based on the dynamic behavior.

In section 2 we explain the reliability problems of interest, define useful reliability measures, and summarize results of previous work. Section 3 deals with detailed description of the methodology proposed for analyzing dynamic behavior of computer networks. In section 4 we illustrate the methodology of dynamic reliability analysis by a detailed study of the bridge network.

2. DYNAMIC RELIABILITY ANALYSIS

For reliability analysis, a computer network or a distributed processing system is usually represented by a graph G(V,E) where V and E are respectively, the set of nodes (representing the computers) and the set of directed or undirected arcs representing the communication links. The number of nodes is N=|V| and the number of links is L=|E|. The links and nodes are labeled as x_i's and x_{i+L}'s respectively. The component set of the network is given as C=$\{x_1, \cdots, x_L, x_{L+1}, \cdots, x_{L+N}\}$. In the static reliability analysis, the processing nodes and the communication links are associated with reliabilities, i.e., probabilities of being operational. The reliability of i-th element is given by,

$$p_i = Pr(i^{th}\ element\ is\ working)$$

$$q_i = 1 - p_i$$

It is generally assumed that there is no correlation between failures of different links and nodes, i.e., the probability of failures of the elements are statistically independent. Further, it is assumed that the characterization of individual element failure behavior is sufficient to perform reliability analysis.

Most researchers performing computer network reliability analysis assume that the component (link or node) reliabilities p_i are constant during the time interval in which the reliability of the network is being examined. Additionally, no distinctions are made between the reliability and availability of networks under different operating environments. For example, in [FRAT 73], an average **network terminal-pair reliability** is defined and computed for a repairable network in which the individual components are assumed to be undergoing failure and subsequent repair (or recovery). The expression derived for this measure in actuality represents a **Steady-state terminal-pair availability.**

In this section, we will attempt to clarify the terminology used in the network reliability problems by giving precise definitions Of the measures. The **events** whose probability of success is of interest in a network are: terminal-pair connectivity, broadcast connectivity, multi-terminal connectivity, etc. Occasionally, these events are also required to satisfy some **performance constraints** specified by the user. The most common constraints include delay (time delay or hop length), flow (capacity or throughput), and survivability of distributed programs and data.

The network may be operated as a **closed system**, i.e., no repair of failed elements (nodes and links) is possible during the time interval of interest. If the failed network elements are **repaired** and made operational while the rest of the network may be still providing acceptable level of service, we are interested in the gain in the probability of successful completion of the events defined above.

We can define two representative states taken by each component and also by the system (i.e., the network): operating and failed. The component failure behavior is simple to understand. The system is said to be failed if at any time instant it **cannot maintain** a specified level of service (an event under some constraints). The dynamic reliability measures which are found very useful in the design and evaluation of computer networks and distributed systems are defined below (these measures are defined for computer systems in books on reliability) [SIEW 82].

Reliability: Given that the network was fully operational (all the components operating) at time t=0, the reliability of the system (R(t)) is the probability that it continues to provide the specified level of service at time t=T. There may be many failures of components, but the network remains operational throughout the interval [0,T].

Mean Time to First Failure: The MTFF of the network is the average time it takes for the network to enter the failed state (i.e., failure to satisfy the specified service request) for the first time, given that it was fully operational at time t=0. In the context of computer networks, an example of service for which MTFF is of interest is file transfer between a source node and a destination node.

Note that definitions of R(t) and MTFF apply to both closed and repairable networks.

Availability: Given that the network was initially (at time t=0) working in full configuration, the availability (A(t)) is the probability that the network is successful in providing the specified level of service (completing an event under a constraint) at any time instant t=T. The network might have undergone one or more failures during the interval (0,T), but it was made operational again by repairing or replacing the failed elements.

Mean Time Between Failures: For repairable networks, MTBF is the average time between two system level failures. MTBF for the network will be higher than that of a single component.

Steady State Availability: The equilibrium or steady state availability (SA) of a network gives the long term probability of maintaining the specified level of service given that the repair is provided on demand throughout the lifetime of the network. It is a measure of the fraction of time the communication system is able to exchange information between a set of nodes.

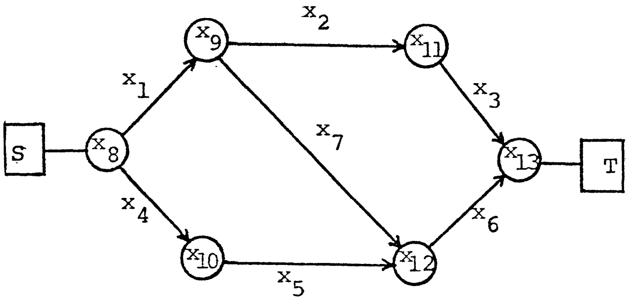


Figure 1. A Typical Computer Network

Several methods are reported in the literature for terminal reliability analysis and computation using a Boolean algebraic approach [FRAT 73, FRAT 76, FRAT 78, GRNA 79]. These methods start by considering all the simple paths between a given pair of nodes, and then performing some Boolean operations to arrive at the Boolean expression for the probabilistic event of interest. This expression is then used to obtain a terminal reliability expression by using the corresponding network element reliabilities. As an example, in the computer network shown in Figure 1, there exists three different paths between the source S and the destination T. We assume that the nodes are perfectly reliable and write the paths in terms of the links. The paths are: 1) $x_1x_2x_3$, 2) $x_4x_5x_6$, and 3) $x_1x_7x_6$. The terminal reliability between S and T is given by the probability of the event

P(path 1 up) + P(path 1 down)*P(path 2 up)
+ P(paths 1 & 2 down)*P(path 3 up)

The Boolean expression for this event is given by,

$$x_1x_2x_3 + \overline{x_1x_2x_3}\, x_4x_5x_6 + \overline{x_2x_3}\, \overline{x_4x_5}\, x_1x_6x_7$$

A highly efficient algorithm for terminal reliability analysis has been proposed by Grnarov et al., [GRNA 79, GRNA 80a]. This algorithm for symbolic reliability analysis is based on the representation of simple paths by "cubes" (instead of prime implicants). The algorithm introduces a new operation for manipulating the cubes, and the interpretation of resulting cubes in such a way that Boolean and arithmetic reduction are combined. The details of the derivation of the algorithm can be found in [GRNA80a]. The symbolic terminal reliability expression for the above example can be obtained as:

$$R(S \rightarrow T) = p_1p_2p_3 + (1-p_1p_2p_3)p_4p_5p_6$$
$$+ (1-p_2p_3)(1-p_4p_5)p_1p_6p_7 \qquad (1)$$

where S→T represents S-T connectivity.

In a distributed processing system with redundant resources, we will be interested in multi-terminal connectivity, which is needed when running a program at a site that requires files residing at different sites [HILB 80, MERW 80, GRNA 81]. In [GRNA 81] an efficient algorithm was proposed for multi-terminal reliability analysis and is based on the derivation of a Boolean function for multi-terminal connectivity. This algorithm is an extension of the algorithm presented in [GRNA 80a, GRNA 80b] to handle both reliability computation and symbolic reliability analysis.

Another reliability measure of interest in studying computer networks is tree connectivity. The probabilistic event of interest is that there exists at least one path from a particular node to a set of nodes. This is useful in studying the reliability of broadcasting of information from a given node to a set of nodes. For example, referring to Figure 1, we might like to evaluate the reliability of connection from node 8 to nodes 11, 12, and 13 at the same time. The evaluation is more than a simple multiplication of terminal reliabilities, since there are some dependencies. Actually, we can perform a Boolean AND operation on the set of paths for each node pair and obtain the Boolean expression for the tree connectivity. The source S can successfully broadcast information to nodes 11, 12, and 13 if all of the following events are true: 1) x_1x_2, 2) $x_1x_7 + x_4x_5$, and 3) $x_1x_2x_3 + x_1x_7x_6 + x_4x_5x_6$.

We first perform the Boolean AND operation on these three set of paths:

$$(x_1x_2)(x_1x_7 + x_4x_5)(x_1x_2x_3 + x_1x_7x_6 + x_4x_5x_6)$$

After simplifying the Boolean expression becomes:

$$x_1x_2x_3x_7 + x_1x_2x_6x_7 + x_1x_2x_3x_4x_5 + x_1x_2x_4x_5x_6$$

The corresponding reliability expression can then be obtained by using Grnarov's algorithm as:

$$R(S \rightarrow \{11,12,13\}) = p_1p_2p_3p_7 + (1-p_3)p_1p_2p_6p_7$$
$$+ (1-p_7)p_1p_2p_3p_4p_5 + (1-p_3)(1-p_7)p_1p_2p_4p_5p_6 \qquad (2)$$

In the next section, we show the method of deriving the time-dependent reliability and availability expressions by the extension of the Boolean approaches discussed above. This method of dynamic analysis will be shown to be equivalent to the analysis of the Markov modeling approach used to study fault-tolerant computers.

3. DESCRIPTION OF THE METHODOLOGY

The following notions are important to keep in mind when performing the dynamic reliability analysis. First, the network reliability measures should always be qualified with i) the probabilistic event for which the measures are evaluated, ii) performance constraints, and iii) the time interval. For example, the reliability and availability are denoted as:

R(event, constraint, time)
A(event, constraint, time)

The first argument should always be explicit. The second argument may be absent if no performance constraints are specified by the user. The third argument is not needed for the following measures.

MTFF(event, constraint)
MTBF(event, constraint)
SA(event, constraint)

For example, we may be interested in finding time T for which the availability of terminal-pair connectivity is greater than, say, A_0. To include performance constraint, we may want to find the terminal-pair reliability R(t) such that the message delay between source and destinaltion is less than d_0.

The second notion is to describe more realistically, the reliability behavior of the individual network elements. A single probability of success p_i for the i-th element is inadequate. In addition to the topological parameters such as the connection matrix for the network, we need the failure rates and repair rates for the nodes and links. Under the Poisson assumptions for the arrivals of failures and exponential distribution for the repair times, the reliability and availability of i-th element can be expressed as

$$R(x_i,t) = e^{-\lambda_i t}$$

$$A(x_i,t) = \frac{\mu_i}{\lambda_i+\mu_i} + \frac{\lambda_i}{\lambda_i+\mu_i} e^{-(\lambda_i+\mu_i)t}$$

where λ_i and μ_i are the constant failure rate and repair rate (Mean Time To Repair, MTTR, is $\frac{1}{\mu}$) respectively of the i-th element. The expressions for MTFF, MTBF, and SA are simple functions of λ_i and μ_i.

$$MTFF(x_i) = \int_0^{\infty} R(x_i,t)dt = \frac{1}{\lambda_i}$$

$$SA(x_i) = A(x_i,\infty) = \frac{\mu_i}{(\lambda_i+\mu_i)}$$

$$= \frac{MTTR}{MTFF + MTTR}$$

$$MTBF(x_i) = MTFF + MTTR = \frac{1}{\lambda_i} + \frac{1}{\mu_i}$$

If the element failures and repairs are described by general probability distribution functions, we have to resort to Laplace transform techniques to solve for the reliability measures of network elements.

The decomposed reliability model of a network obtained by applying the Boolean algebra rules on the path sets for a given event can now be transformed into a time-dependent model by substituting p_i with $R(x_i,t)$ for a non-repairable network, or with $A(x_i,t)$ for a repairable network. Therefore, for the non-repairable network of Figure 1, referring to equation 1, the time-dependent reliability can be written as:

$$\begin{aligned} R(S\to T,t) &= e^{-(\lambda_1+\lambda_2+\lambda_3)t} + (1-e^{-(\lambda_1+\lambda_2+\lambda_3)t})e^{-(\lambda_4+\lambda_5+\lambda_6)t} \\ &+ (1-e^{-(\lambda_2+\lambda_3)t})(1-e^{-(\lambda_4+\lambda_5)t})e^{-(\lambda_1+\lambda_6+\lambda_7)t} \\ &= e^{-(\lambda_1+\lambda_2+\lambda_3)t} + e^{-(\lambda_4+\lambda_5+\lambda_6)t} + e^{-(\lambda_1+\lambda_6+\lambda_7)t} \\ &- e^{-(\lambda_1+\lambda_2+\lambda_3+\lambda_6+\lambda_7)t} - e^{-(\lambda_1+\lambda_4+\lambda_5+\lambda_6+\lambda_7)t} \\ &- e^{-(\sum_{i=1}^{6}\lambda_i)t} + e^{-(\sum_{i=1}^{7}\lambda_i)t} \end{aligned} \qquad (3)$$

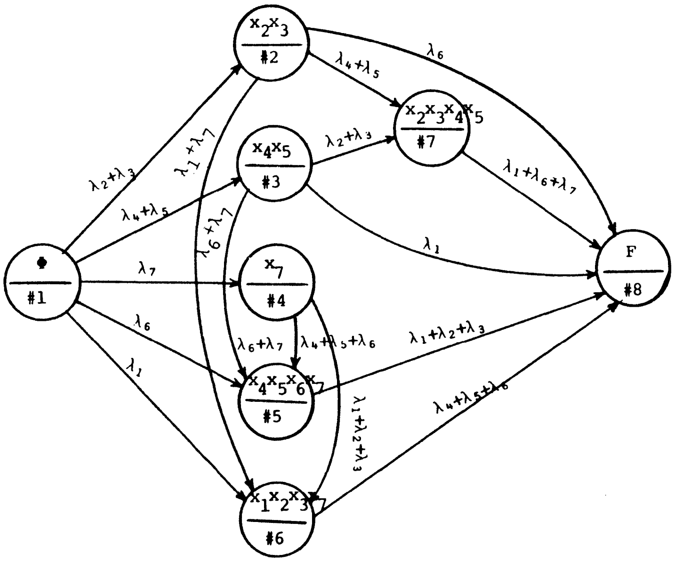


Figure 2. Markov State Transition-Rate Diagram

It can be shown that the above equation is exact and that the analysis is equivalent to the Markov reliability analysis technique. A Markov state transition-rate diagram shown in Figure 2 can be developed for the network of Figure 1. Figure 2 shows the states in which the network satisfies the S-T connection and one failure state (state #8). The transitions between the states represent failure of links. One interesting point to be noted here is that the failure of certain links eliminate some links in series (eg., x_2 eliminates x_3). The failed elements do not contribute to either the reliability or unreliability of the network. The network reliability is given as the sum of the operational state probabilities. The state probabilities are obtained by solving the following set of linear differential equations.

$$\dot{\underline{P}}(t) = Q\,\underline{P}(t) \tag{4}$$

where Q is the state transition-rate matrix and $\underline{P} = (P_1, P_2, P_3, P_4, P_5, P_6, P_7)$. The state probabilities for each of the operational states in Figure 2 can be obtained by solving Equation 4 using Laplace transforms.

$$P_1^*(s) = \frac{1}{(s+\sum_{i=1}^{7}\lambda_i)}$$

$$\begin{matrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{matrix}$$

$$P_7^*(s) = \frac{1}{(s+\lambda_1+\lambda_6+\lambda_7)}\left[\frac{(\lambda_4+\lambda_5)P_1^*(s)}{(s+\lambda_1+\lambda_4+\lambda_5+\lambda_6+\lambda_7)} + \frac{(\lambda_2+\lambda_3)P_1^*(s)}{(s+\lambda_1+\lambda_2+\lambda_3+\lambda_6+\lambda_7)}\right]$$

$$\begin{aligned} R(S\to T,t) &= P_1(t) + P_2(t) + P_3(t) + P_4(t) \\ &\quad + P_5(t) + P_6(t) + P_7(t) \\ &= e^{-(\lambda_1+\lambda_2+\lambda_3)t} + e^{-(\lambda_4+\lambda_5+\lambda_6)t} + e^{-(\lambda_1+\lambda_6+\lambda_7)t} \\ &\quad - e^{-(\lambda_1+\lambda_2+\lambda_3+\lambda_6+\lambda_7)t} - e^{-(\lambda_1+\lambda_4+\lambda_5+\lambda_6+\lambda_7)t} \\ &\quad - e^{-(\sum_{i=1}^{6}\lambda_i)t} + e^{-(\sum_{i=1}^{7}\lambda_i)t} \end{aligned} \tag{5}$$

For complex networks, clearly, the Markov modeling approach becomes very difficult and time-consuming because of the state space explosion. For each probabilistic event considered, the number of states in the Markov model is directly proportional to the branching factor (for example, x_1 and x_4), existence of cross links (for example, x_7), and the depth of the network (for example, x_1, x_2, x_3). When availability is needed, we have to expand the state diagram to account for the non-homogeneity when the repair rates are different for different elements. In view of this drawback of the Markov modeling approach, the Boolean algebraic approach provides an attractive means to achieve both efficiency and functionality. The Boolean method can be applied to all the reliability problems except when the reliability and MTFF are needed for repairable networks. The reason is that the element reliabilities are not dependent on the repair rates whereas the system reliability does. In general, for the same event and the same constraint,

$$R_{closed}(t) < R_{repairable}(t) \le A_{repairable}(t).$$

It is interesting to note that the number of terms in the reliability polynomial (Equation 5) is the same as the number of operational states in the Markov state diagram (Figure 2). Therefore, we can extract the state information by fully expanding the time-dependent reliability expression after substituting R(i,t) for p_i in the symbolic expression (Equation 1). **The number of operational states does not increase exponentially with the number of elements because of the dependencies caused by series elements.**

4. ANALYSIS OF THE BRIDGE NETWORK

In this section, we perform a detailed dynam**ic reliability analysis of the bridge network shown in** Figure 3. This network has five links marked x_1 x_2, ..., x_5 which are bidirectional, and four nodes marked x_6, x_7, x_8, x_9. This network is one of the stan**dard networks analyzed by many researchers. We first use the efficient Boolean technique explained** in the previous section to obtain a reliability ex**pression for the event of interest and then translate it to a time-dependent expression by using the corresponding element reliability or availability. We assume that the i-th link has a constant failure rate λ_i and a constant repair rate μ_i.**

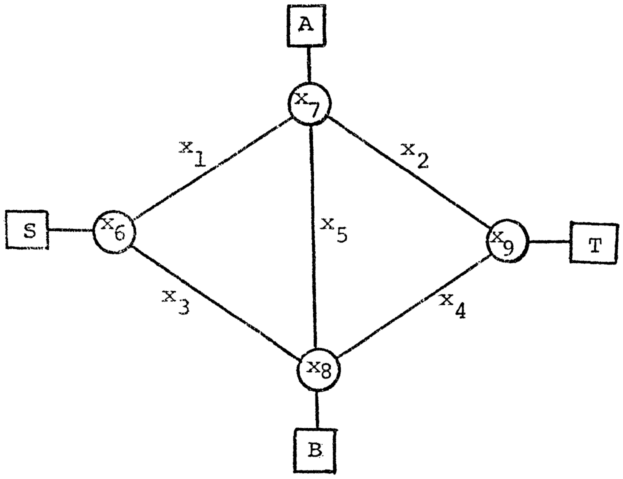


Figure 3. The Bridge Network

We first analyze terminal pair connectivity between S and T. The terminal reliability expression for this event using the Boolean algebraic approach [GRNA 80a] is:

$$\begin{aligned} R(S\to T) &= p_1p_2 + p_3p_4(1-p_1p_2) + p_1p_5p_4(1-p_2)(1-p_3) \\ &\quad + p_3p_5p_2(1-p_1)(1-p_4) \end{aligned}$$

Now the time-dependent terminal pair reliability expression when no repair is possible is given by,

$$R(S\to T) = e^{-(\lambda_1+\lambda_2)t} + e^{-(\lambda_3+\lambda_4)t} + e^{-(\lambda_1+\lambda_4+\lambda_5)t}$$

$$+ e^{-(\lambda_2+\lambda_3+\lambda_5)t} - e^{-(\lambda_1+\lambda_2+\lambda_4+\lambda_5)t}$$

$$- e^{-\sum_{i=1}^{4}\lambda_i t} - e^{-(\lambda_1+\lambda_3+\lambda_4+\lambda_5)t} + 2e^{-\sum_{i=1}^{5}\lambda_i t}$$

$$- e^{-(\lambda_1+\lambda_2+\lambda_3+\lambda_5)t} - e^{-(\lambda_2+\lambda_3+\lambda_4+\lambda_5)t}$$

Assuming that all links have the same failure rate λ, terminal reliability between S and T becomes:

$$R(S\to T) = 2e^{-2\lambda t} + 2e^{-3\lambda t} - 5e^{-4\lambda t} + 2e^{-5\lambda t}$$

We can calculate the mean time to first failure as a function of failure rates using the definition of Section 2 as:

$$MTFF(S\to T) = \frac{1}{(\lambda_1+\lambda_2)} + \frac{1}{(\lambda_3+\lambda_4)} + \frac{1}{(\lambda_1+\lambda_4+\lambda_5)}$$

$$+ \frac{1}{(\lambda_2+\lambda_3+\lambda_5)} - \frac{1}{(\lambda_1+\lambda_2+\lambda_4+\lambda_5)}$$

$$- \frac{1}{(\lambda_1+\lambda_2+\lambda_3+\lambda_4)}$$

$$- \frac{1}{(\lambda_1+\lambda_3+\lambda_4+\lambda_5)} + \frac{2}{(\lambda_1+\lambda_2+\lambda_3+\lambda_4+\lambda_5)}$$

$$- \frac{1}{(\lambda_1+\lambda_2+\lambda_3+\lambda_5)} - \frac{1}{(\lambda_2+\lambda_3+\lambda_4+\lambda_5)}$$

When all links have the same failure rate we have:

$$MTFF(S\to T) = \frac{2}{3\lambda} - \frac{1}{4\lambda} + \frac{2}{5\lambda}$$

$$MTFF(S\to T) = \frac{49}{60\lambda}$$

It is interesting to observe that MTFF for the S-T connection is less than that of a simplex link for which MTFF is $\frac{1}{\lambda}$. This is due to the fact that the increased hardware complexity of the links involved in the communication causes the average time-to-failure to reduce.

We compute the availability for the event of terminal pair connectivity between S and T with the assumption that all links have the same failure rate λ and same repair rate μ. The time dependent terminal-pair availability expression can be obtained from the reliability expression given above by substituting the following expression for x_i.

$$x_i = \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu}e^{-(\lambda+\mu)t}$$

A much simplified expression can be obtained for steady state availability by substituting $a = \frac{\mu}{\lambda+\mu}$ for x_i in the reliability expression. That is,

$$SA(S\to T) = 2a^2 + 2a^3 - 5a^4 + 2a^5$$

The mean time between failures is given by the same expression with $a = \frac{1}{\lambda} + \frac{1}{\mu}$.

Next we consider a tree connectivity event, which is simultaneous connection between S and A, and S and B. The paths in Boolean terms is:

$$S\to A = x_1 + x_3x_5 + x_3x_4x_2$$
$$S\to B = x_3 + x_1x_5 + x_1x_2x_4$$

We take the logical AND of these two to obtain Boolean terms for the tree connectivity as:

$$S\to\{A,B\} = x_1x_3 + x_1x_5 + x_3x_5 + x_1x_2x_4 + x_2x_3x_4$$

From this we obtain the reliability expression for this event as,

$$R(S\to\{A,B\}) = p_1p_3 + (1-p_3)p_1p_5 + (1-p_1)p_3p_5$$

$$+ (1-p_3)(1-p_5)p_1p_2p_4 + (1-p_1)(1-p_5)p_2p_3p_4$$

With the assumption that all links have the same failure rate λ, the time dependent expression for this event is:

$$R(S\to\{A,B\},t) = 3e^{-2\lambda t} - 4e^{-4\lambda t} + 2e^{-5\lambda t}$$

The mean time to first failure is then,

$$MTFF(S\to\{A,B\}) = \frac{3}{2\lambda} - \frac{1}{\lambda} + \frac{2}{5\lambda}$$

which simplifies to:

$$MTFF(S\to\{A,B\}) = \frac{9}{10\lambda}$$

When repair is available, the steady state availability for this tree connectivity event is given by:

$$SA(S\to\{A,B\}) = 3a^2 - 4a^4 + 2a^5$$

where $a = \frac{\mu}{\lambda+\mu}$.

The mean time between failures is given by the same expression with $a = \frac{1}{\lambda} + \frac{1}{\mu}$.

5. SUMMARY

Dynamic reliability modeling and analysis of computer networks and distributed processing systems were presented. A systematic method of obtaining time-dependent reliability expressions for various events such as terminal-pair connectivity, tree connectivity, and multi-terminal connectivity were discussed. The approach uses well known Boolean technique to obtain reliability expressions and then transforming it to the corresponding time-dependent expressions. Other important reliability measures such as availability, MTFF, and MTBF were also studied. A detailed analysis of the bridge network was also presented. Further work involves in studying these reliability problems of computer networks under different performance constraints such as delay, throughput, and resource allocation.

ACKNOWLEDGMENTS

The authors wish to thank Prof. A. Avizienis for his encouragement and support during this research and Ms. Mildred Covin for her help in the preparation of the final manuscript.

REFERENCES

[BALL 80] M. O. Ball, "Complexity of Network Reliability Computations," Networks, Vol. 10, 1980, pp. 153-165.

[FRAT 73] L. Fratta, U. G. Montanari, "A Boolean Algebra Method for Computing the Terminal Reliability in a Communication Network", IEEE Trans. on Circuit Theory, Vol. CT-20, No. 3, May 1973, pp 203-211.

[FRAT 76] L. Fratta, U. G. Montanari, "Synthesis of Available Networks", IEEE Transactions on Reliability, Vol. R-25, No. 2, June 1976, pp 81-86.

[FRAT 78] L. Fratta, U. G. Montanari, "A Recursive Method Based on Case Analysis for Computing Network Terminal Reliability, IEEE Transactions on Communications, Vol. COM-26, No. 8, August 1978, pp 1166-1177.

[GRNA 79] A. Grnarov, L. Kleinrock, M. Gerla, "A New Algorithm for Network Reliability Computation", Computer Networking Symposium, Gaithersburg, Maryland, December 1979, pp 17-20.

[GRNA80a] A. Grnarov, L. Kleinrock, M. Gerla, "A New Algorithm for Symbolic reliability Analysis of Computer Communication Networks", Pacific Telecommunications Conference, January 1980.

[GRNA80b] A. Grnarov, L. Kleinrock, M. Gerla, "A New Algorithm for Reliability Analysis of Computer Communication Networks", UCLA Computer Science Quarterly, Spring 1980.

[GRNA 81] A. Grnarov, M. Gerla, "Multiterminal Reliability Analysis of Distributed Processing Systems", Proc. of the 1981 International Conference on Parallel Processing, August 1981, pp 79-86.

[HANS 72] E. Hansler, "A Fast Recursive Algorithm to Calculate the Reliability of a Communication Network, IEEE Transactions on Communications, Vol. COM-20, No. 3, June 1972.

[HANS 74] E. Hansler, G. K. McAuliffe, R. S. Wilkov, "Exact Calculation of Computer Network Reliability", Networks, 4, (1974), pp 95-112.

[HILB 80] G. Hilborn, "Measures for Distributed Processing Network Survivability", Proc. of the 1980 National Computer Conference, May 1980, pp 157-163.

[MERW 80] R. E. Merwin, M. Mirhakak, "Derivation and Use of a Survivability Criterion for DDP Systems", Proc. of the 1980 National Computer Conference, May 1980, pp 139-146.

[ROBE 70] L. G. Roberts, B. D. Wessler, "Computer Network Development to Achieve Resource Sharing", AFIPS Conference Proceedings, SJCC, New Jersey 1970.

[SIEW 82] D. P. Siewiorek, R. S. Swarz, "The Theory and Practice of Reliable System Design" Digital Press, 1982.

FUNCTIONAL SPECIFICATION OF DISTRIBUTED SYSTEMS

Gruia-Catalin Roman and Robert K. Israel
Department Of Computer Science
Washington University
Saint Louis, Missouri 63130

Abstract -- A formal Distributed Systems Design Language (DSDL) is proposed. The language design places a very strong emphasis on the systematic application of the principle of separation of concerns. In DSDL, systems are described as nets of communicating processes. The language allows designers to define arbitrary communication protocols and provides a means for protocol encapsulation. DSDL is illustrated by means of a highly simplified annotated example representative of the nature of the language.

Acknowledgements -- This work was partially supported by Rome Air Development Center and by the Defense Mapping Agency under contract F30602-80-C-0284. The contribution of J. T. Love to the initial work on DSDL is also acknowledged.

Introduction

This paper reports on one effort to develop a formal Distributed Systems Design Language (DSDL). In DSDL, systems are described as nets of communicating processes. Each process in the net has its own local data over which it has sole control and procedures that specify primitive and indivisible operations over the data. The process also has the ability to exchange messages with other processes in the net. The behavior of the process specifies the order in which its procedures are invoked. Sequences of procedures are allowed to execute concurrently within the process.

Several considerations have influenced heavily the nature of the DSDL: an emphasis on formality, a desire to promote the principle of separation of concerns, a need to support hierarchical specifications, and an aim toward generality. Formality is achieved through the use of set theoretical models for data representation and by employing predicate calculus in defining the procedures (using input/output assertions). The principle of separation of concerns is reflected by the manner in which the definitions of the net and of the process are structured; they are meant to enhance the designer's ability to describe the system in terms of clean abstractions. Hierarchical descriptions of the system are enabled by the fact that processes may be refined into nets. Finally, the generality of the language is enhanced by its capacity to describe a variety of communication structures and protocols.

DSDL is described below by presenting the formal model of distributed systems on which it is based and a small example of its use.

Language Definition

Process Definition

The process is the basic functional unit of DSDL, and is similar to the guardian [1] and the monitor [2] (except that internal parallelism is allowed). It serves to encapsulate data as in the abstract data type [3,4], and has sole access to its own data. In addition, a process is able to receive and send information via messages as in [5]. A set of procedures are defined for the process which perform indivisible operations on data or communicate with the environment. The behavior of a process is then defined as the set of allowable sequences of procedure invocations within the process.

```
A process p is defined as a five-tuple
p = (Dp, Tp, Rp, Sp, Bp)
where
Dp = (Qp, Hp, Ip)
     Dp - process data definition
     Qp - set of data entities
     Hp - data invariant assertion
     Ip - initial value assertion
Tp = {z | z = (Ain(Dp),  Aout(Dp, Dp'))}
     Tp   - transformational procedures
     Ain  - input assertion
     Aout - output assertion
Rp = {z | z = ('true',  Aout(Dp))}
     Rp - message receiving procedures
Sp = {z | z = (Ain(Dp),  'true')}
     Sp - message sending procedures
Bp SUBSET.OF (Tp U Rp U Sp)*
     Bp - process behavior, given as a set
          of procedure invocation sequences
```

Data. The first element in the 5-tuple representing the process p is the data Dp which belongs to that process. This in turn is defined as a triple (Qp, Hp, Ip). Qp is a set of data entities controlled by p and whose elements may be accessed only by procedures within p. Hp is the data invariant, which is a predicate describing the properties which must be possessed by the elements of Qp both before and after all data transformations The last element, Ip, is a predicate defining the initial values for each data entity in Qp.

The data controlled by a process appears in the "DATA" section of the process definition. Both variables and constants may be declared using statements whose syntax resembles Pascal. The variable declarations are placed side by side with predicates that are taken to be parts of the invariant Hp (e.g., VAR n: INTEGER; n>0;). Both variables and constants are either sets or

0190-3918/83/0000/0503$01.00 © 1983 IEEE

elements of sets. Some sets are assumed to be built-in (e.g., INTEGER) while others are constructed by enumeration (e.g., S={1,2,3}), by providing an intensional definition (e.g., S={z ¦ 0<z<4 AND INTEGER(z)}), or by means of standard set operations (e.g., union, intersection, etc.). In addition to sets, a notation for functions and relations is also available.

Procedures. Process activities, data transformations, and message exchanges are defined by the procedures it controls. Transformational procedures, given in terms of input and output assertions, describe the state changes which occur during process execution. Message exchanges are carried out by two built-in procedures (SEND and GET) whose semantics are stated in the communication section of the net definition.

The use of nonprocedural specifications in defining the meaning of the transformational procedures enhances the understandability of the process specification. Furthermore, by treating procedure invocations as primitive operations over the data, the need for synchronization within a process is avoided in the same way as it is done in the monitor concept of concurrent Pascal [6] (but without prohibiting concurrency from occurring in the process).

Syntactically, the definition of transformational procedures is straightforward. Pairs of input ("IN:") and output ("OUT:") assertions are used to cover distinct cases. When an input assertion is followed by several output assertions a conjunction between them is implied. An exception ("EXPT:") assertion may be provided to indicate the action to be taken in case all input assertions fail (e.g., NIL, RESTART, ABORT, etc.). Standard predicate calculus is used in constructing the assertions (AND, OR, XOR, NOT, IF-THEN-ELSE, i.e., implication), and a name followed by a single quote within an assertion denotes the value of a data item after the completion of the procedure.

Behavior. The behavior of a process is defined as the set of all allowable sequences of events within a process, where an event is defined as an invocation of a procedure. Constructs available for the behavior specification include event sequence (BEGIN - END), concurrent event sequences (PARBEGIN - PAREND), conditional (IF - THEN - ELSE), nondeterministic selection (CASE), and repetition (WHILE, DOUNTIL, LOOP).

Net Definition

In order to specify a distributed system, the concept of a net is included in DSDL. A net is defined as a set of independent, concurrent processes which communicate among themselves by means of messages [2,7,8]. Messages are sent over abstract communications paths called links. The behavior of these links, along with the behavior of each process, determines the behavior of the net as a whole.

A net n is defined as a four-tuple

```
n = (P, P', L, C)
where
P  = { p ¦ p is a process }
P' SUBSET.OF P
   P' contains those processes considered
   to be part of the environment.
L  = { l ¦ l SUBSET.OF P }
   where a link l is defined by the set of
   processes that may use it.
C  : L --> POWERSET.OF ((UNION OVER ALL p OF
             (Sp UNION Rp))* )
   with
   C(l) SUBSET.OF (UNION OVER ALL p
             MEMBER.OF l OF (Sp UNION Rp))*
   i.e., C(l) establishes the link behavior
   which determines the set of allowable
   sequences of send and receive events over
   the link.
```

Processes. The first two elements in the 4-tuple describing the net are the set of processes P and a set P' such that P' SUBSET.OF P. P' contains those processes in P which are considered to be part of the environment. In DSDL, these processes are declared with an "EXTERNAL" attribute. Whenever the environment plays only a marginal role in the specification of the system, however, the external processes and the links that connect them to the system may be declared to be undefined.

Processes at one level of the specification may represent abstractions of entire nets to be identified later. DSDL allows one to state this fact through the use of the attribute "REFINEMENT.OF" as in the example below:

```
     NET netname; REFINEMENT.OF pname.
     ...
     END-NET netname.
```

where the net "netname" is identified to be a refinement of the process "pname". Furthermore, any entity in the net may be declared to be a refinement of some entity in the process as long as consistency is preserved.

Links. The last two elements in the net definition are the set of links L and the communications protocol C. The links represent the available paths of communication within the net. For each link l a set C(l) of allowable sequences of send and receive type events in the processes that it connects is defined. This set describes the communications protocol on the link, and is referred to as the link behavior. The link behavior is specified in the same way as the process behavior.

In DSDL, each link is defined by a unique name, the processes connected by it, and a description of its behavior given in the section on communication. More than two processes may have access to the same link and the same two processes may have more than one link in common. The motivation for this approach is to be found in the desire to enable the description of arbitrary interconnection structures. Furthermore, a link may be later refined as a net that implements the behavior of the link. A packet switching net, for instance, may be described first as a link between all the nodes it services and may be subsequently refined to include the switching nodes and their protocols.

For every link in the net, a behavior description has to be included in the communication definition section. The link behavior defines the communication protocol associated with the link, i.e., the semantics of the GET and SEND commands. In defining the link behavior, the designer may use the same means of specification as in the description of a process behavior except that the set of events that may be involved is restricted to the invocation of receiving procedures, the invocation of sending procedures, and the resumption of processing after the invocation of sending or receiving procedures in processes associated with the link. The resumption of an event sequence in a process is specified by the event "pname:GO" in the link behavior.

Conclusions

DSDL has been exercised on several small problems. These exercises were useful in demonstrating the language's power of expression. Nevertheless, there are many unresolved issues. First of all, a meaningful evaluation of the language demands its use on a real-life project of adequate complexity. Second, future advances in the study of the formal aspect of the language must be carried out in preparation for potential incorporation of DSDL in a computer-aided design system. As of now, a definition for consistency between levels has been proposed but proof strategies need to be developed. Finally, a complete system specification language ought to include the capability to define processors and their characteristics, the rules for allocating processes among processors, and performance specifications. Research in these areas is currently under way and the results will be reported elsewhere.

References

[1] Liskov, B., and Berzins, V., "An Appraisal of Program Specifications," Research Directions in Software Technology, P. Wegner, editor, MIT Press, pp. 276-301, 1979.

[2] Riddle, W. E., et al, "Behavior Modeling During Software Design," IEEE Trans. on Soft. Eng. SE-4, No. 4, pp. 671-678, July 1978.

[3] Guttag, J., "Abstract Data Types and the Development of Data Structures," CACM 20, No.6, pp. 396-404, June 1977.

[4] Wulf, W. A., London, R. I., and Shaw, M., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Soft. Eng. SE-2, No. 4, pp. 243-265, December 1976.

[5] Hoare, C. A. R., "Communicating Sequential Processes," CACM 21, No. 8, pp. 666-677, August 1978.

[6] Hansen, B., The Architecture of Concurrent Programs, Prentice-Hall, 1977.

[7] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.

[8] Feldman, J. A., "High Level Programming for Distributed Computing," CACM 22, No. 6, pp. 353-368, June 1979.

Example

```
NET consumer_producer.

     PROCESS producer.
     DATA.
        VAR count : INTEGER; count > 0;
     INITIALIZATION.
        count = 1;
     PROCEDURE w := prepare.
        IN:  count = n AND n > 0;
        OUT: w' = (n,t) AND CHARSTRING(t) AND
             count' = n + 1;
        EXPT: NIL;
     BEHAVIOR.
        LOOP {w := prepare;
              SEND w TO consumer ON channel;}
     END-PROCESS producer.

     PROCESS consumer.
        VAR count : INTEGER; count > 0;
     INITIALIZATION.
        UNDEFINED.
     PROCEDURE use(w);
        IN:  w = (n, t) AND n MEMBER.OF INTEGER
             AND n > 0 AND CHARSTRING(t);
        OUT: count' = n;
        EXPT: NIL;
     BEHAVIOR.
        LOOP {GET(w) FROM producer ON channel;
              use(w);}
     END-PROCESS consumer.

     LINKS.
        channel: (producer, consumer);

     COMMUNICATION.
        channel:
        LOOP {
           producer:SEND(z) TO consumer;
           consumer:GET(z) FROM producer;
           {producer:GO; // consumer:GO;}  }

END-NET producer_consumer.
```

MOPAC
A PARTITIONABLE AND RECONFIGURABLE MULTICOMPUTER ARRAY

Wong-Hua Lee and Miroslaw Malek

Department of Electrical Engineering
University of Texas at Austin
Austin, Texas 78712

Abstract

The design of a VLSI compatible, reconfigurable, partitionable multicomputer array is presented in this paper. The resources of the system can be divided into arbitrary size rectangular partitions, and various computational structures can be configured on each partition by distributed configuration process.

A scheme to provide each partition with a private bus is introduced. This bus is essential for instruction broadcast and also augments the communication capability of each partition.

Design of a special purpose hardware unit responsible for the process of partition allocation is described. This unit assures that the allocation process will not degrade the entire system performance.

I. Introduction

Due to the recent advances in VLSI technology, parallel processing systems which consist of thousands of processors have become feasible [1]. To reduce design turn around time for such enormous systems and to achieve high densities in VLSI, simple and regular interconnection schemes are highly desirable [2]. Out of the many multicomputer interconnection structures that have been proposed, the mesh type networks such as those employed in ILLIAC IV[3] and systolic arrays[2] are especially appealing because they possess the property of simplicity, regularity, modularity, linear cost growth, and ease of routing. All these merits make the mesh type networks suitable for VLSI implementation of large scale computer systems.

In this paper, the system design of the Mesh Organized Partitionable Array Computer (MOPAC), a partitionable SIMD/MIMD (PSM)[4] multicomputer system with an architecture based on the two dimensional mesh type interconnection network, is described. Some of the features of MOPAC are:

. The processors in the mesh network can be partitioned into rectangular submeshes (partitions). Each user's job can be executed on one or more of these partitions. Each partition can work in either the SIMD or the MIMD mode.

. A single user's partitions can communicate with one another and different users' partitions are isolated from one another.

. The size of a rectangular partition can be arbitrary. There is no limitation on the size of the smallest partition that needs to be allocated, therefore high resource utilization is possible.

. The processors in each partition can be configured into various computational structures. Distributed algorithms are designed to carry out the configuration process.

Since each partition may have to operate in the SIMD mode, it must be provided with a communication medium for the broadcasting and receiving of common instructions and operands. Moreover, these media among different partitions should not interfere with one another. In MOPAC, a scheme called the Partitionable Bussing System has been developed to equip each partition with a private bus. In addition to being used for broadcasting, this bus also allows nonadjacent processors in a partition to exchange information directly, therefore enhances the general communication and synchronization capability of the partition.

One of the most time consuming operations for the control unit of MOPAC is the allocation of partitions from the mesh network according to each job's demand. The speed of this process profoundly affects the entire system performance. Due to the lowering hardware costs and speed improvements, it has become natural for system designers to incorporate many of the repetitious operating system functions into hardware modules. In MOPAC, a special purpose hardware mechanism called the Partition Allocation Unit is designed to administer the allocation process. This unit reduces the overhead incurred by allocation to the minimum.

In Section II the overall organization and operation of MOPAC is described. Section III presents the Partitionable Bussing System, and in Section IV the operation of the Partition Allocation Unit is discussed. The conclusion follows in Section V.

II. System Organization and Operation

Each processing element (PE) in MOPAC consists of three components: an application processor (AP), a memory unit, and a communication processor(CP). The application processor is responsible for the execution of users' programs, which are stored in the memory unit along with data. The communication processor is responsible for communicating with other PE's and the Host.

This research was supported in part by the National Science Foundation Grant MCS-8116099.

All communication processors in the system are organized as an nxn two dimensional mesh, with corresponding pairs of edge processors connected to form the topology equivalent to a torus (Fig.1).

Each communication processor is also connected to segments of the Partitionable Bussing System (PBS). When a particular structure on a partition is formed, the PBS bus segments of the individual PE's in this structure will be tied together as a single bus.

The Host is the unit which coordinates the activities at the system level. Its responsibilities include user program development, input and output handling, determining the proper size of the partition(s) required for a user's job, allocation of the partitions from the mesh network, and generation of the initial structure configuration messages.

All CP's and the Host are connected to the System Bus (SB). It is through this bus that the Host can communicate with each PE when required. The principal uses of the System Bus are:

.After a partition is formed, the Host will use SB to transfer initial configuration messages to a proper cell (called the Initial Cell) in the partition to start the structure configuration process.

. When a job in a partition finishes execution, the Initial Cell will report this status to the Host through SB.

. If a user's job has two or more partitions executed in parallel, these partitions can exchange information through SB.

. For fault diagnosis and fault tolerance purposes, x copies of a single job may run on x partitions simultaneously. The partitions may use the SB to communicate with each other and verify the results.

The partitioning philosophy of MOPAC is as follows. For a large class of numerical problems, image processing problems[5], and systolic algorithms[2], the two dimensional mesh is one of the most appropriate interconnection structure. It is also observed [6][7] that many other SIMD or MIMD type of computation structures, such as linear array, pipeline, binary tree, and ring, can be embedded in rectangular mesh. In addition, to partition a mesh network, the simplest and most natural way is to partition it into sub-meshes. Considering the above, MOPAC will allocate rectangular partitions of the mesh for each user's job.

The allocation process of MOPAC is outlined below. After the Host has compiled a user's program and determined the dimensions of the rectangular partition(s) needed[7], the Host will have to allocate the partitions from the available PE's on the mesh. Since there might have other user's jobs (i.e. other partitions) which occupy part or all of the PE's in the mesh network (Fig.2), it becomes quite difficult for the Host to search for a partition of the required size. This searching process is aided by the Partition Allocation Unit (PAU). If PAU cannot find the desired partition, the job will have to wait in a queue, until some of the jobs on the mesh finish processing and release their partitions. If the partition is found, the configuration process for the particular job structure may start.

The organization of the communication processor and the distributed configuration process are similar to those as given in [6]. There are five registers defined in the communication processor:

ID Register: Contains the address (row and column indices) of the cell.

Mode Register(MR): Indicates the mode of operation of each cell. Each cell can be either a processing cell (PC) participating in the processing of a given structure or a connecting cell (CC) whose role is to transfer data in different directions without performing any kind of processing.

Configuration Register (CR): Contains information about the present structure, and the logic address of the cell within the structure.

Predecessor Register: A four bit register which stores the directions of the predecessor cells in each structure. Each bit corresponds to an element of {W,S,N,E}, the four directions on a compass.

Successor Register: Similar to the Predecessor Register except it stores the directions of the successor cells in each structure.

The configuration process starts when the Host sends a configuration message to the Initial Cell in the partition. The general form of the configuration message is:

M(structure type, size of the structure, size of the partition p and q, level within a structure, direction)

The structure type is the code name for the current structure that is being configured, such as LA stands for linear array. Size of the structure indicates the size of the current structure. For example if the configuration is a linear array with k elements, size of the structure will be k. Size of the partition p and q indicates the row and column widtn of the current partition that the job got allocated. This part of the message is vital because by knowing this information, the CP's will not try to construct a structure outside its own partition, therefore preserve independence among different partitions. Level within a structure indicates the level within a structure that the next processing cell should assume. For example the first element in a linear array structure usually assumes level 0, the second element level 1, etc. This level number may also be viewed as the logical address of the cell in a structure. Direction d is a parameter which directs the transmission direction of future configuration messages. One of the functions defined on d is op(d), which is the 180-degree opposite direction to d. For instance op(S)=N, op(E)=W, etc.

After the Initial Cell receives the initial configuration message, it will examine the contents of the message, modify the values of some of the parameters, and pass the message to proper neighboring cells in a similar format. The neighbors will repeat the same procedure, until the whole structure is configured on the partition. Since the configuration is done in such a distributed manner, the burden that would have been put on the Host and System Bus if the configuration is done in a centralized way is eliminated.

For illustration, the distributed algorithm to configure a linear array of size k on a pxq ($pq\geq k$) partition is shown below. The position of the Initial Cell is at the northwestern corner of the partition, and the initial configuration message generated by the Host is M(LA,k,q,0,q,E).

```
For each cell:
Receive M(LA,k,q,l,c,d) from predecessor;
Begin
   CR:=LA,l;(*configure to the lth element
              of the linear array*)
   if l<>k-1
    then (*configuration process not
           complete*)
      if c=1
       then(*come to the edge of
             partition*)
         transmit M(LA,k,q,l+1,q,op(d)) to
                                 S-neighbor
        else
         transmit M(LA,k,q,l+1,c-1,d) to
                                d-neighbor
    else (*l=k-1*)
      configuration process complete
End.
```

The message transmission pattern is shown in Fig.3.

The c parameter in the message is used as a counter to detect if the message has reached a CP on the edge of the partiton. At the beginning of each row in the transmission path, c is set to q, the column width of the partition. As the message is passing from CP to CP, c is decremented by 1 each time, until when the processor at the edge of the partition sees c=1. At this point, the edge processor will reset the value of c to q, reverse the direction parameter, and pass the message to the S-neighbor.

Configuration algorithms for structures such as square array, ring, and tree can be found in [6][7].

The private bus for the partition is also formed during the configuration process (Section III). When the process is finished, the last CP will broadcast "configuration complete" signal to other CP's in the partition through the private bus, and the execution of the job can begin.

III. The Partitionable Bussing System

As shown in Fig.4, Each CP has four PBS bus segments. For CP(i,j), the E-segment is connected to the W-segment of CP(i,j+1), the N-segment is connected to the S-segment of CP(i-1,j), etc. Intervening between a pair of segments is a switch, whose state will determine if the two segments are connected.

The state of the switch is controlled by the setting of the bits in the Successor Register. When the configuration message for a particular structure is transferred from a cell to its d-neighbor ($d\in\{W,S,N,E\}$), the d-bit in the Successor Register of this cell is set. Since this bit also controls the state of the corresponding switch, the bus segments between this cell and the d-neighbor can now communicate via the turned on switch.

As the configuration messages are being transferred from CP to CP, the switches between predecessor cells and successor cells are turned on accordingly. After the configuration process is completed, the bus segments belonging to the cells of the same structure are all connected, and they together serve as one single bus for the partition. Since the configuration message for a job is designed to run in its own partition only, switches among CP's in different partitions will not be activated, so the bussing system among different partitions will be isolated from one another.

After a job finishes execution, each CP in the partition will clear its individual Successor Register, and the PBS segments are disconnected from one another. The released resources in this partition are now ready to participate in new partitions.

Providing an additional bus for the computational structure configured on a partition augments the communication capability of the structure. It was shown in [8] how the complexity of finding the maximum element on a square array structure is reduced by using the global bus. Another example is the binary tree structure. Though it is expected that most communications would occur locally between parent and child nodes, for the occasional situation that remote leaf nodes need to communicate, the global bus can be used without having to relay the messages up and down the entire tree.

IV. The Partition Allocation Unit

The core of PAU (Fig.5) is an associative memory, where searching for a particular bit pattern can be conducted by each word in parallel. The bit pattern is stored in the Comparison Register, and for those bits that do not participate in a certain search, they can be masked out by setting corresponding bits in the Mask Register to logic 0. If the search is successful in a memory word, the corresponding bit in the Match Register will be set to 1, otherwise it will be set to 0.

The size of the associative memory (AM) is nxn, the same as that of the mesh network. The contents of each cell in AM is set to 1 if the corresponding processor in the mesh is free, or to 0 if the processor is busy. After each allocation for a new partition, and after the completion of the job on an existing partition, the state of the corresponding cells in AM will be updated accordingly. All faulty processors will also be marked as busy. In other words, the AM serves as a hardwired bit map which keeps the status of all processors in the mesh network.

For the purpose of PAU operation, all the bits in the Comparison Register are set to 1, and use only the Mask Register (which is connected as a circular shift register) to control the searching process. When a job demands a rectangular partition of size pxq, the Host will issue a search command to the control unit of PAU. The search process starts by setting the first q bits in the Mask Register to 1 and the initiation of the AM operation. All words in AM will examine in parallel whether their first q bits are 1, and if so, the corresponding flags in the Match Register are set.

The result of the above process actually tells the availability of any partition with a column width of q in the first q columns of the mesh. The remaining task of selecting the partition with row width p lies in examining the contents of the Match Register. If there are p consecutive bits in the Match Register that are set, a partition of the desired size is found. The <u>AND Network</u> following the Match Register is designed to provide this function.

There are n stages in the AND Network, and each stage has n outputs. Let the kth bit of the Match Register be denoted by MR(k), and the output of the jth AND gate in stage i ($i\epsilon[1,n], j\epsilon[0,n-1]$) be denoted by OUT(i,j), then

$$\mathrm{OUT}(i,j)= \bigcap_{k=j}^{j+i-1 (\mathrm{mod}\ n)} \mathrm{MR}(k)$$

where $\cap$ denotes the AND operation

In words, the state of OUT(i,j) tells whether there are i consecutive 1's starting with the jth bit in the Match Register. If the desired dimension of partition is pxq, then the function

$$\mathrm{FOUND}= \bigcup_{j=C}^{n-1} \mathrm{OUT}(p,j)$$

where $\cup$ denotes the OR operation, would reveal if at least one such partition exists.

To know if a partition exists is not enough. If it does, the Host must also know the location of the partition. This is the function of the priority encoder shown below the AND Network in Fig.5. The connection between the outputs of each stage of the AND Network and the inputs to the priority encoder is controlled by a set of n OUT-CONTROL (OC) signals. When OC(p) is set, only the outputs from stage p, i.e. OUT(p,j), $0\leq j\leq n-1$, will be fed into the priority encoder. If there is only one OUT(p,j) true, j will appear at the output of the priority encoder, which is marked as ROW-ADDRESS in Fig.5. If there are more than one j's such that OUT(p,j) is true, then only the smallest of such j will be selected as ROW-ADDRESS by the priority encoder.

The above procedure constitutes one cycle of operation of PAU. At the end of the first cycle, if control unit sees that FOUND is false, this indicates that there is no partition of size pxq exists in the first q columns of the mesh. To determine if such partition exists in the second q columns, the control unit will shift the Mask Register right by 1 bit, and a second search cycle begins. This process continues until at a certain cycle when FOUND turns out to be true, or until after n cycles, when FOUND is false in every cycle, then the control unit can tell that there is no partition of size pxq exists. If FOUND is true in cycle c ($c\epsilon[0,n-1]$), the row address of the cell in the northwestern corner of the partition is shown at the output of the priority encoder, ROW-ADDRESS. The column address is c.

The PAU is amenable to be implemented on a special purpose VLSI chip. The structure of the associative memory and the AND network are regular, therefore the design of elementary cells and the layout of them are straightforward. In addition the circuit complexity of PAU is proportional to N, the number of PE's in the mesh, while the number of pins required is proportional to log $\sqrt{N}$. This high circuit complexity to pin ratio makes PAU ideal for single chip implementations.

V. <u>Conclusion</u>

As the design cost and design cycle time seem to be two major obstacles in the development of large scale VLSI systems, a highly regular and reconfigurable architecture such as the MOPAC system becomes especially valuable. Since MOPAC is regular, the initial design man-year effort is significantly reduced. Since MOPAC is reconfigurable, it can provide a wide range of capabilities and the custom design cost for individual applications is eliminated. In addition, the independent bus within each partition combined with direct interprocessor connections provide an excellent environment for jobs exhibiting strong locality of shared resources. Thus it is concluded that MOPAC can be truly considered as one of the viable VLSI architectures.

<u>References</u>

[1] L.S.Haynes, R.L.Lau, D.P.Siewiorek, D,W.Mizell, " A Survey of Highly Parallel Computing," <u>Computer</u>, Jan. 1982. pp. 9-24.

[2] C.A.Mead, L.A.Conway, <u>Introduction to VLSI Systems.</u> Reading, MA. Addison- Wesley, 1980. Section 8.3.

[3] W.J.Bouknight, S.A.Denenberg, D.E. McIntyre, J.M.Randall, A.H.Sameh, D.L.Slotnick, " The ILLIAC IV System," <u>Proc.</u> <u>IEEE</u>, April 1972. pp. 369-379.

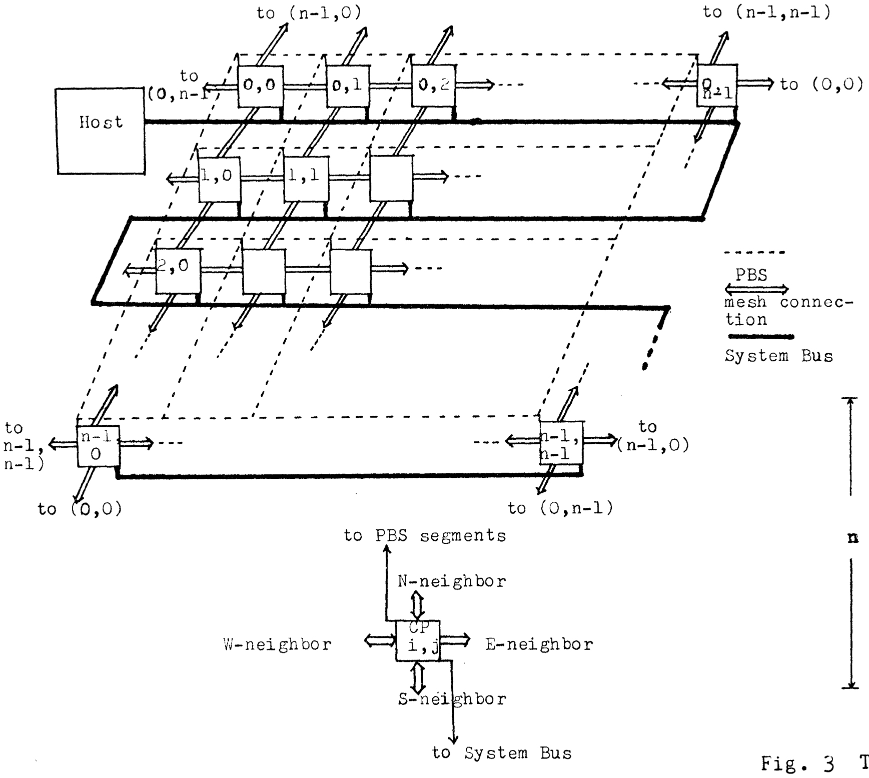
[4] H.J.Siegel, R.J.McMillen, P.T. Mueller, " A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems," <u>Proc. AFIPS 1979 NCC,</u> June 1979. pp. 529-542.

[5] A.Rosenfeld, " Parallel Image Processing Using Cellular Arrays," <u>Computer</u>, Jan. 1983. pp. 14-20.

[6] I.Koren, " A Reconfigurable and Fault Tolerant VLSI Multiprocessor Array," <u>Proc. of the 8th Annual Symposium on Computer Architecture</u>, 1981. pp. 425-442.

[7] W.H.Lee and M.Malek, " Design of a Partitionable and Reconfigurable Multicomputer Array," Technical Report, Dept. of Electrical Engineering, Univ. of Texas at Austin. Dec. 1982.

[8] S.H.Bokhari, " Max, An Algorithm for Finding Maximum in an Array Processor With a Global Bus, " Proc. of the International Conference on Parallel Processing, 1981. pp. 302-303.



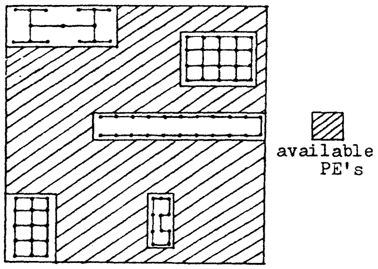


Fig.2 An instance of partial allocation on the mesh

Fig. 1 Interconnection structure for communication processors

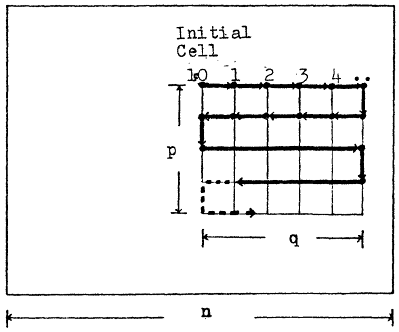


Fig. 3 Transmission pattern for a linear array structure on a pxq partition

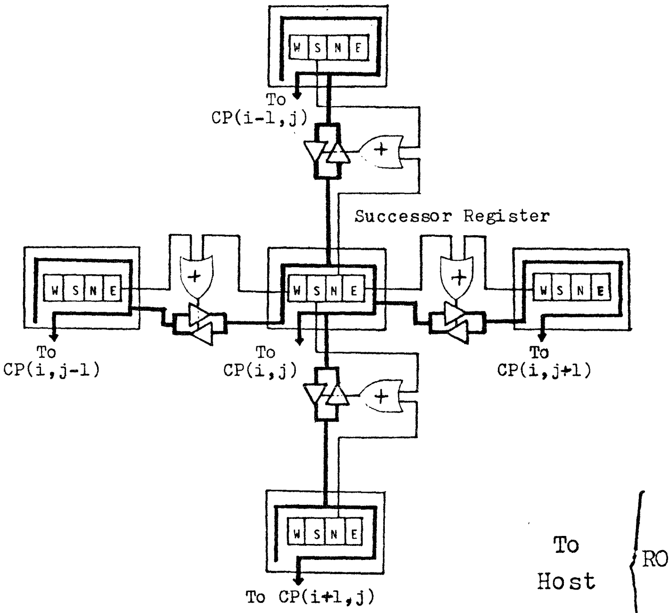


Fig. 4 Organization of PBS segments for one cell and the interconnection to its four neighbors

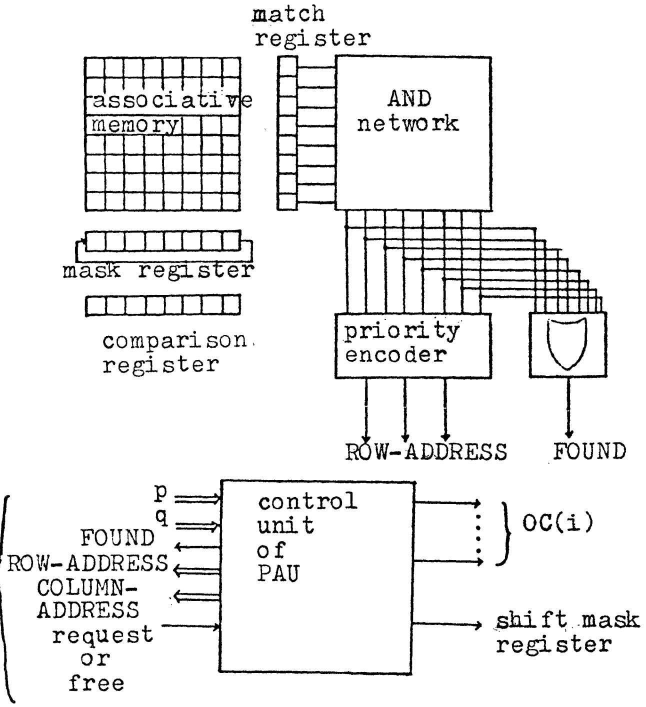


Fig. 5 Block diagram of the Partition Allocation Unit

The Multiprocessor EMPRESS
A Useful Tool for Studying Parallelization Concepts

Hans-Joerg Brundiers, Richard E. Buehrer, member, IEEE,
Hansmartin Friess and Milan Tadian, member, IEEE
Swiss Federal Institute of Technology, ETH
CH-8092 Zurich, Switzerland

Abstract. This study describes preliminary results of general interest based on first utilization of the multiprocessor EMPRESS of the ETH Zurich. To carry out the study, a electrical-engineering reference problem was solved in the parallel mode using the parallelized version of the program PSCSP (which numerically simulates continuous systems using the integration method of Taylor series).

Results are presented for the following asynchronous parallelization concepts:
- single-stage parallelization
- double-stage parallelization
- improved single-stage parallelization using a new parallel algorithm for Taylor series and a partially distributed control of the parallel processes,
- double-stage parallelization, simulation of optimal hardware by scaling the instruction timing.

Two further parallelization methods are proposed to be tested on EMPRESS:
- nearly completely distributed control of parallel processes and
- complete prescheduling and synchronous operation.

Introduction

Multiprocessor Hardware

The multiprocessor EMPRESS is a model computer of the MIMD type assembled at the ETH Zurich. The system comprises 16 LSI-11 processors (called execute processors=EP) and one PDP-11/34 (called supervisor). The newly developed communication hardware consist of two networks (for a detailed description see [1]):
- the intercommunication memory 'intercom' is organized as a quadratic matrix of 17x17 RAM segments. It allows simultaneous and delayless exchange of data.
- in the process distribution system, a hard wired 'job controller' monitors the 16 EP's via a fast parallel bus.

The hardware supports two stages of parallelization. In the first stage the supervisor distributes processes among the EP's. In the second stage a 'master' EP can form a logically contiguous group together with other 'slave' EP's and distribute sub-processes among the latter.

Application Software

As first application, a parallel algorithm using Taylor series for the numerical integration of differential equations [2] was tested. The basic concept of this method is the piecewise approximation of the solutions by expansion into Taylor series. Higher terms of the series are calculated recursively from previously calculated terms. The algorithm provides a decomposition of arithmetic expressions within the differential equations into single preprogrammed recursion formulae. Parallelism is exploited on the level of arithmetic expressions as well as on the level of recursion formulae. For nonlinear recursions the formulae contain typically summations of the following form (where the inherent parallelism depends on the order n):

$$R_n = \sum_{\lambda=k}^{m(n)} a_\lambda * P_{m-\lambda} * Q_\lambda$$

The algorithm is realized (in sequential form) in the program PSCSP [3] for the numerical simulation of continuous systems developed at the ETH. A restricted version of this package has been adapted to the multiprocessor. Its main parts are mentioned briefly (see also [1]):
- The separate precompiler accepts the user-provided definitions of the differential equations and produces a Fortran program containing calls to recursion formula.
- The code generator does a symbolic execution of this program accounting for run time values of parameters. Thus it generates an optimized branchless code sequence for the calls.
- The scheduling routine establishes a tabular process dependency graph corresponding to the code sequence.
- The process controller guides the asynchronous parallel execution of the processes by interpreting the dependency graph. (These parts run in the supervisor)
- The library of recursion formulae preprogrammed to run in the EP's in 2 versions:
  - evaluation of a formula is one process in one EP
  - formulae are sub-parallelized; sub-processes run simultaneously on various processors.

This hardware and software system offers several possibilities for performance measurements and for testing other parallelization methods.

The next paragraph introduces the refence problem used for perfomance tests followed by the results obtained using the basic system. Section IV describes results obtained using a improved software. In section V we present results obtained by simulating optimal hardware. In the final section we mention two other parallelization concepts to be tested on EMPRESS.

II Reference Problem

As reference problem, we chose the following system of differential equations describing the dynamics of an electrical synchronous machine [4].

$$\frac{\partial}{\partial t}\underline{I} = \underline{\underline{L}}^{-1} * \underline{V}(\underline{I},\delta,s),$$

$$\frac{\partial}{\partial t}\delta = f(s), \qquad \frac{\partial}{\partial t}s = g(\underline{I})$$

$\underline{I}$ designates a vector of 5 currents, $\underline{\underline{L}}$ a 5x5 coefficient matrix, $\underline{V}$ a vector of 5 voltage functions, s the load angle and δ the slip.

0190-3918/83/0000/0511$01.00 © 1983 IEEE

This problem has an adequate degree of parallelism (seven integrators), nonlinearity (which means subparallelism in recursion formulae) and coupling (which means complexity in the dependency graph). Fig. 1 shows the dependency graph pertaining to this problem and a certain set of parameters. The numbers correspond to processes (=evaluation of recursion formulae) to be executed in the EP's. The bold typed numbers indicate recursive processes whose internal parallelism and time behaviour depend on n. To get the n'th term in the series axpansion, this graph has to be processed for the order n, where n goes from 0 to 15.

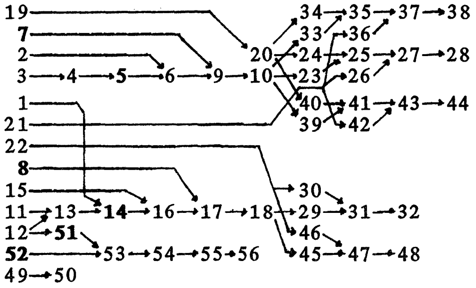


Fig. 1

Dependency Graph

III Basic System

The operation of the multiprocessor in the single-stage mode is illustrated in the process flow diagram in fig.2a, for a pass through the graph with n=9. At the beginning, the supervisor signals to the job controller that the 'root processes', like 19, 7, ... are ready to be started. After finishing a process, the EP sends a process identification to the supervisor, which in turn transfers the results to the common data region of the intercom and checks whether any follower process is ready to be started, until the last 'leaf process' terminates.

In this mode the recursive processes lead to a delay, as expected. Further delays (e.g. 4 which follows 3) are due to overloads in the supervisor process controller. The mean speed-up is 3.4. The graph shows a mean parallelism of 5.6, defined as (total number of processes)/(number of processes on critical path).

Fig.2c shows how the system works in the double-stage mode. The recursive processes are now split into a master process and several slave processes. There is only a small gain in the mean speed-up, due to overhead in the administration of sub-processes and non optimal choice of the number of sub-processes.

IV Improved Software

In ref.[5] another parallel algorithm for the recursive calculation of Taylor series is proposed. This algorithm has been installed in the basic system: Let j_n be a recursive process in the graph, like 14, used to calculate the following sum:

$$R_n = \sum_{\lambda=o}^{n} P_{n-\lambda} * Q_\lambda$$

This sum is no longer split into sub-processes at the time j_n is processed, but two new processes j''_{n-1} and j'_{n-1} are prescheduled to do the partial sums depending on terms of the order n-2 and n-1 respectively. When processing the graph for n-1, j" is started as a root process, and j' as follower of j. j'' and j' are no longer on the critical path, as shown in fig.2b (for j=52 j'' and j' could be taken together because j finished before j'').

A process control routine was installed in the EP's in order to distribute partially the process control load. This routine has direct read-only access to the dependency table stored in the common data region. Having completed a process, the EP undertakes the next awaiting follower process, if any. The supervisor remains engaged in communicating data to other followers and starting them. Fig.2b demonstrates the effect of both improvements.

V Simulation of Optimal Hardware

In ref.[6] hardware improvements for the multiprocessor are proposed including:

- EP's with registers and instructions dedicated to the process distribution hardware
- intercom with access time same as for local memory and automatic wait before reading results from other processes
- faster process distribution bus
- dedicated process distribution hardware in the supervisor.

Such hardware were simulated in the basic system by delaying instructions for process control and execution in the supervisor and EP's in such a way that non optimizable instructions are delayed by a factor of 10 and others by an estimated value <10. Fig.2d indicates a significantly higher speed-up.

VI Conclusions

The speed-up can be summarized as follows:

	n=0	n=15	⟨n⟩
1-stage parallelization, basic	2.9	3.9	3.4
1-stage parallelization, improved	3.7	7.5	5.4
2-stage parallelization, basic	2.7	4.8	3.5
2-stage optimal hardware (x10)	5.4	8.6	7.1

Encouraged by the improvements obtained so far, we plan to test two more parallelization methods:

- Using the 2nd stage, hardware process control can be distributed completely: if an EP finishes a process with more than one follower, the EP can assign remaining followers to slave EP's (except in deadlock situations).
- It is possible to resolve a sequence of recursion formulae (as well as other expressions) into operations of similar length, which can be prescheduled by assigning a logical processor number and time slot to each operation [5]. Such a problem can run synchronously on EMPRESS: One master EP synchronizes the remaining slave EP's in processing the parallel operations.

A future object in view would be a 'parallel operating system' for EMPRESS supporting parallel processing of more general applications.

References

[1] R. E. Buehrer, H. J. Brundiers, H. Benz, B. Bron, H. Friess, W. Haelg, H. J. Halin, A. Isacson, M. Tadian, ''The ETH-Multiprocessor EMPRESS: A Dynamically Configurable MIMD System'', IEEE Transactions on Computers, Vol C-31, pp 1035-1044, Nov. 1982.

[2] H. J. Halin, R. Buehrer, W. Haelg, H. Benz, B. Bron, H. J. Brundiers, A. Isacson and M. Tadian, ''The ETH Multiprocessor Project: Parallel Simulation of Continuous Systems'', Simulation, pp 109-123, Oct. 1980.

[3] H.J . Halin, ''The Applicability of Taylor Series Methods in Simulation'', Proc. 1983 Summer Comp. Simul. Conf., Vancouver, B.C .

[4] P. Wegmann, H. NourEldin, P. Wehrli, ''Digital Simulation of a Synchronous Machine with Motion Equation'', AIE-Report No. 78-06, Institut fuer Automatik und industrielle Elektronik der ETH, Zurich, Switzerland, 1978

[5] M. Tadian, R. E. Buehrer, W. Haelg, ''Parallel Simulation by Means of a Prescheduled MIMD System Featuring Synchronous Pipeline Processors'', in Proc. 1982 Int. Conf. Parallel Processing, M. T. Liu and J. Rothstein, Eds., 1982.

[6] R. Buehrer, Hardware eines dynamisch konfigurierbaren Multiprozessors, Ph.D.dissertation 6930, Swiss Federal Inst. Technol., Zurich, Switzerland, 1981

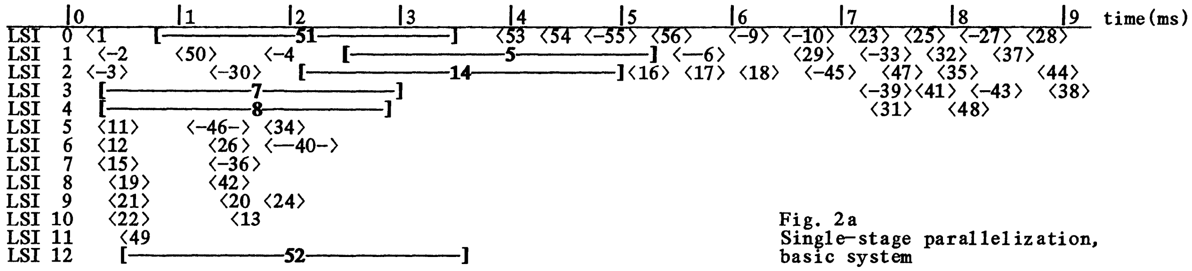


Fig. 2a
Single-stage parallelization, basic system

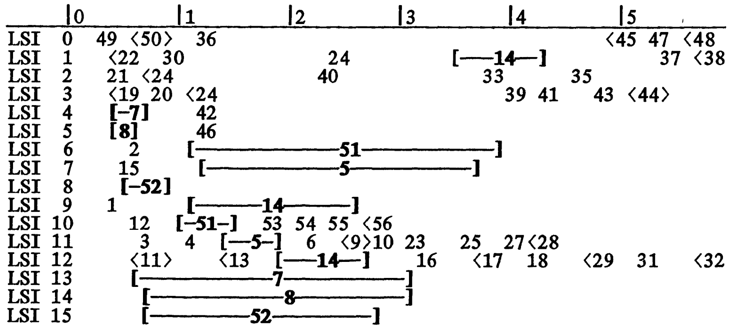


Fig. 2b
Single-stage parallelization, improved system

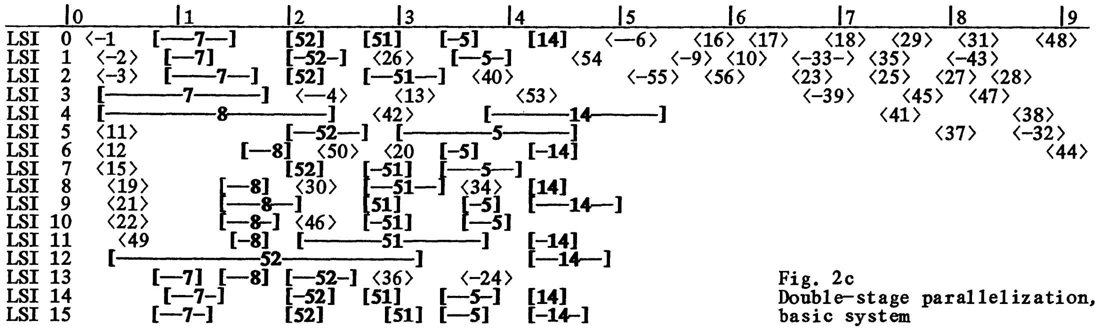


Fig. 2c
Double-stage parallelization, basic system

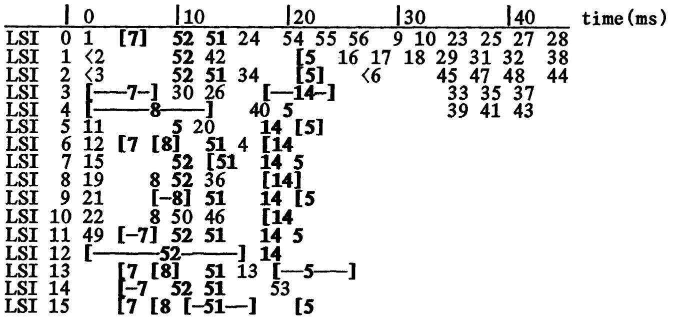


Fig. 2d
Double-stage parallization, simulating optimal hardware

Figure 2 Process flow diagrams

PERFORMANCE OF A MODULAR INTERACTIVE DATA ANALYSIS SYSTEM (MIDAS)*

Creve Maples, Daniel Weaver, Douglas Logan, and William Rathbun
Lawrence Berkeley Laboratory, University of California
Berkeley, California 94720

Abstract -- A processor cluster, part of a multiprocessor system named MIDAS (Modular Interactive Data Analysis System), has been constructed and tested. The architecture permits considerable flexibility in organizing the processing elements for different applications. The current tests involved 8 general CPUs from commercial computers, 2 special purpose pipelined processors and a specially designed communications system. Results on a variety of programs indicated that the cluster performs from 8 to 16 times faster than a standard computer with an identical CPU. The range represents the effect of differing CPU and I/O requirements - ranging from CPU intensive to I/O intensive. A benchmark test indicated that the cluster performed at approximately 85% the speed of the CDC 7600. Plans for further cluster enhancements and multi-cluster operation are discussed.

Background and Objective

MIDAS, a Modular Interactive Data Analysis System being developed at the University of California Lawrence Berkeley Laboratory, is based on the concurrent operation of multiple asynchronous processors. The architecture is designed to provide a highly-interactive, graphics-oriented, multi-user environment that permits programs to dynamically utilize multiple processors in a manner appropriate for the calculation. The system was specifically oriented towards handling scientific calculations, particularly those involving data analysis. This criteria necessitates that the facility be able to accomodate data bases on the order of 200 to 3000 Mbytes per user.

The basic project objectives were to achieve high processing speeds, optimized I/O handling, modular hardware and software structure, flexible architecture, and a fault-tolerant environment. Initially the system should be capable of achieving processing speeds approximately equivalent to that of a CDC 7600 **per problem.** The design is expandable, however, and ultimately should be able to provide between 10 and 100 times this performance. In order to efficiently handle potentially large volumes of information, the architecture permits information transfers to be optimized to minimize the effects of such operations on calculations. A high degree of modularity is required to support system expansion, to allow future utilization of different CPUs and mass storage devices, and to facilitate a fault-tolerant structure.

The effective utilization of multiple processing elements in a wide variety of applications requires a high degree of architectural flexibility. Programs must be permitted to organize and control processors and communication in a manner appropriate to the particular problem. Error recovery and human engineering are important concerns in any computer system and particularly so in a multiprocessor environment. The design criteria requires that the system be able to identify and isolate failures at the module level and, in most cases, circumvent the failure by employing alternate modules (with, however, a possible degradation in throughput). Similarly, diagnostic tools are required that allow users to easily debug code in a real-time environment of multiple asynchronous processors.

Architecture

Essentially the MIDAS design organizes multiple processing elements into clusters. Each cluster combines a number of central processing units from commercial computers with independently developed specialized processors and a specially designed communications system [1,2,3]. The basic architectural structure is a three-level hierarchy of computer processors, organized in a general tree-structure and integrated with independent 'intelligent' mass storage and interactive systems. The three processing levels consist of a Primary Computer, Secondary Computers, and Multiple Processor Arrays.

A. Multi-Processor Arrays

For the purpose of control, MIDAS organizes groups of processors into clusters called Multiple Processor Arrays. A simplified version of such an array, containing ten active processors, is shown in Figure 1. Two of these processors, the Input and Output Formatter, are specialized pipelined devices designed to handle information flow into and out of the cluster. They operate independently at a 200 ns clock cycle on two separate external 20 Mbytes/sec. I/O busses (32-bit data, 8-bit control). These processors may, depending on their programming, select or reject information (filtering); expand or compress data (format); manipulate data (mask, shift, etc.); or route specified information as required by other processors in the cluster.

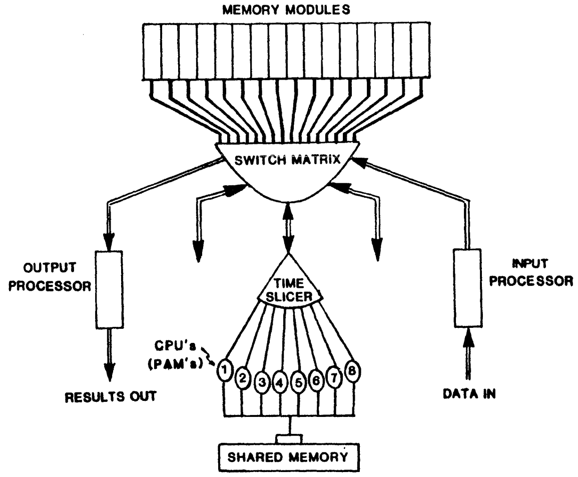


Figure 1. A single Multiple Processor Array

*This work supported by the Director, Office of Energy Research, Division of Nuclear Physics, Office of High Energy and Nuclear Physics and Nuclear Science of the Basic Energy Science Program, U.S. Dept. of Energy, Contract No. DE-A03-76SF0098

0190-3918/83/0000/0514$01.00 © 1983 IEEE

The eight general purpose CPUs, illustrated in Figure 1, are referred to as Programmable Arithmetic Modules (PAMs). They are standard commercial CPUs with dedicated memory, able to handle scientific calculations in general, and floating point operations and Fortran codes in particular. For the initial development the ModComp 7870 CPUs were selected. These processors support 64-bit floating-point hardware, pipelined operation, and up to 4 Mbytes of memory. The CPUs are, for comparison, roughly 15% slower than the DEC VAX 11/780.

B. Communication

As previously discussed, data flow into and out of a cluster is handled by two independent, 20 Mbytes/second busses. Within a cluster, however, interprocessor communication may be handled either by controlling access to independent memory units or via global shared memory. Each of the sixteen independent memory modules, illustrated in Figure 1, has a dedicated memory bus and may contain up to 256 KB of memory. A 5 x 16 cross bar switch allows any memory module to be dynamically attached to any of the five processor busses shown. Since information transfer between a memory module and a CPU (PAM) is considerably faster than the cycle time of the CPU, it was possible to time-multiplex 8 independent memory-CPU connections on the same bus with essentially no degradation of access time. Time-multiplexing these connections was an implementation, not an architectural, decision. Functionally the communication access operates as a 12 x 16 cross bar switch.

Any memory module may thus be attached to any processor at any time. Switching a memory module between available processors requires about 50 ns. This use of bank-switched memory units for multiprocessor communication is similar to the S1 Multiprocessor architecture under development at Lawrence Livermore Laboratory [4]. One major difference in implementation, however, is that MIDAS currently prohibits a memory module from being simultaneously accessed by more than one processor. When a processor is attached to a module, it has exclusive access to that memory until it relinquishes it (or until the supervisory CPU forces a relinquish). Each memory module is also equipped with auto-zeroing hardware and a current destination directory. Thus, when a module is released by a processor the directory pointer is incremented and the memory is attached to the next class of processor specified. If all processors of the specified class are busy, the memory will remain unattached until one becomes available. Once a processor-memory connection occurs, there is no functional distinction between the switched memory and the processor's local dedicated memory. The memory module is accessed by standard load and store instructions, rather than by I/O commands. Thus from a programmer's point of view, the switched memory is simply a particular common block.

TABLE 1

DISTRIBUTED SUBSYSTEM

Functional Memory Classification

		MEMORY	
CLASS	TYPE	ACCESS	AVAILABILITY
1. Dedicated	Program/Data	Fixed	Always
2. Switchable	Program/Data	All	Request
3. Shared	Program/Data	CPU's	Always - Queued
4. Storage	Data	All	Always - Indirect

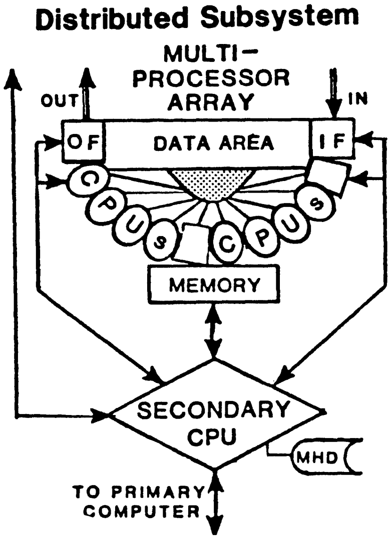


Figure 2. A Distributed Subsystem.

General communication between processors may also occur by means of a global shared-memory unit. Access to this memory is given on the basis of a demand queue. For store operations, a processor may lock out other processors until all memory updates are completed. Since frequent accesses of the global shared memory could slow the parallel operation of the processors, its use should be minimized. Each processor may also communicate directly with the supervisory computer described in the next section.

An independent bulk memory unit, with a 32 Mbyte capacity, is also available for data storage. CPU (PAM) access to this unit is indirect in that information must be transferred via the switchable memory modules. This mode of accessing bulk memory is quite efficient with respect to CPU utilization since a PAM continues operation immediately after releasing a memory module, and does not wait until the data transfer to bulk memory is complete. The bulk memory has dual ports which can be utilized either in a standard DMA transfer or in an address incrementing (+1) mode. Table 1 summarizes the classes of memory available within cluster, and the attributes of each.

C. Control

Each multiprocessor cluster is controlled and monitored by a standard commercial mini-computer, called a Secondary CPU. For simplicity of operation, this computer should be compatible with the CPUs utilized in the Multiple Processor Array. This two-level structure, illustrated in Figure 2, is called a Distributed Subsystem, and forms the basic processing unit of the MIDAS structure. The function of a subsystem is to receive specific problems (from the Primary Computer, the highest control level in the system) and to break the problem into components (e.g., input, output, initialization, computational stages, etc.) which can be carried out in parallel. It then establishes the corresponding processor sequence and communication paths in the Multiple Processor Array, and loads the appropriate code into each processing element. During execution the Secondary monitors and, if necessary, controls the operation of the Multiple Processor Array, and maintains contact with its supervisor, the Primary Computer.

In order to perform these operations the Secondary has its own dedicated disc drive, and runs an essentially standard operating system. It therefore can, for example, compile, assemble, and link programs. The console-control functions of all the standard CPUs (in the MPA) are interfaced into the Secondary, which therefore has complete control and monitor capability. It can run, halt, resume, or single-step each PAM independently or collectively and can monitor or modify selected registers or memory locations. The Secondary is, however, too slow to directly control the high-speed, asynchronous memory switching potentially required in the MPA. The actual switching of memory modules and interprocessor communication is handled by a special hardware device termed the Conductor. The Conductor is functionally controlled by the Secondary and can supply the Secondary with detailed information on system activity when requested.

The distributed subsystem has four communication channels to the external environment: the two I/O busses associated with the Multiple Processor Array; an independent bus attached to the Secondary; and a direct communication channel with the Primary Computer. The first three busses are under the control of the Primary and may be switched to whatever external devices (or processors) that are appropriate.

The function of the Primary Computer is to handle all user interaction and to allocate and manage all the resources of the system. In this regard it oversees the operation of the multiple subsystems and controls all intersubsystem connections. Further, at the request of users, it will supply on a real-time basis, status information on interim results of executing problems.

A fundamental objective for MIDAS was the support of a highly interactive computing environment. The monitoring capabilities of the Secondary Computers were explicitly designed to support such operation. Within its cluster and without perturbing the actual calculations, a Secondary can examine data and obtain detailed information (e.g., status, partial results, etc.) from the bulk memory unit (via a separate memory port) or from any PAM. This information, when requested, would be transmitted to the Primary Computer for presentation to the user.

Performance

Development of the MIDAS project was planned in three phases: prototype construction of a simplified distributed subsystem (to test the basic switching, communication, and control concept); development of a complete model of a fully operational subsystem under control of a Primary (to test performance); and then implementation of a full MIDAS architecture with multiple subsystems. The project was begun in the fall of 1979 and the prototype was operational in January 1982. It consisted of 8 memory modules, input and output processors, a bulk memory unit, 4 CPUs, and the high-speed switching hardware (Conductor).

Table 2

Results of Benchmark Tests on MIDAS Prototype System

	Running Time in Seconds (Ratio)			
	I/O Limited	Average Mix		CPU Limited
Single Computer	272 (6.3)	612 (5.5)	2314 (4.9)	(~4)
MIDAS (3)*	49 (1.1)	159 (1.4)	673 (1.4)	(1.4)
MIDAS (4)*	<43> (1)	112 (1)	472 (1)	(1)

*Indicates number of parallel CPUs

Based on the results of software simulations, the performance of the system was expected to be dependent on the relative I/O versus CPU requirements of each problem. This is to be expected since the system was designed to minimize I/O impact on calculations. Benchmark testing therefpre utilized various existing programs with different calculational conditions. These programs were written in three different languages, with Fortran being most common. In order to operate in a parallel environment some modifications of the original programs were required. The required modifications were not major, and considerable effort was made to minimize such changes in order to permit accurate benchmark testing. The program structures were, therefore, deliberately not optimized to take full advantage of the architecture.

Table 2 compares the time required to run four different problems on a standard ModComp computer and on MIDAS, utilizing 3 and 4 ModComp CPUs, respectively. The speed increases observed in the CPU-limited problem (specifically a Monte Carlo fission simulation program) tracked exactly with the number of CPUs used, as was expected. The I/O-intensive problem, however, ran over six times faster on MIDAS (with 4 CPUs) than on the standard computer. This relative performance increase is attributable both to the existence of independent I/O processors within MIDAS and the more efficient handling of information transfer (the problem required the examination and analysis of approximately 10 Mbytes of information). The value of 43 seconds, obtained with the 4-CPU MIDAS prototype, was subsequently determined to be an artificial limitation imposed by a problem in an external commercial disc controller. It was unable to supply the continuous high-speed data rate MIDAS was capable of accepting. The accuracy of calculational results were independently verified in all cases.

The second phase of development, the construction of a complete subsystem, was completed in February 1983. The layout of this model is shown in Figure 3. Performance of the system was again tested with a similar mix of programs and the results of these tests are shown in Figure 4. The observed speed enhancement (with respect to a single computer) is plotted versus the number of CPUs involved. If processor contention is negligible and each processor is able to contribute its complete capability towards the solution of the problem, the relative performance would simply equal the number of CPUs employed. For the CPU-bound problems investigated, the lower curve indicates that this was indeed the case. The middle line represents a more typical problem (with modest I/O requirements) and exhibits a speed enhancement approximately 50% greater than would be expected from the addition of single processors. The upper curve, which is a highly I/O intensive program, exhibits an even greater increase in relative speed.

The Multiple Processor Array used in this Phase 2 test can, as shown in Figure 3, receive external data from either the Primary Computer or a specialized disc system supervised by a Multiported Programmable Controller (MPC). This high-speed controller is part of the intelligent mass storage system scheduled for operation in the next phase of the project. Since development of this controller is not complete, data required by the test programs were supplied from disc drives on the Primary Computer. The maximum steady state data rate which the commercial disc controller package could sustain was approximately 650 KB/sec. In the I/O intensive test results shown in Figure 4, five CPUs were more than sufficient to handle the analysis of this continuous data stream. The further addition of CPUs therefore resulted in no improvement in the processing speeds. This temporary restriction in handling highly I/O intensive problems will be alleviated when the new MPC system is operational. A single such controller will feature 3 independent channels of look-ahead dual-track buffering and will be expected to handle sustained speeds of approximately 1.2 Mbytes/sec.

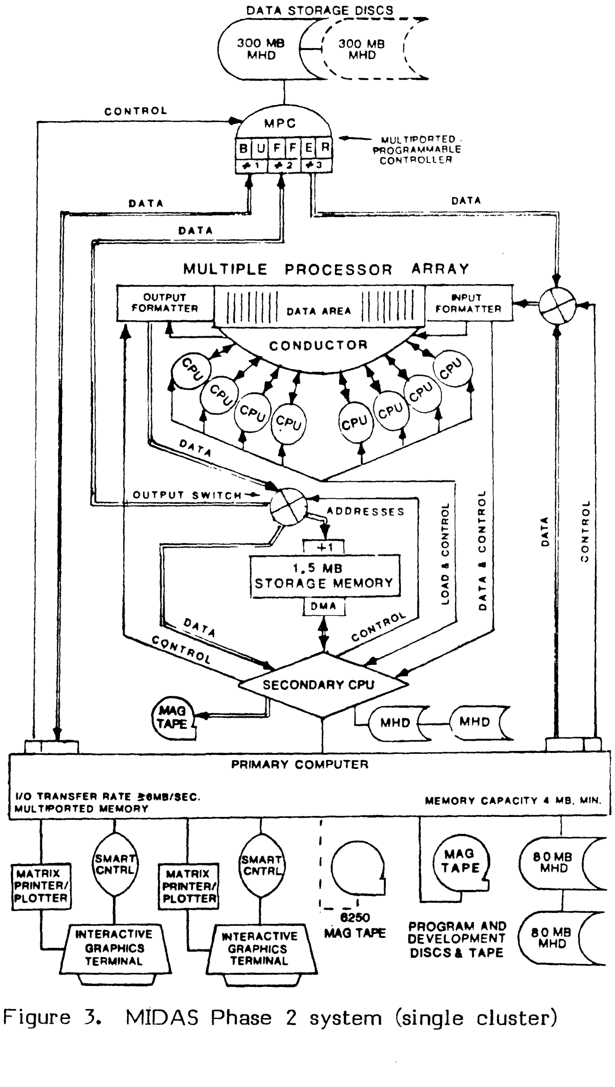


Figure 3. MIDAS Phase 2 system (single cluster)

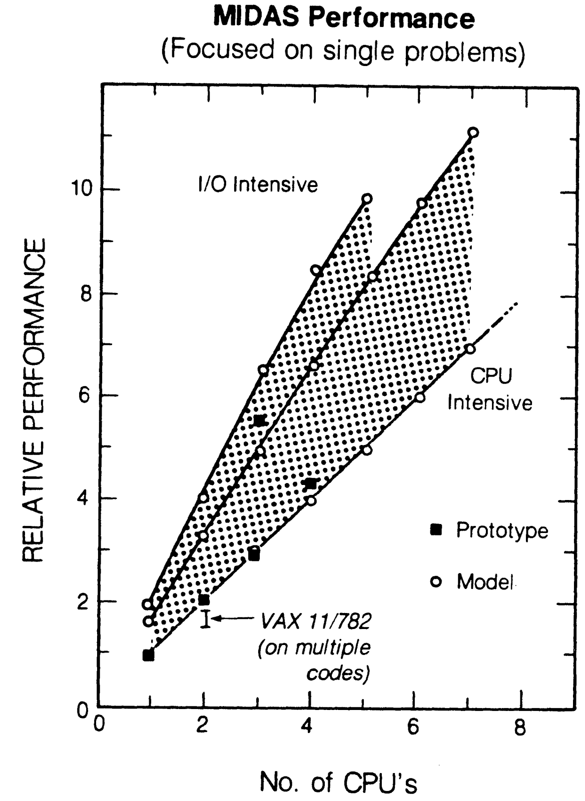


Figure 4. Test results for MIDAS, Phase 2

It should be emphasized that although the Phase 2 model shown in Figure 3 could run eight different programs simultaneously, the test results shown in Figure 4 require that the processors work cooperatively on a single problem. Due to problems of communication, control and memory conflicts, commercial multiprocessor computer systems are not able to deliver one-to-one or even linear speed increases with an increasing number of processors. This is true even when the CPUs are operating on totally independent problems. For purposes of comparison the relative performance of a DEC VAX 11/782 (containing two 11/780 CPUs) is also shown in Figure 4. According to DEC's published measurements, the addition of a second CPU results in a 60-80% increase in performance. Two new systems, the ELXSI and Denelcor's HEP, are specifically designed to reduce multiprocessor conflicts.

Table 3 shows timing comparisons for a program running on both the CDC 7600 and the MIDAS Phase 2 model. The CDC value reflects only the actual processing time and **excludes** both system overhead and I/O time. The MIDAS values are total processing times, including I/O. The MIDAS results, labeled 'standard code' in Table 3, were obtained by utilizing as close a representation of the original CDC program as possible. Although the program heavily utilizes floating point calculations, it also requires the I/O transfer of about 560 Kbytes of information during execution. After the initial comparison tests, the standard code was modified slightly to explicitly take advantage of some of the parallel aspects of the architecture. These minor changes resulted in a 20% increase in MIDAS processing speed, as shown in the last column of Table 3. Further investigation suggests that additional MIDAS-specific modifications to the program may effectively double the performance of the program. Additional information on programming MIDAS in general, and this problem in particular, is contained in Reference [5].

Future Directions

Figure 5 illustrates a potential layout for the next, or third phase of the MIDAS project. In this example, the Primary Computer is controlling five processing clusters. Such a structure would contain between 55 and 145 processors, depending on the implementation plan adopted. The control and communication between processor clusters adds a new dimension to the operation of the facility. The Primary Computer may utilize multiple clusters on single problems in an analogous fashion to the way the Secondary controls multiple processors. The Phase 3 effort will, in addition, require the development of a multi-bussed, parallel-processor mass storage environment and a high-speed, interactive system.

Subsequent development will also continue to increase the processing capability within a cluster. As illustrated in Figure 1, the existing Multiprocessor Array has two unused busses on the switch (Conductor). The number of CPUs within a cluster could, therefore, be tripled by duplicating the current 8-CPU, time-multiplexed handler on each of these additional memory busses (the number of memory modules would also have to be increased). Alternatively the addition of multiple vector (array) processors to one bus is being investigated. Such units would operate in parallel with all other elements of the system. They would be independently programmable and, like other processors, could receive and transmit information by means of switchable memory blocks. Communication, frequently a bottleneck in standard CPU-array processor configurations, would effectively occur at memory speeds with both processors free to continue concurrent operation. The Array Processors could thus be used to augment the standard processors for those portions of a problem amenable to the vector approach.

Table 3

MIDAS Performance Relative to CDC 7600

(Execution Time in Seconds)

		MIDAS	
# CPUs	CDC 7600[1]	"Standard" Code[2]	Enhanced Code[3]
1	38.5	447	355.
2		223	176.
3		147	117.
4		111	87.3
5		89	70.1
6		74	58.8
7		63	50.2
8		55	44.3

1) CPU seconds only
2) Essentially identical to the original CDC code
3) Code modified to utilize some specific MIDAS capabilities

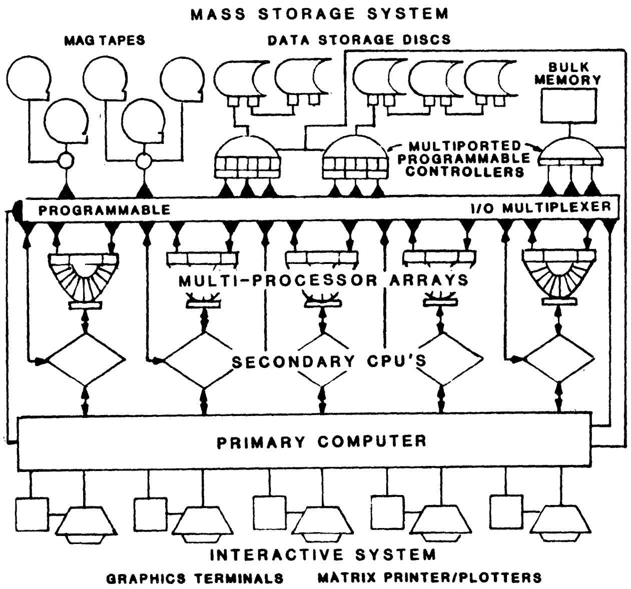


Figure 5. Potential layout of a Phase 3 MIDAS system, containing 5 multi-processor clusters.

Another area of active study will be the dynamic utilization of special purpose processors. Such processors could perform in hardware specific calculations, or algorithms, which are not amenable to parallel approaches (either vector or multiprocessor). These hardware modules (ultimately VLSI processors) could be incorporated on a vacant memory bus (Figure 1) and accessed from code in a manner analogous to a hardware subroutine. To further increase the processing power of a cluster, the Phase 3 development will also replace existing CPUs with new commercial processors at least 3 to 4 times as fast. The performance of a number of new (or planned) CPUs, including micro-processors, are therefore being investigated.

The main software thrust has been to permit existing programs to operate in a parallel environment with minimum code changes. MIDAS is, of course, different from standard computers and is capable of handling problems in ways not possible on other machines. As was indicted previously, even small structural changes in code can significantly increase the execution speed on MIDAS [5]. For this reason considerable effort is underway to investigate different computational approaches to important problems and to develop tools and language constructs which permit users to exploit the full, and currently untapped, potential of this system.

Summary

Computers continue to assume an increasingly important role in scientific research. Traditional serial computers are, however, approaching the theoretical limits imposed by heat dissipation and the speed of signal propagation. In the decade of the 60's computing speeds increased by a factor of about 100 while during the 70's, however, only a factor of 10 increase in speed was realized. In order to achieve significant improvements in future computer processing, it is necessary to explore new parallel architectures. Until recently effort in this direction has focussed primarily on parallel instruction execution (SIMD architectures), as in vector machines (e.g., Cyber 205 or CRAY-1). While such architectures can provide substantial processing capability for 'vectorizable' programs, many problems, particularly those dominated by conditional testing and branching, do not appear amenable to this approach.

The MIDAS project represents research in advanced computer architectures and their specific application in a scientific environment. The performance of the Phase 2 multiprocessor cluster was tested on a variety of programs having total sustained I/O requirements spanning over 2 orders of magnitude (5 to 1000 KBytes/sec.) for periods of 15-20 minutes. In the worst case, the cluster performed n-times faster for n-CPUs. The maximum performance realized was approximately twice n. The tests were performed using existing programs and real data. A CPU-intensive problem run on both the CDC 7600 and MIDAS indicated that the current Phase 2 cluster was the equivalent of at least 85% of the 7600. The Phase 2 cluster required approximately $360K to develop and about 18 man-years of effort.

The utilization of multiple processors operating concurrently, such as employed in the MIDAS architecture, provides an alternative to achieve necessary high-speed computation in the future. The MIDAS Project is designed to fill a specific need within the scientific community. Experience at other research institutions and examination of current computer performance suggest that this need cannot reasonably be filled by currently available computers in any cost-effective manner.

References

[1] Creve Maples, William Rathbun, Daniel Weaver, and John Meng, "The Design of MIDAS - A Modular Interactive Data Analysis System", IEEE Transactions on Nuclear Science, (October 1981), pp. 3746-3753.

[2] Creve Maples, Daniel Weaver, William Rathbun, and John Meng, "The Utilization of Parallel Processors in a Data Analysis Environment", IEEE Transactions on Nuclear Science, (October 1981), pp. 3880-3888.

[3] W. Rathbun, C. Maples, J. Meng, and D. Weaver, "A Fast Time-Sliced Multiple Data Bus Structure For Overlapping I/O and CPU Operations," IEEE Transactions on Nuclear Science, (October 1981), pp. 3875-3879.

[4] Ronald D. Levine, "Supercomputers," Scientific American, (January 1982), pp. 118-135.

[5] Douglas Logan, Daniel Weaver, Creve Maples, and William Rathbun, "Application Programming for MIDAS, a Multiprocessor System," IEEE Transactions on Nuclear Science, (October 1983), in press

THE HOMOGENEOUS MULTIPROCESSOR ARCHITECTURE - STRUCTURE AND PERFORMANCE ANALYSIS

Nikitas Dimopoulos
Electrical Engineering Department
Concordia University
Montreal, Quebec, Canada H3G 1M8

Abstract — In this work, we present the structure of the Homogeneous Multiprocessor. The Homogeneous Multiprocessor is a tightly coupled MIMD architecture composed of two parts. The multiprocessor proper where each processing element communicates directly with either one of its two immediate neighbors through a distributively controlled switching network, and the H-Network which is a high speed (7 Mbytes/sec) local area network.

A performance analysis of the two components of the architecture as well as possible applications of the Multipiocessor are also presented.

I. Introduction

In recent years multiprocessors have become important in solving problems where a large amount of computation is needed. Several multiprocessors have been proposed and built.

A major architectural issue involved in the design of such machines is the availability of information paths that would enable the exchange of information between processors. Most of the existing MIMD designs have opted for a complete graph solution incorporating crossbar switching or microprogrammed controllers rendering the system either expensive or slow.

The Homogeneous Multiprocessor discussed in this work resulted from the realization that in many applications (relaxation processes [6], neural network simulation [4]) the complete graph solution is not a necessary attribute of the design, since such problems can be formulated in such a way so that each computational subtask would require information from only its neighboring subtasks to complete the computation. Moreover, such problems would benefit from architectures that limit the scope of interprocessor communication, but make the limited communication paths fast.

In this work we will describe one such multiprocessor architecture, the Homogeneous Multiprocessor.

II. Structure

The Homogeneous Multiprocessor system, shown in Fig. 1, is a tightly coupled MIMD structure, and it is composed of two parts. The Homogeneous Multiprocessor proper and the H-Network.

The Homogeneous Multiprocessor proper consists of k processors, k memory modules and k local buses used by each processor to access its local memory module. Interbus switches, normally open, isolate the processor-memory complexes from each other allowing thus maximum throughput. Also the interbus switches provide the means of communication between neighboring processors. Upon a request from either one of its two neighboring processors a switch may close connecting thus two neighboring local buses to form an extended bus. An extended bus is then utilized by the requesting processor to access the memory module of its neighbor and this in turn provides for interprocessor communications. As it may be seen in Fig. 1, there is no central controller to control the switches. Rather, associated with each one of the switches, there exists a switch controller which decides on the next state of the switch based on the status of the two neighboring switches and the presence of a request for the switch to close.

Thus a switch s_i can only exist in one of three states and it receives requests for service from its two adjacent processors p_i and p_{i+1}. The next state transition is based on the current state of the switches s_{i-1}, s_i and s_{i+1} and it is computed according to Algorithm 1.2.

The three states a switch can exist at, are as follows:

- OPEN: Denoted by O. This state signifies that no requests exist or if a request exists it will not be honoured in the immediate future.
- GRAY: Denoted by G. This signifies that a request has been acknowledged and service (i.e. switch closure) will be granted in the immediate future.
- CLOSED: Denoted by C. This state signifies that conditions are compatible for establishing an "extended bus".

The operational Algorithm 1.2 that decides the next transition is as follows:

Algorithm 1.2

```
     For the switch $s_i$
1.   If no requests exist it becomes Open;
     if a request exists then:
2.   If Open it becomes Gray provided that the
     switch to its left ($s_{i-1}$) is Open. Otherwise
     it remains Gray.
3.   If Gray it becomes Closed provided that the
     switch to its right ($s_{i+1}$) is Open. Other-
     wise it remains Gray.
4.   If Closed it remains Closed.
5.   Switches $s_0$ and $s_{k+1}$ are always open.
```

A Pascal description of the above Algorithm 1.2 is given in Fig. 2. It has been proven [3] that this algorithm provides for safeness (i.e. no

This research is supported by the Natural Sciences and Engineering Research Council, Canada under grant #A1337 and by Fonds F.C.A.C. pour l'aide et la soutien â la recherche under grant #EQ2007.

0190-3918/83/0000/0520$01.00 © 1983 IEEE

two neighboring switches will close at the same time) and liveness (i.e. if a switch is requested to close, it will do so in the near future).

The second component of the architecture is the H-Network [2]. This is a high speed base-band local area network with a structure which resembles that of the Ethernet [5], yet it utilizes separate pathways for data transmission, network acquisition, and collision detection. These separate pathways allow the processes of data transmission and collision detection to proceed in parallel while the network acquisition interval is reduced to less than 100 ns. This short network acquisition interval increases the probability that only one station will become master of the network at any given time which results to an increased network utilization.

As shown in Figs. 1 and 3, the H-network consists of four pathways plus Network Stations which interface the network to each processor of the Homogeneous Multiprocessor proper. The pathways are:

(a) The H-bus. This is the data highway which consists, in the present implementation, of 16 data lines. Each Network Station interfaces with the H-bus via an input and an output buffer which are implemented by using fast FIFO's (128 words x 16 bits/word). These buffers are used to capture an incoming packet and to hold an out-going packet until the station controller achieves mastership of the H-Network.

(b) The Access Line. This is a control line and it is used to ensure the mastership of the network by a single station with high probability. A fast test and set module is used by the station controllers so that they can test and set the condition of the Access line in a very short interval of time (~ 100 ns).

(c) The ID line is used to detect possible collisions. A station controller while master of the network, transmits, and at the same time listens, its unique identification code over the ID line. If more than one station has gained mastership of the network their codes collide on the ID line. Such a collision can be detected by the transmitting stations and the transmission aborted.

(d) Timing and control. This group of lines facilitates the actual transmission of data through the H-bus. The data transmission protocol adopted is an asynchronous one with handshaking. Such a protocol is immune to varying signal delays imposed by the physical placement of the modules or by the technology used for their implementation.

The basic unit of information transmitted over the H-bus is the packet. As it can be seen in Fig. 4 each packet consists of three parts, namely the header, the body and a checksum. The total length of the packet may not exceed, in this implementation, 128 words. This length coincides with the depth of the FIFO's used in the network stations. Three words have been reserved for the header.

The first word carries the origin and destination of the packet. The first word of the header is captured by the Temporary Register of the network stations. This event happens before the commencement of the transmission of the remaining of the packet. Thus, the origin and the destination of any packet are made available to all the network stations prior to the packet transmission. Based on this information, the stations decide whether to capture the incoming packet, and communicate their decision (through the timing and control lines) to the transmitting station which releases the packet to the H-bus at high data rates (~ 7 Mbytes/sec). Packets with a destination of 00 are accepted by all the stations. Such packets are used for network control and management.

The second word of the header contains the length of the packet plus control information such as type of packet and retransmission number. Finally, the third word of the header has been reserved for use by the higher layers of the network protocol.

III. Performance

The two components of the architecture have been analyzed separately.

The Homogeneous Multiprocessor proper has been simulated and the average memory access time A for neighboring memory modules is given as a function of the average interarrival intervals λ of requests. This function, for a multiprocessor composed of 20 processor-memory complexes with a memory access time of 600 nsec, is given in Fig. 5.

For the H-Network we have assumed a 1-persistent CSMA-CD, where τ is the packet transmission interval, δ is the collision detection interval (δ=10 μsec) and a is the window of vulnerability, during which one or more H-stations may acquire the network (a≃100 nsec). We have also assumed N network stations, each one of which attempts to gain mastership of the network with a period T (T≃3 μsec).

The network acquisition, packet transmission and collision detection cycles are outlined in Fig. 6.

The N stations, which are continuously vying for the acquisition of the network, are modeled as a Poisson process with parameter $G_N = \frac{N}{T}$. Therefore, the utilization factor is calculated as $s = \frac{\bar{U}}{\bar{B}+\bar{I}}$, where $\bar{U}$ is the average period of successful transmission, $\bar{B}$ is the average busy period (collision or successful transmission) and $\bar{I}$ is the average idle period.

The quantities $\bar{U}$, $\bar{B}$ and $\bar{I}$ are calculated as follows:

$$\bar{U} = \tau \cdot P(\text{successfull transmission}) =$$
$$= \tau \cdot P(\text{no requests in interval a})$$

$$\bar{U} = \tau e^{-\frac{N}{T}a}$$

$$\bar{B} = a + \tau P(\text{successful}) + \delta(1 - P(\text{successful}))$$

$$\bar{B} = a + \tau e^{-\frac{N}{T}a} + \delta(1 - e^{-\frac{N}{T}a})$$

and finally, $\bar{I} = \frac{1}{G_N} = \frac{T}{N}$.

Therefore, $$s = \frac{\tau e^{-\frac{N}{T}a}}{a + \tau e^{-\frac{N}{T}a} + \delta(1 - e^{-\frac{N}{T}a}) + \frac{T}{N}}$$

A plot of S as a function of N for various τ(in μsec) is given in Fig. 7.

IV. Discussion

The two components of the Architecture, (i.e. the Homogeneous Multiprocessor proper and the H-Network) play a different role in the function of the Multiprocessor.

The Homogeneous Multiprocessor proper is particularly suited for problems that can be modeled by a set of parallel processes operating independently of each other and occasionally exchanging information or control with their immediate neighbors.

In general, the solution for such problems is calculated as:

$$x_{s_i}^{t+1} = \theta_{s_i}^t [x_{s_{i-1}}^t, x_{s_i}^t, x_{s_{i+1}}^t]$$

where

$$x = [x_{s_1}, x_{s_2} \dots s_{s_k}]^T$$

is the solution and,

$$\theta^t = [\theta_{s_2}^t \dots \theta_{s_k}^t]$$

is the operator describing the problem. Examples of such applications may be drawn from image processing through relaxation [6], digital filtering [1] and neural networks [4].

The H-Network will serve primarily as the main communications link of the Homogeneous Multiprocessor proper and its environment. It will carry user and file traffic in cooperation with Front End and Back End processors.

Also, the H-Network makes it possible for distant processors in the Multiprocessor proper to communicate directly without resorting to time consuming hops of intermediate processors.

In order to enhance reliability and to provide for overall control of the structure, one of the processors of the multiprocessor proper together with its two immediate neighbors will serve as master of the structure. The master will share its data base with its neighbors. The master and its neighbors will repeatedly check for integrity of the structure. In the event that the master malfunctions, then one of the neighbors will take over. This becomes possible since the data base of the master is shared by its two neighbors.

The designation of the master comes about through the H-Network. Upon power up or reset the processor whose network station first achieves mastership of the H-Network becomes master of the structure.

REFERENCES

[1] M.O. Ahmad et. al., "Ladder Realization of a Class of Two-dimensional Analog Voltage Transfer Functions with Application to Wave Digital Filters", Archiv fur Electronik und Ubertrangungstechnik, Vol. 33, #2, (1979), pp. 81-85.

[2] N. Dimopoulos and D. Kehayes, "The H-Network-A High Speed Distributed Packet Switching Local Computer Network" to be presented at "MELECON '83 - Mediterranean Electrotechnical Conference" Athens, Greece, May 24-26, 1983.

[3] N. Dimopoulos, "On the Structure of the Homogeneous Multiprocessor", under review for the IEEE Trans. on Computers.

[4] N. Dimopoulos, "Organization and Stability of a Neural Network Class and the Structure of a Multiprocessor System", Ph.D. thesis, University of Maryland (1980).

[5] R.M. Metcalfe et. al. "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm. ACM, July 1976, pp. 395-404.

[6] S.W. Zucker et. al., "An Application of Relaxation Labelling to Line and Curve Enhancement", IEEE Trans. on Computers, Vol. C-26, (1977), pp. 394-403 and 922-929.

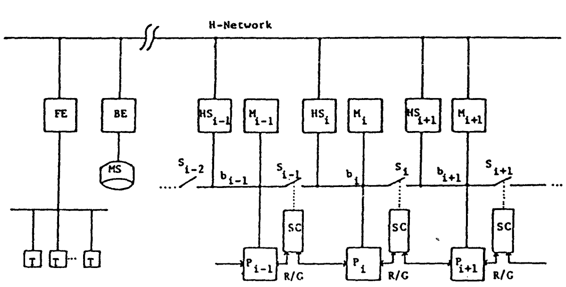


Figure 1. The Homogeneous Multiprocessor

M: Memory P: Processor S: Bus Switch
FE: Front End BE: Back End SC: Switch Controller
T: Terminal b: Local bus HS: Network station
MS: Mass storage R/G: Bus request/grant

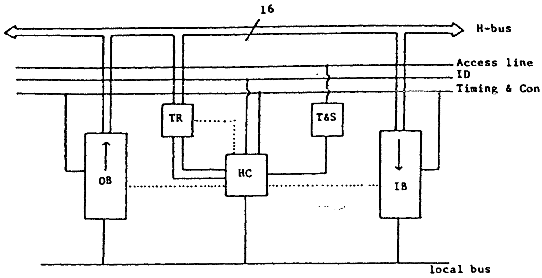


Figure 3: A Network Station

OB: Output buffer IB: Input Buffer
HC: Station controller ID: Collision detection
TR: Temporary Register T&S: Test and set

```
{Synchronization Algorithm 1.2 for the network of switches.
  k is the number of switches in the network.}
program Algorithm;

type states=(open,gray,closed);
state: array[0..k+1] of states;  {array of the switch states}
nxtstate: array[1..k] of states;    {next_state  array}
request: array[1..k] of Boolean;

procedure newstate (var i: integer);
if (request[i]=false) then nxtstate[i]:=open else
   begin
       case state[i] of
       open: if(state[i-1]=open) then nxtstate[i]:=gray else nxtstate[i]:=open;
       gray: if(state[i+1]=open) then nxtstate[i]:=closed else nxtstate[i]:=gray;
       closed: nxtstate[i]:=closed;
       end {case}
   end
end; {newstate}

begin
state[0]:=open; state[k+1]:=open;
    while true do
    begin
       parbegin
          newstate(1);
          newstate(2);
             .
             .
             .
          newstate(k);
       parend;

       parbegin
          state[1]:=nxtstate[1];
          state[2]:=nxtstate[2];
             .
             .
             .
          state[k]:=nxtstate[k];
       parend;
    end {while}
end. {Algorithm}
```

Figure 2

The Synchronization Algorithm 1.2
for the Network of the s Switches

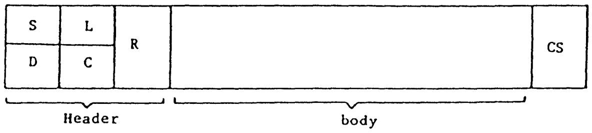


Figure 4. Packet

S: Source	D: Destination	L: Length
C: Control	R: Reserved	CS: Checksum

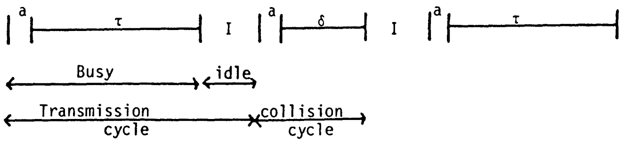


Figure 6. H-Network acquisition, transmission and collision detection cycles.

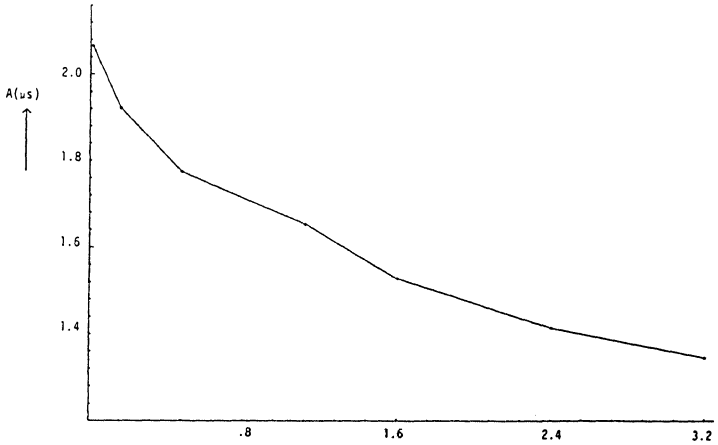


Figure 5 ——→ λ(μs)

Average memory access time for neighboring memory modules A (in μs) vs. the average interarrival intervals of the requests λ (in μs).

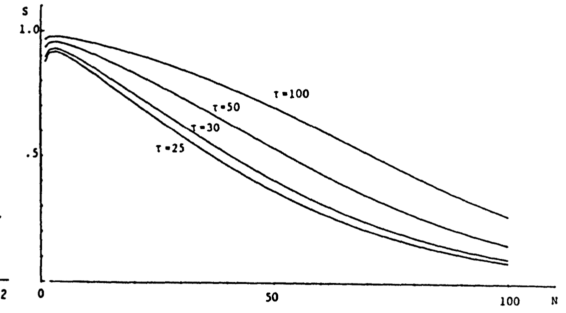


Figure 7. Network Utilization Factor

Cedar—A Large Scale Multiprocessor

Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh

Laboratory for Advanced Supercomputers
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

This paper presents an overview of Cedar, a large scale multiprocessor being designed at the University of Illinois. This machine is designed to accommodate several thousand high performance processors which are capable of working together on a single job, or they can be partitioned into groups of processors where each group of one or more processors can work on separate jobs. Various aspects of the machine are described including the control methodology, communication network, optimizing compiler, and plans for construction.

1. Motivation

The primary goal of the Cedar project is to demonstrate that supercomputers of the future can exhibit general-purpose behavior and be easy to use. The Cedar project is based on five key developments which have reached fruition in the past year and taken together offer a comprehensive solution to these problems.

(1) **The development of VLSI components makes large memories and small, fast processors available at low cost.** Thus, highly parallel (e.g., 1024 processors) systems are not ruled out by cost or physical volume considerations as they have been in the past. Particularly important are the 32-bit 2.5 megaflop chips or chip-sets developed in the past year [WaMc82]. Thus, basic hardware building blocks will be available off-the-shelf in the next few years.

(2) Given the hardware components for a highly parallel system, accessing a parallel shared memory and moving data between memories and processors has been a traditional architectural stumbling block. Many systems have been built that have severe constraints in the memory (e.g., access to columns only) or interconnection network (e.g., nearest neighbors only). **Based on many years of work, it is possible to have a shared memory and switch design which will provide high bandwidth over a wide range of computations and applications areas.**

(3) Compilation for parallel, pipeline, and multiprocessor systems has been another serious traditional problem. **The Parafrase project has demonstrated that by restructuring ordinary programs these supercomputer architectures can be exploited effectively.** It has also been shown that Parafrase can restructure programs to effectively exploit various levels of a memory hierarchy. An important consequence is that a compiler can be used to manage caches in a multiprocessor and thus avoid cache coherency problems.

(4) The control of a highly parallel system is another problem of long-standing concern and controversy. It is probably the most controversial of the five topics listed here, mainly because it seems to be the least amenable to rigorous analysis. From an abstract viewpoint, the traditional dataflow approach seems best because control is distributed out to the level of operations on scalar operands. In practice, it seems that dealing with this low level of granularity has many weaknesses. **By using a hierarchy of control, we have found that dataflow principles can be used at a high level (macro-dataflow), thus avoiding some of the problems with traditional dataflow methods.** We have also demonstrated that a compiler can restructure programs written in ordinary programming languages to run well on such a system.

(5) Algorithms for systems with concurrency have been studied for a number of years. Many successes have been achieved in exploiting the array parallelism of various pipeline and parallel machines. But there have been a number of difficulties as well. It has long been realized that some of these difficulties should be surmountable using a multiprocessor because the parallelism in such a machine is not as rigid as in array-type machines. **Recent work in**

This work was supported in part by the National Science Foundation under Grants No. US NSF MCS81-00512 and US NSF MCS80-01561, the US Department of Energy under Contract No. US DOE DE AC02-81ER10822, and by the Department of Computer Science at the University of Illinois

0190-3918/83/0000/0524$01.00 © 1983 IEEE

numerical algorithms seems to indicate great promise in exploiting multiprocessors without the penalty of high synchronization overheads which has proved fatal in some earlier studies. Furthermore, nonnumerical algorithms have been developed at a rapidly increasing rate in the past few years. These can generally use a multiprocessor more efficiently than a vector machine, particularly in cases where the data is less well structured. Our group has been active in developing both numerical and nonnumerical algorithms.

To reach the goal stated in the opening paragraph, we believe that a two-phase approach is necessary. The first phase is to demonstrate a working prototype system, complete with software and algorithms. The second phase would include the participation of an industrial partner (one or more) to produce a large scale version of the prototype system called the production system. Thus, the prototype design must include details of scaling the prototype up to a larger, faster production system.

Our goal for the *prototype* is to achieve Cray-1 speeds for programs written in high level languages and automatically restructured via a preprocessing compiler. We would expect to achieve ten to twenty megaflops for a much wider class of computations than can be handled by the Cray-1 or Cyber 205. This assumes a four cluster, 32-processor prototype where each processor delivers 2.5 megaflops.

The *production* system will use processors that deliver over 10 megaflops, so a 1024 processor system should realistically deliver (through a compiler) several gigaflops by the late 1980s. Actual speeds might be higher if (as we expect) our ideas scale up to more processors, if higher speed VLSI technology is available, and if better algorithms and compilers emerge to exploit the system.

An integral part of the design for the prototype and final system is to allow multiprogramming. Thus, the machine may be subdivided and used to run a number of jobs, with clusters of eight processors, or even a single processor being used for the smallest jobs.

2. The Cedar Architecture

In order to integrate the discussion, we show in Fig. 1 an overall system diagram. More details of our preliminary view of the system are discussed in [GLPV83].

2.1. Processor Cluster. A Processor Cluster (PC) is the smallest execution unit in the Cedar machine. A chunk of program called a Compound Function can be assigned to one or more PCs.

A PC consists of n processors, n local memories, and a high speed switching network that allows each processor access to any of the local memories. Each processor can also access its own local memory directly without going through the switch. In this way, extra delay is incurred only when the data is not in its own local memory. Furthermore, each processor can directly access global memory for data that is not in local memory. Our compiler is targeted at exploiting this hierarchy of memory access speeds.

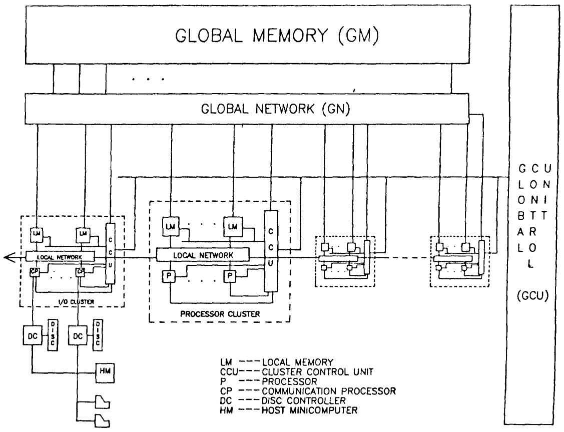


Figure 1. Overall system diagram.

Each processor consists of a floating-point arithmetic unit, address generation unit, and Processor Control Unit (PCU), with program memory. There are no programmer accessible data registers in the processor. However, the local memory is dynamically partitionable into pseudo-vector registers of different sizes, and so it serves really as a large set of general-purpose registers. There are two reasons for this type of cluster organization. Firstly, it simplifies the compiler design. Secondly, there is no need for general-purpose registers since off-the-shelf floating-point arithmetic is an order of magnitude slower than medium size static memories (500 ns vs. 50 ns). Each local memory has its own global memory access unit that allows movement of data between global and local memories to proceed concurrently with the computation.

The entire PC is controlled by the Cluster Control Unit (CCU), which mostly serves as a synchronization unit that starts all processors when the data is moved from global memory to local memory and signals the Global Control Unit (GCU) when a compound function execution is finished.

In this paper we discuss two different machine sizes: the prototype and production machine. The prototype machine is a 4 cluster (8 processors per cluster) machine built for the purpose of debugging architectural and software concepts and justifying performance estimates for a broadly chosen set of applications. An architecturally and technologically upward scalable production machine is a 64-128 cluster (8-16 processors per cluster) high performance supercomputer.

To obtain short design time, we will use for the prototype machine off-the-shelf components, standard memory chips, and gate arrays, while the production machine will use custom VLSI parts and high density packaging technology.

Communication between disks, etc., and global memory will be through a special I/O cluster. An I/O cluster is equivalent to a PC except for the processors themselves. Instead of the usual processors, the I/O cluster will have communication processors. These in turn will connect to Extended Storage (solid state disks) which serves as a buffer between Cedar and the support machines (e.g., VAX) which will provide access to disks, terminals, etc.

2.2. Global Network

Large scale multiprocessors require access to a shared memory system and convenient interprocessor communication. Early parallel computers tended to be mesh-connected—that is, access to neighboring processors and memories was fast, but more global communication/access took proportionally more time. Vast amounts of manpower were expended devising special algorithms which could execute in this type of environment. (Pipeline processors are not immune to this problem—the performance degradation due to non-unit vector strides or irregular addressing patterns are generally recognized problems.)

Our network is based on an extension of the omega network [Lawr75] and is similar in concept to the omega network designed for the Burroughs FMP machine [Burr79], [BaLu81]. That network was nominally 1024x1024, and was a circuit-switching network where the data path at each node was 11 bits wide. They estimated that the minimum time required to set up a connection would be 120 ns.

Our initial design differs in several respects from the FMP design. It is based on the use of 8x8 switches located on 160 pin boards, rather than 2x2 switches. Taking into account expected delays due to conflicts, time multiplexing of 120-bit packets, memory access, and return transmission, we estimate an expected delay time of less than 2 μs/1024 words between processor and memory. (Using the same techniques we can design networks to provide average global memory access in as little as 500 ns, but these designs would require as many as 8 boards per processor.)

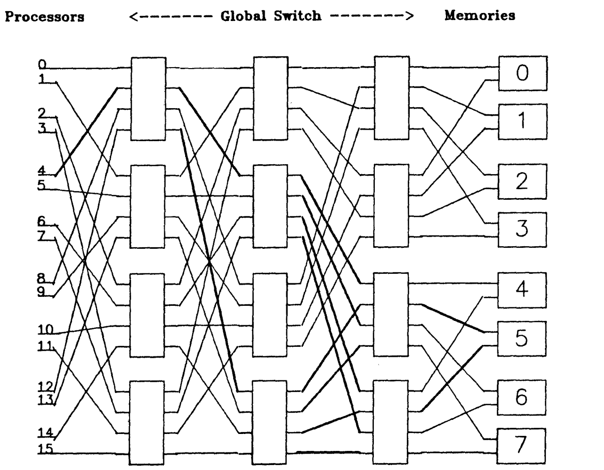


Figure 2. Example of a global network connecting 16 processors with eight memories. Notice the redundant paths from processor 4 to memory 5.

An example of one of these networks connecting 16 processors to 8 memories is shown in Fig. 2. This example uses 4x4 switches, but illustrates the principles we will use in constructing the 1024 port global network. We have discovered ways to add redundancy in larger networks that allow us to use this redundancy both for conflict avoidance and fault tolerance [Padm83]. Notice that unlike the omega network, this network allows more than one path between any processor port and any output port. This path redundancy provides both fault tolerance and conflict avoidance. Thus, from every switch (except the last) there are at least two valid paths. If either of these is either blocked by another message or by a failure, a connection can still be made via an alternate path. (A total blockage can exist if *all* alternate paths are blocked by conflicts with other messages and/or faults. However, analytic and simulation results indicate that the probability of conflicts is significantly lower with the redundant paths than without them, and that the probability of there being enough faults to block a message is so small that the mean time between fault-blocked-messages is on the order of one year even for the production machine.) The control logic which allows this conflict/fault avoidance is distributed throughout the network and is not very different from the classical omega control algorithm.

2.3. Memory System

The overall memory system has a great deal of structure to it, but the user need not concern himself with anything but the global shared memory. However, the fast local memories present in the design can be used to mask thc approximately 2 μs access time to global memory. Each cluster of eight processors contains eight, 16K local memories A given processor can access its own local memory module directly, or any local memory in its cluster through the cluster network.

User transparent access to these local memories will be provided in several ways. First, program code can be moved from global to local memories in large blocks by the cluster and global control units. Time required for these transfers will be masked by computation. Second, the optimizing compiler will generate code to cause movement of blocks of certain data between global and local memory. Third, automatic caching hardware (using the local memories) will be available for certain data where the compiler cannot determine *a priori* the details of the access patterns but where freedom from cache coherency problems can be certified.

All levels of memory include operand level synchronization facilities (similar to the full/empty bit of the Denelcor HEP [Smit82]), and the global shared memory includes the (programmer) option of virtual memory.

Figure 3 shows a programmers' view of the memory system. Both the cluster and local memories include cache space (which is physically implemented in the local 16K memories) for global memory accesses. This cache is different from most cache memory schemes in that not all global memory accesses are cached–only those predetermined by the programmer or compiler. In this way, we avoid the cache consistency problems which plague most multiprocessor cache designs. Thus, only read-only data (or data that is determined by the compiler to be read-only during a short phase of a program) or data that is only written by a single processor (private data) is cached.

Thus a user need only be concerned with a single uniform globally shared memory, and he could quickly design a program to execute from this memory. When he is satisfied with his results, he can use the optimizing compiler to improve his performance by making better use of the entire memory hierarchy and by utilizing more parallelism.

GLOBAL SHARED MEMORY	GLOBAL SHARED VIRTUAL MEMORY	CLUSTER SHARED MEMORY	CLUSTER SHARED CACHE	LOCAL MEMORY	LOCAL CACHE

Figure 3. Programmers' view of the memory system.

2.4. Global Control Unit. The execution of a program is limited by the parallelism exhibited by the control mechanism. In a von Neumann machine, the parallelism is limited by a serial control mechanism in which each statement is executed separately in the order specified by the program.

The execution speed can be increased by using parallel control flow or dataflow mechanisms [TrBH82]. Each of these mechanisms tries to execute all independent operations in parallel, where the operation is a typical arithmetic operation (e.g., addition, multiplication, etc.) or control operation (e.g., decision). However, the number of resources (e.g., operational units) in the machine is limited and sometimes not all independent operations can be executed in parallel. Therefore, the resources must be allocated and deallocated in the order specified by the computation. The price paid for parallelism is in the form of extra time or hardware needed to allocate operational units to instructions and keeping track of the execution order, the process we call scheduling. Proposed dataflow architectures are very inefficient on regular structures because of this fine granularity of their operations. When data is structured (vectors, matrices, records), the control and dataflow is very regular and predictable and there is no need to pay high overhead for scheduling.

In our system, we adapt to the granularity of the data structure. We treat large structures (arrays) as one object. We reduce scheduling overhead by combining together as many scalar operations as possible, and executing them as one object[Corn81]. In our machine, each Processor Cluster (PC) can be considered as an execution unit of a macro-dataflow machine. Each PC executes a chunk of the original program called a compound function (CF).

From the GCU point of view, a program is a directed graph called a flow graph. The nodes of this graph are compound functions and the arcs define the execution order for the compound functions of a program.

The nodes in our graph can be divided into two groups: Computational (CPF) and Control (CTF). All CTFs are executed in the GCU, and all the CPFs are done by clusters. All CPFs have one predecessor and one successor. CTFs are used to specify multiple control paths, conditional or unconditional.

The compound function graph is executed by the GCU. Each node requires two different types of action:

(1) Computation of the original part of the program specified in the CF which may be done by the GCU itself or allocated to PCs. The latter case is for CPFs, and it requires their scheduling and preparation. The CTFs either do not have this part at all, or perform computation related to control.

(2) Graph update after the executable part of a node is done (if it had one). Successors of each node are updated and checked for readiness. The updating consists of recording that the predecessor node was executed. A node is ready when all its predecessors in the graph are done. When a node is finished, the predecessor information is reinitialized for the next execution of the node (if it is a cycle, for instance).

The second problem of dataflow architectures is storage allocation, deallocation and movement of data, resulting in slow data access. In our machine, data is stored permanently in global memory and it can be shared there by all PCs. The data is moved into the assigned PC before the execution of a CF and later stored back to global memory. In this way, the movement of data is minimized while the order and locality of data is preserved. **Thus, the macro-dataflow architecture combines the control mechanism of**

dataflow architectures and storage management of the von Neumann machine.

3. Software

The primary language for Cedar will be Fortran (although we expect several other languages to become available as well). In Fortran, users will have a choice of writing programs directly in an extended Fortran (based on the ANSI 8X standard), or of using their old programs as is. The powerful restructuring capabilities of Parafrase [KuPa79] will usually be brought to bear on programs written in serial Fortran and may also (though not necessarily) be applied to programs written in extended (parallel) Fortran. Since the Parafrase system operates in a source-to-source manner, the user who used Parafrase can then choose to maintain his original code or the new, restructured version (thus obviating the need for further restructuring. [KPSW82])

The compiler will provide the Cedar system with code that may be regarded as a dependence graph containing several types of nodes called compound functions (CFs). This macro-dataflow graph is presented to the global control unit (GCU) which oversees its execution. Some CFs are control functions executed in the GCU itself, while other nodes are computational and are passed down to clusters of processors [GaKP81].

Four important kinds of parallelism may be distinguished in the Cedar macro-dataflow code. The first is parallelism between CFs themselves. This includes executing some CFs on the GCU and some in processor clusters, as well as executing several CFs at once in different processor clusters.

The second kind of parallelism is in the loop control of the computational CFs. For example, all iterations of a loop may be executable at once and so each iteration can be assigned to a different processor. Of course, the GCU may fold a computation onto a limited number of processors and each processor will then do a number of iterations ([KuPa79], [PaKL80]).

Third, is a kind of pipelining effect achieved by moving data from global to local memories before it is needed for the computation of a CF. We can prefetch data for iteration **i+1** while computing iteration **i**, for example, or we can prefetch larger blocks. Experimental evidence shows that this approach will be effective in exploiting the local memories in clusters. This software cache management only works effectively for data that can be guaranteed (by the compiler) to be written at most once in a given phase of a computation. Otherwise, cache coherency problems can develop, and our solution to this is to force any nonsafe code to execute directly from the global shared memory. This will cause some speed decrease (by a constant factor) and should be a rather rare event in any case. The global memory has of course only one copy of the data, and hardware will ensure that the correct value is stored.

The fourth kind of parallelism will be exploited mainly for phases of computations in which there is less parallelism than processors. This involves spreading expressions across more than one processor for execution. For example, if a loop of 50 iterations could be run as 50 independent iterations, but our machine had 100 processors available, two processors could be used for each iteration. This code spreading is entirely within individual compound functions and may involve executing independent assignment statements in distinct processors or even spreading single expressions over two or more processors. Experiments to date show that spreading can be very effective in some cases but it is not a first priority technique.

Some standard operating system functions will be handled by our hardware, e.g., task scheduling in the GCU. The I/O clusters will handle some of the activities that are traditionally at the interface between the compiler, OS, and I/O channels. In particular, we plan to have the I/O clusters execute I/O statements and do format conversions. They will also handle page faults between the global memory and disk system. We also plan to attach front-end processors to the I/O clusters.

A front-end processor will provide various user services. We would expect it to be a network node in any installation and in the Department of Computer Science at the University of Illinois we will attach it to a VAX/Ethernet network within the department. The point is that users should be able to access the system through an interface with which they are familiar and happy (VMS, UNIX, NOS, or whatever). Thus, a user would submit a job through a front-end processor which sends it to an I/O cluster, which in turn can initiate I/O directly or begin execution through the GCU. Results will be returned through the I/O cluster to the front-end processor for output, graphics display, etc. In this way, we hope to make the architectural details of the Cedar system as invisible to the user as possible.

4. Summary

The Cedar architecture nicely integrates the five key developments sketched in the introduction. We believe that the Cedar system will deliver high performance over a much wider range of applications and algorithms than today's supercomputers can handle. Because the Cedar clock speed is slow relative to such systems, the complexities of building and manufacturing this system are substantially reduced. Due to the ever-decreasing costs of integrated circuits and the relative ease with which the Cedar design can be partitioned, we feel that the monetary cost per megaflop will be much lower than could be achieved by attempting to push today's pipelined supercomputers to higher speeds.

REFERENCES

[BaLu81] G. H. Barnes and S. T. Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems," *IEEE Computer,* Vol. 14, No. 12, pp. 31-41, Dec., 1981.

[Burr79] Burroughs Corporation, "Numerical Aerodynamic Simulation Facility Feasibility Study," Mar., 1979.

[Corn81] Cornish, M., "Lecture Notes in Dataflow Computer Architecture," MIT, 1981.

[GaKP81] D. D. Gajski, D. J. Kuck, and D. A. Padua, "Dependence Driven Computation," *Proc. of the COMPCON 81 Spring Computer Conf., San Francisco, CA,* pp. 168-172, Feb., 1981.

[GLPV83] D. D. Gajski, D. H. Lawrie, J-K. Peir, A. Veidenbaum, and P-C. Yew, "Second Preliminary Specification of Cedar," Cedar document no. 8, Univ. of Ill. at Urb.-Champ., Dept. of Comput. Sci., Feb., 1983.

[KPSW82] D. Kuck, D. Padua, A. Sameh, and M. Wolfe, "Languages and High-Performance Computations," Invited paper, *Proc. of the IFIP Working Conf. on The Relationship Between Numerical Computation and Programming Languages,* Boulder, CO, pp. 205-221, Aug., 1982 (North-Holland).

[KuPa79] D. J. Kuck and D. A. Padua, "High-Speed Multiprocessors and Their Compilers," *Proc. of the 1979 Int'l. Conf. on Parallel Processing,* pp. 5-16, Aug., 1979.

[Lawr75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. on Computers,* Vol. C-24, No. 12, pp. 1145-1155, Dec., 1975.

[Padm83] K. Padmanabhan, PhD Thesis, University of Illinois, in preparation, 1983.

[PaKL80] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. on Computers,* Vol. C-29, No. 9, pp. 763-776, Sept., 1980.

[Smit82] B. S. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Proc. of the International Society for Optical Engineering,* Vol. 298, pp. 241-248, 1982.

[TrBH82] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architectures," *Comput. Surveys,* Vol. 14, No. 1, pp. 93-143, Mar., 1982.

[WaMc82] F. Ware and W. McAllister, "C-MOS Chip Set Streamlines Floating-Point Processing," *Electronics,* pp. 149-152, Feb. 10, 1982.

VECTOR OPTIMIZATION ON THE CYBER 205

CLIFFORD N. ARNOLD
SOFTWARE RESEARCH
CONTROL DATA CORPORATION
ARDEN HILLS, MINNESOTA

ABSTRACT. Vector optimization is defined as generating the best object code for a given vector computation for a given machine. In this paper an analytical performance model is developed for both scalar and vector source code as executed on the Control Data CYBER 205. The accuracy of these models is typically within 30% for scalar code and within 10% for vector code. If the compiler can generate more than one version of code for a given parallel computation, then performance estimates from these models can be used to choose which version should be executed. Sixteen FORTRAN kernels with two or more versions of source code were used as a benchmark to test this method. Thirteen kernels were "vector optimized" correctly. The three kernels not properly optimized had an average performance penalty of 17%. Using the set of kernels as a benchmark, vector optimization improved its performance by more than a factor of four, and obtained 98% of the improvement possible had all kernels been correctly optimized.

I. INTRODUCTION

Automatic vectorization is the process by which a serial or scalar version of source code is analyzed by a piece of software to discover the code's inherent parallelism. The result is a transformation of the original code which expresses the parallelism discovered by the analysis. This can take many forms, for example rewritten source code using a parallel dialect, machine dependent object code, and many expressions in a variety of languages in between. A lot of work has gone into automatic vectorization software, both in the computer industry and in academia. Performance analysis of three such software packages is reviewed in [1]. There the concept of vector optimization was introduced; "The goal of vector optimization is to generate the best object code for a given vector computation (for a given computer)." Little study has been done in this area, which is unfortunate. As users of vector computers in the field know all too well, code vectorization to a higher level of parallelism can in some cases slow the computation down.

This paper addresses the following question. As information pertaining to the vectorizability of a kernel is increased, will a kernel always run at least at the same speed as before with the possibility of being sped up many times? Clearly the scalar (original) version of the code can be used, thus guaranteeing the same execution speed as before vectorization. To make vectorization analysis worth the trouble, a vector optimization tool needs to determine correctly when to choose the scalar version of the code and when to choose the vector version. In many cases more than one vector version is possible, and in such cases a decision among these versions also needs to be made. Figure 1 below illustrates the issue. Three different possible code solutions are depicted with their performance assumed to be a function of some data dependent parameter set.

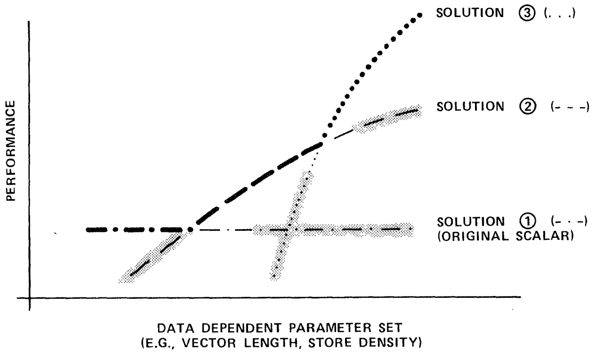


FIGURE 1. Performance Profile of Three Code Versions of the Same Kernel

Vector optimization would hopefully choose the bold face line. If it did not successfully do that, it would successfully avoid the shaded areas.

I have applied a simple but surprisingly successful model to estimate the timings of different solutions based on their respective source code versions alone. These estimates are then used to make the optimization decisions. In this paper I describe the timing model for vector performance (Section II) and scalar perfomance (Section III) for the CYBER 205. Validation of the model against 16 kernels from the Livermore Loops (3) as run through several vectorizers (see [1]) is described in Section IV. Results are discussed in Section V.

0190-3918/83/0000/0530$01.00 © 1983 IEEE

II. MODELING VECTOR PERFORMANCE OF THE CYBER 205

Assume the time to execute a vector operation is a linear function of the vector length n:

$$\text{time} = S + R * n \qquad (1)$$

Equation (1) is rewritten by Hockney and Jesshope [2] as:

$$t = (n + n_{[1/2]})/r_{[\infty]} \qquad (2)$$

where $r_{[\infty]}$ is the asymptotic performance (MFLOPS) at infinite vector length and $n_{[1/2]}$ is the vector length required to achieve half the asymptotic performance. Note that t is then in microseconds. The coefficients in Equation (1) are defined in terms of $r_{[\infty]}$ and $n_{[1/2]}$ as noted below:

$$\begin{aligned} R &= 1/r_{[\infty]} \text{ (microsec/result)} \\ S &= n_{[1/2]}/r_{[\infty]} \text{ (microsec)} \end{aligned} \qquad (3)$$

R and S were measured for 34 vector instructions using vector lengths ranging from 2 to 8192 in powers of 2. The timings proved to follow the model of Equation (1) very well. The results are listed in Table 1. The data for R is significant to two digits and for S to three digits, with a few exceptions shown in the list. Typical values for R range from 0.01 to 0.03 microsec/result, and for S range from 1 to 3 microsec ($n_{[1/2]}$~100).

Test kernels with several vector operations had timings entirely consistent with the timings of the individual vector operations added together. This is predicted by the linear model.

III. MODELING SCALAR PERFORMANCE OF THE CYBER 205

Initially Equation (1) was also applied to scalar loops where n was the trip count of the loop. Over a hundred short test kernels were timed. Though the resulting timings were linear, it was found that R was not constant for a given instruction in different loop contexts. This should not be surprising. For example, if loads and stores for an add operation can be overlapped by a multiply operation in the same loop, the rate is clearly different than if no overlap can be scheduled. R for an add operation was found to range from 0.42 microsec/result to 0.08 microsec/result depending on the context of the loop. Figure 2 shows the dependence of R on the number of operations (I) in the loop. For adds and multiplies this can be fitted well by:

$$R = 0.08 * \exp(1.61/I) \qquad (4)$$

$$t(\text{scalar loop}) = S + R * n * I \qquad (4a)$$

For divide and square root, R is constant:

$$R = 1.1 \text{ microsec/result} \qquad (5)$$

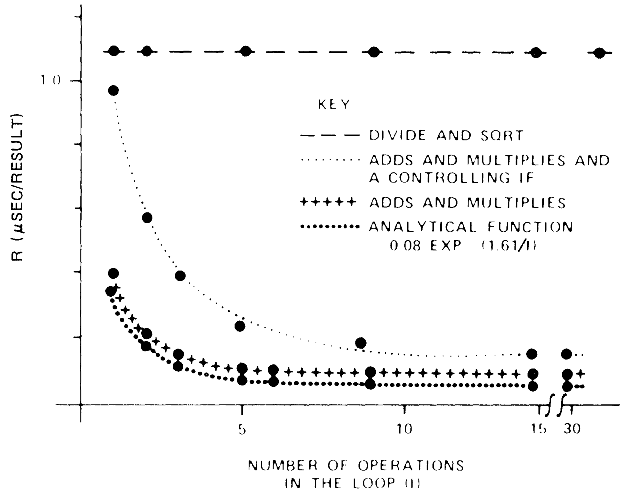


FIGURE 2. R in Scalar Loops

More testing showed R to also be a function of whether the operands were in memory or in the scalar register file (256 registers). All dimensioned variables (indexed operands) are in memory whereas as many scalar variables as possible are in the register file, for which operations are significantly quicker. Thus Equation (4) was fitted to include register to register operations:

$$R[\text{scalar}] = (0.08 - 0.03*(J/(J+K))) * \exp(1.61/I) \qquad (6)$$

where J is the number of scalar references and K is the number of indexed references in the loop. (Note: I ~ J+K).

The behavior of S and R is surprisingly simple for scalar loops. S is always less than 1.0 microsec, and usually about 0.25 microsec. Each IF statement in the loop (assuming no nested IFs) adds 0.7/I microsec/result to R and has no effect on S. For nested structures, like nested Do Loops, simply evaluate the inner most loop first and then have it act as an in-line routine with its characteristic I, J, K and S added to the other statements in the next level of the nest structure.

Initially it seemed unlikely that the scalar unit could be timed as simply as noted above. The FORTRAN compiler's scalar optimizer and the hardware were essentially being treated as a black box. It is interesting and curious to suggest that it can be timed empirically without knowledge of its detailed operation. Investigators might want to try this for other machines.

TABLE 1. INSTRUCTION TIMING
MODEL FOR C205 VECTOR OPERATIONS

Operation	S (microsec)	R (microsec/result)	$n_{[1/2]}$= S/R
ASSIGNMENT	1.00	0.010	100
ADD (Floating Pt.)	1.00	0.010	100
MULTIPLY (Floating Pt.)	1.04	0.010	104
DIVIDE (Floating Pt.)	1.60	0.031	52
ADD (Integer)	1.04	0.010	104
MULTIPLY (Integer)	3.72	0.040	93
DIVIDE (Integer)	2.50	0.041	61
Q8VINTL	0.95	0.020	48
Q8VGATHR (Index List)	1.50	0.027 to 0.081	53
COMPARE (Floating Pt.)	1.35	0.010	135
Q8SSUM	2.50	0.020	125
Q8VSCATR (Index List)	1.38	0.025 to 0.081	55
* Q8SDOT	2.65	0.020	133
Q8VMASK	1.90	0.010	190
Q8VCMPRS	1.38	0.010	138
Q8VEXPND	1.45	0.010	145
Q8VGATHP (Periodic)	0.83	0.029	29
Q8VSCATP (Periodic)	1.50	0.024	125
Q8SMAX	1.50	0.020	75
VSQRT	1.55	0.030	52
* LINK ([S(+)V](+)V)	2.84	0.010	284
* LINK ([V(+)V](+)S)	2.60	0.010	260
VIFIX	1.05	0.010	105
VFLOAT	1.05	0.010	105
Q8VMKO(Z)	1.04	0.0013	800
BIT COMPARE	1.22	0.00125	976
WHERE (Logical)	0.20	0.0 #	
WHERE (Expression)	1.00	0.0 #	
STACKLIB TRIADS	5.-16.	0.144	35 to 111
STACKLIB DIADS	2.- 4.	0.113 to 0.128	18 to 35
Q8VMERG	1.50	0.010	150
VEXP	24.	0.5 to 0.6	40 to 48
(REAL)**REAL	24.	0.05 to 0.06	400 to 480
VSIN	19.	0.28 to 0.42	45 to 68
Q8VCTRL	1.10	0.010	110

* Each result takes two (2) floating point operations.

# The WHERE statement only has start-up time. The time per result is zero. If there is an expression in the WHERE statement its timing must be calculated separately and then the WHERE statement start-up added to it.

(+) This signifies either the multipication or addition operator in the Link instruction. In a given Link on the Cyber 205 there can not be more than one add or multiply. There are other types of operators which can be Linked, but these were not timed.

IV. PERFORMANCE MODEL VALIDATION

A test base of 18 FORTRAN kernels from the Lawrence Livermore Laboratory [3] were used just as in [1]. For each loop there were at least two different source code versions, scalar and vector, and for loops 2, 4, and 18 there were two or more vector versions. Kernels 16 and 17 were discarded because the unknown data dependent branching probabilities made these loops impossible to time. Note that the same was not true of Loop 15 which has many IF statements in the original code.

Using Equations (1), (4a), (5), and (6) and Table 1, the test base kernels were timed using the source code versions alone. Scalar times converted to MFLOPS are listed in Table 2. These are compared to the actual timings with the relative error noted. On the average the predicted performance is 15% too high. More interesting is the spread in the relative error. In statisics this is called the "sample varience" or "standard deviation," and is annotated as "sigma." For this sample the scalar timings have a sigma of about 31%. Thus, over a performance range of 1.7 to 22.4 MFLOPS, scalar timings can be predicted repeatedly to within ±31% using a very simple three parameter equation (S, J, and I). Figure 3 shows the distribution of errors in a histogram format.

Vector timings, converted to MFLOPS are listed in Table 3. The triple entries represent three different vector lengths where typically the second length is twice the first, and the third vector length is twice the second (see [1]). Note that performance is well predicted over a large span of vector lengths and over a performance range of 3.4 to 168 MFLOPS. Loops 2a, 4, 8, 13, 14, 15, 18 are examples of kernels that are tricky timing exercises requiring index list gathers and scatters, partial vectorization, bit vector operations, WHERE blocks, multiple nested loops, and up to 73 timing equations (Loop 15).

Figure 4 shows in a histogram format the distribution of errors for this test base in vector mode. Loop 13 is a statistical "outlier" at more than the 3 sigma level, implying that it is not a statistical anomaly but a technical one not satisfied by the model. Therefore it should not be used in the sample for calculating the sample mean or variance. With that loop eliminated, Loop 14 is a statistical outlier at the 2 sigma level, suggestive, but not a compelling reason to delete it from the sample. Ignoring Loop 13, the average predicted performance is about 5% too high. The spread in the relative error (sigma) is less than 10%.

Loops 13 and 14 both involve random indexing through memory. My timings for Q8VGATHR and Q8VSCATR assumed a "well behaved list," and they likely are not that well behaved. More tests showed the performance could degrade by a factor of 3 in R in the worst case. Thus a range in R is shown in Table 1. These ranges more than make up for the error in the predictions of Loops 13 and 14. A good working value of R for these two instructions is 0.035 microsec/result. Making this correction to the performance predictions of these loops brings these estimates into line with their actual performance.

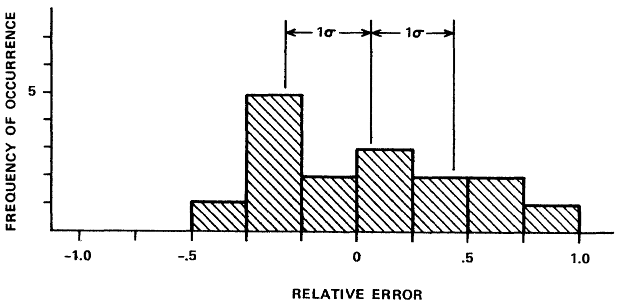


FIGURE 3. Histogram of Relative Error for Scalar Loop Timing Predictions

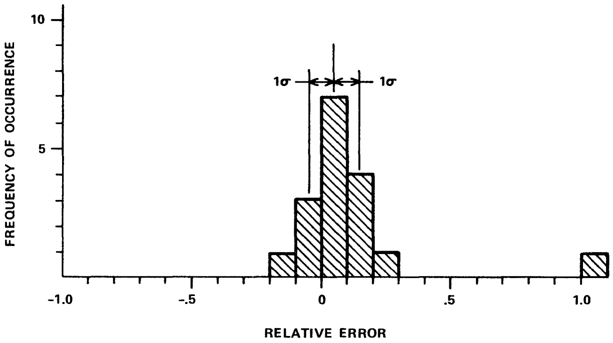


FIGURE 4. Histogram of Relative Errors for Vector Code Timing Predictions

TABLE 2. C205 PERFORMANCE MODEL - Scalar Code

Kernel No.	Predicted (MFLOPS)	Actual (MFLOPS)	Relative Error
1	11.1	9.6	15.6%
2	11.0	12.3	-10.6%
3	6.4	5.9	8.5%
4	5.4	3.3	73.6%
5	9.6	7.9	21.5%
6	8.5	5.2	62.5%
7	15.1	17.0	-11.2%
8	15.2	22.4	-32.1%
9	13.6	13.0	4.6%
10	11.7	8.6	36.0%
11	2.5	1.7	47.0%
12	2.5	2.9	-13.8%
13	4.7	3.1	51.6%
14	4.7	5.5	-14.5%
15	3.8	3.4	11.8%
18	7.3	8.3	-12.0%
			14.9 ± 31.3%

TABLE 3. C205 PERFORMANCE MODEL - Vector Code

Kernel No.	Predicted (MFLOPS)	Actual (MFLOPS)	Relative Error
1	89.7, 116.6, 137.2	94.4, 122, 138.9	-3.5%
2a	12.3, 16.9, 20.8	12.5, 16.2, 19.8	2.6% ± 3.6%
2b	65.4, 79.1, 88.3	64.6, 76.9, 87.2	1.8%
3	65.4, 79.1, 88.3	64.6, 76.9, 87.2	1.8%
4a	13.5, 22.9, 30.0	12.2, 21.5, 29.1	4.8%
4b	9.5, 16.1, 21.1	8.7, 15.3, 20.4	6.0%
5	5.8, 6.2, 6.5	5.1, 5.7, 5.9	10.9%
6	6.1, 6.5, 6.7	5.4, 6.1, 6.4	8.1%
7	93.6, 127.6, 155.8	113, 146, 168	-12.3%
8	3.2, 18.1, 18.9	3.4, 15.9, 16.6	7.3% ± 11.4%
9	52.0, 79.6, 108.5	54.4, 81, 110	-2.5%
10	22.4, 30.5, 37.1	22.7, 30.0, 36.1	1.0% ± 2.1%
11	7.6, 7.9, 8.1	8.0, 8.5, 8.6	-6.0%
12	71.4, 83.3, 90.9	62, 75, 83	11.9%
13	7.7, 8.8, 9.4	3.9, 4.4, 4.5	102%
14	6.2, 6.5, 6.6	5.0, 5.3, 5.2	24.5%
15	21.4, 29.2, 35.7	18.4, 25.6, 30.3	16.1%
18a	44.1, 54.4, 61.6	42.3, 50.4, 55.6	7.7%
18b	38.3, 47.4, 53.8	35.0, 42.4, 47.3	11.7%
18c	4.2, 4.2, 4.2	4.2, 4.2, 4.2	0%
	All 16 Kernels		9.7% ± 23.2%
	All 16 Kernels except #13		4.8% ± 8.4%

V. RESULTS

For 9 of the 16 kernels in this test base (kernels 1, 2, 3, 7, 9, 10, 11, 12, and 15), vector optimization clearly discriminates among the possible source code choices. The likelihood of an error being made in any of these choices is quite small. The differences in their respective scalar and vector performance predictions far exceeds the sum of their expected errors, in all cases by more than 2 sigma and in most by more than 3 sigma. Of these 9 kernels, kernels 2, and 15 were subtle timing exercises (Section IV).

Making the proper choice of source code for kernels 4, 5, 6, 8, 13, 14 and 18 is not so straightforward. Figure 5 shows the probability of making an error in code choice for a kernel as a function of the separation of the performance estimates. An error is made when the code version with the faster time estimate turns out to be the slower running version. In Figure 5, M1 and M2 are the respective performance estimates of two choices of source code for the same kernel. Kernels 2a, 4ab, 5, 6, 8, 13, 14 and 18ab are noted on the figure as examples. The average overestimation of vector performance (5%) and scalar performance (15%) has been factored out in these cases. The standard errors, S1 and S2, for scalar and vector estimates are 31% and 10% of their respective estimates (Mv and Sv for the vector estimate and Ms and Ss for the scalar one). Out of these 8 examples, it is expected that one (rounding to the nearest integer) will be wrong. In fact, three of them are (kernels 6, 8, and 14). The slower versions are 15%, 29%, and 4% slower than their faster estimates not chosen. It should be noted that the scalar speed of kernel 8 is unusually fast (rumor has it that the compiler was turned on this particular piece of code).

Given an incorrect choice of the version of source code, statistics predicts how much error will likely cost in performance. Where Figure 5 shows the probability of making an error, Figure 6 shows the probability of the size of the penalty as a function of the separation of the estimates. This assumes that the error has been made. Kernels 6, 8, and 14 are included as examples. This "penalty" function is fairly flat with the average penalty for a large sample of incorrect choices typically in the range of 5% to 20%. The top curve represents the 90% confidence interval of the penalty. Therefore the probability that the penalty lies below that line is 90%.

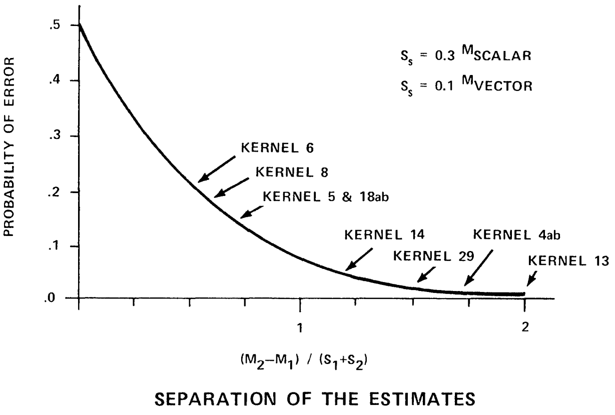


FIGURE 5. Probability of Choosing the Wrong Source Code Version

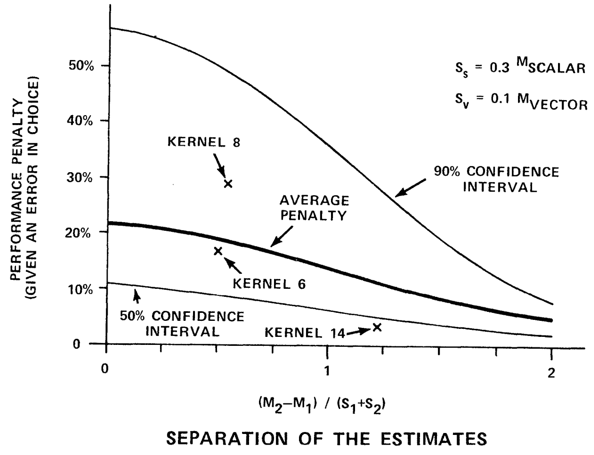


FIGURE 6. Performance Penalty when the Wrong Code Version is Choosen

VI. Conclusions

The analytical timing model presented in this paper estimates the performance of scalar kernels with a standard deviation of 30%. The performance of vector kernels is more accurately predicted with standard deviation being 10%. In many cases these errors are insignificant compared to the difference in predicted timings for two coded versions (e.g. scalar versus vector) of the same kernel. In such cases vector optimization is eminently safe and useful. One class of kernels that will repeatedly fall into this category on the CYBER 205 are the kernels whose vector performance is predicted to exceed 25 MFLOPS. The test base showed some slower kernels for which vector optimization also clearly discriminated the faster version.

When vector optimization chooses between two versions of a kernel whose performance difference is comparable to the expected errors, the probability of making an error in choice is no longer negligible. When choosing between scalar and vector code, the chance of error is 8% when the performance difference is 40% of the average estimate. The chance of error grows to 42% when the performance difference is 10%. When choosing between two vector versions, the probability of error is 0.5% at a performance difference of 40% of the average estimate, and 22% at a performance difference of 10%. When an error is made the typical performance penalty incurred ranges from 5 to 20%. Rarely (less than 10% of the time) would the penalty exceed 50%. These conclusions assume that the errors in timing predictions obey Normal Distribution statistics. This assumption looks correct for the vector timing predictions, but is suspect for the scalar ones. This code sample implies that high performance scalar code is selectively underestimated, while the slow performance scalar code is selectively overestimated. Perhaps the analytical timing model for scalar performance needs a nonlinear term to better match the high and low performance extremes. I think this last point forces more interpretation than is really in the data. The model needs to be tested on more code, preferably real applications, and I intend doing this in the future.

The bottom line for this experiment is that out of 16 kernels, 9 have clear vector optimization choices from among two or more choices. Of the remaining 7 harder to discriminate versions 3 are chosen incorrectly for an average penalty of 17%. Weighting all 16 kernels equally, the mean scalar performance is 8.1 MFLOPS while the mean vector performance is 32.1 MFLOPS (for the shortest vector lengths), 41.6 MFLOPS (for the middle vector length), and 48.3 MFLOPS (for the longest vector lengths). Vector optimization yields a mean performance of 37.7, 47.2, 54.7 MFLOPS respectively for these vector lengths. If scalar code is chosen in all cases except where the best vector performance estimate exceeds the scalar one by 40% (less than 8% chance of slowing the code by choosing vector) then such a vector optimization algorithm would yield an average performance of 37.7, 48.0, 55.1 MFLOPS. If the vector optimization decisions had been all correct (that is, best effort), the mean performance for this test base would have been 37.7, 48.1, and 55.2 MFLOPS.

The vector optimization algorithm presented here should be easy to implement in a compiler or other automatic vectorization software.

VII. References

[1] C. N. Arnold, Performance Evaluation of Three Automatic Vectorizer Packages, International Conference for Parallel Processing 1982, Bellaire, Michigan, August 1982.

[2] R. W. Hockney, C. R. Jesshope, Parallel Computers, Adam Hilger Ltd, Bristol, UK, 1981.

[3] F. McMahon, 1972, Unpublished. (Available from C. N. Arnold on request)

PIPELINED EVALUATION OF FIRST-ORDER RECURRENCE SYSTEMS[*]

Lionel M. Ni
Department of Computer Science
Michigan State University
East Lansing, MI 48824

Kai Hwang
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract -- A first-order recurrence is a sequence of evaluations in which the value of the latest term depends on the previously computed term. Due to the sequential nature, it presents a special problem for parallel processing. For most scientific applications, only the final term is desired. This paper presents various strategies to evaluate the final value of first-order recurrence using pipeline. Two methods, symmetric reduction and asymmetric reduction, are proposed and compared in a static pipeline environment. The pipeline utilization can be further improved when multiple recurrence systems are evaluated.

I. Introduction

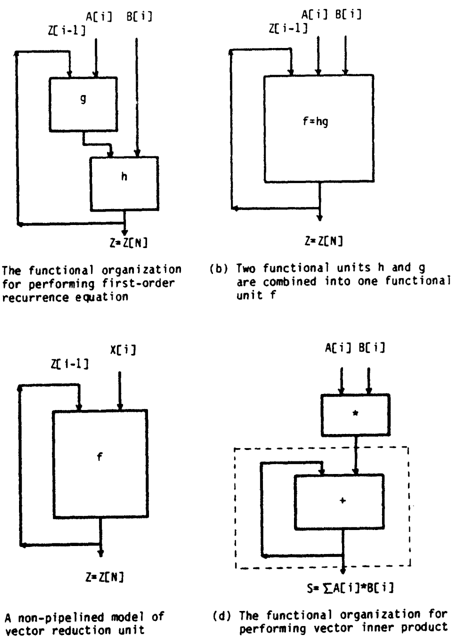
A first-order recurrence is a sequence of evaluations in which the value of the latest term in the sequence depends on the previously computed term. For most scientific applications, only the final term is desired, such as the inner product of two vectors, which forms the basis for most matrix manipulations. The inner product can be evaluated as a linear first-order recurrence. The evaluation of a recurrence presents a special problem for a parallel processing system because the definition itself is given in terms of sequential evaluation [3].

A general class of first-order recurrence equations can be expressed in the following form [7]:

$$Z[1] = B[1]$$
$$Z[i] = h(B[i],g(A[i],Z[i-1])) \quad 2\leq i\leq N \qquad (1)$$

where A[i] and B[i] are external inputs and h and g are index independent functions that satisfy the following restrictions: 1) h is associative; 2) g distributes over h; and 3) g is semiassociative. The external inputs are essentially two vectors with N elements. In this study, we are interested in the final scalar result Z[N] which is reduced from the vector inputs. Thus, the basic primitive operation which requires sequential processing in the evaluation of the first-order recurrence is vector reduction. Vector reduction accepts vector input and produces a single scalar output [5,8]. Perhaps the most common such vector reduction operation is the vector summation, which takes one input vector and produces a single output equal to the sum of the elements of the input. Other vector reduction operations can be found in [12].

In a sequential processor, the evaluation of the first-order recurrence involves a DO loop operating on one element from each vector, A[i] and B[i], at a time as defined in Eq.(1). Figure 1(a) shows a sequential organization for the evaluation of the first-order recurrence. Two functional units, g and f, are connected serially with output of h feeding the input of g. A simplified diagram which combines the functions h and g is shown in Fig. 1(b), where f=hg. We may combine the external input sources, A[i] and B[i], into one external input source X[i], as in the further simplified diagram shown in Figure 1(c). This is the model that we will use in this paper. f is also called a vector reduction operator because it reduces the vector input to a scalar output. Figure 1(d) shows the functional organization for performing inner product $S=\underline{A}\bullet\underline{B}$. The vector reduction operator, "+", is indicated by the dash-line box.



(a) The functional organization for performing first-order recurrence equation

(b) Two functional units h and g are combined into one functional unit f

(c) A non-pipelined model of vector reduction unit

(d) The functional organization for performing vector inner product

Fig.1. Evaluation of first-order recurrence

[*] This research was supported in part by the U.S. Engineering Foundation under grant RI-A-82-11 at Michigan State University and in part by the U.S. National Science Foundation under grant ECS-80-16580 at Purdue University.

Parallelism and pipelining are two major techniques in achieving high performance processor unit design. Evaluation of a recurrence system in an array processor has been extensively studied by many researchers [1,7]. An interconnection network is required in an array processor to provide the necessary routing paths for the purpose of cyclic vector reduction. An array processor which is capable of evaluating multiple recurrence systems has been studied by [6,10].

This paper deals with pipelined evaluation of the first-order recurrence. Pipelining generally takes the approach of splitting the function to be performed into small pieces and allocating separate hardware to each piece, termed a segment. Usually, the rate of data flow through the pipeline is independent of the number of segments and dependent on the rate at which new data may be fed into the pipeline. If a function is capable of being partitioned into K segments, then a pipeline designed to execute the same function repeatedly can perform the function K times faster, at most. This peak performance can be achieved only if the input elements are mutually independent and the input string is very long. However, due to the recurrence relation, peak performance may not be easily achieved.

The pipelined design of a vector reduction unit needs a feedback path from the output to the input of the pipeline. Here only the static pipeline is considered. The existence of feedback implies a certain sequentialism to the function being evaluated. Since each output of the pipeline depends on previous outputs, pipelining does not help in the direct implementation of Eq.(1). Improper or inefficient control of such feedback around a pipeline can destroy its efficiency and decrease its throughput. This paper studies the construction and scheduling of pipelined reduction units while maintaining a high throughput rate.

Vector summation or inner product instructions have been implemented in many vector processors, for example, IBM 3838, TI-ASC, STAR-100, CRAY-1 [4] and ESL systolic processor [10]. This paper provides a generalized model for the pipelined evaluation of first-order recurrence under different scheduling strategies. Both single vector input and multiple vector inputs are considered for an environment consisting of one pipelined reduction unit.

In section II, a traditional recursive reduction method is briefly described. A symmetric reduction method is then introduced. A more faster asymmetric reduction method is proposed in Section III. The above methods are also compared in terms of control complexity, processing speed, and the buffer requirements. Section IV discusses the situation in which multiple vector inputs request to perform the same recurrence operation in a single pipeline unit. An interleaved scheduling of multiple vector inputs sharing a single pipeline is proposed. Finally, an example is demonstrated with mixed single and multiple vector processing by chaining several pipeline units.

II. Symmetric Single Vector Reduction

A first-order recurrence system has a single vector input. It is assumed that a static pipeline with K segments is used in evaluating a vector input of N elements, where N and K are arbitrary integer values. Further, each memory or buffer can supply one element per pipeline cycle time.

In order to evaluate the final scalar result, a recursive reduction method has been proposed by Kuck [9] with N being an integer power of 2. The N elements are divided into two halves. N/2 pairs of elements are processed in the first iteration through the pipeline. The intermediate result vector of N/2 elements is then again divided into two halves, N/4 elements each, in the second iteration. After $\log_2 N$ iterations, the final scalar result is obtained.

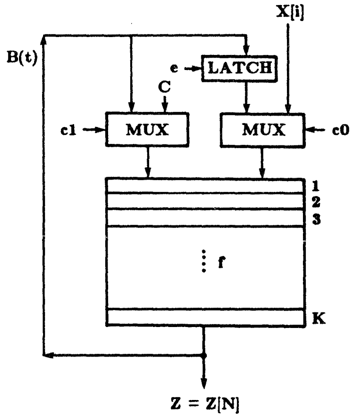
A generalized procedure based on the recursive reduction of a vector with an arbitrary value of N was specified in [12]. In addition to storing the input vector or the intermediate result in the memory, two extra buffers are needed to hold the divided subvectors. When N is large, the buffer will significantly increase the hardware cost and time delays. Let T(N,K) denote the number of cycles required to evaluate an input vector with N elements in a pipeline with K segments. The following result was obtained [12].

Theorem 1:

The recursive reduction method requires $T_{RR}(N,K)$ cycles, where

$$T_{RR}(N,K)=(K-1)\lceil\log_2 N\rceil+3(N-1) \qquad (2)$$

The excessive buffer can be eliminated by using a feedback path from the output to the input of the pipeline as shown in Fig. 2. B(t) is the feedback input at the t-th pipeline cycle and C is a constant. The feedback input will be latched if e=1. B(t-j) indicates the feedback input at the (t-j)-th cycle.



c1	c0	INPUT PAIR
0	0	C , X[i]
0	1	C , B(t-j)
1	0	B(t) , X[i]
1	1	B(t) , B(t-j)

Fig.2. The hardware organization of a pipelined processor with K segments

The reduction operator, f, is commutative. For ease of explanation, we assume N≥K. The N elements can be partitioned into K groups.

$$Z = Z[N] = f(P(1), P(2), ..., P(K)) \qquad (3)$$

where

$$P(i) = f(X[N-i+1], X[N-K-i+1], X[N-2K-i+1], ...) \quad \text{for } 1\leq i\leq K \qquad (4)$$

If (N mod K)=0 then P(i) is the reduction result of N/K elements for all i; otherwise, P(i) is the reduction result of $\lceil N/K \rceil$ elements for $1\leq i\leq (N \bmod K)$ and of $\lfloor N/K \rfloor$ elements for $(N \bmod K) < i\leq K$. T(N,K) can be divided into four phases.

$$T(N,K)=T_i(N,K)+T_p(N,K)+T_m(N,K)+T_d(N,K) \qquad (5)$$

where T_i is the time needed to fill up the pipeline, T_p is the time needed to partition the N elements into K groups, T_m is the time required to merge K groups into one group, and T_d is the time required to drain the pipeline. Since the input elements X[i]'s are supplied at the rate of one element per cycle, the following results are obtained.

$$T_i(N,K) = \text{Min}\{N,K\} \qquad (6)$$

$$T_p(N,K) = \text{Max}\{0,N-K\} \qquad (7)$$

In total, T_i and T_p add to N cycles. The input elements may come from local memory or from the output of another pipeline. The control inputs will be (c1,c0,e)=(0,0,0) and (c1,c0,e) = (1,0,0) for the evaluation of Eqs.(6) and (7), respectively. The constant input C is chosen so as to make f(C,X)=X. In the case of vector summation, C will be 0. After N cycles, the external input, X, to the pipeline will be set to the same value as C.

A pipeline segment is productive for a given clock period if the segment is actively involved in the computation during that period; otherwise, the segment is called unproductive. This implies that the result generated by an unproductive segment will be ignored. After N cycles (if N≥K), all K segments are productive and the i-th segment will be operating on P(i). The merging of K groups is done by combining two groups at a time. The number of groups is reduced by half after each iteration. Thus, $\lceil \log_2 K \rceil$ iterations will result in a single group. Two methods are proposed for the group-merging phase. The symmetric reduction method is described first. The asymmetric reduction method will be discussed in the subsequent section.

Once all groups are merged into one group, K cycles are needed to drain the pipeline in the last phase. The drain delay is equal to

$$T_d(N,K)=K \qquad (8)$$

In the symmetric reduction (SR) method, two consecutive productive segments are merged at one time as depicted in Fig. 3. Figure 3 shows the pattern of productive segments in each iteration for a pipeline of 10 segments, where K_i denotes the number of productive segments after the i-th iteration and K_0=K. When K_i is odd, the group in the first segment will be merged with a dummy group. Note that the group in the first segment always includes the original group P(1) at the beginning and end of each iteration. Thus, the time required for each iteration is the time needed to route the group in the first segment, possibly merged with another group, back to the first segment. The distance between two consecutive productive segments is always 2^i after the i-th iteration. The control sequence is illustrated below for merging K' groups in a pipeline with K'≤K. The function, E(x), is defined to be 1 if the integer x is odd. If x is even, then E(x)=0.

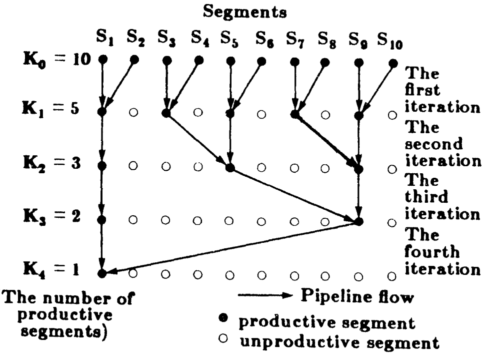


Fig.3. The pattern of productive segments before and after each iteration for the symmetric reduction in a pipeline with K=10 segments

ALGORITHM-SGM: Symmetric Group-Merging
Input: K' productive segments (K'≤K).
Output: (1) The control sequence (c1,c0,e), and
(2) The group-merging time $T_{m(SR)}(K')$ of of merging K' groups into one group.

```
Procedure:
Begin
  t=0;
  for i=1 to $\lceil \log_2 K' \rceil$ do begin
    if K' < K then {for t=t+1 to t=t+K-K', set
                (c1,c0,e)=(0,0,0)};
    if $(K'-1)\bmod(2^{i-1}) > 0$ then {for t=t+1 to
       t=t+$(K'-1)\bmod(2^{i-1})$, (c1,c0,e)=(0,0,0)};
    G=0; (* G is a boolean variable *)
    D=E($\lceil K'/2^{i-1} \rceil$) (* D is a boolean variable *)
    for j=1 to $\lceil K'/2^{i-1} \rceil$ - D do begin
      for t=t+1, set (c1,c0,e)=(G,G,G');
      if i>1 then {for t=t+1 to t=t+$2^{i-1}$-1,
                  set (c1,c0,e)=(0,0,0)};
      G=G';
    end; (* end for *)
    for t=t+1 do (c1,c0,e)=(D,1,0);
  end; (* end for *)
  $T_{m(SR)}(K')$=t;
End.
```

If N<K, then K'=N; otherwise, K'=K. The above algorithm can be applied to an arbitrary value of N and $T_{m(SR)}(N,K)=T_{m(SR)}(K')$. The number of cycles required for each iteration will be the time required to go through the whole pipeline once, i.e., K cycles, plus a certain amount of delay cycles in the latch if the number of productive segments is odd. Table 1(a) shows the contents of pipeline segments and the latch during the merging of six groups in each cycle using the SR method. The following lemmas are used in evaluating the time required in the group-merging phase. Lemma 1 can be proved by induction.

Lemma 1:

The number of productive segments after the i-th iteration equals

$$K_i = \lceil K_{i-1}/2 \rceil = \lceil K/2^i \rceil \qquad (9)$$

Lemma 2:

If K is not an integer power of 2, then

$$\sum_{i=0}^{\lceil \log_2 K \rceil -1} 2^i E(K_i) = 2^{\lceil \log_2 K \rceil} - K \qquad (10)$$

Proof: Eq.(10) can be proved by induction. However, the following equation must be proved first.

$$\sum_{i=0}^{\lceil \log_2 K \rceil -1} 2^i E(K_i) - \sum_{i=0}^{\lceil \log_2 (K+1) \rceil -1} 2^i E((K+1)_i) = 1 \qquad (11)$$

Since K is not an integer power of 2, K can be written as $K=2^m d$, where $m \geq 0$ and d is an odd number. Thus, $E(K_i)=0$ for $i<m$ and $E(K_m)=1$. Then we want to show that $E((K+1)_i)=1$ for $i<m$ and $E((K+1)_m)=0$.

If K is odd, then m=0 and K+1 is even. Thus, $E(K_0)=1$, and $E((K+1)_0)=1$.

If K is even, then K+1 is odd and $(K+1)_0=2^m d+1$. $(K+1)_i$ can be expressed as

$$(K+1)_i = \lceil 2^{m-i} d + 2^{-i} \rceil = 2^{m-i} d + 1 \quad \text{for } i<m$$

Thus, $E((K+1)_i)=1$ for $i<m$. Since d is odd and $(K+1)_m = \lceil d+2^{-m} \rceil = d+1$, we have $E(K+1)_m)=0$ and $(K+1)_{m+1} = \lceil (d+1)/2 \rceil = \lceil d/2 \rceil = K_{m+1}$. From Lemma 1, we have $K_i=(K+1)_i$ for $i>m$.

Since K is not an integer power of 2, this implies that $\lceil \log_2 K \rceil = \lceil \log_2 (K+1) \rceil$. Eq.(11) thus can be proved. By induction, Eq.(10) can be easily derived.

Q.E.D.

Theorem 2:

In the group-merging phase of the symmetric reduction method, the total time delay equals

$$T_{m(SR)} = \begin{cases} g(K) & \text{if } N \geq K \\ g(N)+(K-N)\lceil \log_2 N \rceil & \text{if } N<K \end{cases} \qquad (12)$$

where $g(M)=M\lceil \log_2 M \rceil + 2^{\lceil \log_2 M \rceil} - M$.

Proof of the Theorem 2 follows directly from the Lemma 2. Note that if K_{i-1} is even then K cycles are needed in the i-th iteration; otherwise, it takes an extra 2^{i-1} cycles to merge with a dummy segment as stated in the Algorithm. If N<K, we may consider a pipeline having N segments and K-N dummy segments. Thus, each iteration will required (K-N) extra cycles due to the delay in dummy segments. Since it takes $\lceil \log_2 N \rceil$ iterations, Eq.(12) is achieved.

III. Asymmetric Single Vector Reduction

The symmetric reduction method is good for microprogrammed control because the generation of control sequence has a regular pattern. However, it takes more cycles as shown in Table 1(a). In the second iteration, four cycles later, three groups have been merged into two groups. But it takes four more cycles to make the first pipeline segment containing P(1). The asymmetric reduction (AR) method can eliminate unnecessary delays as illustrated in Table 1(b). However, the control sequence is no longer a regular pattern because it is very difficult to express the distance between productive segments mathematically. A hardwired control can be easily implemented to generate the control sequence.

With the AR method, the pipeline processor needs to record the state of each segment as well as the latch. Denote the state of the latch as S_0 and the states of pipeline segments as S_1, S_2, ..., S_K. $S_i=1$ indicates that the i-th segment is productive; otherwise, it is 0. The state of the pipeline is expressed by a K+1 tuple $(S_0,S_1,\ldots,S_K)$. Initially, we have $S_0=0$

Table 1. Contents of pipeline segments (K=6) of each cycle during group-merging phase

	i=1						i=2								i=3					
L	6		4		2			5-6	5-6			1-2	1-2			3-6	3-6	3-6	3-6	
1		5-6		3-4		1-2				3-6				1-2						1-6
2	1		5-6		3-4		1-2				3-6				1-2					
3	2	1		5-6		3-4		1-2				3-6				1-2				
4	3	2	1		5-6		3-4		1-2				3-6				1-2			
5	4	3	2	1		5-6		3-4		1-2				3-6				1-2		
6	5	4	3	2	1		5-6		3-4		1-2				3-6				1-2	

(a) Symmetric Reduction (SR) method

	i=1						i=2				i=3					
L	6		4		2			5-6	5-6			1-2	1-2	1-2	1-2	
1		5-6		3-4		1-2				3-6						1-6
2	1		5-6		3-4		1-2				3-6					
3	2	1		5-6		3-4		1-2				3-6				
4	3	2	1		5-6		3-4		1-2				3-6			
5	4	3	2	1		5-6		3-4		1-2				3-6		
6	5	4	3	2	1		5-6		3-4		1-2				3-6	

(b) Asymmetric Reduction (AR) method

and $S_1=S_2=\ldots=S_K=1$, if $K\leq N$, and $S_1=S_2=\ldots=S_N=1$ and $S_{N+1}=\ldots=S_K=0$, if $K>N$. The group-merging operation is terminated if $S_1=1$ and $S_2=\ldots=S_K=0$.

The control signals are primarily determined by the current states of S_0, S_1, and S_K. The latch is enabled if $S_0=0$ and $S_K=1$. If the latch is occupied and the last segment is productive, these two groups will be merged in the next cycle. The control outputs can be expressed as follows:

$$\begin{aligned} c1 &= c0 = S_0S_K \\ e &= \overline{S}_0S_K \end{aligned} \tag{13}$$

A state transition table can be easily derived. The first segment becomes productive if two groups from the latch and the last segment are merged. The next state is expressed by the present state as follows:

$$\begin{aligned} S_0 &= S_0 \oplus S_K \\ S_1 &= S_0S_K \\ S_i &= S_{i-1} \qquad 2\leq i\leq K \end{aligned} \tag{14}$$

The above equations can be easily modified to cover three other phases. Theorem 3 states the total number of cycles required for the group-merging phase in the AR method.

Theorem 3:

In the group-merging phase of the asymmetric reduction method, the total time delay equals

$$T_{m(AR)}=\begin{cases} h(K) & \text{if } N\geq K \\ h(N)+(K-N)\lceil\log_2 N\rceil & \text{if } N<K \end{cases} \tag{15}$$

where $h(M)=M\lceil\log_2 M\rceil-2^{\lceil\log_2 M\rceil}+M$.

Proof: Using the AR, the number of productive segments in each iteration is same as with the SR method. However, the distance between any two consecutive productive segments is no longer a constant. But the distance between the first and the second occupied segment is still 2^{i-1} before the i-th iteration. If the number of productive segments, K_{i-1}, is odd before the i-th iteration, the last merge will be on the 2nd and the 3rd productive segments. The first productive segment does not have to go through the pipeline. Thus, if K_{i-1} is odd, it takes $K-2^{i-1}$ cycles for the i-th iteration; otherwise, it takes K cycles. Eq.(15) thus can be achieved by applying Lemma 2.

Q.E.D.

Table 2. Number of cycles required in each iteration of the group-merging phase for different size of pipeline segments

K	i=1			i=2			i=3			i=4			g(K)	h(K)
	PS	SR	AR	PS	SR	AR	PS	SR	AR	PS	SR	AR	SR	AR
1													0	0
2	1	2	2										2	2
3	2	4	2	1	3	3							7	5
4	2	4	4	1	4	4							8	8
5	3	6	4	2	7	3	1	5	5				18	12
6	3	6	6	2	8	4	1	6	6				20	16
7	4	8	6	2	7	7	1	7	7				22	20
8	4	8	8	2	8	8	1	8	8				24	24
9	5	10	8	3	11	7	2	13	5	1	9	9	43	29
10	5	10	10	3	12	8	2	14	6	1	10	10	46	34
11	6	12	10	3	11	11	2	15	7	1	11	11	49	39
12	6	12	12	3	12	12	2	16	8	1	12	12	52	44
13	7	14	12	4	15	11	2	13	13	1	13	13	55	49
14	7	14	14	4	16	12	2	14	14	1	14	14	58	54
15	8	16	14	4	15	15	2	15	15	1	15	15	61	59
16	8	16	16	4	16	16	2	16	16	1	16	16	64	64

PS: Productive segments after the i-th iteration
SR: Symmetric reduction method
AR: Asymmetric reduction method

Table 2 shows the total number of cycles required for group-merging for K=1 to 16 and for the SR and the AR methods, respectively. It also shows the number of cycles required and the number of productive segments in each iteration. Note that the number of cycles required for the SR method is at least K for each iteration, whereas it is at most K for the AR method.

Precisely, the SR method requires $2(2^{\lceil\log_2 K\rceil}-K)$ more cycles than the AR method, if $N\geq K$. If $K=2^k$ for some k, the total processing time will be the same for both methods. In the worst case of $K=2^k+1$, the SR methods requires 2(K-2) more cycles than the AR method. The number of pipeline segments, K, in a typical vector processor is in the range (2,15) [2]. Thus, when N is much greater than K, the difference in processing time between the SR method and the AR method is at most 2(K-2), which is insignificant. Table 3 lists the total vector reduction time for some typical values of N and K under three different reduction methods. In general, if N is much greater than K, the saving in vector reduction time in these two proposed methods is $2N+O(K(\log_2 N-\log_2 K))$ cycles over the recursive reduction method. Furthermore, the extra buffer space is eliminated.

Table 3. Comparison of three different methods in terms of the total processing time for some typical values of N and K

T(N,K)	N = 10			N = 100			N = 1000		
	RR	SR	AR	RR	SR	AR	RR	SR	AR
K= 3	35	24	14	311	110	108	3017	1010	1008
K= 5	43	33	27	325	123	117	3037	1023	1017
K= 8	55	42	42	346	132	132	3067	1032	1032
K=15	83	66	54	395	176	174	3137	1076	1074

RR: Recursive Reduction
SR: Symmetric Reduction
AR: Asymmetric Reduction

IV. Interleaved Multiple Vector Reduction

A set of independent recurrence systems with the same function may request to evaluate their final values. For example, a matrix and vector multiplication involves many independent vector inner product operations. The scheduling of multiple vector inputs in a single pipeline processor is studied below. Assuming M independent vectors of N elements each, the scalar result for the j-th vector is defined by

$$Y[j]=f(X[j,1], X[j,2], \ldots, X[j,N]) \quad 1\leq j\leq M \quad (16)$$

The memory is assumed to supply one element per pipeline cycle. The simplest approach to evaluate Eq.(16) is to process one vector at a time. The total processing time will be $M\cdot T(N,K)$. With careful control, draining the pipeline of the last vector input can overlap with filling up the pipeline of the next vector input. Thus, the total processing time can be reduced to $M\cdot T(N,K)-K(M-1)$. Even with this overlapping operations between vectors, the pipeline is still not fully utilized during the group-merging phase. A more efficient scheduling approach is to interleave the processing of multiple vector inputs.

Consider the case of M=K first. For interleaved processing, the memory will supply X[1,1], X[2,1],...,X[M,1] for the first M cycles, X[1,2], X[2,2], ..., X[M,2] for the next M cycles, and so on. After the first M cycles, the i-th pipeline segment will be operating on X[M+1-i,1] for $1\leq i\leq K$. Another M cycles later, the i-th pipeline segment will be operating on X[M+1-i,1] and X[M+1-i,2]. After M·N cycles, the i-th pipeline segment will be operating on X[M+1-i,1], X[M+1-i,2], ..., and X[M+1-i,N]. At the end, K=M more cycles are needed to drain the pipeline. The total processing time will be K(N+1).

Obviously, this approach is considerably faster than processing each vector sequentially because no merging phase is involved. Pipeline segments are unproductive only at the initial filling up phase and at the last draining phase. The idea of interleaved processing is to allow all pipeline segments to be shared by as many vectors as possible. It can be considered to have M virtual nonpipelined processors as shown in Fig.1(c). Each virtual processor is dedicated to one vector input with one input element per K cycles.

If M>K, not all M vectors can be allocated with pipeline segments. In order to save the intermediate results of the remaining M-K vectors, M-K dummy segments are introduced as shown in Fig. 4. Since the number of input vectors may vary, the length of the dummy segment buffer, D, is adjustable by program control. There are virtually M nonpipelined processors. Each virtual processor has a vector input with one element per M cycles. Physically, each vector input is assigned with one pipeline segment. With D dummy segments inserted, the pipeline can be viewed as an M-segment pipe. The total processing time will be M(N+1).

If M<K, some vectors may be allocated with more than one segment, and different vectors may have a different number of segments. In this case, the control of the pipeline will be very difficult because the procedure of merging groups for each vector will be input-dependent. Since K is not always an integer multiple of M, one way to allocate vectors with an equal number of segments is to leave some segments idle which results in low pipeline utilization. With the help of a dummy segment buffer, the pipeline can be fully utilized by choosing $D=\lceil K/M\rceil M-K$.

Let $Q=\lceil K/M\rceil=(D+K)/M$ be the number of segments allocated to each vector including dummy segments. By feeding the elements into the pipeline in an interleaved fashion as before, MN cycles later, all D+K segments will be occupied by groups of vectors. Each vector has Q groups in Q segments. Since Q>1, a group-merging phase is required. In Fig. 2, a latch was used to hold the group to be merged with the next group. For multiple vector inputs, each vector needs its own latch. A FIFO latch buffer is provided in Fig.4 for this purpose. Again, the system can be considered to have M virtual processors. Each processor is a pipeline organization with Q segments and each segment takes M cycles.

Either the SR method or the AR method can be used during the group-merging phase. However, the control sequence needs to be modified. The control sequence for each vector is the same as that of single vector reduction. The number of iterations will be $\lceil \log_2 Q\rceil$ during the group-merging phase. Since M vectors are processed in an interleaved fashion, each control output must be repeated M times, one for each input vector. The size of the FIFO buffer is obviously chosen to be K-1 because M<K. The total processing time in this case will be $M\cdot N+Q\cdot T_m(N,Q)+K$. The quantity $T_m(N,Q)$ depends on the reduction method to be used. This was evaluated in Theorem 2 and Theorem 3 for the SR and the AR methods, respectively.

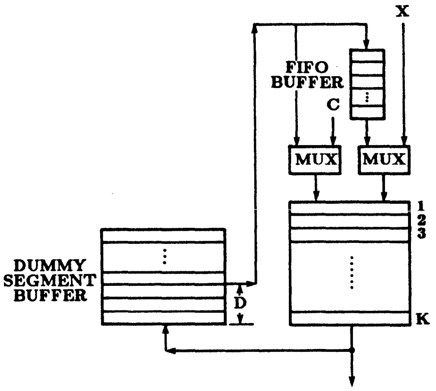


Fig.4. The hardware organization of a pipelined processor for interleaved multiple vector reduction

If the number of input vectors is too large, the dummy segment buffer may not be able to provide enough dummy segments. In this case, the vector inputs must be partitioned such that one block of input vectors is processed at a time according to the above procedures. In multiple vector processing, the low pipeline utilization due to group-merging can be essentially eliminated. The following example may further demonstrate the usage of a pipeline processor with a direct feedback path.

Example:

Let $\underline{A}$ be an NxN matrix and $\underline{B}$ be an Nx1 vector. $\underline{C}$ is an Nx1 vector obtained by performing matrix-vector multiplication of $\underline{A}$ and $\underline{B}$. Given $\underline{A}$ and $\underline{B}$, we want to find the maximum element in vector $\underline{C}$.

Three pipeline processors - multiplier, adder, and comparator - are chained together as shown in Fig. 5 with K_m, K_a, and K_c segments, respectively. For simplicity, N is assumed to be equal to nK_a. $\underline{A}$ and $\underline{B}$ are stored in two independent memory modules. The $\underline{A}$ matrix is partitioned into n blocks. Each block is a K_a by N submatrix.

The pipeline adder is needed to perform multiple vector reduction of N input vectors and to result in N elements of the $\underline{C}$ vector, whereas the pipeline comparator is used to perform single vector reduction and to produce the desired scalar output. To allow for overlapping of two blocks of data in a single pipeline processor, the feedback input either comes from the output of the pipeline or is supplied with a dummy input. The switching in the output of the pipeline can be easily controlled by a counter of value K_aN.

In the first N*N cycles, two input elements, one from the $\underline{A}$ matrix and one from the $\underline{B}$ vector, will be fetched from the memory modules and fed into the pipeline multiplier. At the c-th cycle, $1 \leq c \leq N^2$, elements a_{ij} and b_k are fetched, where $0 \leq i,j,k \leq N-1$ and

$$i=K_a\lfloor(c-1)/K_aN\rfloor+\lfloor((c-1)\text{mod}(K_aN))/N\rfloor$$

$$j=((c-1)\text{mod}(K_aN)) \text{ mod } N \qquad (17)$$

$$k=\lfloor((c-1)\text{mod}(K_aN))/K_a\rfloor$$

The input pattern of matrix $\underline{A}$ and vector $\underline{B}$ is also shown in Fig.5. The matrix $\underline{A}$ is fetched block by block. Each block is fetched in column-major and takes K_aN cycles. Corresponding to the input of one block of matrix $\underline{A}$, the vector $\underline{B}$ will provide N elements and each element will be repeatedly used K_a cycles.

The external input to the adder comes from the output of the multiplier. The first input to the adder occurs when $c=K_m+1$. The adder will produce K_a outputs, which are inputs of the comparator, every K_aN cycles. The comparator will be fed with K_a consecutive elements from the output of the pipeline adder and will be fed with dummy input for the next $K_a(N-1)$ cycles. This process will be repeated n times. The $\underline{C}$ vector will be merged into K_c groups after $K_m+N^2+K_a$ cycles. If the AR method is used, $h(K_c)$ cycles are needed to merge these K_c groups into one group. K_c more cycles are needed to drain the pipeline. In total, $K_m+N^2+K_a+h(K_c)+K_c$ cycles are required to obtain the final result. When N is very large, the total processing time is dominated by the one-pass fetch of the $\underline{A}$ matrix.

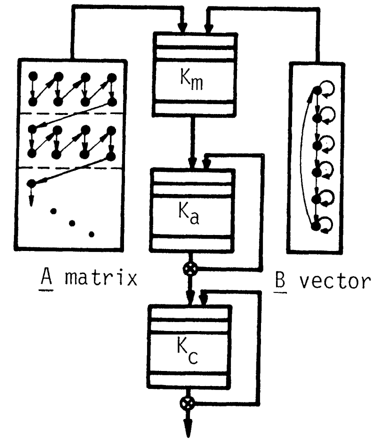


Fig.5. Three pipeline units and two memory modules are organized to perform the operation described in the example.

By partitioning the matrix inputs, scheduling the input elements, and employing more pipeline units, the above method can be extended to evaluate many other matrix operations. This method has been successfuly applied in designing a VLSI systolic architecture for the purpose of pattern clustering [13].

V. Conclusion

New scheduling methods which can efficiently evaluate the first-order recurrence system in a pipeline processor have been demonstrated. It has been shown that the asymmetric reduction method is faster while the symmetric reduction method is good for microprogrammed control. When the number of segments is an integer power of two, the symmetric reduction method behaves the same as the asymmetric reduction method. If the length of the input vector is very long, the difference of their processing times are not significant. Both of these methods are better than the conventional recursive reduction method for reducing processing time and eliminating temporary buffer.

To evaluate multiple first-order recurrence systems in a single pipeline, the pipeline utilization can be further increased by interleaving multiple vector inputs. A physical pipeline shared by many vector inputs can be viewed as having many virtual reduction processors, in which each virtual processor is dedicated to one vector reduction operation. The pipeline utilization is further increased by totally or partially eliminating the group-merging phase.

Several pipeline units can be chained together and the vector inputs can be partitioned to facilitate interleaved processing as indicated in the example shown in Section IV. The design is actually a two-level pipelined architecture. Due to the feedback loop involved, scheduling of the external inputs and connecting of different pipeline units must be carefully considered to avoid conflict. A system including multiple pipeline units and parallel memory modules can provide a more efficient systolic architecture for evaluating various matrix manipulations and is suitable for VLSI implementation.

REFERENCES

[1] Chen, S.C. and Kuck, D.J., "Time and parallel processing bounds for linear recurrence systems," IEEE Trans. on Computers, Vol.C-24, July 1975, pp.701-717.

[2] Cray Research Inc., Cray-1 Computer System Hardware Reference Manual, Pub.No.2240004, Minnesota, 1977.

[3] Hockney, R.W. and Jesshope, C.R., Parallel Computers, Adam Hilger Ltd, Bristol, 1981.

[4] Hwang, K. and Briggs, F.A., Parallel Computer Architecture, McGraw-Hill Book Co., New York (in press to appear).

[5] Hwang, K., Su, S.P. and Ni, L.M., "Vector computer architecture and processing techniques," Advances in Computers, Vol.20, M. Yovits (Ed.), Academic Press, Inc., 1981, pp.115-197.

[6] Hwang, K. and Ni, L.M., "Resource optimization of a parallel computer for multiple vector processing," IEEE Trans. on Computers, No.9, Sept. 1980, pp.831-836.

[7] Kogge, P.M. and Stone, H.S., "A parallel algorithm for the efficient solution of a general class of recurrence equations," IEEE Trans. on Computers, Vol.C-22, August 1973, pp.786-793.

[8] Kogge, P.M. The architecture of Pipelined Computers, McGraw-Hill Book Co., 1981.

[9] Kuck, D.J., The Structure of Computers and Computations, Vol.1, John Wiley & Sons, Inc., 1978.

[10] Kulkarni, A.V. and Yen, D.W.L., "Systolic processing and an implementation for signal and image processing," IEEE Trans. on Computers, Vol.C-31, October 1982, pp.1000-1009.

[11] Ni, L.M. and Hwang, K., "Performance modeling of shared-resource array processors," IEEE Trans. on Software Engineering, Vol. SE-7, July 1981, pp.386-394.

[12] Ni, L.M. and Hwang, K., "Vector reduction methods for arithmetic pipelines," Proc. of the 6th Int'l Sym. on Computer Arithmetic, Aarhus, Denmark, June 20-22, 1983.

[13] Ni, L.M. and Jain, A.K., "A systolic architecture for pattern clustering," Technical Report, Dept. of Computer Science, Michigan State University, June 1983.

THE SOLUTION OF LINEAR RECURRENCE RELATIONS ON PIPELINED PROCESSORS

W.Oed
Zentralinstitut fuer Angewandte Mathematik (ZAM)
Kernforschungsanlage Juelich GmbH
5160 Juelich - West Germany

O.Lange
Allgemeine Elektrotechnik und Datenverarbeitungssysteme
Rheinisch-Westfaelische Technische Hochschule Aachen
5100 Aachen - West Germany

Abstract Recurrence relations are frequently to be solved iteratively on a computer. Since the computation of the i-th value usually depends on the (i-1)-th value, pipelined processors cannot be used to their full potential. A transformation method for obtaining an equivalent recurrence relation not depending on the previous value together with some stability considerations concerning equivalent transformations are presented.

Introduction

Various aspects of the efficient solution of linear recurrence relations on parallel or pipelined processors have been investigated; e.g.[1]-[4]. We focus on the solution of linear recurrence relations on architectures where floating-point addition and floating-point multiplication are performed by pipelined functional units.

Our aim is to speed up recurrence relations on pipelined processors, where the speedup S is defined as the ratio of the straight forward, in a sense 'sequential' output latency l_s over any optimized output latency l_o

$$S = l_s/l_o \ . \tag{1}$$

For a pipelined processor the stage utilization indicates for each stage how often on the average that stage processes data within a regarded interval [6]. The length of the interval is the latency l_s for the sequential case, and l_o for the optimized case respectively. The stage utilization for the j-th stage in the sequential case is given by

$$U_{sj} = \frac{1}{l_s}\sum_{i=1}^{l_s} q_{ij} \leq 1 \tag{2}$$

with $q_{ij}=0$ if stage j is idle, $q_{ij}=1$ if stage j is busy at cycle i. If U_{sm} is the maximum value of U_{sj}, $j=1,2,\dots,k$, then the optimized output latency will be

$$l_o = l_s U_{sm} \ . \tag{3}$$

Linear Recurrence Relations and Pipelining

First we will investigate an m-th order linear homogeneous recurrence relation with constant coefficients of the type

$$u_i - a_{m-1}u_{i-1} - a_{m-2}u_{i-2} - \dots - a_0 u_{i-m} = 0 \tag{4}$$

with given initial values $u_0,u_1,\dots,u_{m-1}$ and real coefficients $a_{m-1},a_{m-2},\dots,a_0$.

Consider a second order recurrence relation with given initial values u_0 and u_1

$$u_i = a_1 u_{i-1} + a_0 u_{i-2} \ . \tag{5}$$

As an example, a computer is taken with one pipelined floating-point multiplier and one pipelined floating-point adder. Let k_{MUL} be the number of stages for the multiplier, and k_{ADD} the number of stages for the adder. Figure 1 depicts in a reservation table the timing for this recurrence, with $k_{MUL}=3$ and $k_{ADD}=2$; (these are for instance the characteristics of the FPS-164 processor). As can be seen, every k time-units with

$$k = k_{MUL}+k_{ADD} = l_s$$

a new u_i is computed, once the computation has been set up. Note that some overlap is taking place even in the 'sequential' case, since a_0u_{i-1} can be computed already before u_i becomes available. However, with $U_{sm}=2/5$, we do not have an optimal computation.

In an m-th order recurrence relation m multiplications and (m-1) additions are to be performed. This implies $l_o=m$ to be the optimal output latency in the case of one multiplier and one adder thus yielding a speedup of

$$S = l_s/l_o = k/m \ .$$

Equivalent Transformation of Recurrence Relations

In order to achieve the desired speedup, the recurrence relation has to be transformed in such a way that the computation of a u_i does not depend on the immediate predecessor. For improving l_s

$$l_g = l_s(1-U_{sm}) \tag{6}$$

cycles can be gained per interval by 'backstepping' as we will call it. The resulting recurrence relation should be of the type

$$v_i - c_{m-1}v_{i-\alpha} - c_{m-2}v_{i-\beta} - \dots - c_0 v_{i-\mu} = 0 \tag{7}$$

with $\alpha,\beta,\dots,\mu$ natural numbers and the property $1 \leq \alpha < \beta < \dots < \mu$, where

$$l = 1+l_b = 1+\lceil l_g/m \rceil \tag{8}$$

with $l_b=\lceil l_g/m \rceil$ the number of 'backsteps'.

The new recurrence relation is called an equivalent transformation if $u_i=v_i$ for all i. Since the transformed recurrence relation is of higher order, more initial values are needed. They are to be computed from the original recurrence relation.

One method for obtaining an equivalent transformation would be repeated substitution. Consider the second order linear recurrence relation (eq.(5)) which also can be written as

$$u_{i-1} = a_1 u_{i-2} + a_0 u_{i-3} \tag{9}$$

By substituting u_{i-1} in eq.(5) by eq.(9), u_i can be computed without depending upon u_{i-1}, which results in

$$u_{i-1} = (a_0+a_1^2)u_{i-2} + a_0 a_1 u_{i-3} \tag{10}$$

thus achieving a backstep of one. By further substitution the desired l_b backsteps may be achieved in this fashion. However this procedure is rather awkward and may cause stability problems as will be shown later.

Another method for obtaining an equivalent transformation uses some results regarding conditional recurrence sequences [7]. In a conditional

0190-3918/83/0000/0544$01.00 © 1983 IEEE

recurrence sequence the values u_i are obtained by a set of different recurrence relations, from which the recurrence relation for calculating the next u_i is chosen upon a certain condition. Consider for example a sequence with initial values u_0 and u_1 where the values are generated by the alternate application of three different recurrence relations

$$\begin{aligned} &u_i-a_1u_{i-1}-a_0u_{i-2} = 0 \quad \text{for } 0 = (i \bmod 3) \\ &u_i-a'_1u_{i-1}-a'_0u_{i-2} = 0 \quad \text{for } 1 = (i \bmod 3) \qquad (11) \\ &u_i-a''_1u_{i-1}-a''_0u_{i-2} = 0 \quad \text{for } 2 = (i \bmod 3)\ . \end{aligned}$$

A shift-operator z is defined as

$$zu_i = u_{i-1} \qquad (12)$$

and in general

$$z(z^{j-1}u_i) = z^ju_i = u_{i-j}\ .$$

With this shift operator eq.(11) can be written as

$$\begin{aligned} &(1-a_1z-a_0z^2)u_i = f(z)u_i = 0 \quad \text{for } 0=(i \bmod 3) \\ &(1-a'_1z-a'_0z^2)u_i = g(z)u_i = 0 \quad \text{for } 1=(i \bmod 3) \qquad (13) \\ &(1-a''_1z-a''_0z^2)u_i = h(z)u_i = 0 \quad \text{for } 2=(i \bmod 3) \end{aligned}$$

where $f(z),g(z),h(z)$ are polynomials applied to u_i.

In general a conditional recurrence sequence is generated by means of a set $P=\{f(z),g(z),\ldots\}$ of r m-th order polynomials and a decision function $Q(i)$ by which a polynomial is selected from P for application to u_i. In our example we have r=3 and $Q(i)=(i \bmod 3)$.

The theory describes a method which constructs from r polynomials of set P a single polynomial $F(z)$, which is a polynomial in z^r. When $F(z)$ is applied unconditionally to u_i the same recurrence sequence is generated.

The procedure is demonstrated for the polynomials of eq.(13). First $f(z),g(z),h(z)$ each are split into polynomials of powers of z^r, in our example z^3, where

$$f(z) = f_0(z^3)+zf_1(z^3)+z^2f_2(z^3)$$

with

$$f_0(z^3) = 1 \quad , \quad f_1(z^3) = -a_1 \quad , \quad f_2(z^3) = -a_0$$

for f(z) as it is given in eq.(13); g(z) and h(z) are treated correspondingly. Eq.(13) can then be written as

$$\begin{aligned} &[f_0(z^3)+zf_1(z^3)+z^2f_2(z^3)]u_{3j} = 0 \\ &[g_0(z^3)+zg_1(z^3)+z^2g_2(z^3)]u_{3j+1} = 0 \qquad (14) \\ &[h_0(z^3)+zh_1(z^3)+z^2h_2(z^3)]u_{3j+2} = 0 \end{aligned}$$

with $j=0,1,\ldots,n/3$. Using the shift operator (eq.(12)), the system (14) may then be written in matrix form

$$\begin{pmatrix} f_0(z^3) & z^3f_2(z^3) & z^3f_1(z^3) \\ g_1(z^3) & g_0(z^3) & z^3g_2(z^3) \\ h_2(z^3) & h_1(z^3) & h_0(z^3) \end{pmatrix} \begin{pmatrix} u_{3j} \\ u_{3j+1} \\ u_{3j+2} \end{pmatrix} = 0. \qquad (15)$$

It is proved in [7] that the same recurrence sequence is obtained if eq.(15) is written as

$$F(z)u_i = 0$$

with F(z) the value of the determinant

$$F(z) = \begin{vmatrix} f_0(z^3) & z^3f_2(z^3) & z^3f_1(z^3) \\ g_1(z^3) & g_0(z^3) & z^3g_2(z^3) \\ h_2(z^3) & h_1(z^3) & h_0(z^3) \end{vmatrix}. \qquad (16)$$

Multiplication of the j-th row by z^j and division of the j-th column by z^j does not change the value of the determinant (16) and finally

$$F(z) = \begin{vmatrix} f_0(z^3) & z^2f_2(z^3) & zf_1(z^3) \\ zg_1(z^3) & g_0(z^3) & z^2g_2(z^3) \\ z^2h_2(z^3) & zh_1(z^3) & h_0(z^3) \end{vmatrix}$$

is produced, which is a polynomial in z^3.

This theory also holds for the special case that all polynomials are identical, i.e. $f(z)=g(z)=h(z)$; an important notion for our purpose. Recall that in order to achieve an optimal speedup at least l_b backsteps are required. Therefore all we have to do is to replicate the polynomial $f(z)$ $l_b+1=l$ times (eq.(8)) into the set $P=\{f(z),f(z),\ldots,f(z)\}$ thus artificially creating a 'conditional' recurrence sequence. It will become again unconditional after the transformation has been carried through. The resulting polynomial $F(z^l)$ applied to the original m-th order recurrence relation (eq.(4)) will yield

$$u_i-c_{m-1}u_{i-l}-c_{m-2}u_{i-2l}-\cdots-c_0u_{i-ml} = 0\ . \qquad (17)$$

As an example take a look at eq.(5), whose sequential calculation is depicted in figure 1. With $U_{sm}=2/5$ a speedup of $S=5/2$ should be gained. With eq.(6) and eq.(8) we find $l_b=2$ and consequently $l=3$. Therefore the set $P=\{f(z),f(z),f(z)\}$ with

$$f(z) = 1-a_1z-a_0z^2$$

will be used to achieve

$$\begin{vmatrix} 1 & -z^2a_0 & -za_1 \\ -za_1 & 1 & -z^2a_0 \\ -z^2a_0 & -za_1 & 1 \end{vmatrix} = 1-c_1z^3-c_0z^6$$

with $c_1=3a_0a_1+a_1^3$ and $c_0=a_0^3$. The equivalent recurrence relation to (10) will then be

$$u_i = c_1u_{i-3}+c_0u_{i-6}\ .$$

The optimized case is illustrated in figure 2 with an optimal output latency of $l_o=2$.

Some Stability Considerations

Two criteria for stability of linear homogeneous recurrence relations are [8]:

- absolute stability, if for all roots t_i of the characteristic polynomial

$$|t_i| < 1 \quad \text{for } i = 1,2,\ldots,m$$

- relative stability, if

$$t_d = t_{max}$$

with

$$t_{max} = \max\{|t_i|\} \quad \text{for } i = 1,2,\ldots,m$$

the root with maximal absolute value and

$$t_d = \max\{|t_i| \ ,\ c_i \neq 0\} \quad \text{for } i = 1,2,\ldots,m$$

the dominant root, i.e. the maximal absolute root with a coefficient $c_i \neq 0$ in the general solution.

We will assume that the original recurrence relation is stable. However by its transformation instability might be introduced, because the transformed relation is of higher order with additional roots in the general solution.

An example will demonstrate what might happen by using the substitution procedure (eq.(10)), and it also will be shown that the transformation described above does not introduce instability.

Consider eq.(5) with $a_1=7/6$ and $a_0=-1/3$, which is absolutely stable, since the characteristic polynomial

$$t^2 - \frac{7}{6}t + \frac{1}{3} = (t - \frac{1}{2})(t - \frac{2}{3})$$

has $t_{max}=2/3$ as the maximal root. By the substitution procedure the recurrence relation will become

$$u_i = \frac{-37}{36}u_{i-2} - \frac{7}{18}u_{i-3}$$

with the characteristic polynomial

$$t^3 - \frac{37}{36}t - \frac{7}{18} = (t - \frac{1}{2})(t - \frac{2}{3})(t + \frac{7}{6})$$

which no longer is absolutely stable, since $t_{max}=7/6>1$ does not fulfil the stability criterion. Note that the dominant root is $t_d=2/3$ since the same sequence is to be generated.

Applying the transformation as discussed above, any m-th order linear recurrence relation with the characteristic polynomial

$$\begin{aligned} &t^m-a_{m-1}t^{m-1}-a_{m-2}t^{m-2}-\ldots-a_0 \\ &= (t-t_1)(t-t_2)(t-t_3)\cdot\cdot(t-t_m) \qquad (18) \end{aligned}$$

will be transformed to a recurrence relation of the type (17), with the characteristic polynomial

$$t^{ml}-c_{m-1}t^{(m-1)l}-c_{m-2}t^{(m-2)l}-\ldots-c_0 \ . \qquad (19)$$

Substituting t^l by y the polynomial (19) can be written as

$$\begin{aligned} &y^m-c_{m-1}y^{m-1}-c_{m-2}y^{m-2}-\ldots-c_0 \\ &= (y-y_1)(y-y_2)\ldots(y-y_m) \\ &= (t^l-y_1)(t^l-y_2)\ldots(t^l-y_m) \\ &= (t-t_{11})(t-t_{12})\ldots(t-t_{1l})(t-t_{21})\ldots(t-t_{ml}). \qquad (20) \end{aligned}$$

For each of the roots y_r, $r=1,2,\ldots,m$ we obtain by factorization

$$t_{rj} = \sqrt[l]{|y_r|}e^{i\frac{2\pi j+\omega r}{l}} \qquad (y_r = |y_r|e^{i\omega r}) \qquad (21)$$

with $j=1,2,\ldots,l$. Since the new recurrence relation is equivalent to the original recurrence relation, the roots $t_1,t_2,\ldots,t_m$ of the original polynomial (eq.(18)) are included in the set of roots $t_{11},t_{12},\ldots,t_{ml}$ and the l different roots t_{rj} for each r differ only in the angle, while the absolute values are the same. Therefore it is assured that the equivalent transformation of a recurrence relation as described above, does not change the stability behaviour of the original recurrence relation.

Conclusion

Since the speedup potential of a recurrence relation can be predicted for a given machine architecture, the transformation presented can be performed automatically. In particular low order recurrence relations, which are the most common ones can be speeded up considerably.

Acknowledgements We would like to thank Dr.F.Hossfeld, W.Meyer, Dr.R.von Seggern, and Dr.B.Steffen for valuable hints and discussions.

References

[1] Kogge P.M., Stone H.S.
"A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations" IEEE Trans. on Comp. vol.C-22 no.8 Aug.1973 pp786-792

[2] Kogge P.M.
"Maximal Rate Pipelined Solutions to Recurrence Problems" Proc. 1st Ann.Conf. on Comp.Arch. Dec.1973 pp71-80

[3] Chen S.C., Kuck D.J.
"Time and Parallel Processor Bounds for Linear Recurrence Systems" IEEE Trans. on Comp. vol.C-24 no.7 July 1975 pp701-717

[4] Hyafil L., Kung H.T.
"The Complexity of Parallel Evaluation of Linear Recurrences"
J.ACM vol.24 no.3 July 1977 pp513-521

[5] Kuck D.J.
"The Structures of Computers and Computations - vol.1" John Wiley&Sons 1978

[6] Kogge P.M.
"The Architecture of Pipelined Computers"
McGraw-Hill 1981

[7] Simons J.L.
"Conditional Recurring Sequences"
PhD thesis, Technical University Delft 1976

[8] Cash J.R.
"Stable Recursions"
Academic Press 1979

Figures

$A_0 \cdot U_0$	$A_1 \cdot U_1$				$A_0 \cdot U_1$	$A_1 \cdot U_2$				$A_0 \cdot U_2$	$A_1 \cdot U_3$
	$A_0 \cdot U_0$	$A_1 \cdot U_1$				$A_0 \cdot U_1$	$A_1 \cdot U_2$				$A_0 \cdot U_2$
		$A_0 \cdot U_0$	$A_1 \cdot U_1$				$A_0 \cdot U_1$	$A_1 \cdot U_2$			
				$A_0U_0+A_1U_1$					$A_0U_1+A_1U_2$		
					$A_0U_0+A_1U_1$					$A_0U_1+A_1U_2$	
						U_2					U_3

figure 1. sequential computation of a second order linear recurrence

$C_0 \cdot U_0$	$C_1 \cdot U_3$	$C_0 \cdot U_1$	$C_1 \cdot U_4$	$C_0 \cdot U_2$	$C_1 \cdot U_5$	$C_0 \cdot U_3$	$C_1 \cdot U_6$	$C_0 \cdot U_4$	$C_1 \cdot U_7$	$C_0 \cdot U_5$	$C_1 \cdot U_8$
	$C_0 \cdot U_0$	$C_1 \cdot U_3$	$C_0 \cdot U_1$	$C_1 \cdot U_4$	$C_0 \cdot U_2$	$C_1 \cdot U_5$	$C_0 \cdot U_3$	$C_1 \cdot U_6$	$C_0 \cdot U_4$	$C_1 \cdot U_7$	$C_0 \cdot U_5$
		$C_0 \cdot U_0$	$C_1 \cdot U_3$	$C_0 \cdot U_1$	$C_1 \cdot U_4$	$C_0 \cdot U_2$	$C_1 \cdot U_5$	$C_0 \cdot U_3$	$C_1 \cdot U_6$	$C_0 \cdot U_4$	$C_1 \cdot U_7$
				$C_0U_0+C_1U_3$		$C_0U_1+C_1U_4$		$C_0U_2+C_1U_5$		$C_0U_3+C_1U_6$	
					$C_0U_0+C_1U_3$		$C_0U_1+C_1U_4$		$C_0U_2+C_1U_5$		$C_0U_3+C_1U_6$
						U_6		U_7		U_8	

figure 2. overlapped computation of a second order linear recurrence

DATA-STATIONARY INSTRUCTIONS AS A WAY TO MINIMIZE LONG DISTANCE COMMUNICATIONS IN VLSI

John Robert Burger*
School of Engineering
University of the Pacific
Stockton, California 95211

Abstract

Long lines on a chip cause problems. There are unavoidable performance degradations. For example, capacitance either causes delays, or necessitates space and power consuming drivers. Moreover, inexpensive chips are essentially 2-dimensional, so there simply is no space for long control lines, especially if they must overlap. One alternative is to allow instructions to move in step with the operands. This necessitates a special system controller, and extra instruction buffers. The advantage is high throughput for programs which can be reduced to straightforward procedures. Floating point addition, e.g., is a procedure which requires up to 2N segments in a pipeline, N being the number of bits in the fractional part. In comparison, signed floating point multication (or division) uses only about N segments. CORDIC, the most complex routine considered here, needs roughly 16N segments. The execution of data-stationary instructions has been simulated at the bit level using Fortran. Such simulations help establish the necessary size and shape of the chip surface to execute given instructions, and provide detail for design. Overall, the computer architecture presented below should be relatively easy to implement.

I. Introduction

Data-stationary instructions in a pipeline have the major advantage of being natural to VLSI, which needs maximum local communications [1,2]. Architectures based on fan out to parallel processors usually cannot be used if the components have to be hardwired. Hardwiring implies the use of 3-dimensions, while chips are largely 2-dimensional. Examples of 3-dimensional systems, and other types of parallel processors may be found in a recent excellent survey [3].

It is awkward to program a pipeline for certain instructions, e.g., conditional branches. Never-the-less, general purpose programming clearly is possible if certain design features are used [4]. The features include universal programmable segments, variable numbers of segments depending on the instruction, and the possibility of (non-local) communication between segments and main memory. Kogge focused on data stationary instructions, showing that they are easier to microprogram, and to optimize, that complex time-stationary programs become shorter, that pipelines naturally fill and empty [5,6].

A new computer system based on data-stationary instructions hopefully would be able to run conventional code; this prevents existing software from being outdated. Moreover, the resultant system has to be efficient in array processing, this being a bottleneck in present inexpensive computers.

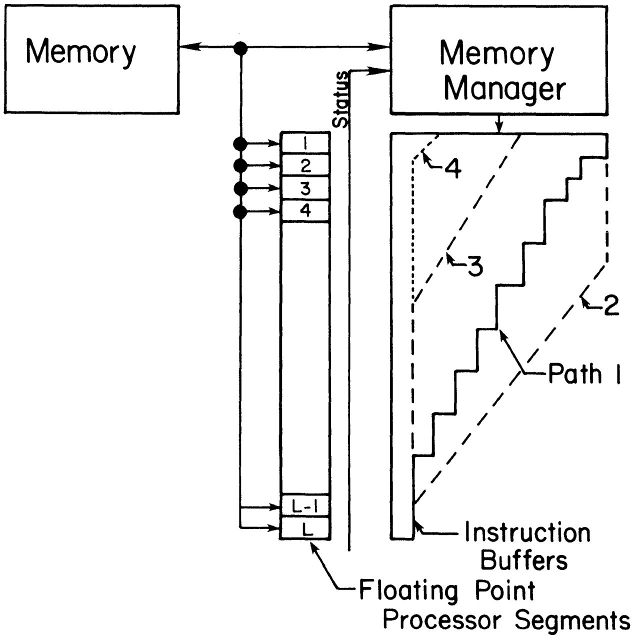


Figure 1.
An Architecture for Data-stationary Instructions

II. System Considerations

Figure 1 shows a typical design. Instruction codes shift left as they execute, and flow down along with the operands in the processor segments. Each instruction is allowed to use a variable number of processor segments to complete execution; data in the pipeline is not latched until each segment is ready. Preliminary questions to be answered are, what is a convenient value for L, the number of processor segments, and how many words of memory are needed for the instruction buffers? The shape of the necessary area for instruction buffers depends on the type of instructions being executed. Path 1 in Figure 1 is the endpoint of a row of instructions which execute at roughly a uniform rate, instructions such as add, subtract, multiply, and divide. Array operations such as inner product also use a buffer area roughly the shape of an upsidedown right triangle. If the number of words in the base of the triangle is specified as W, the instruction buffers use up to about WL/2

*This work is sponsored in part by an engineering research initiation grant from the Engineering Foundation and the I.E.E.E.

0190-3918/83/0000/0547$01.00 © 1983 IEEE

words.

Longer procedures such as CORDIC prevent a uniform flow of instructions: CORDIC followed by shorter instructions may cause the last instruction in a row to follow Path 2. Path 2 breaks as soon as the CORDIC instruction is done. Shorter instructions followed by CORDIC may cause the CORDIC instruction to follow Path 3, or Path 4. A stream of instructions which follow Path 4 would free the vast majority of buffers.

The Memory Manager (MM) in Figure 1 must keep track of the types of instructions which are loaded. Overflow beyond L segments should be prevented; otherwise recycling will be necessary. Instruction recycling is undesirable, since unlucky instructions may be waiting for operands from main memory which must be generated by the recycled instructions. Note that each processor segment is connected to main memory via a bus.

Reading or writing in the system of Figure 1 may involve timing difficulties when sequential code is being run. Instructions which must not execute ahead of time are: a) a system must not load from a register in memory which has not yet received the proper data due to an unexecuted store to the register. b) a system must not store to a register whose data has not yet been used due to an unexecuted load from the register. Clearly, the nature of the code is important, since it may avoid using the same addresses for temporary storage. Data flow concepts apparently reduce the need for addressable memory; but languages and compilers are needed to support the data flow concept [7].

One function of the MM is to check instructions and addresses for those which must not be loaded into the instruction buffers due to poor timing. The MM may instead insert NO OPS until the next instruction can be loaded, or it may interrupt the program and run another. An interrupt involves storing recovery information in a stack. Interrupts are efficient when many structured subprograms are waiting to be run. It may happen that the programming is such that mainly the first processor in the pipeline is active, the others being idle. This is, in effect, an automatic reversion to a single processor computer, something which is considered acceptable by the author. As long as arrays run efficiently, a typical program would execute at a fairly steady rate from the user's point of view. Again we stress that faster execution occurs via better coding.

The M M also identifies and regulates branches, e.g., the IF ( ) THEN ( ) ELSE ( ). There are 2 options: (a) interrupt, running something else until the condition which determines branch direction is calculated. (b) run as much code as possible in both branches, deleting the unneeded results after the branch condition is calculated. Code can minimize conditional branches, e.g., by using fixed numbers of iterations in loops. The number of instructions in a loop may be used to determine the method of implementation. Loops with only a few instructions and a fixed number of iterations may be loaded by the MM as one long sequence of code. Longer loops can also be straightened for the pipeline, but may need a way to exit under given conditions. The MM can detect the condition and stop feeding replicated code. Hence, a LOOP UNTIL ( ) is possible using the STATUS line in Figure 1.

III. The Matrix Processing Feature

Matrix processing is implemented in a quite ordinary way; vector operands and partial products move in the same direction; array operands go diagonally. In comparison, Kung's systolic processing may have operands and partial products going to opposite directions; array operands move in at right angles [1, 8]. The instruction buffers may be loaded with data, or addresses to be processed in parallel. The options are: A) data can be put immediately into the buffers; B) addresses can be put into the buffers in any order for direct or extended addressing; C) the addresses can be stepped one unit at a time from an index placed in the buffer. A useful feature used in array processing is the End of Data (EOD) mark in the data stream to bring the system out of the array processing mode.

Figure 2 illustrates the data flow for a simple inner product (c = ab).

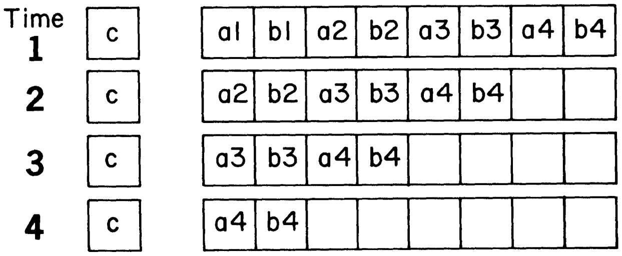


Figure 2.
The Data Flow for an Inner Product (c=ab)

Each block for c represents roughly N segments, within which it is possible to execute a floating point multiply-add operation. Notice that processor usage is complete, and that the data buffer space really has a triangular shape. Values are shifted left and down. The operands, e.g., a_1, b_1, must be stacked together as shown.

In a 1-dimensional pipeline, Matrix-vector multiplication (c = Ab) involves keeping the b vector in the first row until the matrix is fully loaded (see Figure 3). Although processor usage is complete, portions of b are repeated in the buffer space. The important thing is that a version of array processing indeed meshes with data-stationary instruction processing. If 2-dimensional pipelines were allowed, 1 layer would suffice to load a matrix-matrix multiplication (C=AB), as illustrated in Figure 4.

A_{11}, for example, is transmitted to the right column of C, i.e., C_{11}, C_{12}, C_{13}, and C_{14}. B_{11} is transmitted to the first row of C, i.e., C_{11}, C_{21}, C_{31} and C_{41}. The 4 x 4 matrix-matrix multiplication is completed after 4 time frames (not all shown). Note in the figure that arrays are in

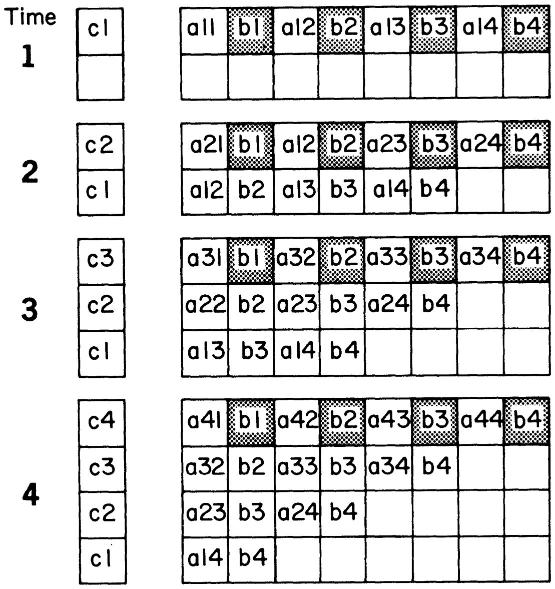


Figure 3.
The Data Flow For a Matrix-Vector Multiplication (c = Ab)

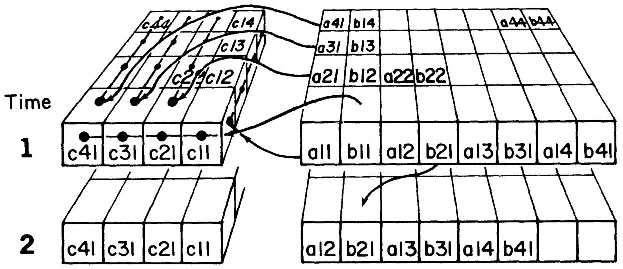


Figure 4.
Matrix-Matrix Multiplication (C = AB) in a 2-dimensional Pipeline

standard form when viewed from the bottom. The processors are used completely. Each layer does a matrix-matrix multiplication, so a sequence of layers could do certain tensor operations. The problem is that today's VLSI is mostly 2-dimensional. When 3-dimensional technologies become popular, 3-dimensional pipelines may conveniently perform sophisticated iterations, recursions, and picture reconstructions. Meanwhile, this paper will be concerned with only 2-dimensions.

IV. The Design of a Segment

A standard method starts with a list of desired machine instructions, and ends with a timed logic circuit [9]. Standard methods are systematic, but iterative. Especially interesting is the microprogramming for data-stationary instructions; the microprogram must proceed in space and time from segment to segment along with the instructions. Each segment is self-timed, and is organized for floating point (FLP) arithmetic as is usually required in a general computer.

The Control Logic

Unclocked combinational logic is located between the latches of the pipeline as in Figure 5.

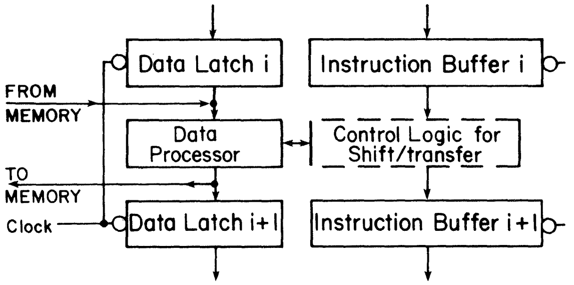


Figure 5.
The General Plan for a Processing Segment

Figure 6 shows the overall structure for the circuitry between the instruction buffers.This circuitry serves to shift, and to transfer instructions or numerical data. An instruction which has been shifted into the leftmost part of the buffer will activate control lines and begin to execute. The signals in the control lines are sequenced by counts stored in the registers Seq^0, seq^1, Seq^2. These counters can be incremented (or set to 0) by INC0, INC1, and INC2. The control circuits are nested finite state machines. Future counts occur in subsequent segments. The PLAs decode the instruction and the count; microprograms in the PLAs must be replicated for each segment. The Address Bus Control places addresses on the address bus to memory (dashed line) for reading or writing data. The address field associated with an instruction is assumed to vary from 0 to 8 bytes; the instruction itself is allowed 1 byte. The Shift Left block in Figure 6 may need to shift up to 9 bytes. There is an option in the Shift Left block to shift addresses or data, but not the instruction. This option is used for inner products.

The MM prevents reading and writing which would result in numerical error in sequential code, as mentioned above. Permissible load and store commands may have to wait a short time to use the bus. The pipeline is held back until each load and store is accomplished; an inhibit line is included in the data bus.

Load/Save logic in each segment generates read and write signals, and helps manage the bus traffic. Outputs waiting to be saved are serviced before input data is read, since outputs depend on the values in the input latches. The writes to memory are done one at a time, each processor having a status which the central memory manager can poll.

Inputs to be read from memory may also have to wait a short time to use the bus. Reads can be

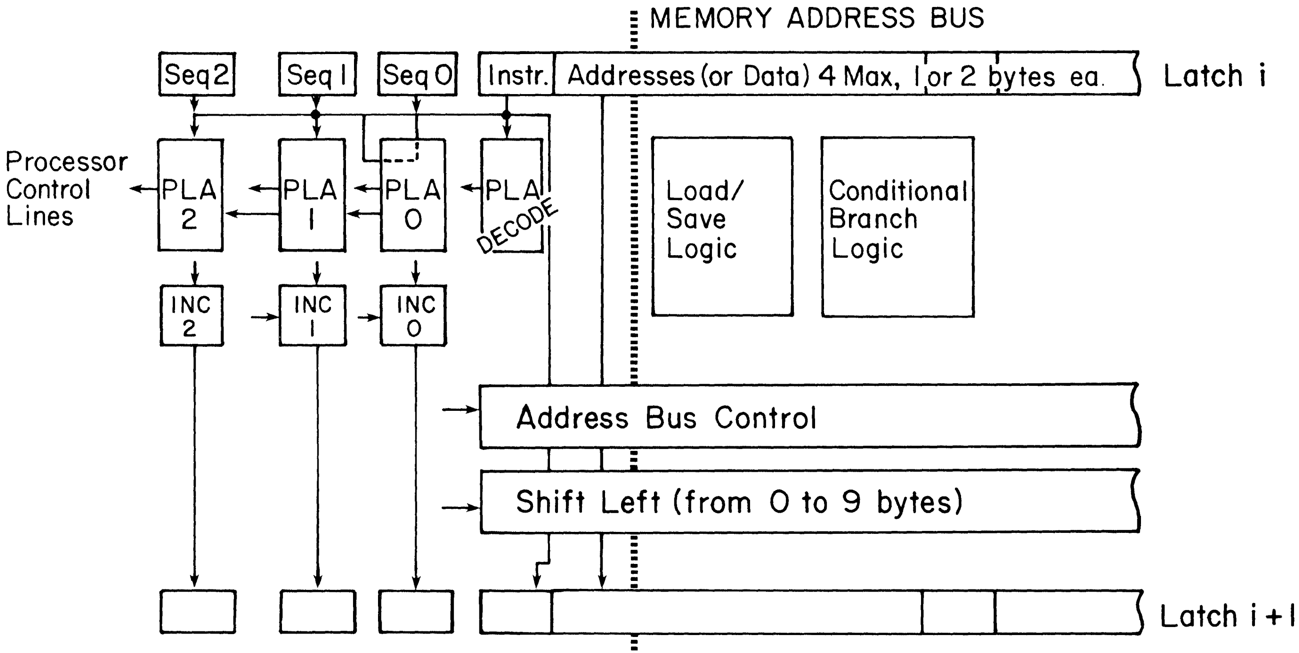


Figure 6. The Control Section of a Segment

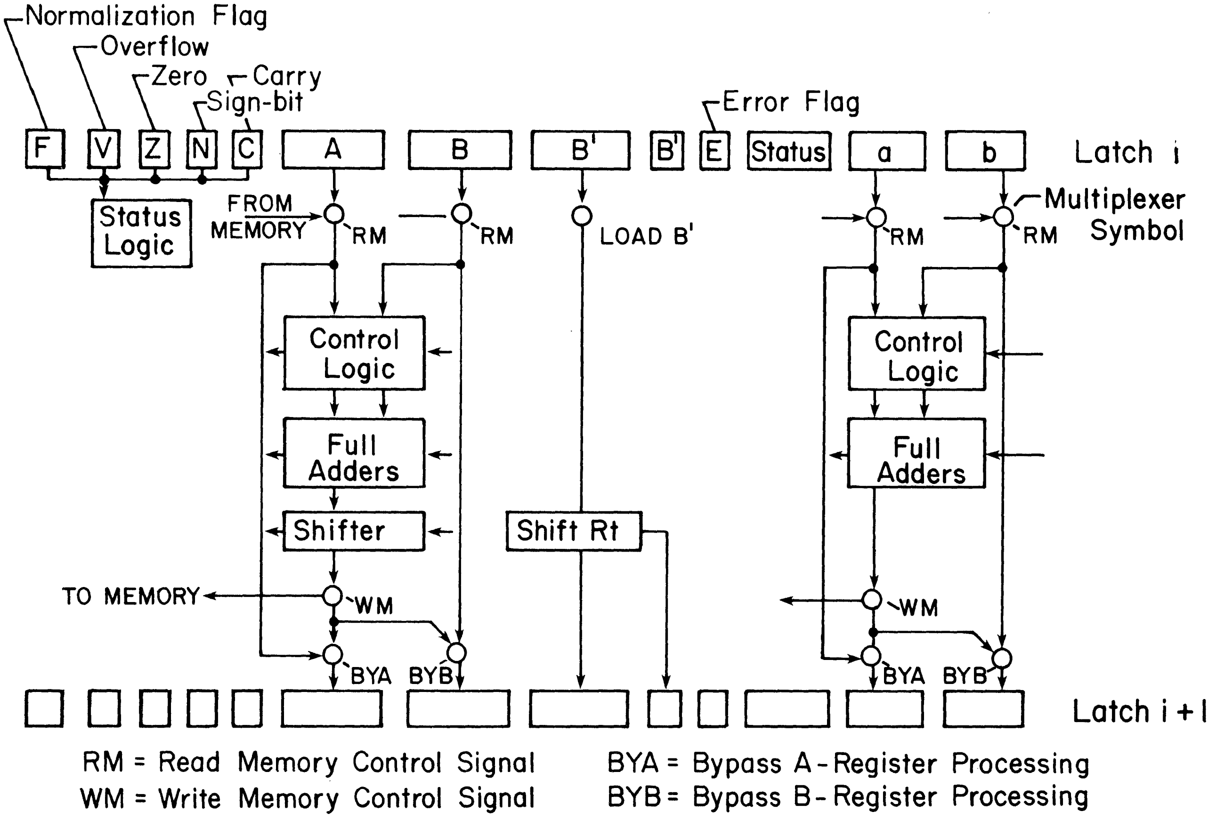


Figure 7. The Processing Section of a Segment

implemented by controlling the clock to the segment. Segments with a load command usually would latch the input data which is read. Microcode has to allow for this method of reading memory. The reads are done one at a time, after the writes to memory. The bus management is fairly standard. The Conditional Branch Logic causes startup addresses to be moved from the interrupt stack kept by the MM.

The Floating Point Processors

The FLP processor segments have to be simple adders with a few gates to direct data. The processors should be simpler than most currently available microprocessors since there is a need to save space. Algorithms which use adding and shifting are easily pipelined [10 - 15]. Iterations are accomplished in successive segments rather than by looping. Figure 7 shows a typical design. Fewer components might give perhaps only 1 gate delay per segment for higher throughput, but would increase the complexity of the microprogramming. Referring to Figure 7, the fractional parts of numbers are kept in latches A and B in signed 2s complement form; exponentials of the same form are in a and b. B' is for storage of operands, counts, and constants; b_0' receives the rightmost bit in B_0' after a shift right. Various status bits are defined in the figure.

The Microprogramming

The machine instructions may be broken into microinstructions complete with sequencing information. Unfortunately, the microprograms cannot be published here due to limited space. The overall features are at least summarized: The Floating Point Addition (FLP ADD) takes 7 lines of microprogramming (not shown). The align exponents section may take up to N segments; likewise for postnormalization, where N is the precision. The total is about 2N segments for FLP ADD. Further information about floating point systems may be found in Reference 10, Chapter 9.

FLP SUB is very similar; FLP MUL based on Booth's pairs needs 6 lines of microprogramming (not shown). It needs about N segments. FLP DIV based on a modification of Guild's method needs 8 lines for signed division. About N segments are required.

FLP CORDIC

The nested finite state machines permit several methods for function evaluation, but CORDIC is most interesting as an example [15]. It evaluates more than one function at once, if desired; and it may be used for plane matrix rotations [16]. To simplify, the easily implemented pre-normalizations and post-normalizations are omitted from this discussion. CORDIC needs 9 lines of microprogramming, and also parts of the FLP ADD routine. Each addition is assumed to take 1 segment to prenormalize, 1 to add (or subtract), and 1 to postnormalize. N bits of precision are generated in N mathematical iterations; it is estimated that 16N segments are needed in a CORDIC evaluation.

FLP IP (Inner Product)

FLP IP is little more than FLP MUL; register B,b accumulates the inner product. The ITERO command keeps adding partial products until an EOD mark in the data stream is detected by circuitry. The DONE is then enabled, feeding the next instruction, with the option to store the inner product in memory. Three lines of microprogramming are needed, and also parts of the FLP MUL routine.

V. Fortran Simulation

Ordinary Fortran using integer arrays filled with ones and zeros is useful for digital simulations. The author's simulations used a 'PLA' subroutine to set control lines, and a main 'SEGSIM' program to call for various register transfers, e.g., APB, which adds B to A in binary. The reasoning behind the above microprogramming was thus verified. Simulations at the bit level bring one closer to the necessary detail for VLSI design where trial and error methods can be extremely expensive.

VI. Summary and Conclusions

A designer has several particular trade·offs: 1) Only one bus is proposed to conserve space and to minimize non-local communications within the chip. An alternative is to load all addresses and data into the head of the pipeline, and to receive information only from the foot, where results are written to memory. This would eliminate non-local communications and may be a good thing in some applications. It is not clear that conventional sequential code would run well without a bus, particularly when conditional branching is necessary. 2) Conditional branches are handled by waiting until the branch condition is decided before loading more code. The option is to load and run as much code as possible, selecting the desired results after the branch condition is calculated. Running more than one branch direction would complicate the Conditional Branch logic. 3) Feedback in the pipeline is avoided to reduce non-local connections on the chip. Simple accumulators and more complex pipelines with feedback are not used in this architecture, although feedback has certain advantages [5]. 4) Loops are implemented by loading the code within the loop repeatedly as it is being run. Loops are straightened; true loops are not used in this design. The option is to use feedback and microprogram loops; unfortunately, the pipeline would then be stopped until individual loops are satisfied. 5) CORDIC is proposed without the help of extra local registers; this simplifies the circuitry and the microprogramming, but function evaluation depends heavily on the memory bus. If CORDIC had to be called frequently, local registers for the 3 variables, and for the constants would be well worthwhile.

The architecture presented for data-stationary instructions minimizes long distance communications on a chip; the approach is reasonably programmable. Assuming a modest floating point system with a precision (N) equal to 8, a pipeline with L = 128 segments should be quite useful. VLSI with hundreds of thousands of gates could support the pipelined structure; there would even be space for a small memory. Interfacing serially, 1 word per clock pulse, is within reason, allowing video to be processed for T.V., for example. Currently, a data-stationary architecture appears to be a good compromise for VLSI, plus it is relatively easy to implement.

Acknowledgments

The helpful comments of the reviewers are gratefully acknowledged.

References

1. C.Mead and L. Conway, Introduction to VLSI Systems. Reading, Mass. Addison-Wesley, 1980.
2. D.G. Fairbairn, "VLSI: A New Frontier for Systems Designers," IEEE Computer, Vol. 15, pp 87-96, Jan. 1982.
3. L.S. Haynes, R.L. Law, D.P. Siewiorek, and D.W. Mizell, "A Survey of Highly Parallel Computing," IEEE Computer, Vol. 15, pp 9-24, Jan. 1982.
4. C.R. Vick, S.P. Kartashev, and S.I. Kartashev, "Adaptable Architecture for Supersystems," IEEE Computer, Vol. 13, pp 17-25, Nov. 1980.
5. P.M. Kogge, "The Microprogramming of Pipelined Processors," Symposium on Computer Architecture Proceedings, Fourth Annual, IEEE/ACM, pp 63 - 69, March, 1977.
6. P.M. Kogge, The Architecture of Pipelined Computers. N.Y., N.Y. McGraw-Hill, 1981.
7. D.D. Gajski, D.A. Padua, D.J. Kuch, and R.H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," IEEE Computer, Vol. 15, pp 58 - 69, Feb. 1982.
8. H.T. Kung, "Why Systolic Architectures," IEEE Computer, Vol. 15, pp 37-46, Jan. 1982.
9. M.M. Mano, Digital Logic and Computer Design. N.J.: Prentice-Hall, 1979.
10. K. Hwang, Computer Arithmetic: Principles, Architecture, and Design. N.J.: J. Wiley, 1979.
11. A.K. Kamal, et al., "A Generalized Pipeline Array," IEEE Trans. Comput., Vol. C-23, pp 533-536, May 1974.
12. A. D. Booth, "A Signed Binary Multiplication Technique," Quart. Journ. Mech. and Applied Math., Vol. IV, Pt. 2, pp 236 - 240, 1951.
13. H.H. Guild, "Some Cellular Logic Arrays for Nonrestoring Binary Division," The Radio and Elec. Engr., Vol. 39, pp 345 - 348, June 1970.
14. E.E. Swartzlander, Jr., "Arithmetic for Ultra-high-speed Tomography," IEEE Trans. Comput., Vol. C-29, pp 341 - 353, May 1980.
15. J.S. Walther, "A Unified Algorithm for Elementary Functions," Spring Joint Computer Conference, 1971, pp 379 - 385.
16. H.M. Ahmed, J. Delosme, and M. Morb, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," IEEE Computer Magazine, Vol. 15, pp 65 - 82, Jan. 1982.