1979 CONFERENCE ON PARALLEL PROCESSING Oscar N. Garcia 79CH1433-2C

PROCEEDINGS

OF THE

1979 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

OSCAR N. GARCIA
Editor

Cosponsored by the

Wayne State University

and the

IEEE Computer Society

IEEE Catalog No. 79CH1433-2C

Ram Gupta

PROCEEDINGS OF THE 1979 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

OSCAR N. GARCIA
University of South Florida
Editor

Papers presented on
August 21-24, 1979

Co-Sponsored by

Department of Electrical and Computer Engineering
WAYNE STATE UNIVERSITY
Detroit, Michigan

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

Additional copies are available from:
IEEE Computer Society
5855 Naples Plaza, Suite 301
Long Beach, CA 90808

or

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

IEEE Catalog No. 79CH 1433-2C
Library of Congress No. 79-83622

PREFACE

The 1979 International Conference on Parallel Processing is the eighth of a series of annual meetings initiated in 1972 at Sagamore, N. Y. A tradition has developed characterizing this conference: the papers presented are heavily oriented towards research topics but with a very pragmatic flavor. Also, the remoteness of the meeting location and the informal atmosphere have fostered the spirit of exchange among participants.

We hope to encourage the continuation and expansion of this excellent tradition. This year we received a total of 93 papers of which 23 came from 10 countries other than the U. S. despite the increasing number of emerging conferences on closely related topics. In an effort to better serve the participants we have planned to have these Proceedings printed and available at the time of the conference. You will not, therefore, find the awards for most original paper and best presentation announced in the following pages, but the selection will take place at the conference as usual. For this same reason session chairmen are not acknowledged at this time. The new publication schedule has unquestionably imposed tighter time constraints than ever before on authors and reviewers alike. To them should go the credit of the accomplishment and my heartfelt appreciation for their willing cooperation.

The present program reflects some new and broader trends towards concurrent computing. It is illuminating to analyze the changing interests of the authors. My assessment is that the emphasis has shifted from the hardware organization features to more conceptual language translator and algorithmic topics. More attention is given to the synchronization and control issues in the languages and models of parallel architectures as noticed in the first two sessions. I consider this a very natural and healthy development. Also, the topics of searching and reconfigurable systems showed more strength than in the past when compared to better established subjects such as performance evaluation, parallel arithmetic and pipelining. The session on networks for interconnection promises to be one of the strongest ever; some fascinating concepts are presented on array processors and there are novel results on special purpose multiprocessor architectures.

Tse Feng comes first in my list of acknowledgments on the organization of this conference as General Chairman, particularly when I conside his many other present commitments. He and his assistants handled the arrangements, publicity and tentative program printing and distribution. Next Annette Krygiel has planned a panel sessior in which practitioners will focus on the most ur gent problems facing parallel computing. A form program committee was not appointed and I believ this contributed to both the spontaneity and heterogeneous nature of the papers received. It did make the task of the Program Chairman more involved and I would recommend such a committee for the future on that basis. The difficulties were ameliorated by a number of reviewers who contributed well beyond the call of duty in a variety of circumstances. From the list on 157 reviewers recognized later in these Proceedings, I would like to particularly thank Tilak Agerwal Jean-Loup Baer, Bruce Berra, Dave Davis, Mario Gonzalez, Robert Keller, Willis King, Jack Lipovski, Mike Liu, Nancy McDonald, Ken Thurber, Kishor Trivedi, and Dave VanVoorhis, among other in that category. Also Mariagiovanna Sami and Chris Vissers publicized enthusiastically our call for papers in Europe and to them we express our gratitude.

The presentation by our keynote speaker, Dr Paul Schneck which is appropriately the only unrefereed contribution to these Proceedings, sets the pace for the papers that follow. We appreciate his willingness to accomodate to our schedule and his sharing his views with us.

Last but foremost, I want to acknowledge thc assistance of the Secretary of our Computer Science Program at the University of South Florida, Mrs. Brenda Malowney, who is responsible for the organization, addressing and copies of more than 600 pieces of mail involving authors, reviewers and other participants.

We hope that the labors of those involved may be repaid by your enjoyment of the conference.

[illegible]
Oscar N. García
Program Chairman

Tampa, Florida

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT'D)

TABLE OF CONTENTS (CONT'D)

TABLE OF CONTENTS (CONT'D)

ISSUES IN PARALLEL COMPUTING: A NON-EUCLIDEAN EXAMINATION

Paul B. Schneck
Office of the Director
NASA/Goddard Space Flight Center
Greenbelt, MD 20771

Abstract -- In this talk I will review some of the identifiable milestones of the evolution of computers toward what we currently term parallel systems. A significant observation is that parallelism has been present in computing systems from the beginning, but that programmers did not have to deal with it until recently. One might even conclude that the difficulties currently associated with parallel computers originate with software and programming and not exclusively with the innovative architectures of those machines. We will discuss this issue and explore potential approaches to a solution.

Introduction

I am going to to identify some of the broader underlying issues relating to parallel processing. Only after we have identified the appropriate issues can we begin to make substantial progress toward their resolution.

As a starting point, let us note that 55 papers will be presented this year, an increase of more than 25 percent over the 42 papers presented last year. The magnitude of our parallel processing vector has clearly grown. What about its orientation? To understand the orientation of the conference we will classify the papers (somewhat arbitrarily) into one of three possible subject areas: hardware, software, and algorithms.

We begin by briefly looking at the group of papers classified as hardware related. The thrust in this area has shifted somewhat from geometric considerations to those dealing with concurrency. The lock-step model for parallel processing has begun to fade. This year there are sessions devoted to synchronization and to serial-by-bit arithmetic. What is the underlying theme?

In the area of software, the number of papers has declined, even as the conference has grown. Where this stems from a reduction to practice of state-of-the-art software, the field has grown. But where this is a result of our inability to cope with facing the problems of parallel computing, we may be in trouble. The traditional software work in languages continues. In fact, there is already talk of standardizing the language for array processing. The decreased emphasis in the software area may result from a lack of either "push" or "pull." A "pull" comes about in response to requirements for improvements which can be brought about by software. Surely there is no lack here. A "push" develops when new algorithms are available for implementation. It appears that this is the bottleneck.

If there has been a bottleneck due to the lack of availability of algorithms for parallel processing, then this conference brings reason to expect an end to that situation. The growth from last year to this year occurs in the area of algorithms.

There seems to be a recognition that the mere existence of a problem's solution, perhaps demonstrated by software, does not result in a practical application of parallel processing. If this is the case, then we may look forward to renewed vigorous efforts in software, based on a sound algorithmic foundation.

Parallel Computer Development: The Process

In this section we will examine the process leading to the development of parallel computers with an eye toward overcoming any apparent deficiencies.

The Traditional Approach. The traditional approach to development of a parallel processing capability is shown in figure 1. We note that the engineering inspiration which underlies all of the following activities may be decoupled from the discipline activities for which the system will be used.

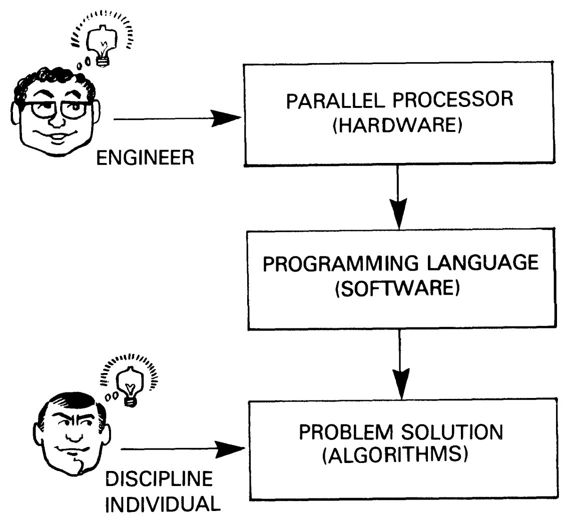

PARALLEL PROCESSOR (HARDWARE)
ENGINEER
PROGRAMMING LANGUAGE (SOFTWARE)
PROBLEM SOLUTION (ALGORITHMS)
DISCIPLINE INDIVIDUAL

FIG. 1. THE TRADITIONAL APPROACH TO PARALLEL PROCESSING

What does the engineer use as a basis for his design? While there is no definitive answer to this question, there does seem to be a reasonable response: Lacking other inputs, the engineer attempts to optimize the performance of the computer (and here we are referring only to the mainframe) in terms of potential results per unit of hardware. It is this phenomenon that was behind the slow acceptance of floating point and its eventual incorporation into hardware. After all, the existence of separate execution facilities capable of handling both fixed point arithmetic means that when one facility is in use, the other is idle--perhaps a convenience, but not an optimization. The quest for component utilization is so deeply embedded that the requirement for general availability of floating point stimulated two departures in architecture.

In the first instance, typified by the CDC 6600 and IBM 360/91, additional control circuitry was added to the instruction unit of the computer so that both the floating point and fixed point execution units could perform concurrently. In the second instance, which developed at about the same time, microprogramming was implemented as a means of control of a processor. (The IBM 360/30 and 360/40 are excellent examples of this.) This permitted a single hardware unit, i.e., a serial-by-byte adder or multiplier, to be used for either floating point or fixed point operations, as appropriate.

By analogy, the introduction of parallel computers (or vector computers) was merely responding to the same thrust to make optimum use of hardware in a design. There was no coupling with a discipline area. Thus, it is not surprising that the utility of these first machines has been questioned, that their acceptance has been slow in coming.

The Modified Approach. Because of difficulties experienced with these initial design efforts we have adopted a modified approach to the use of parallel machines. This modified approach is depicted in figure 2. We note that the basic approach to hardware design remained unchanged. What has changed is the earlier interaction of individuals skilled in the discipline areas which will utilize the machine. This earlier involvement results in new techniques which are responsive to the special abilities (as well as the relative inabilities) of parallel processors.

I will cite a particular example to demonstrate the importance of algorithmic interaction at this point in the process. In the solution of the finite difference approximation to the partial differential equation representing the heat flow in a solid, the Gauss-Seidel method, or method of successive displacement converges twice as fast as the Jacobi method, or method of simultaneous displacement. The numerical analysis of the two methods reveals that the eigenvalues of the former method are the square of eigenvalues of the latter method. Thus, two iterations of the latter method are necessary for each iteration of the former method. Now, the Fortran representation of the succssive displacement scheme appears something like:

```
          DO  1  I =  2, N-1
          DO  1  J =  2, N-1

1  A(I,J)=2.5*(A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))
```

While the Fortran representation of the simultaneous displacement scheme appears as:

```
          DO   1   I = 2,N-1
          DO   1   J = 2,N-1

1  B(I,J)=.25*(A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))

          DO  2  I = 2,N-1
          DO  2  J = 2,N-1

2    A(I,J)  =  B(I,J)
```

Clearly the method of successive displacement is not only faster, but less cumbersome, easier to read, and occupies only half as much space for data as compared with the method of simultaneous displacement. Naturally, we almost exclusively see the method of simultaneous displacement implemented for conventional, sequential machines.

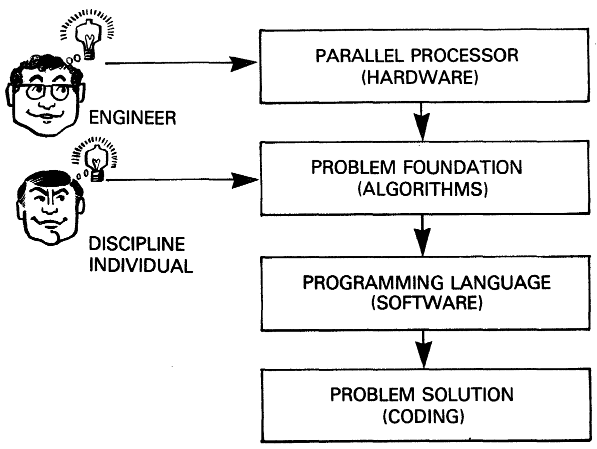

ENGINEER
DISCIPLINE INDIVIDUAL
PARALLEL PROCESSOR (HARDWARE)
PROBLEM FOUNDATION (ALGORITHMS)
PROGRAMMING LANGUAGE (SOFTWARE)
PROBLEM SOLUTION (CODING)

FIG. 2. MODIFIED TRADITIONAL APPROACH TO PARALLEL PROCESSING.

A New Approach. What, if anything, does this have to do with parallelism? The answer is that when we aim to solve a heat flow problem on a parallel machine we need to reexamine the way in which the hardware will perform the Fortran statements. In the classical case of a lock-step parallel processor (i.e., the ILLIAC-IV) the method of simultaneous displacement is the natural mode of operation. In order to implement the method of successive displacement it would be necessary to operate with only one processing element at a time, defeating the potential advantage of a parallel processor.

To summarize, the method of simultaneous displacement uses all of the processing elements e.g. 64 for the ILLIAC-IV, and requires twice as many iterations as the method of successive displacement, which results in an advantage of e.g. 32 for this "inferior" algorithm.

Let us now consider how we find the maximum value of a set. With a sequential machine we merely step through the entire set, retaining the maximum, until we reach the end. At that time the value retained is the maximum for the set. This is depicted in Fortran as:

```
          AMX  =  A(1)
          DO 1 =  I=2,N
1         AMX =   AMAX(AMX,A(I))
```

On a parallel machine we want to take advantage of the simultaneous availability of computing resources. This is a crucial point, so I will clarify the intent of that statement. We do not wish merely to maximally utilize the processor's resources. We do wish to use the resources to reduce problem solution time.

When dealing in these circumstances we are no longer interested in the strict computational complexity of an algorithm. It may be preferable to perform more operations to solve a problem and yet obtain a faster solution.

In obtaining the maximum of a set of N elements many of the processing elements will remain idle during the $\log_2 N$ steps necessary.

Thus, we arrive at the process depicted in figure 3 as a recommendation for a system approach to problem solving.

The Role of Software

Following are some general observations about software. They are particularly pertinent to all of us, as practitioners of parallel processing:

1. Computer instruction sets typically have 100-200 instructions.
2. The majority of programs are written in compiler languages (e.g. Fortran, COBOL, Ada).
3. Compilers usually generate only 50-60 different instructions.

During a project on compiler portability, I transferred to a CDC 6600 compiler for "LITTLE" to an IBM 360. This was accomplished in a straightforward manner by transliterating the instructions generated for the CDC 6600 to their counterparts for the IBM 360. There are a few instructions of course, e.g. population count and pack, which do not have direct counterparts. These required special treatment but do not conflict with the above observations.

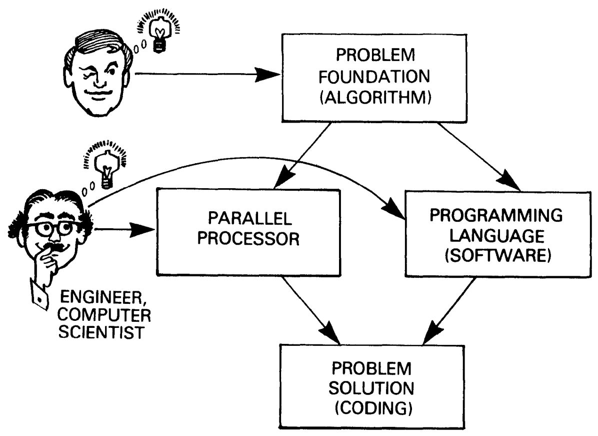

PROBLEM FOUNDATION (ALGORITHM)
PARALLEL PROCESSOR
PROGRAMMING LANGUAGE (SOFTWARE)
ENGINEER, COMPUTER SCIENTIST
PROBLEM SOLUTION (CODING)

FIG. 3. A NEW APPROACH TO PARALLEL PROCESSING

Based on these observations, one can infer that:

1. The full range of instructions available on present computers is not being utilized.
2. Broadening the semantic range of programming languages might result in significant improvements in ease of programming, speed, and space utilization.

This view of the state of programming languages and compilers is depicted in figure 4. The key issue is brought to the fore with this figure. We do not know, indeed, we cannot determine, the performance of the software that is the channel between the discipline activity and the computer system.

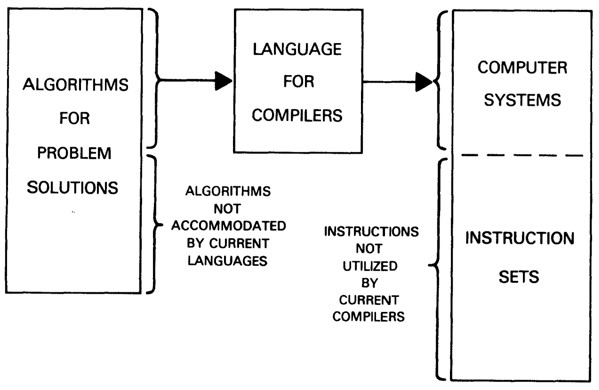

ALGORITHMS FOR PROBLEM SOLUTIONS
LANGUAGE FOR COMPILERS
COMPUTER SYSTEMS
INSTRUCTION SETS
ALGORITHMS NOT ACCOMMODATED BY CURRENT LANGUAGES
INSTRUCTIONS NOT UTILIZED BY CURRENT COMPILERS

FIGURE 4. THE ALGORITHM-COMPUTER IMPEDANCE PROBLEM

We find ourselves without a metric. At best it is difficult to make progress in this mode. We cannot easily determine whether we are approaching our target or withdrawing from it. We do not know how far we have to go to reach our goal. In the case of building circuits to perform binary addition there was a continual refinement and improvement of performance. Relatively late in the era of building such circuits a metric was devised. Only then did we know for certain that we were truly near our target.

A similar result for matrix multiplication removed the previous barrier to performance which had long stood unchallenged. Recent results have lowered the number of multiply operations a second time.

A Spectrum of Parallel Processors

There has been steady progress across the spectrum of computer organizations from initial, fully sequential processors to parallel processors. While this has occurred the structure of programming languages has not kept pace.

In figure 5 we see that the first steps in this progression were not visible to the programmer. In fact, they were almost invisible, even at the instruction set level. Except for differences in timing there were no functional changes caused by early parallelism. Compilers with instruction schedulers made it possible for programmers to completely ignore these new capabilities. Later steps in this direction have radically changed the instruction set capabilities and induced some changes in programming languages. This is a specific case of the situation to which we referred earlier.

As a computer user, I can measure a system's effectiveness only in broad terms. Programming languages and compilers must now advance so that we can exploit the advantages of parallel architectures.

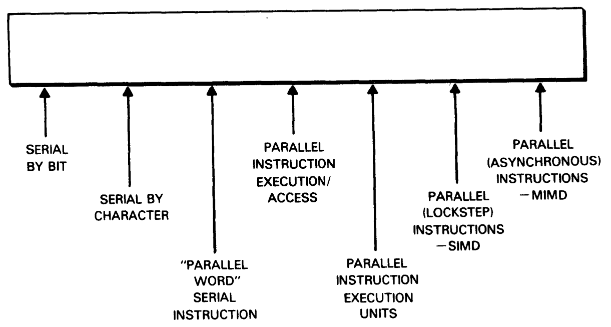

SERIAL BY BIT
SERIAL BY CHARACTER
"PARALLEL WORD" SERIAL INSTRUCTION
PARALLEL INSTRUCTION EXECUTION/ ACCESS
PARALLEL INSTRUCTION EXECUTION UNITS
PARALLEL (LOCKSTEP) INSTRUCTIONS —SIMD
PARALLEL (ASYNCHRONOUS) INSTRUCTIONS —MIMD

FIGURE 5. SPECTRUM OF PARALLELISM

Conclusion

We need to address the problems of conception, design, and software for parallel processor systems in a new fashion. We must look at all the elements of this assemblage as a system requiring optimization. If we address any individual component we may achieve only a local maximum and forego an opportunity to improve the system.

I will conclude by providing questions, not answers, to this audience.

Why does current software frequently blur the distinction between parallel processors and vector processors?

What is the critical element in parallel processing: data structure, execution scheduling, algorithm, etc.?

Can we develop a metric with which we can judge the relative advantages and disadvantages of alternatives in hardware, software, and algorithms?

What are the primitives on which concepts of parallelism are based?

The dialogue generated by this conference will lead toward resolution of those issues.

HIGH-SPEED MULTIPROCESSORS AND THEIR COMPILERS*

D. J. Kuck and D. A. Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract -- High speed multiprocessors are seen as a means of speeding up a wide class of computations that are not amenable to array processing. We discuss the structure of such machines and compare them to other organizations. The key to their efficient use is good compiler algorithms, and we present several approaches to compilation.

1. Computer Structures

Introduction

Parallelism in computer systems arises for two different reasons. One is to increase the speed of execution of a single program, and the other is to increase the throughput of a multiprogrammed system. Among existing computers, most parallel and pipeline array processors have been designed with the first reason in mind and most multiprocessors with the second.

Most present multiprocessors do not seem to consider in their architecture any features aimed specifically at the speedup of a single program, except for the fact that they have a memory which can be accessed by every processor. This is obviously good enough for multiprogramming or for speeding up programs capable of decomposition into processes requiring a low frequency of interaction. However, it is clear that by allowing a higher frequency of interaction, not only could more speedup be obtained, but also the class of programs capable of speedup would increase noticeably.

In this paper we address the question of whether additional architectural features could enhance the performance of a multiprocessor with respect to the speed of single programs. The best way to proceed seems to be to consider structures frequently found in programs and to evaluate the impact of different designs upon their execution time. In contrast to the language-directed approach to machine design, we advocate the program-directed approach. We have measured FORTRAN program characteristics,with the goal of discovering the limitations of array machine structures,for some time [KuMC72], [KBCD74]. The same methods have been used for COBOL [Stre74], GPSS [Davi72], and SNOBOL [Chen77], some of which led to multiprocessors, as well. We are now making measurements of programs that are not suitable for array processing, with the goal of designing a high-speed multiprocessor. Such a machine could be used to speed up a single program or to enhance throughput by multiprogramming.

1.1 A Machine Survey

In [Kuck78, Sect. 4.2.5], a control unit taxonomy was presented that is similar to that of [Flyn72] but can be extended to more classes of machines; 16 categories were mentioned. The idea is to view a control unit as accepting one or more instruction sequences and generating execution sequences for the rest of the machine. For the moment, we will be concerned only with execution sequences and will restrict our attention to four kinds of machines:

SES:	single execution, scalar;
MES:	multiple execution, scalar;
SEA:	single execution, array; and
MEA:	multiple execution, array.

An SES machine is a traditional serial computer. Many SEA machines have been built, including pipeline and parallel processors for which single instructions generate an array of executable operations. In fact, pipeline array processors attached to minicomputer hosts are now in widespread use. A number of array machines (including the CRAY-1 and TI ASC) can execute several array operations simultaneously and may be regarded as MEA machines. The MES category includes several types of machines that can execute multiple scalar operations at once; for example, the CDC 6600 or various multiprocessors. Clearer distinctions between machine types can be made by also considering instruction sequences (as mentioned above), but that is beyond our present

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS77-27910.

scope. Any SEA, MEA or MES machine may be regarded as a multioperation machine, because several operations are being carried out at once. However, we are concerned with high-speed execution of single programs, and will consider the shortcomings of traditional array processors together with how a kind of multiprocessor can be used to improve the situation.

Historically, when SES machines were seen to be too slow and technology was not speeding up fast enough, architectural innovations were used to achieve faster turnaround. For example, the CDC 6600 allowed several operations to be carried out at once and the 360/91 added pipelining to this idea. A combination of compiler software, control unit hardware, and processor hardware was used to attain speed improvements over SES machines. These machines (and their successors) may be called "data-flow" machines because data and control dependences are used to determine when operations can be executed, so some operation overlap is possible. Research in this area continues [DeMi74], [KeLP79], [Davi79]. On the other hand, SEA machines generally rely on a user or software vectorizer to introduce explicit array statements in a source program, and these are then executed at high speed. It is sometimes difficult for users to rethink and rewrite their programs properly. Automatic vectorizers, although they can be much more powerful than hand reprogramming, have not been commercially available until recently.

It may be observed that certain algorithms are amenable to substantial speedup by a combination of these ideas. For example, the merging and sorting networks of Batcher [Batc68], the FFT network of Pease [Peas68], the arithmetic expression tree evaluation of Kuck and Muraoka [Swan72], [VaZi78], etc.

Machines in which the completion of one operation triggers the next can become complex and cumbersome when one attempts to push the idea too far. Array machines and vectorizers are the fastest available systems today, but it can be shown abstractly and empirically, that they are inefficient for certain classes of computations. What then is the proper next step toward "ultimate speed" machines that are useful in a wide range of applications?

We believe that since a fairly wide class of computations can be successfully vectorized for array machines [KuMC72], [KBCD74], [Kuck77], [CKTB79], one should take this as given and study the difficulties with the remaining computations. One should look for additional compiler algorithms and hardware flexibilities that lead to substantial speedups on a much wider class of computations. Thus we are led to a class of machines that can behave as a high-speed array processor when appropriate and can behave as a high-speed multiprocessor when necessary. The data and control flow notions of earlier machines must, of course, be exploited by such a system in an efficient manner.

1.2 A High-Speed Multiprocessor

The following sketch of a high-speed multiprocessor is preliminary. We have studied a number of algorithms and programs and believe this offers a substantial improvement over current machines. Ideas about compiling for this machine will be presented in later sections. We are in the process of implementing these and, after analyzing our collection of over 1,000 programs, we expect the architecture to become clearer.

First, we describe a processor cluster (PC) that can behave as a SIASEA (single instruction, array; single execution, array) machine or as a MISMES (multiple instruction, scalar; multiple execution, scalar) machine. Each processor in the cluster is a fairly traditional machine, with a scalar control unit and processor, together with a local memory. Thus, each processor may carry out an independent computation. The processor capabilities can be chosen to meet the intended application areas, but the use of LSI processors is clearly attractive (e.g., 32-bit floating-point microprocessors). The cluster size is also a design variable, but from the viewpoints of both technology and applications effectiveness, 8 to 16 processors per cluster seems appropriate.

Fig. 1 shows two such processor clusters (containing c processors each) interconnected via a set of local alignment networks (LANs). These alignment networks can be used to communicate within one cluster (each can use half of the LAN independently) or adjacent clusters can intercommunicate through the shared LAN. Each cluster also has an array control unit that accepts an array instruction set and drives all c processors in lock-step fashion.

The alignment networks can be used to communicate data between processors and memories. They may also be used, simultaneously, to send status bits between processors. One important feature of each processor control unit is that the execution of each instruction is conditional on a set of status bits. These can be set by other processors in the cluster (or outside the cluster). Thus, a set of computations running in a PC can be made dependent at the instruction level--so, for example, data computed in processor 1 can be stored in a local memory, processor 1 can set an appropriate bit in processor 2, and processor 2 can fetch the result of processor 1 to proceed with its computation. All of this can be accomplished in a matter of a few clocks, so very tight coupling is possible. For example, if each processor is computing one iteration of a loop and there are data dependences from one statement to the next between iterations, these can be quickly satisfied. As another example, a PC could evaluate an arithmetic expression tree quickly by appropriate alignments, see [Swan72]. We shall discuss the use of such a PC in more detail later with respect to program structures. For the moment it is clear that a PC can operate as an MES or SEA machine.

Next, consider a collection of PCs to form a complete system. Fig. 2 shows a collection of n PCs plus a set of global control units (GCUs), a global alignment network (GAN), and a global memory (slower levels in the memory hierarchy are ignored here). The GCUs allow a collection of PCs to work together either in an array or scalar fashion. Thus, the entire system could be partitioned to handle several jobs at once, but the key point is that any job may be handled in either an SEA, MEA or MES mode. Program memory is associated with each control unit.

Array access and alignment from the global memory through the global alignment network is well understood [BuKu71], [KuSt79], [LaVo79], [Lawr75], and the same ideas can be used in each PC for array operations. Also, memory access for independently distributed addresses is well understood [ChKL77]. Thus, a set of array computations with linear addressing patterns can be assumed to work well. Memory stores and fetches for MES operation as well as most subscripted subscript array accesses should work well also. The problem of aligning data in these latter cases has been unsolved until recently. We will discuss this in more detail in Section 4.

2. Language/Machine Relations

The flow of data and control in programs are fundamental considerations in the design and performance of a machine to execute those programs. We are interested in relating various program constructs to various computer organizations. In order to proceed, we shall first give a broad paradigm for the languages and programs we are considering. Then we will relate these to the machine organizations discussed earlier.

The language paradigm emphasizes those aspects of programs that most concern us in compiling programs and executing them with high performance on the given machine structures. Many details are omitted in the interest of brevity. The presentation contains three parts:

1) π-blocks;
2) DAGs of π-block clusters; and
3) Control structures for (1) and (2).

2.1 π-Blocks

The name "π-block" means a block that is derived from a partition of a program's data and control dependence graph. The discussion here is a generalization of that in [Kuck78], which was given in a narrower language setting.

We shall consider π-blocks as the smallest objects of concern to a compiler in scheduling a computation on a machine. It is assumed that atomic operations exist in any given machine, and that dependence graphs of such atomic operations are contained within each π-block. Several examples of π-blocks follow.

An arithmetic assignment statement is a π-block with atomic arithmetic operations connected by a data dependence tree, as well as an atomic assignment operator. Similar statements can be made about Boolean assignment statements or character string assignment statements. Also, cons(car(x),cdr(y)) might be a LISP π-block, or a complex APL expression might be a π-block. In programs for a sorting or merging network machine [Batc68], the comparison of two numbers and transmission of the greater in one direction and lesser in the other direction can be regarded as a π-block. A similar definition could be made with respect to an FFT network [Peas65], etc. Other examples of π-blocks are decision trees that branch to one of several program locations, and conditional expressions that may do a branch or select one sequence of several assignment statements [Kuhn79].

Thus, π-blocks may have as values data items of any type, or program addresses. They can also be subject to mode bits that determine whether or not particular values are to be computed.

Programs are collections of π-blocks of the types mentioned above. In order to be able to deal with acyclic graphs of objects later (in scheduling and in the hardware), we define a π-block to be a single node of the type mentioned above, or a maximal cycle of such nodes formed by data and control dependences.

2.2 DAGs of π-Block Clusters

Just as there were dependences within π-blocks, there are dependences among the π-blocks of a program. A graph formed using π-blocks as nodes and dependences as arcs is, however, a directed acyclic graph (DAG), because all dependence cycles are within π-blocks. For purposes of compilation and execution, we may be interested in forming nodes that are clusters of several π-blocks; such clusters may be formed from π-blocks accessing the same variables, or requiring similar data alignments for processing, etc. Thus, we will now consider clusters of π-blocks interconnected in the form of a DAG.

The arcs in this DAG may represent dependences due to control or data flow, and there are three types of the latter: data dependence, antidependence and output dependence (cf., [Kuck78]). Associated with each of these types of arcs is a set of distance vectors, one for each pair of array variables causing a dependence. The distance vector indicates the difference in subscript values in each position.

Most programming languages have some kind of repetition statement (e.g., DO, FOR, etc.). We will assume that the control for such repetitions has been distributed down to the level of π-block clusters (see [Kuck78], [CKTB79], or [KuMC72] for more details). If a cluster contains π-blocks that originated in different repetition statements, they may be combined using mode bits, for example.

Henceforth, we will consider DAGs of π-block clusters. The clusters are interconnected by data dependence arcs labeled with distance vectors, and repetitions are associated with π-block clusters. Thus, we can deal with a graph that consists of antichains of π-block clusters, any of which can be executed at once, and the program can be executed by simply observing the dependences between the antichains.

2.3 Statement Execution Ordering

The atomic operations within a π-block usually have well-known dependence relations (e.g., operator precedence for arithmetic expressions). At higher levels, it is necessary to assume or explicitly specify these dependences. For example, most programming languages for traditional computers assume that statements are normally executed one after the other from the top to the bottom of a written page or memory area. When hardware parallelism is available, programming or compiler techniques are needed to exploit it.

For blocks of assignment statements, various statement execution orderings were specified in [Kuck78]. Without repeating the formal definitions we will sketch the ideas here, and then we will extend these ideas to a block of assignment statements (or π-block cluster) with an associated repetition statement.

The two broad classes of statement execution ordering are SEQ and SIM. SEQ means that all the normal data dependence, antidependence and output dependence arcs are followed in executing a program, but any statements without such dependences between them may be executed in any order. If a number of assignment statements are to be executed with SIM ordering, all right-hand side atoms must be fetched before any left-hand side results are stored. These two classes have an intersection that contains a class specified by TOG; such statements may be executed together, i.e., in any order at all, since there are no dependences between them. SEQ contains a class called SF (store all previous left-hand sides before fetching the next right-hand side) which corresponds to the strict sequential ordering implied by traditional languages running on traditional serial machines. Other execution orderings are specified in [Kuck78].

These ideas can be extended to repetition statements as follows. Let I be an ordered set $(\mathrm{I},<i_1,\ldots,i_m>)$ called an index set, let B be a block of assignment statements with a specified execution ordering, and let control represent SEQ, SIM, SF, TOG or any other execution ordering. Then by

DO control $I[B]$

we mean the following.

1) Expand B according to its statement execution ordering.
2) Copy the result of (1) for i_1, i_2, ..., and i_m from left to right.
3) Apply the execution ordering specified by control to this set of m sequences.

Example A traditional loop (e.g., a FORTRAN DO loop) can be specified as

DO SF $I[\mathrm{SF}[S_1;\ldots;S_n]]$,

where $I = (\mathrm{I},<i_1,\ldots,i_m>)$.

The inner SF requires that statements S_1 through S_n be executed in a serial way with the left-hand side of S_i being stored before any of the right-hand side atoms of S_{i+1} are fetched. This sequence is computed m times: first for i_1, then for i_2, ..., and finally for i_m. ■

Other examples will be given in the following section.

2.4 Machine Considerations

In this section we will discuss how one singly-nested loop can be mapped onto each of the several kinds of machine structures discussed earlier. The statement execution orderings of the previous section will be used to illustrate what a user might write or what a pre-compiler might generate from a traditional source language program. Later we will show how these statements can be compiled for high-speed multiprocessors.

The four types of machines mentioned in Section 1 were SES, MES, SEA and MEA. The example of Section 2.3 showed how to specify a purely sequential loop for a traditional SES machine. This may, in fact, be regarded as the meaning of a FORTRAN DO loop. For each of the other machine organizations, it is important to know some details of the statements in the loop. For an SEA machine, some parts of a loop may need to be executed as on an SES machine, but others can be executed as

SF[DO SIM $I[S_1]$; DO SIM $I[S_2]$; ...;
DO SIM $I[S_n]$],

which corresponds to a sequence of array operations. The goal of vectorizers for array machines is just this kind of code, see [KuMC72], [KBCD74].

An MEA machine has the additional flexibility of being able to execute several array operations at once. Thus, in general, the outer SF of the SEA machine can be replaced by SEQ to allow as many simultaneous array operations as the program has and the machine can handle. So we have

SEQ[DO SIM $I[S_1]$; DO SIM $I[S_2]$; ...;
DO SIM $I[S_n]$]

as the canonical MEA machine program.

Finally, consider the MES machine and several types of programs. In the simplest case, we can execute the same serial program independently in each available processor, once for each loop repetition. Thus, we can write

$$\text{DO TOG } I[\text{SF}[S_1;S_2;\ldots;S_n]]$$

to denote a set of m independent repetitions of n statements. Note that if some of the S_i are conditional expressions, separate paths may be followed for each of the m cases. Also, this idea can be generalized to a set of distinct and independent blocks as might arise from a sequence of loops.

Next, assume an MES machine with a program that requires data to be passed between processors. In this case we can write

$$\text{DO SEQ } I[\text{SF}[S_1;S_2;\ldots;S_n]]$$

to denote a set of m SF sequences, each of which can be executed at once, subject to whatever dependences exist between them as indicated by SEQ. In order to have such a program execute efficiently, a tight interprocessor coupling is necessary. Note that in contrast with the SEA case, we have the index on the outside and SF on the inside here. The SEA machine executes a sequence of array operations, whereas the MES machine executes an array of serial computations, for the same given source program.

3. The Compiler

We now proceed to consider a methodology to translate programs written in a sequential language like FORTRAN into code suitable for fast execution in multioperation machines.

The first step is to translate the original program into a DAG of π-block clusters. Techniques to do this have been developed and implemented during the last few years [CKTB79], [Wolf78].

The transformations that should be applied to the DAG of π-block clusters are the subject of the remainder of this section. For reasons of space, we had to choose in our description between clarity and precision. We will strive to obtain the first, relying mostly on examples (see [Padu79] for more details).

Transformations on DAGs of π-block clusters can be classified as follows:

1) π-block transformations;
2) π-block cluster transformations; and
3) DAG of π-block cluster transformations.

When the target machine is of the SEA type, the π-block cluster transformations and some of the π-block transformations will not be applicable.

We will study each one of these types of transformations in turn. We will try to make our considerations as machine independent as possible, making use of the language of SEQs, SIMs, etc., wherever convenient. In Section 3.4, we will consider the influence of particulars of the target machine, giving special emphasis to the machine described in Section 1. Throughout this section, our goal is obtaining as fast execution time as possible. However, even if we consider an ideal target computer (like Murtha's IT machine [Murt66]) and very simple programs, we find that algorithms to obtain the optimum execution time are impossible in practice (they are NP-complete). For very simple problems, like bin packing, it has been proved that some heuristics give results very close to the optimum. The proofs, however, are sometimes quite elaborated [Grah76]. In our case, the problem of finding an optimal algorithm or analyzing a heuristic are even harder because machines are not ideal and programs are, in general, very complex to analyze involving, for example, IF statements and, therefore, probabilistic execution times. We are forced, then, to abandon any search for optimal transformation and content ourselves with engineering judgment and experimental evaluation of our techniques. For these reasons, the statements made here about the different transformations are tentative; a more concrete assessment must await experimentation.

3.1 π-Block Transformations

We will consider three types of π-block transformations. They should be applied in the same order they are presented here, as shown in the overall algorithm presented in Section 3.1.4.

3.1.1 Partition

A π-block can be assumed to represent an SF statement. Let us say that its form is

$$\text{DO SF } I\{\text{SF}[S_1;S_2;\ldots;S_n]\} \qquad (1)$$

where, by definition of a π-block, there is a path in the graph of dependences from S_i to S_j for all i, j $\varepsilon\{1,2,\ldots,n\}$ if cycles are present.

In cases where it is applicable, the partition method will produce a TOG statement semantically equivalent to (1) of the following form:

$$\begin{aligned}\text{TOG}\{&\text{DO SF } I_1[\text{SF}\{S_1;S_2;\ldots;S_n\}];\\ &\text{DO SF } I_2[\text{SF}\{S_i;S_2;\ldots;S_n\}]; \qquad (2)\\ &\quad\cdots\\ &\text{DO SF } I_m[\text{SF}\{S_1;S_2;\ldots;S_n\}]\}\end{aligned}$$

where the I_is, i = 1, 2, ..., m are pairwise disjoint and their union is I. We will demonstrate its applicability by several examples.

Example 3.1 The loop

```
    DO  I = 1, M
S1:     A(I) = B(I) + C(I)
    ENDO
```

represents a vector operation (here S_1 is a π-block by itself). It is easy to see that the following transformation is correct:

```
DO SF(I,<1,2,...,M>) [$S_1$] =>

    TOG{DO SF(I,<1>) [$S_1$];

        DO SF(I,<2>) [$S_1$];                    (3)

          . . .

        DO SF(I,<M>) [$S_1$]}.
```

The last TOG statement can be written more compactly as

```
DO TOG(I,<1,2,...,M>) [$S_1$]
```
■

Transformations similar to the ones in Example 3.1 can always be done when no cycles are involved in the dependence graph of the π-block. This type of transformation may be called <u>total partition</u>.

Sometimes it is possible to apply partition to π-blocks involving cycles as shown in the next example.

<u>Example 3.2</u> Consider the loop

```
      DO  I = 3, M
$S_1$:        A(I) = A(I-2) + 1
      ENDO
```

We can partition this loop using the following transformation

```
DO SF(I,<3,...,M>) [$S_1$] =>

    TOG{DO SF(I,<3,5,...,$\lfloor\frac{M-1}{2}\rfloor$*2+1>) [$S_1$];

        DO SF(I,<4,6,...,$\lfloor\frac{M}{2}\rfloor$*2>) [$S_1$]}
```
■

To be able to do transformations like the one in Example 3.2 (called <u>partial partition</u>), it is necessary to make use of the distance vectors. For a similar method called splitting, see [BCKT79]. Partial partition may not help in the case of single-instruction stream machines because the cycles could involve IF statements.

In the previous two examples, we considered only singly-nested loops. Generalization to loops with more levels of nesting should be obvious. There is, however, a method that is useful in some cases for the partition of multiply nested DO loops. This method, loop interchange, is described in [CKTB79] and [Wolf78]. The methods of this section would be useful with the DOALL construct of [Burr79].

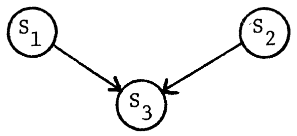
<u>3.1.2 Algorithm Change</u> If the method of partition does not work, this second method will be applied. The idea is to use as much information as possible from a π-block in order to detect what sort of algorithm is represented by it. Thus, if the π-block is recognized as a linear recurrence, the best parallel algorithm known for the particular target machine should be applied. Algorithms for linear recurrences have been widely studied, some results and references can be found in [Kuck78].

<u>3.1.3 Loop Freezing</u> When everything else fails, the π-block will have to be executed serially. In this case, the body of the π-block can be considered as a program segment and global transformations can be applied to it.

<u>Example 3.4</u> The following loop cannot be partitioned, and no known algorithm can be applied to speed it up.

```
      DO  I = 1, M
$S_1$:        A(I) = A(I-1) * A(I-2) * C(I-2) + X
$S_2$:        D(I) = D(I-1) * A(I-1) * C(I-1) + Y
$S_3$:        C(I) = C(I-1) * A(I) * D(I) + Z
      ENDO
```

If we freeze this loop (i.e., serialize it and consider its body as a program segment), we obtain the following graph of dependences



Now, applying to the body a global transformation (see Section 3.3), we obtain

```
DO SF(I,<1,2,...,M>) {SF[TOG{$S_1$;$S_2$};$S_3$]}
```
■

<u>3.1.4 Overall Strategy</u> We conclude our discussion on π-block transformations with a description of the overall strategy. This is shown in the following algorithm:

<u>Algorithm 3.1</u>

```
Input:  π-block P of the form DO SF $I${SF[$S_1$;$S_2$;
        ...;$S_n$]}

Output: Execution structure

If partition can be applied to P
  Then
    Transform DO SF $I${SF[$S_1$;$S_2$;...;$S_n$]}
    to        TOG{DO SF $I_1$[SF{$S_1$;$S_2$;...;$S_n$}];
                    . . .
                  DO SF $I_m$[SF{$S_1$;$S_2$;...;$S_n$}]}
    For k = 1 to m apply Algorithm 3.1
         to DO SF $I_k${SF[$S_1$;...;$S_n$]}
  Else
    If algorithm change can be applied
      Then apply algorithm change
```

Else apply global transformation to the body of P (i.e., to SF[S_1;S_2;...;S_n]). ■

To see how the algorithm will work, we will use the following example.

Example 3.5 Let us apply Algorithm 3.1 to the loop

```
      DO  I = 1, MI
          DO  J = 3, MJ
S1:           A(I,J) = A(I,J-2) + 1
          ENDO
      ENDO
```

Originally, S_1 is a π-block by itself with the following representation

DO SF(I,<1,2,...,MI>) × (J,<3,4,...,MJ>) [S_1] (5)

When we apply Algorithm 3.1 to (5), we will have the following sequence of transformations

Step 1

DO SF(I,<1,2,...,MI>) × (J,<3,4,...,MJ>) [S_1]

$\Downarrow$

TOG{DO SF(I,<1>) × (J,<3,4,...,MJ>) [S_1];

DO SF(I,<2>) × (J,<3,4,...,MJ>) [S_1];

. . .

DO SF(I,<MI>) × (J,<3,4,...,MJ>) [S_1]}

Step 2

Applying the algorithm to DO SF(I,<k>) × (J, <3,4,...,MJ>) [S_1] for k ε{1,2,...,MI}, we obtain

TOG{DO SF(I,<k>)(J,<3,5,...,$\lfloor\frac{MJ-1}{2}\rfloor$*2+1>) [$S_1$]; (6)

DO SF(I,<k>)(J,<4,6,...,$\lfloor\frac{MJ}{2}\rfloor$*2>) [$S_1$]}

as in Example 3.2.

Step 3

Finally, we can apply a linear recurrence algorithm to the DO SFs in (6). ■

3.2 π-Block Cluster Transformations

In this section we will start by considering a cluster of π-blocks with the following characteristics:

1) All π-blocks come from the same loop in the original source program;
2) The DAG of dependences of the π-blocks in the cluster has the form of a chain; and
3) The execution time of every iteration of any given π-block is constant.

We will study how to transform this cluster, which we will call a simple cluster, to produce good machine code. Later, we will mention briefly how to extend the techniques studied to more general clusters. We will consider the case of a MES machine only. The case of a MEA machine can be treated similarly.

Let us say that the statements in the cluster are S_1, S_2, ..., S_n. One way of executing this cluster is the one shown at the end of Section 2.4, namely,

DO SEQ I[SF[S_1;S_2;...;S_n]].

Let us consider the consequences of executing a recurrence computation using this method. Suppose that statement S_j is of the form

$X_i \leftarrow X_{i-1} * A_i + B_i$.

Assuming that statements S_1 through S_{j-1} had no recurrence, all processors in an MES machine could reach statement S_j within a short time interval of each other. However, S_j would be executed first on processor 1, then on processor 2, then on processor 3, and so on, with a gap of length $O(m)$ in processor m as it waits for X_{m-1}. Thus the execution time for a program with a single recurrence, assuming m loop repetitions and n statements, would be $O(m+n)$ steps. Since the serial computation takes $O(mn)$ steps, the efficiency (speedup/processors) is $O(n/(m+n))$.

Now let us study an alternative scheme for executing such a computation, where the machine will execute the first π-block on processor 1, the second π-block on processor 2, and so on. This computation can be represented by

SEQ[DO SF I[SF[S_1;...;S_p]]; ...;

DO SF I[SF[S_q,...;S_n]]], (7)

where S_1 ... S_p form the first π-block and S_q ... S_n form the last π-block. Each π-block is executed serially by DO SF I; however, the ensemble of cluster repetitions is executed with SEQ control, so they proceed as much in parallel as possible. We call this method pipelining because a loop is broken into a number of smaller loops that may be chained together. Note that in this case, assuming that clusters are small, $O(n)$ processors are used, and assuming that dependences exist across the entire set of processors, the computation time is $O(m+n)$ steps (as it was above), but the efficiency is now $O(m/(m+n))$. Thus, pipelining tends to be more efficient when the number of loop repetitions is large, relative to the number of loop statements. Of course, if we need not pay time for dependences across the entire array, then the pipeline time is $O(m)$ so the speedup and efficiency increase.

Henceforth, we will assume that the number of loop repetitions is large relative to the number of statements. We will, therefore, only consider pipelining. The application of what follows when the number of statements is large is immediate.

When translating to an instruction controlled machine (as opposed to a data controlled or data-flow machine) we, for simplicity, will use a synchronization per iteration approach. The resulting structure for the case of a singly-nested loop is shown next.

```
parbegin
  π_1: DO I = 1, M
          $S_1$; $S_2$; ...; $S_p$; V($\sigma_1$)
       ENDO;
         . . .
  π_g: DO I = 1, M
          P($\sigma_{g-1}$); $S_\ell$; ...; $S_k$; V($\sigma_g$)          (8)
       ENDO;
         . . .
  π_m: DO I = 1, M
          P($\sigma_{m-1}$); $S_q$; ...; $S_n$
       ENDO
parend
```

Here, as before, S_1, S_2, ..., S_p form the first π-block, S_ℓ ...S_k form the g^{th} π-block, and S_q ... S_n form the last π-block of DO SF $I[S_1;\dots;S_n]$.

The statements P and V are the well-known synchronization primitives, and the σ_i, i = 1, ..., m - 1 are semaphores. From the definition of a π-block, it is easy to see that (8) is a correct transformation.

In the segment of program shown in (8) we see that for large enough M the execution time is dominated by the <u>bottlenecks</u>, which are defined as those π-blocks of maximum execution time.

Given that pipelining can be applied to a cluster of π-blocks, we are faced with the problem that local transformations can also be applied to the individual π-blocks. We now proceed to state an algorithm that integrates the local transformations of 3.1 with pipelining.

<u>Algorithm 3.2</u>

Input: A simple cluster of π-blocks

Output: The cluster transformed for parallel execution; and its execution time.

<u>Step 1</u>

Compute the execution time of the cluster when executed as a pipeline. To compute this time, we should attempt to decrease the size of the bottlenecks by applying loop freezing and global transformations to their bodies. Call the pipeline execution time T_{pip}, and the program structure resulting PS_{pip}.

<u>Step 2</u>

Let us say that the π-blocks in the cluster are π_1, π_2, ..., π_m and that π_g is the first bottleneck. Then we proceed as follows:

1) Apply local transformations (Algorithm 3.1) to π_g. Let T_g be the execution time of the resulting program structure and PS_g the structure itself.
2) Apply Algorithm 3.2 to the chain π_1, π_2, ..., π_{g-1}. Let T_1 be the execution time of the resulting structure and PS_1 the structure itself.
3) Same as (2) but for chain π_{g+1}, ..., π_m. Let T_2 be the execution time and PS_2 the resulting structure.

Finally, let $T_{g\ell} = T_1 + T_g + T_2$ and $PS_{g\ell} \equiv SF[PS_1; PS_g; PS_2]$.

If $T_{pip} \leq T_{g\ell}$, then the result is T_{pip} and PS_{pip}

else the result is $T_{g\ell}$ and $P_{g\ell}$. ■

<u>Example 3.7</u> Consider the following loop:

```
       DO  I = 1, 10
$S_1$:     A(I) = B(I) + C(I)
$S_2$:     D(I) = A(I) + 1                         (10)
$S_3$:     E(I) = D(I) + 2
       ENDO
```

Here each statement is a π-block by itself. If we only count arithmetic operations when computing the execution time, then we have $T_{pip} = 2 + 10 = 12$.

When Step 2, Algo. 3.2 is executed, S_1 (the first bottleneck) will be partitioned as a vector operation (total partition). The execution time when P processors are available is $\lceil\frac{10}{P}\rceil$.

Finally, if it is assumed that P > 3, then $T_{pip} > 3\lceil\frac{10}{P}\rceil = T_{g\ell}$. Therefore, (10) should be executed as a sequence of vector operations. ■

Example 3.8   In the loop

```
      DO  I = 1, 100
S1:      A(I) = B(I) + 1                          (11)
S2:      C(I) = C(I) * C(I-1) + A(I) * C(I-2)
S3:      D(I) = C(I) + 1
      ENDO
```

each statement is a π-block. The pipeline execution time (assuming as in Example 3.7, that only arithmetic operations count) is $T_{pip} = 2 + 4 \times 100 = 402$.

Applying Algorithm 3.2 to (11), we find that the bottleneck cannot be improved. Therefore, $T_{g\ell} = 2 \times \lceil\frac{100}{P}\rceil + 4 \times 100$, and $T_{pip} \le T_{g\ell}$ when $P \ge 3$. We conclude that pipelining should be used for (11). ∎

In general, we will have to deal with more general cases than a chain of π-blocks. Some of the considerations required to extend our previous algorithm are given next.

1) In general, the execution times of the π-blocks will not be constant. For example, if IF statements are involved, the execution time could be random.

In this case, we could be very precise when computing execution times if all probabilities are known. This computation can, however, become lengthy and furthermore, the probabilities are not always known. We should try, then, to use a simple value like maximum possible execution time when computing the overall execution time of the cluster.

2) Cluster of π-blocks could have a DAG of dependences more general than simple chains. If the number of iterations of the original DO loop is much bigger than the number of statements, we could apply topological sorting [Knut73] to obtain a chain, and then apply Algorithm 3.2. However, if the number of iterations is comparable to the number of π-blocks, a different algorithm should be applied.

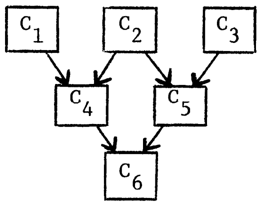
3) The π-blocks in a cluster do not have to originate in the same loop of the source program. By using techniques like fusion [AbKL79], π-blocks coming from different loops could become part of the same cluster.

3.3 DAG of π-Block Clusters Transformations

Once the π-block clusters have been transformed, it remains to transform the whole DAG of π-block clusters.

If the target machine is of the SEA type, topological sorting should be done on the DAG, and then we can execute the clusters sequentially. On the other hand, for MES or MEA computers as many clusters as possible should be executed simultaneously.

Example 3.9   Consider the following DAG of clusters of π-blocks


$$\begin{tikzcd} \boxed{C_1} \arrow[dr] & \boxed{C_2} \arrow[d] \arrow[dr] & \boxed{C_3} \arrow[d] \\ & \boxed{C_4} \arrow[d] & \boxed{C_5} \arrow[dl] \\ & \boxed{C_6} & \end{tikzcd}$$

The antichains are C_1, C_2 and C_3; C_4 and C_5; and C_6. Execution of this DAG in a MEA or MES computer, given that enough processors are available, will be as follows:

SF[TOG{C_1;C_2;C_3}; TOG{C_4;C_5};C_6]

In a SEA computer, the DAG could be executed as

SF[C_1;C_2;C_3;C_4;C_5;C_6] ∎

3.4 Transformations Dependent on Machine Particulars

We now briefly consider some transformations that could be helpful in the particular machine we showed in Section 1. These transformations are intended as illustrative and are by no means exhaustive.

3.4.1 Data Transmission and Synchronization

One of the goals in code generation should be to try to make as little use as possible of the global alignment network. Also, if the processors have general registers, we should try to use them instead of the local alignment network. Transformations to achieve this goal are not always straightforward.

Example 3.10   Consider the following program

```
      DO  I = 1, N
S1:      A(I) = B(I) + C(I)
      ENDO                                   (12)
      DO  I = 1, N
S2:      D(I) = A(I-8) + 2
      ENDO
```

If (12) is executed in such a way that the ith iteration of S_i is executed in the same processor as the ith iteration of S_2, it will be necessary to use the local alignment network (or even the global alignment network if the clusters are very small)for data transmission (and synchronization if the global control unit is not used). A better solution is to execute the ith iteration of S_1 in the same processor as the (i+8)th iteration of S_2.

Then data transmission and synchronization are avoided. ■

Example 3.11 Suppose we want to pipeline the following DAG of π-blocks.

$$\begin{tikzcd} & & & & & S_0 \arrow[lllll] \arrow[d] \\ S_1 \arrow[r] & S_2 \arrow[r] & S_3 \arrow[r] & S_4 \arrow[rr] & & S_5 \end{tikzcd}$$

If we have the machine of Fig. 1 with c = 2, there will be some allocations that will need use of the global alignment networks and others that won't, as shown in the next figure.

Cluster	1		2		3	
Processor	P_1	P_2	P_3	P_4	P_5	P_6
A good allocation	S_1	S_2	S_3	S_0	S_4	S_5
A bad allocation	S_0	S_1	S_2	S_3	S_4	S_5

■

3.4.2 Reclustering to Increase Efficiency of Pipelining

Let us start with one example.

Example 3.11 If in the chain of π-blocks

$$\begin{tikzcd} \boxed{S_1} \arrow[d] \\ \boxed{S_2} \arrow[d] \\ \boxed{S_3} \end{tikzcd} \qquad (12)$$

One iteration of S_3 takes two units of time and S_1 and S_2 take one unit of time per iteration each, then if we execute (12) (using a canonical transformation like (8)) as

$$\text{SEQ}\{\text{DO SF } I[S_1];\ \text{DO SF } I[S_2];\ \text{DO SF } I[S_3]\}, \qquad (13)$$

we will obtain an efficiency close to 2/3. However, if we cluster S_1 and S_2 and execute as

$$\text{SEQ}\{\text{DO SF } I[\text{SF}\{S_1;S_2\}];\ \text{DO SF } I[S_3]\} \qquad (14)$$

we will obtain an efficiency close to 1. Notice that (13) and (14) will take the same amount of time. ■

The goal of the process of reclustering is to increase efficiency without increasing time. The complexity of the problem of finding the best possible clustering can be easily shown to be NP-complete; therefore, some heuristics should be used.

3.4.3 Allocation Overhead If dynamic processor allocation is allowed, the time to allocate a processor could have a determining effect on the degree of parallelism that can be used profitably. We have looked at a very simplified version of this problem and it turns out that even if allocations times are constant and an unlimited amount of processors are available, the problem is NP-complete [Padu79].

4. Conclusion

Throughout this paper we have discussed methods of compilation for a high-speed multiprocessor and we have referenced similar papers for array processors. Table I summarizes some general conclusions about the performance of two types of machines. Two points that are frequent sources of difficulty in array machines are conditional branching inside loops and the accessing of arrays (or other data structures) in irregular ways. Both of these subjects will be treated here.

IF Trees

Conditional statements inside loops can frequently be turned into array tests for fast execution [Bane79]. In fact, nearly all of the programs analyzed in [KBCD74] achieved substantial SEA speedup despite IF statements inside DO loops. Nevertheless, in some cases, IF statements inside loops can present serious difficulties to fast SEA execution.

For example, a loop with many, equally likely paths and relatively few iterations could lead to many, very short vector operations. In such cases, it may be desirable to partition the loop for execution in a high-speed multiprocessor. Thus, each processor can follow a separate path through the loop as in each iteration of a serial machine execution. Synchronization may be needed, but it can also be traded for some redundant computation.

Additional speedup may be achieved by equipping each processor with a parallel IF-tree evaluator [Kuhn79]. After the conditions of the tests are computed, the path through the decision tree is evaluated in time (gate delays) proportional to the log of the number of IFs in the loop (we are assuming there are many IFs). After the proper outcome is selected, final computations are made for each iteration in its processor. This technique may be viewed as an array of sequential computations, and this is indicated in Table I.

Interconnections

It is well known that the fastest possible processor-memory switch is the crossbar. However, the cost of the crossbar switch for a large number of processors and memories is quite high. Alternatives to the crossbar switch have been developed using the concept of perfect shuffle [Ston71].

Table I

	Array Processor Vectorization (sequence of arrays)	Multiprocessor (array of sequences)	Pipelining Across Multiprocessors
1) Number of processors for maximum speedup	* Proportional to the number of iterations of the loop.	* Proportional to number of iterations of the loop.	* Proportional to the number of statements.
2) Speedup	* High if π-blocks are vector operations or if they are linear recurrences. Equal or worse than pipelining if this is not the case.	* High	* Proportional to number of processors.
3) Efficiency	* High for vector operations and linear recurrences with small band. Drops off in other cases.	* High	* Over 70% in most cases.

For array machines, we could use the omega network [Lawr75] which allows fast access to rows, columns and diagonals of two-dimensional matrices when these are suitably distributed among memory modules. This same omega network could be used in multiprocessors as proposed in [Burr79].

Another possibility is the one-stage perfect shuffle with queueing on each comparator [Lang76]. The queueing works well for array machines; however, when requests to memory are random, as could be the case in a multiprocessor, the queues could become too long. An alternative to queueing we have been studying is to set at random the two input modules when conflict arises. One of the results we have obtained is that, if two one-stage perfect shuffles are present in a system with n processors and n memories, the average delay for a request between processor and memory will take $O(\sqrt[5]{n})$ stages [Padu79]. This magnitude can be greatly decreased using other techniques.

References

[AbKL79] W. Abu-Sufah, D. Kuck, and D. Lawrie, "Automatic Program Transformations for Virtual Memory Computers," Proc. of the 1979 Nat'l. Computer Conf. (June, 1979), pp. 969-974.

[Bane79] U. Banerjee, Ph.D. thesis, in preparation, Dept. of Comput. Sci., University of Illinois at Urbana-Champaign (1979).

[Batc68] K. E. Batcher, "Sorting Networks and Their Applications," Proc. AFIPS Spring Joint Comput. Conf. (1968), pp. 307-315.

[CKTB79] S. C. Chen, D. J. Kuck, R. A. Towle, and U. Banerjee, "Time and Parallel Processor Bounds for Fortran-like Loops," to appear in IEEE Trans. on Comput. (1979).

[BuKu71] P. Budnik, and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Trans. on Comput. (Dec., 1971), pp. 1566-1569.

[Burr79] Burroughs Corporation, "Numerical Aerodynamic Simulation Facility Feasibility Study," (Mar., 1979).

✓ [Chen77] W. K. Cheng, "Multiprocessor for String Manipulation," M.S. thesis, Dept. of Comput. Sci., Univ. of Ill. at Urb.-Champ., Rpt. No. 77-907, (Oct., 1977).

[ChKL77] D. Chang, D. J. Kuck, and D. H. Lawrie, "On the Effective Bandwidth of Parallel Memories," IEEE Trans. on Comput. (May, 1977), pp. 480-490.

[Davi72] E. W. Davis, Jr., "A Multiprocessor for Simulation Applications," Ph.D. thesis, Dept. of Comput. Sci., Univ. of Ill. at Urb.-Champ., Rpt. No. 72-527, (June, 1972).

[Davi79] A. L. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality," AFIPS Conf. Proc. (June, 1979), pp. 1079-1086.

[DeMi74] J. B. Dennis, D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proc. 2nd Annual Symp. on Comput. Architecture (Dec., 1974), pp. 126-132.

[Flyn72] M. J. Flynn, "Some Computer Structures and Their Effectiveness," IEEE Trans. on Comput. (Sept., 1972), pp. 948-960.

[Grah76] R. L. Graham, "Bounds on the Performance of Scheduling Algorithms," in Computer and Job-Shop Scheduling Theory, E. G. Coffman, Jr., ed., John Wiley & Sons, Inc., New York (1976).

[KBCD74] D. Kuck, P. Budnik, S-C. Chen, E. Davis, Jr., J. Han, P. Kraska, D. Lawrie,

Y. Muraoka, R. Strebendt, and R. Towle, "Measurements of Parallelism in Ordinary FORTRAN Programs," IEEE Computer (Jan., 1974), pp. 37-46.

[KeLP79] R. M. Keller, G. Lindstrom, and S. Patil, "A Loosely-Coupled Applicative Multiprocessing System," AFIPS Conf. Proc. (June, 1979), pp. 613-622.

[Knut73] D. E. Knuth, The Art of Computer Programming, Vol. 1/Fundamental Algorithms (2nd Ed.), Addison-Wesley, Reading, MA, (1969).

[Kuck77] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," ACM Computing Surveys (Mar., 1977), pp. 29-59.

[Kuck78] D. J. Kuck, The Structure of Computers and Computations, Vol. I, John Wiley & Sons, Inc., NY, (1978), 611 pp.

[KuMC72] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-Up," IEEE Trans. on Comput. (Dec., 1972), pp. 1293-1310.

[KuSt79] D. J. Kuck, and R. Stokes, "The Burroughs Scientific Processor (BSP)," submitted for publication (1979).

[Lang76] T. Lang, "Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network," IEEE Trans. on Comput. (May, 1976), pp. 496-503.

[LaVo79] D. Lawrie, and C. Vora, "The Prime Memory System for Array Access," submitted for publication (1979).

[Lawr75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Trans. on Comput. (Dec., 1975), pp. 1145-1155.

[Murt66] J. C. Murtha, "Highly Parallel Information Processing Systems," in Advances in Computers, Vol. 7, F. L. Alt and M. Rubinoff, eds., Academic Press, NY (1966), pp. 1-116.

[Padu79] D. A. Padua, Ph.D. thesis, in preparation, Dept. of Comput. Sci., Univ. of Ill. at Urb.-Champ. (1979).

[Peas68] M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," Journ. of the ACM (April, 1968), pp. 252-264.

[Ston71] H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Comput. (Feb., 1971), pp. 153-161.

[Stre74] R. E. Strebendt, "Program Speedup Through Concurrent Record Processing," Ph.D. thesis, Dept. of Comput. Sci., Univ. of Ill. at Urb.-Champ., Rpt. No. 74-638, (Oct., 1974).

[Swan72] L. A. Swanson, "Simulation of a Tree Processor," M.S. thesis, Dept. of Comput. Sci., Univ. of Ill. at Urb.-Champ., Rpt. No. 72-503, (Jan., 1972).

[VaZi78] T. Vanaken, and G. Zick, "The X-Pipe: A Pipeline for Expression Trees," Proc. of the 1978 Int'l. Conf. on Parallel Processing (Aug., 1978), pp. 238-245.

[Wolf78] M. J. Wolfe, "Techniques for Improving the Inherent Parallelism in Programs," M.S. thesis, Dept. of Comput. Sci., Univ. of Ill. at Urb.-Champ., Rpt. No. 78-929, (July, 1978).

Fig. 1

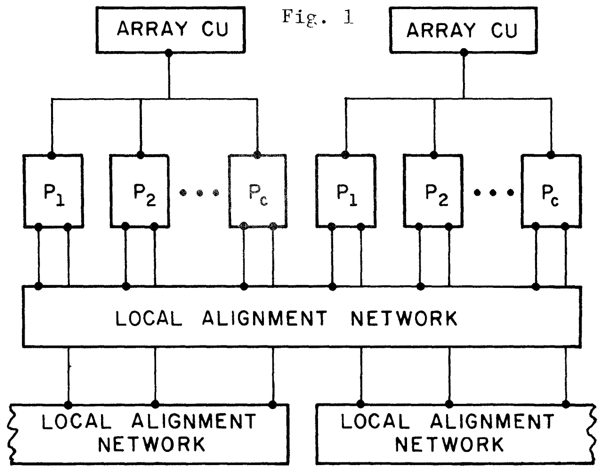
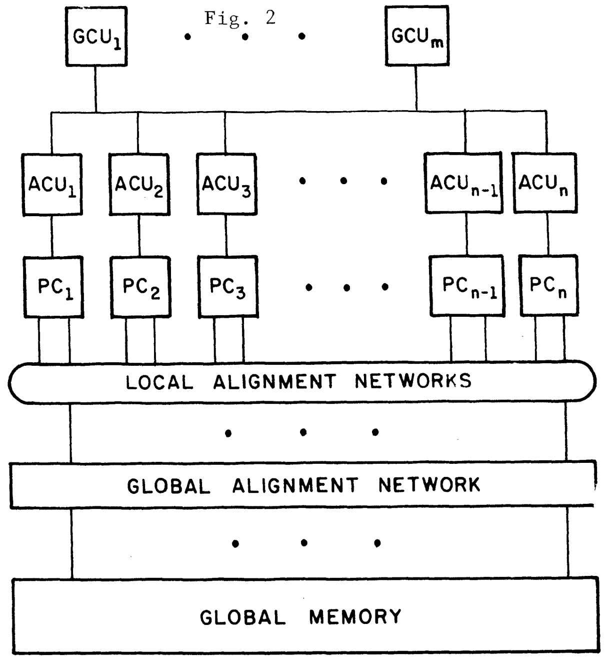


Fig. 2



PARALLEL PROCESSING OF HIGH-LEVEL LANGUAGE PROGRAMS*

Pong-sheng Wang and Ming T. Liu

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract -- The unifying effects of progress in language and computing theory, coupled with the advent of inexpensive microprocessor, has led many computer architects to consider the modular design of computer systems incorporating multiple microprocessors to implement various functions of the overall system. This paper is concerned with the parallel processing of high-level language programs by a multiple microprocessor system. The notion of a Parallel Execution String (PES) is first introduced as a representation of expressions for parallel execution. The PES approach is then applied to detect the parallelism at both the statement and the block levels. The advantage of the PES approach over the conventional "parallel by level" approach is discussed, and two algorithms are given to convert expressions into PES's. Finally, the organization of a multiple microprocessor system designed for parallel processing of PES's is presented. Code generation and optimization techniques are also discussed.

I. Introduction

The advent of inexpensive microprocessors has led many computer architects to consider the design of computer systems incorporating multiple microprocessors to implement various functions of the overall system [1]. Examples are the implementation of a general-purpose pipelined CPU [2], an emulator for the Sperry Univac 1108 mainframe [3], the Cm* [4], the Multi Associative Processor (MAP) [5], the Distributed Function Multiple Processor (DFMP) [6], the direct-execution computer organization [7], etc. This paper is concerned with the parallel processing of high-level language programs.

In this paper, we investigate the problems involved in parallel execution of arithmetic expressions in high-level programming languages. We are not concerned with the tree-height reduction techniques as proposed by Baer and Bovet [8], by Ramamoorthy and Gonzalez [9], by Squire [10], and by Stone [11]. Rather, we are dealing with a representation of parallelism, a way the expressions are executed, and an appropriate computer organization for carrying out the execution. The concept and notion of a Parallel Execution String (PES) is introduced as a representation of expressions for parallel execution. The PES approach is then applied to detect the parallelism at both the statement and the block levels in Section II and Section III, respectively. Two algorithms are then given to convert expressions into PES's and to schedule them for execution in Section IV. A machine organization suitable for carrying out parallel operations with our approach is described in Section V. Finally, code generation and code optimization techniques are discussed in Section VI and Section VII, respectively.

II. Parallel Processing of Expressions

It is well known that an expression can be represented by a rooted tree, with its internal nodes denoting operators and its external nodes variables and constants [12]. The son-nodes of a node are the operands of that node. For two operator nodes in the tree, if neither is the ancestor of the other, these two operators are independent and thus can be executed in parallel. In order to take advantage of the parallelism during the execution of an expression, there should be an intermediate form, into which the expression can be transformed, that shows the parallelism explicitly. One method (i.e. the conventional parallel-by-level approach [13]) is to group together the operations that are at the same level in the tree, and then to execute the operations in a group in parallel. The result of each operation is represented by some external symbol, which is not in the expression, and the symbol is then used as the operand of some operation on a higher level. The implication of this method is that we should have assumed that all operations take equal length of time, for otherwise there will be instances that the executions of some operations are delayed unnecessarily. However, this assumption does not hold for most real computers. Thus we propose another scheme to represent an expression in such a form that is appropriate for parallel executions regardless of the execution time of various operations. Later we will see that this scheme also has some additional advantages. First, let us give some definitions:

* Research reported herein was supported in part by grants AFOSR-77-3400 and NSF-MCS-77-23496.

definition

In an expression tree, an operator node is called

type 1 -- if all of its operands are variables or constants;
type 2 -- if exactly one of its operands is an operator; and
type 3 -- if more than one of its operands are operators.

If we consider only unary and binary operators, then the definition of type 3 becomes:

type 3 -- if it has two operands being operators.

For simplicity reasons, from now on we will only consider unary and binary operators. However, the proposed scheme can be easily extended to handle operators with more than two operands.

If we consider the type 1 nodes as starting points toward the root of the tree, then there are as many paths as type 1 nodes. Each path passes through a sequence of operators and uniquely defines a string of operators and operands, starting at a type 1 node and ending at the root. The string is to be called a Parallel Execution String (PES). These paths merge together on their ways toward the root and eventually converge at the root, where the last operation is performed. Note that each path has a type 1 node at one end and the root of the tree at the other end, and all of the intermediate nodes on each path are of type 2 or type 3. Type 3 nodes are merging points of paths, whereas the others are of type 2.

We observe that for each path, all the operations on that path have to be executed sequentially, beginning from the starting node and heading toward the root. However, any two nodes on two different paths prior to the merging point of these two paths (even though they are at different levels), are independent and thus can be executed in parallel. From these observations we can see that there exists more parallelism among the PES's of an expression than what can be exploited by the parallel-by-level approach. In a formal way, we can define a PES as follows:

Definition

A Parallel Execution String (PES) of an expression is a sequence

$D_1 \; T_1 \; D_2 \; T_2 \; \ldots . \; D_{n-1} \; T_{n-1} \; D_n$

where

(1) $T_1, T_2, \ldots . T_{n-1}$ are operator nodes in the tree, T_1 is a type 1 node, $T_2, \ldots,$ T_{n-1} are of type 2 or 3, and T_{n-1} is the root node;

(2) for i=1,2,...,n-2, T_{i+1} is the father node of T_i;

(3) D_1 and D_2 are operands of T_1;

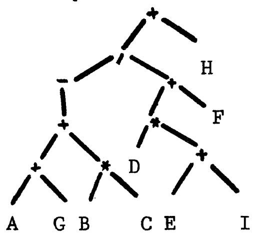
(4) for every i such that T_i is of type 2, D_{i+1} is either empty or an operand of T_i, depending upon whether T_i is an unary or binary operator;

(5) for every i such that T_i is of type 3, D_{i+1} is represented by "#k", where k is a number uniquely identifying the node T_i among all the type 3 nodes; and

(6) if T_i corresponds to a non-commutative operator X, and T_{i-1} is the right son-node of T_i, then T_i is represented by X'.

Figure 1 is an example of representing an expression in PES notations. (In Section IV, we will present an algorithm for converting an arithmetic expression into PES's.)

The expression -(A+G+B*C)/(D*(E+I)+F)+H can be represented as a tree:



It can be compiled into PES's as:

A + G + #1 - / #2 + H
B * C + #1 - / #2 + H
E + I * D + F /' #2 + H

Fig. 1 Example of translating an expression into PES's

With the definitions above, we thus propose a scheme to decompose an expression into PES's and execute them in parallel as follows. Whenever a processor is free, it will pick up one of the PES's that have not yet been taken, and start executing that PES from left to right. The first operation is always of type 1, which means all of its operands are variables or constants, so that it can be executed immediately. If it is a unary operator, it performs the operation and keeps the result in the processor for the next instruction. If it is a binary operator, it loads the first operand into the processor, performs the operation on the first and second operands, and keeps the result for the next instruction.

When the processor reaches a type 2 node, nothing will prevent it from performing that operation. Because this node has exactly one of its operands as an operator which has already been executed immediately prior to this node by the same processor. Note that the result of the previous operation is still kept in the processor and can be readily used as the operand for this type 2 node. The processor will execute the type 1 and type 2 nodes in the PES one by one, independent of other PES's.

When the processor reaches a type 3 operator, i.e. the operator with a #k operand, it will either continue executing the PES or save the partial result obtained thus far and then give up the PES, depending upon whether or not any of the other PES's, passing through the same type 3 node, has been executed up to this node. One possible machine organization to implement this scheme is described in Section V. Each PES will be executed only by one processor, although the processor may give up that PES before it reaches the end.

As mentioned earlier, the PES scheme has the advantage that it can exploit all the parallelism within an expression regardless of the operations that may take unequal length of time. In addition we also find that it has the following advantages:

1. The execution of a PES is done straightforward from left to right. No precedence relation between the operators needs to be considered, and it does not need any stack.

2. If the operations in the expressions are limited to unary and binary operations, it only needs a one-address instruction to perform the operation for each of type 2 and type 3 nodes, and two one-address instructions for each of type 1 nodes.

3. The partial results need not be stored for any type 1 and type 2 nodes, and they remain in the processors and will be used in subsequent operations. Even though the partial result may have to be stored for a type 3 node, it occurs only when the other path has not yet reached that node, so that storing the partial result in this case will not really increase the total execution time.

4. When a PES is assigned to a processor, a sequence of operations will be performed by the processor without the intervention of others, so that the execution can be done as fast as possible. This also makes it possible to employ some techniques (e.g., pipelining instruction fetching, decoding, and execution) to increase the execution speed further.

5. As will be seen in the next section, the PES scheme allows detection of parallelism between the operations in different statements.

III. Parallel Processing of Statements

It can be argued that expressions in most programs tend to be short and hence the scheme described above will not speed up the execution too much. Therefore, we need to go a step further to investigate how the PES scheme can be applied to exploit the parallelism among a block of statements.

To execute two statements in parallel, a condition must be satisfied: the input set and the output set of either statement cannot have any variable in common with the output set of the other statement. There are two methods that we can use to exploit the parallelism between statements to make the PES scheme more practical. The first method is to make the statements independent of each other by using a technique called forward substitution [12]. After applying forward substitution, the expressions become independent of each other and usually become more complicated and have more parallelism to be exploited, so that we can use the PES scheme to execute the statements in parallel.

In this paper we propose another method to detect the parallelism between the operations in different statements. Our approach is based on the concept introduced in Section II that an expression can be represented by one or more Parallel Execution Strings (PES's). The PES's of the statements in a block are tested for dependency and scheduled for execution according to the same condition mentioned above for statements, but here the tasks being tested and scheduled are the PES's generated from the statements, instead of the statements themselves. The advantage of this scheme is apparent in the example below. In the scheduling process, each PES will be assigned to a specific execution stage. Program execution will be done stage by stage. Those PES's which are assigned to the same stage can be executed in parallel.

The following is an example for this scheme. The original program consists of three assignment statements:

```
X := A * B * C + D
X := C * X / (D + E * (F - G))
A := D * E + C * B
```

The PES's for these three statements are:

```
A * B * C + D → X
C * X / #1 → X
F - G * E + D /' #1 → X
D * E + #2 → A
C * B + #2 → A
```

These PES's will be scheduled to two stages for execution:

Stage 1

```
A * B * C + D → X
F - G * E + D /' #1
C * B + #2
```

Stage 2

```
C * X / #1 → X
D * E + #2 → A
```

From the example above we can see that this scheme has the advantage that, even though Statement 1 and Statement 2 are not independent, a subexpression of Statement 2 can be executed concurrently with Statement 1. Similar situation also exists between Statement 1 and Statement 3.

IV. Compiling Algorithms

In this section we will present two algorithms: Algorithm A is to convert an arithmetic expression into Parallel Execution Strings, and Algorithm B is to detect the parallelism between PES's in a block and to schedule them for execution. Algorithm A requires only one pass through the input expression and each PES corresponds to one of the execution paths described in Section II. Algorithm B is applied to the PES's in a block one by one.

For simplicity reasons, Algorithm A will assume that the input expressions have been translated into reversed Polish strings. However, the algorithm can be easily modified to accept arithmetic expressions without any preprocessing. Figure 2 is the flowchart of Algorithm A. Since scanning a reverse Polish string corresponds to the post-order traverse of the expression tree, a stack is used in Algorithm A to keep the operands before their operators are scanned. If the operand stored in the stack is an operator, it will be represented as $n, where n is the highest numbered PES passing through the operator node.

When a variable or constants is scanned, it is always pushed onto the stack. When an operator is scanned, it will be handled according to its type. In Step 5 of Algorithm A, binary operators are processed in Case 1 through Case 3, for type 1 through type 3 nodes, respectively. For type 1 nodes (Case 1), a new PES is generated for the operation, and the operands on the stack are replaced by the string number. For type 2 nodes (Case 2), the operator and the constant or variable operand are appended to each of the PES's passing through the operator node, and the operand is deleted from the stack. For type 3 nodes (Case 3), the operator and a "#i" are appended to each of the PES's passing through the operator node, where i is a unique integer for the type 3 node, and the top two elements on the stack are replaced by the larger of the two. Unary operators are processed in Step 6. There are only two possible types for unary operators: type 1 and type 2. For type 1 nodes (Case 1), a new PES is generated for the operation and the operand on the stack is replaced by the string number. For type 2 nodes (Case 2), append the operator to each of the PES's passing through the operator node. No changes to the stack will be made.

Algorithm A

```
1. Convert the expression into a reverse Polish
   string. This procedure can be found in
   Hamblin [14] and is omitted here. Here we
   may use the tree-height reduction
   techniques [8-11] to obtain a modified
   Polish string.

2. Initialize i ← 1, j ← 1, where
     i is an index for temporary storage and
     j is an index for generated strings.

3. From left to right scan the Polish string
   for the next symbol S.
   If it is the end of string, the procedure is
   done, and String 1, String 2, ....,
   String (j-1) are outputs.

4. If the symbol S is an operand, push it onto
   the stack, then go to Step 3.

5. If the symbol S is a binary operator, the
   top two elements on the stack have the
   following possibilities:

   case 1  Both are operands:
     1) Create a new String j.
     2) Pop the top element off the stack, and
        let it become the third symbol of
        String j; then let the operator S be
        the second symbol of String j, pop
        stack again, and make it the first
        symbol of String j.
     3) Push the symbol $j onto the stack.
     4) j ← j+1.
   End case 1

   Case 2  One is an operand, and the other is
        a $k:
     Let e be the number such that $e is the
        next $'s appearing in the stack below
        $k. If no such $e exists, e is 0.
     1) If $k is on top of the stack and the
        operator is not commutative, append the
        "reverse operator" of S to each of:
        String k, String (k-1), ...,
        String (e+1). Otherwise, append the
        operator S to each of the same set of
        strings. (The "reverse operator" means
        that the order of its operands is
        reversed).
     2) Pop up the stack twice. Append the one
        which is an operand to the same set of
        strings as in 1.
     3) Push $k onto the stack.
   End case 2

   Case 3  Both of the top two elements on the
        stack are $'s:
     Let the top one be $n, the second one be
   $m, and m < n.
  Let e be the number such that $e is the
     next $'s appearing in the stack below
     $m. If no such $e exists, e is 0.
  1) Append the operator S to each of:
     String m, String (m-1), ...,
     String (e+1).
  2) If the operator S is commutative,
     append it to each of: String n,
     String (n-1), ..., String (m+1).
     Otherwise, append its reverse operator
     to each of these strings.
  3) Append a symbol #i to each of:
     String n, String (n-1), ..., String m,
     ..., String (e+1).
  4) Pop up the stack twice, and push $n
     onto the stack.
  5) i ← i+1.
End case 3

Go to Step 3.

6. If the symbol S is a unary operator, there
     are two possible cases:
  Case 1 Top of the stack is an operand:
    1) Create a new String j.
    2) Pop the top element of the stack. Let
       it become the first symbol of String j.
    3) Let the operator S become the second
       symbol of String j.
    4) Push $j onto the stack.
    5) j ← j+1.
  End case 1

  Case 2 Top of stack is $k:
    Let e be the number such that $e is the
       next $'s appearing in the stack below
       $k. If no such $e exists, e is 0.
    Append the operator S to each of:
       String k, String (k-1), ...,
       String (e+1).
  End case 2

  Go to Step 3.

                                End of Algorithm A
```

Figure 3 shows the flowchart of Algorithm B. Algorithm B will detect the parallelism across statements based on the PES scheme. It uses a symbol table to keep track of the variable names used in the block. For each variable in the symbol table, there are two fields associated with it: LAST-FETCHED and LAST-STORED. We will use LF(X) and LS(X) to denote the two fields associated with variable name X. These two fields contain the stage numbers at which a variable name was last fetched and last stored, respectively. Any PES's changing the variable X will be scheduled for a stage later than the larger of LF(X) and LS(X), because they cannot change the value of X until the prior fetching and storing operations of X are completed. Case 1 in Step 1 is to handle this situation. Any PES's using the variable X as input will be scheduled for a stage later than LS(X), because they have to wait until the storing operation of X is completed. This situation is handled by Case 2 in Step 1.

In Algorithm B, we use an array TMP whose size is at least the maximum number of temporary storage elements used in a block. TMP(K) keeps the largest stage number of the scheduled PES's that contain the symbol #k. TMP is used for eliminating unnecessary conflict checking and for assigning sub-expressions to the earliest stage possible. During the scanning of a PES, if we find any temporary storage symbol which has been scheduled before, there is no need to continue on the current PES. This situation is handled by Case 3 in Step 1. During the scanning of a PES, a variable STG is updated to the largest stage number in which variable conflicts will prohibit the execution of the current PES. When it comes to Step 2, the current PES has been determined not to be executed in or before stage STG. Therefore, the PES is scheduled for stage STG+1. In Step 3, the table entries of LF, LS, and TMP are updated to reflect the results of scheduling up to the current PES.

Algorithm B

```
0) Clear the TMP array.
   Apply the following steps to each of the
   PES in the block one by one.

   Set STG to 0.
1) Scan the PES from left to right. If it
   reaches the end of the PES, go to Step 2.
   Otherwise, get next symbol S.
   If S is an operator, ignore it and get the
   next symbol.
   If it is an operand, there are three
   possible cases:

   Case 1 If S is the output variable of the
          assignment:
     1.1) STG ← max[STG,LF(S),LS(S)]
     1.2) go to Step 1.
   End case 1

   Case 2 If S is an input variable of the
          assignment statement:
     1.3) STG ← max[STG,LS(S)]
     1.4) go to Step 1.
   End case 2

   Case 3 If S is a temporary storage for
          partial result, say #k:
     If TMP(k) = 0, go to Step 1.
     If TMP(k) > STG, go to Step 2.
     If 0 < TMP(k) ≤ STG, then
          S ← end of string,
          go to Step 2.
   End case 3

2) STG ← STG+1.

3) For each operand T to the left of S in the
   string, do the following:
   If T is the output variable of the
   assignment,
             LS(T) ← STG.
```

If T is an input variable of the assignment,

$LF(T) \longleftarrow \max[STG, LF(T)]$.

If T is a temporary storage for the partial result, say #j,

$TMP(j) \longleftarrow STG$.

End of Algorithm B

V. Machine Organization

A possible machine organization for parallel processing of PES's is shown in Figure 4. In this system, there is a variable number of identical microprocessors. Each microprocessor has its own program counter PC, accumulator AC, busy bit indicator B, and ALU. It is capable of fetching, decoding, and executing the instructions stored in the main memory. At the completion of a non-branching instruction, the microprocessor increments its own program counter and starts executing the next instruction fetched from the main memory. The instruction set of each microprocessor will have a special "operate-or-store" type of instructions, which is an ordinary operator except that the execution will depend upon the condition of the operand. If the operand in the PR memory (to be explained below) shows a "not ready" condition, the execution will not proceed any longer, the microprocessor will become free, and its B indicator will be reset to 0. Instead of performing the operation, it simply stores the content of its AC into the storage location addressed by the operand field of the instruction. If the operand is ready, the instruction will be performed as an ordinary operation. Instructions of this type are used for binary operators of which both of their operands are the results of some other operators, i.e. the type 3 nodes in the tree representation.

There is a Partial-Result (PR) memory in the system. Its purpose is to store the partial result obtained by the microprocessor that reaches a Type 3 node first. Each location in the PR memory has a status bit which indicates the availability of the partial result. The status bits are reset initially. The first "operate-or-store" instruction ac essing a PR location will store its partial result and set the status bit. The second "operate-or-store" instruction ac essing the same location will use its content and reset the status bit. While a microprocessor is executing the "operate-or-store" type instructions, no other microprocessors will be allowed to access the same location in the PR memory. This is to insure that only one of the two operands for a type 3 node will be stored into the PR memory.

In the system, there is an "Entry-Point-List" (EPL) memory, which consists of pointers pointing to the starting points of each PES. There is a pair of registers "Front" (F) and "Rear" (R), and an indicator "Need-Processor" (NP). F and R contain pointers to the EPL memory. Before an execution stage starts, F points to the beginning of the EPL for that stage, and R points to the beginning of the EPL for the next stage. F is incremented by one when a PES is taken by a processor for execution. The indicator NP indicates 1 when F is not equal to R, and 0 when F = R. Thus NP indicates whether or not the stage needs any more processors for execution.

There is an FR memory to store the pointers that will be loaded into F and R throughout the execution. The "Central-Program-Counter" (CPC) register is a pointer pointing to the FR memory for the current execution stage. Changing the execution sequence is done by changing the content of the CPC register.

In order to detect the completion of an execution stage, a "Number-of-Parallel-Strings" (NPS) register is used. When new values are load into F and R, NPS is set to a value equal to the difference between the values of R and F. It is then decremented at the completion of each PES. The completion of an execution stage is indicated by a zero in the NPS.

The control sequence for each microprocessor can be summarized as follows:

```
0. Idles. When NP=1, go to 1.
1. PC ← EPL(F), F ← F+1, B ← 1.
2. Fetch, decode, and execute instructions.
   Repeat until it encounters an IDLE
   instruction or the operand of an
   "operate-or-store" instruction is not ready.
3. NPS ← NPS-1, B ← 0, go to 0.
```

The control sequence for CPC, FR, F, R, and NPS can be summarized as follows:

```
0. R ← FR(CPC), CPC ← CPC+1.
1. F ← R, R ←FR(CPC), CPC ← CPC+1.
2. NPS ← R-F.
3. When NPS=0, go to 1.
```

The design of a multi-microprocessor system using Am2900 bit-slice microprocessors can be found in [15]. The system has the capability of performing parallel operations with the PES approach, and the capability of multi-processing sequential programs. It also has an efficient and flexible interrupt handling mechanism. The memory system is also designed to match the high throughput of the multi-microprocessor system.

VI. Code Generation

For each expression, several sequences of instructions will be generated. Each sequence of instructions corresponds to one of the PES's generated by Algorithm A described above, which also corresponds to one path in the tree representation. The last instruction of each sequence is always an IDLE instruction, which will set the processor free. However, if there is only one sequence in each of two consecutive execution stages, the first sequence will not have the IDLE instruction at its end. This is to eliminate the overhead in rescheduling processors in the case that only one processor is needed in each of the two consecutive execution stages.

For each sequence of instructions, an EPL entry, which is the address of the first instruction in the sequence, is generated in the EPL memory. For each execution stage, an FR entry, which is the address of the first EPL in the execution stage, is generated in the FR memory. Again, if there is only one sequence in each of two consecutive execution stages, there will be no FR entry generated for the second stage. This code generation scheme will insure that a strictly sequential program can be executed by a multi-processor computer system as fast as by a uni-processor computer system, while the programs with parallelism exploited by the PES scheme will be executed by a multi-processor computer system faster than by a uni-processor computer system.

The translations from the PES's to machine instructions are straightforward. It can be summarized as follows:

1. The first symbol in the PES is always an operand. Generate a LOAD instruction to load that operand into the accumulator. Continue scan the PES from left to right and do the following steps.

2. If it is a unary operator, generate an instruction to perform that operation on the content of the accumulator. The result of the operation will be in the accumulator.

3. If it is a binary operator, generate an instruction to perform the operation on the next symbol. The operation has an implied operand which is in the accumulator, and the result is stored in the accumulator.

4. In step 3, if the next symbol is a numerical symbol preceded by a #, then the instruction generated is a special "operate or store" instruction.

For example the PES: A + G * #1 / #2 + H will be translated into:

```
LDA    A     ;load A into accumulator
ADD    G     ;add G to accumulator
MOS    #1    ;multiply #1 to ACC or
             ;  store ACC to #1 then
             ;  idle
DOS    #2    ;divide ACC by #2 or
             ;  store ACC to #2 then
             ;  idle
ADD    H     ;add H to ACC
IDLE         ;end of PES, free
             ;  the processor
```

VII. Code Optimization

There are two types of code optimization that can be done with the proposed compiling scheme. The first one is to eliminate the redundant portion of a PES to save space. When the PES's generated from an expression are scheduled to be executed in more than one execution stage, the redundant portions of some PES's can be eliminated as follows. If a PES contains a partial result symbol (i.e. the symbol preceded by a "#") which also appears in some PES's scheduled for a later execution stage, then the substring to the right of that symbol can be eliminated. That substring is redundant because the execution of the PES will never proceed beyond that partial result symbol.

The other type of optimization is to reorder the PES's to minimize the total execution time. If the number of processors is equal to or greater than the number of PES's in an execution stage, the reordering of the PES's will not affect the total execution time. But if there are more PES's than processors, it might be advantageous to reorder the PES's. The tasks to be scheduled are the PES's in an execution stage. The execution time for each PES can be estimated by adding up the execution time for all the operators in the PES. The goal is to find a optimal non-preemptive m-processor schedule for the PES's in an execution stage to minimize the total execution time. However, it is known that the problem of finding the optimal non-preemptive schedule for n independent tasks with unequal length executed by m processors is NP-complete even for m 2 [16]. The problem is complicated even more by the fact that reordering the PES's will vary the execution time for each PES. Therefore, we can only find some near optimal scheduling strategies. A simple and intuitively sound strategy for this problem is the longest-processing-time scheduling, which gives the tasks with the longest processing time the highest priority. Simulations based on randomly generated PES's show that the longest-processing-time schedules have near-optimal performance, despite that the estimated execution time for a PES is usually not its actual execution time.

VIII. Conclusion

In this paper, we have first proposed the PES scheme for compiling the expressions in high-level language programs into an intermediate form suitable for parallel processing. We then went a step further to apply the scheme to a block of statements to exploit more parallelism. The advantages of this approach were briefly discussed. It should be noted that the PES approach we have proposed does not preclude the use of the tree-height reduction techniques and other techniques of program analysis for parallel processing [12].

Note that the PES scheme can be applied to a system with conventional machine instruction set as well as the indirect-execution high-level language computers [17]. In the latter, the PES notation is used as an intermediate language which is ready for parallel execution by the hardware. The translation from the source language to the intermediate language is not complicated. The intermediate language clearly indicates the parallelism exploited in the source language programs.

We also presented the organization of a multi-microprocessor system suitable for parallel processing of PES's. The code generation and optimization techniques for such a system were also discussed. The proposed machine organization and code generation method have the advantage that it minimizes the overhead of executing a strictly sequential program or program segment while the programs with parallelism can be executed faster on such a multi-processor system compared with a uni-processor system. The optimization techniques described here can be used with other machine-independent optimization techniques in compiling programs.

References

[1] A special issue on "LSI Modular Computers and Networks," Computer, Vol. 11, No. 7, July 1978.

[2] R.R. Ramseyer and A. van Dam, "A Multi-Microprocessor Implementation of a General Purpose Pipelined CPU," Proc. of the 4th Annual Symp. on Computer Architecture, pp.29-34, March 1977.

[3] B.R. Borgerson, et al., "Mainframe Implementation with Off-the-Shelf LSI Modules," Computer, Vol.11, No.7, pp.42-48, July 1978.

[4] S.H. Fuller, et al., "Multi-Microprocessors: An Overview and Working Example," Proc. IEEE, Vol.66, pp.216-228, Feb. 1978.

[5] G.J. Nutt, "Microprocessor Implementation of a Parallel Processor," Proc. of the 4th Annual Symp. on Computer Architecture, pp.147-152, March 1977.

[6] F.C. Colon, et al., "Coupling Small Computers for Performance Enhancement," Proc. NCC, Vol.45, pp.755-764, June 1976.

[7] Y. Chu, "An LSI Modular Direct Execution Computer Organization," Computer, Vol.11, No.7, pp.69-76, July 1978.

[8] J.L. Baer and D.P. Bovet, "Compilation of Arithmetic Expressions for Parallel Computations," Proc. IFIP Congr. 1968, pp.340-346.

[9] C.V. Ramamoorthy and M.J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," Proc. FJCC 1969, pp.1-15.

[10] J.S. Squire, "A Translation Algorithm for a Multiprocessor Computer," Proc.18th ACM Nat. Conf., Denver, Colorado, 1963.

[11] H.S. Stone, "One-Pass Compilation of Arithmetic Expressions for a Parallel Processor," Communications of ACM, Vol.10, No.4, pp.220-223, April 1967.

[12] D.J. Kuck, "Parallel Processing of Ordinary Programs," in Advances in Computers, Vol. 15, pp.119-179, Academic Press, 1976.

[13] H. Hellerman, "Parallel Processing of Algebraic Expressions," IEEE Trans. on Elec. Computers, Vol. EC-15, No.1, pp.82-91, Feb. 1966.

[14] C.L. Hamblin, "Translation to and from Polish Notation," Computer Journal, Vol.5, pp.210-213, Oct. 1962.

[15] Pong-sheng Wang and Ming T. Liu, "A Multi-Microprocessor System for Parallel Computations," to appear in Proc. of the ACM Second Annual Symp. on Small Systems, Oct. 1979.

[16] M.R. Garey and D.S. Johnson, Computers and Intractability: A guide to the theory of NP-completeness, W.H. Freeman and Company, 1978.

[17] Y. Chu (editor), High-level Language Computer Architecture, Academic Press, 1975.

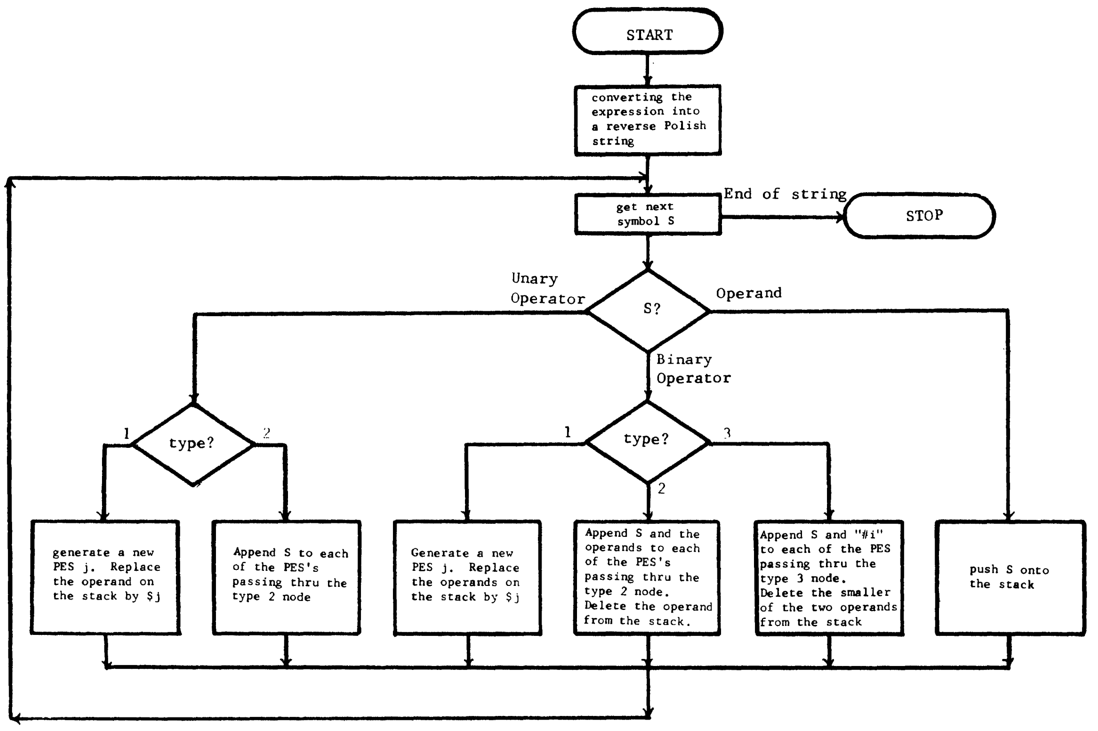

START
converting the expression into a reverse Polish string
get next symbol S
End of string
STOP
Unary Operator
S?
Operand
Binary Operator
1
type?
2
1
type?
3
2
generate a new PES j. Replace the operand on the stack by $j
Append S to each of the PES's passing thru the type 2 node
Generate a new PES j. Replace the operands on the stack by $j
Append S and the operands to each of the PES's passing thru the type 2 node. Delete the operand from the stack.
Append S and "#i" to each of the PES passing thru the type 3 node. Delete the smaller of the two operands from the stack
push S onto the stack

Figure 2 Flowchart of Algorithm A

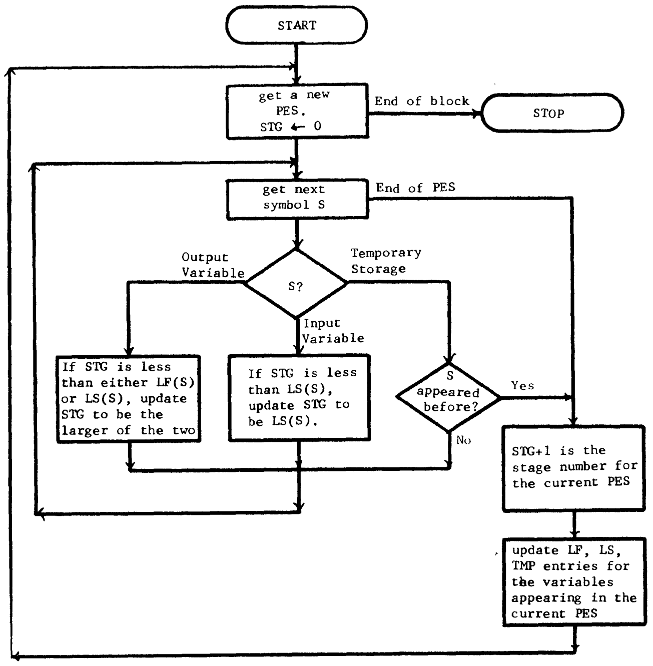

START
get a new PES. STG ← 0
End of block
STOP
get next symbol S
End of PES
Output Variable
S?
Temporary Storage
Input Variable
If STG is less than either LF(S) or LS(S), update STG to be the larger of the two
If STG is less than LS(S), update STG to be LS(S).
S appeared before?
Yes
No
STG+1 is the stage number for the current PES
update LF, LS, TMP entries for the variables appearing in the current PES

Figure 3 Flowchart of Algorithm B

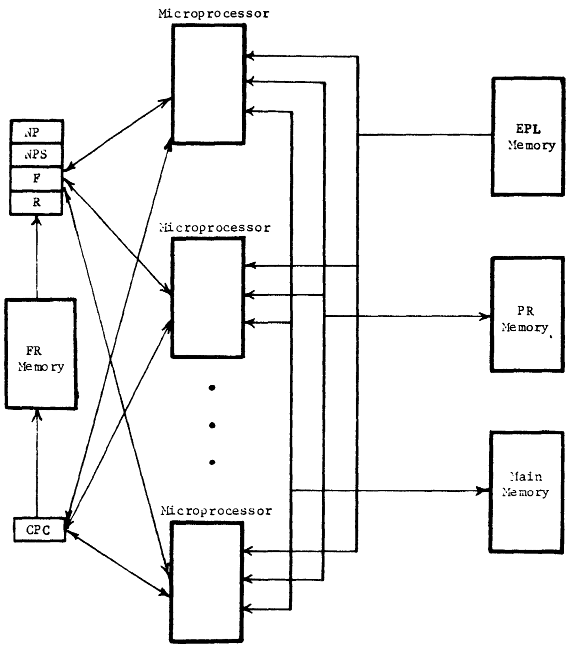


Figure 4 Machine Organization

A FLOW ANALYSIS PROCEDURE FOR THE TRANSLATION OF HIGH LEVEL LANGUAGES TO A DATA FLOW LANGUAGE*

Stephen J. Allan**
Arthur E. Oldehoeft
Department of Computer Science
Iowa State University
Ames, Iowa 50011

Abstract -- A data flow analysis procedure is described which may be used in the translation of high level languages to parallel target languages. The technique analyzes the data dependencies which exist between statements in a high level program and constructs an intermediate form amenable to optimizing transformations and code generation. An example illustrates how information provided by the analysis may be used in generating code for a highly parallel data flow machine. Within the framework of the described data flow analysis procedure, extensions to the high level language are discussed which allow for higher utilization of the data flow machine.

Introduction

The user acceptance of a data flow computer to some extent will be influenced by the ability of the user to program application programs in a high level language. This calls for a translator to translate the high level language to a data flow language. This paper describes a technique for data flow analysis which may be used in this translation. The technique is useful for a broad class of high level languages which include the sequential von Neumann type high level languages in common use as well as nonsequential high level languages such as the so called single assignment languages [1,6,7,16]. In order for this flow analysis technique to be applicable to single assignment languages, it is required that the definition of a value precede any use of that value in the text of the high level program. In some single assignment languages this is required by the definition of the language while others would require a preprocessor to topologically order the statements.

The target language could be the instruction set provided by any of a variety of parallel architectures. In this paper, however, the assumed target machine is a highly parallel data driven machine [5,9,15,17]. The underlying assumption behind a data driven machine is that a program is not a sequence of instructions that cause changes to a memory space, but instead a program is a collection of computations related to each other by the need for data values that are produced and consumed. The order of execution of the computations is not directly stated by the program but rather by the partial ordering provided by the data dependencies. The purpose of the data flow analysis is to determine this partial ordering so that target code may be generated to exploit the inherent parallelism in the program. While other techniques [2,4,10,11] provide a general basis for the analysis, additional information must be gathered to generate code for such a parallel execution environment.

A compiler using the data flow procedure described in this paper has been implemented to translate programs written in a typical von Neumann type high level language to the language of a simulated data flow machine [14]. The discussion of the flow analysis technique is presented with this implementation used as an example. Some extensions to the language are then discussed which rely on this same technique of flow analysis but allow higher utilization of a data flow machine.

High Level Language and Data Structure for Internal Representation

A high level programming langauge, devoid of features incompatible with the notion of functionality of programs, was designed as a vehicle to help achieve the objectives of the research [14]. The language is similar in appearance to Pascal but, at the moment, has about the same expressive power as Algol 60. A program consists of a main procedure with declarations, including the definitions of other procedures and functions, and a body of statements. Procedures and functions may be recursively applied. Statements in the language include assignment, conditional (i.e., if-then and if-then-else), iterative (i.e., while-do, repeat-until and for), procedure call, and input and output. The for statement is translated directly into a while-do statement and no further reference to it is made in the paper. Parameters of procedure calls and definitions must have an "in" and/or "out" directionality

*Research reported herein was supported by the National Science Foundation under NCS77-02467 and by the Sciences and Humanities Research Institute of Iowa State University.

**Present address: Computer Science Department, Colorado State University, Ft. Collins, Colorado 80523.

attribute. Integer, real and boolean data types are currently supported along with a full complement of operators and intrinsic functions which can operate on identifiers declared with the above data types. The array is the only data structuring facility which exists at the present time. An array may have any number of dimensions and may be dynamically declared at run time upon procedure entry. Transfer of control (e.g., goto's) and global references are not allowed in the language.

The compiler translates the source text of a program written in the high level language into an intermediate form. This intermediate form is recorded as a table of relatively high level entries and is hereafter referred to as the IFT. Initially, the IFT is a representation of the tree structure of the high level program. After later phases add data flow information, the IFT becomes a representation of a data flow graph, amenable to optimizing transformations and code generation.

Each entry in the IFT consists of four major fields as shown in Figure 1.

TYPE	I	O	TREE

Figure 1 Entry in the IFT

TYPE is the field that indicates the type of statement represented by the entry; I is the set of input values for the entry; O is the set of output values for the entry; and TREE is the syntax tree for the entry, if one exists.

A separate IFT is generated for every procedure and function defined in the program. Entries are created in the IFT during the parse phase and are threaded to represent the ordering of the statements as they were encountered in a sequential scan of the high level program. Each high level statement results in the generation of one or more entries in the IFT where the data flow information is maintained. The general form of the different high level statements and the types of entries in the IFT generated by the compiler are given in Figure 2. A simple high level statement (i.e., assignment, procedure or function call, and procedure or function heading) generates only one IFT entry in which the data flow information for that statement will be maintained. For compound statements that are conditionally executed (i.e., bodies of while, repeat and if constructs) an "interface" entry is generated to maintain the cumulative data flow information for the condition and block of statements within the body. These interface entries are denoted in Figure 2 by "if", "while" and "repeat". An interface represents a staging area for the values used by the condition and block and for the values defined by the block. All information used by the block is conceptually passed from preceding statements through the interface and all information defined by the block is conceptually passed to succeeding statements through the interface. This allows for local flow analysis of blocks of statements. A "close" or "end" entry is generated to mark the end of a repeat, while, if, procedure or function block and contains no data flow information. The "then" and "else" entries are also generated to mark the start of the then and else bodies. The input/output statement generates an "input"/"output" entry in the IFT for each component in its list. If an input or output component involves an implied do loop, a while loop is generated with corresponding entries placed in the IFT.

High level statement	Entries in the IFT
procedure (function) statement list end	procedure (function) entries for statement list end
input/output a1,...,an	input/output for a1 . . . input/output for an
x := expression	assign

if condition
then statement list1
{else statement list2}
	if
	condition
	then
	entries for statement list1
	{else
	entries for statement list2}
	close

while condition
statement list
end
	while
	condition
	entries for statement list
	close

repeat
statement list
until condition
	repeat
	entries for statement list
	condition
	close

x(in(...),out(...))	call

Figure 2 High level statements and entries in the IFT

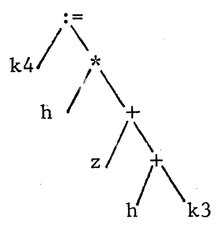
Figure 3(a) shows a segment of a high level program using the Runge-Kutta method for finding the numerical solution to the ordinary differential equation $y'=x+y$ with $y(0)=1$. Figure 3(b) shows the corresponding set of entries in the IFT. The actual data flow information for these entries will be illustrated later. Figure 3(c) shows the syntax tree for entry 8.

```
x := 0
y := 1
i := 1
repeat
  z := x+y
  k1 := h*z
  k2 := h*(z+h/2+k1/2)
  k3 := h*(z+h/2+k2/2)
  k4 := h*(z+h+k3)
  y := y+(1./6.)*(k1+2*k2+2*k3+k4)
  x := x+h
  i := i+1
until n < i
```

(a) High level program segment

	TYPE	TREE		TYPE	TREE
0.	assign	T0	7.	assign	T6
1.	assign	T1	8.	assign	T7
2.	assign	T2	9.	assign	T8
3.	repeat		10.	assign	T9
4.	assign	T3	11.	assign	T10
5.	assign	T4	12.	condition	T11
6.	assign	T5	13.	close	

(b) IFT entries



(c) Syntax tree T7

Figure 3 High level program segment and IFT entries

Data Flow Analysis

The data flow analysis technique presented here assumes the IFT as an internal form of the program and also assumes that the target machine provides direct semantic support for those operations which are implicit in the high level program (i.e., arithmetic operators, array selection and appendage, procedure call). The general technique is a top-down recursive descent flow analysis [10]. Since the IFT is a highly structured representation of the program and since procedures are free of side effects, the flow analysis is highly simplified.

The total data flow analysis is performed in three phases. In the first phase, the input and output sets for each statement are collected. The second phase generates the use and definition information about each value and the third phase performs the live value analysis. The three phases are outlined in subsequent sections and described in detail elsewhere [3].

Collection of Input and Output Sets

This section describes the generation of input and output sets for each type of entry in the IFT.

The calculation of the input set and the output set for a non-interface IFT entry is straightforward, as illustrated in Figure 4.

The input and output sets for interface entries for compound blocks of statements (if, while or repeat) are somewhat more complicated, depending on conditionally defined values and values which are used in a block prior to their redefinition.

Entry Type	Input and Output Sets for Entry E0
assign	I(E0) = {x:x is referenced by the assignment statement} O(E0) = {x:x is defined by the assignment statement}
condition	I(E0) = {x:x is referenced by the condition} O(E0) = ∅ (null set)
input	I(E0) = {x:x is referenced by the input statement} ∪ {input filename} O(E0) = {x:x is defined by the input statement} ∪ {input filename}
output	I(E0) = {x:x is referenced by the output statement} ∪ {output filename} O(E0) = {output filename}

call, function or procedure	I(E0) = in(S)
	O(E0) = out(S)
	where in(S) and out(S) are the sets of parameter values in the high level statement with the corresponding directionality attribute.

Figure 4 Calculation of input and output sets for single IFT entry blocks

Let E = E1,...,En be any set of entries in the IFT corresponding to a compound block of statements. Disregarding conditionally defined values, this set of sequential entries, E, has its input and output sets defined to be

$$I(E) = I(E1) \cup \{\bigcup_{i=2}^{n} (I(Ei) - \bigcup_{j=1}^{i-1} O(Ej))\} \text{ and}$$

$$O(E) = \bigcup_{i=1}^{n} O(Ei).$$

This means that the input set for E consists of values which are used before their redefinition within the corresponding block of statements being processed and the output set contains all values defined within the block.

Suppose that x is conditionally defined in such a block and x is used in some subsequent computation. The value for this use of x may depend on its conditional definition or on some previous definition. This is portrayed in Figure 5(a). In order to simplify the data flow analysis, the previous definition of x is added to the input set to denote unconditional production of the most recent value of x whether it comes from within the conditional block or from the previous definition. This is portrayed in Figure 5(b).

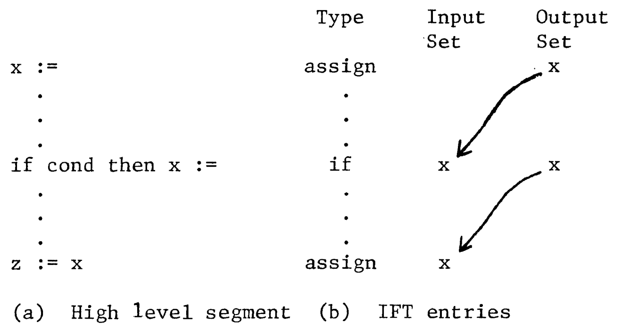


Figure 5 Conditional definition

The calculation of the input and output sets for interface entries representing if, while and repeat blocks is given in Figure 6. An input set contains the upward exposed uses of values appearing in the block of statements constituting the body along with values which are conditionally defined by the block. The collection of this information is readily implemented by a top-down recursive descent parse.

Interface entry E0 for block of the form	Input and Output Sets

if C then E	I(E0) = I(C) ∪ I(E) ∪ O(E)
	O(E0) = O(E)

if C then E1 else E2	I(E0) = I(C) ∪ I(E1) ∪ I(E2) ∪ {O(E0) - (O(E1) ∩ O(E2))}
	O(E0) = O(E1) ∪ O(E2)

while C do E	I(E0) = I(C) ∪ I(E) ∪ O(E)
	O(E0) = O(E)

repeat E until C	I(E0) = I(E) ∪ (I(C) - O(E))
	O(E0) = O(E)

Figure 6 Calculation of input and output sets for interface entries

Figure 7 shows the IFT entries for the Runge-Kutta program segment (presented in Figure 3) expanded to include the input and output sets.

Entry		Input Set	Output Set
0.	assign	∅	x
1.	assign	∅	y
2.	assign	∅	i
3.	repeat	x,i,y,h,n	z,k1,k2,k3,k4,y,i,x
4.	assign	x,y	z
5.	assign	h,z	k1
6.	assign	h,z,k1	k2
7.	assign	h,z,k2	k3
8.	assign	h,z,k3	k4
9.	assign	y,k1,k2,k3,k4	y
10.	assign	x,h	x
11.	assign	i	i
12.	condition	i,n	∅
13.	close		

Figure 7 Input and output sets calculated for IFT

<u>Generating the Use and Definition Sets</u>

After the input and output sets have been constructed, the dependency relationships must be established between the definition of values and their subsequent use. This is done by matching names of values (names of identifiers in original source code) in the corresponding input and output sets of entries in the IFT. For all entries producing a value, a list is constructed showing all the entries where that value is used. Thus, use(x,Ei) denotes the set of entries at the same nesting level as Ei which use the value of x defined in Ei. For each value x used by an entry Ej, an ordered list def(x,Ej), having maximum length of two, is constructed showing the entries where the value was defined. The entry which defines a value can be found by a backward scan of the preceding entries until the value appears either in an output set of an entry (corresponding to a statement at the same level within the same block) or in the interface entry of the enclosing statement. If it is not found, the value has no definition. If x is used in Ej, then def(x,Ej) = (a,(b,c)) denotes the definition set of x. For a non-interface entry, this set consists of only a, the first element. For an interface entry, this set contains two elements a and (b,c). The element a identifies where the value was most recently defined outside the block and (b,c) identifies the last definition within the block. Except for the case of an if-then-else, c is null. A member of the set is denoted by def(x,Ej)(u) where u is one of a, b, or c.

The use and definition analysis is presented in Figure 8 as a recursive top-down procedure which produces the use and def sets for the entire IFT. Suppose H denotes the interface entry for the block of statements to be analyzed and E denotes the set of entries corresponding to statements within the block. This procedure modifies the IFT entries by attaching the use and def sets.

```
procedure useanddef(in(E,H),out(E,H))
  elseflag := false
  for i = 1 to |E| do
    if TYPE(Ei) = (else or then) then
      if (TYPE(Ei) = else) then
        elseflag := true
      end if
    else
      for each x ε I(Ei) do
        finddef(in(i,x,E,H),out(E,H))
      end for
      for each x ε O(Ei) do
        if x  ε O(H) then
          if elseflag then
            def(x,H)(c) := Ei
          else def(x,H)(b) := Ei
          end if
        end if
      end for
      if TYPE(Ei) = (while or repeat or if) then
        U := {x:x is a subblock of Ei}
        useanddef(in(U,Ei),out(U,Ei))
      end if
      if TYPE(Ei) = (while or repeat) then
        for each x ε I(H) - O(H) do
          def(x,H)(b) := H
          use(x,H) := use(x,H) ∪ H
        end for
      end if
      if TYPE(Ei) = (while or repeat or if or
                procedure or function) then
        for each x ε O(H) do
          if def(x,H)(b) ≠ ∅ then
            SB := def(x,H)(b)
            use(x,SB) := use(x,SB) ∪ H
          end if
          if def(x,H)(c) ≠ ∅ then
            SB := def(x,H)(c)
            use(x,SB) := use(x,SB) ∪ H
          end if
        end for
      end if
    end if
  end for
end procedure

procedure finddef (in(i,x,E,H),out(E,H))
  found := false
  for j = i-1 to 1 while not found do
    if x ε O(Ej) then
      def(x,Ei)(a) := Ej
      use(x,Ej) := use(x,Ej) ∪ Ei
      found := true
    end if
  end for
  if not found then
    if x ε I(H) then
      def(x,Ei)(a) := H
      use(x,H) := use(x,H) ∪ Ei
    else def(x,Ei)(a) := ∅
    end if
  end if
end procedure
```

Figure 8 Use and definition analysis procedure

Figure 9 shows the IFT entries for the Runge-Kutta program segment (given in Figure 3) expanded to include the use and definition information.

Entry	Input Set			Output Set	
	val	def	use	val	use
0. assign	∅			x	3
1. assign	∅			y	3
2. assign	∅			i	3
3. repeat	x	0,10	4,10	x,k1,	?
	i	2,11	11	k2,k3,	?
	y	1,9	4,9	k4,y,	?
	h	?	5,6,7,	z,i	?
			8,10		
	n	?	12		
4. assign	x	3		z	3,5,6,
	y	3			7,8
5. assign	h	3		k1	3,6,9
	z	4			
6. assign	h	3		k2	3,7,9
	z	4			
	k1	5			
7. assign	h	3		k3	3,8,9
	z	4			
	k2	6			
8. assign	h	3		k4	3,9
	z	4			
	k3	7			
9. assign	y	3		y	3
	k1	5			
	k2	6			
	k3	7			
	k4	8			
10. assign	x	3		x	3
	h	3			
11. assign	i	3		i	3,12
12. condition	i	11		∅	
	n	3			
13. close					

Figure 9 Use and definition for IFT

Live Value Analysis

Live value analysis provides necessary information for certain optimizing transformations. A value is defined to be live at a given point in a program if it has a subsequent use. Live value analysis requires information gathered in the first two phases of data flow analysis. Associated with each value x in the output set of entry Ei is a boolean value, live(x,Ei), which indicates whether x is live at this point.

A top-down recursive descent algorithm called liveanalysis is used to generate this information The algorithm analyzes values in the output set for each entry starting with the first entry of a procedure. If an interface entry is encountered, a recursive call on the procedure liveanalysis is made (propagating known live information inward) to analyze entries in the inner nesting level.

The algorithm for live value analysis is given in Figure 10. This procedure modifies each value in the output set of the IFT entries by attaching a boolean value. The initial call would take the form liveanalysis(in(E,H),out(E,H)) where E is the set of entries corresponding to a procedure H. Figure 11 shows the IFT for the Runge-Kutta program segment (given in Figure 3) expanded to include the live value analysis information.

```
procedure liveanalysis(in(E,H),out(E,H))
  for i = 1 to |E| do
    for each x ε O(Ei) do
      live(x,Ei) := false
      if use(x,Ei) ≠ ∅ then
        if use(x,Ei) = {H} then
          if TYPE(H) = (while or repeat)
             and x ε I(H) then
            live(x,Ei) := true
          else
            if TYPE(H) = (procedure
                             or function) then
              live(x,Ei) := true
            else live(x,Ei) := live(x,H)
            end if
          end if
        else live(x,Ei) := true
        end if
      end if
    end for
    if TYPE(Ei) = (while or repeat or if) then
      U := {x:x is a subblock of Ei}
      liveanalysis(in(U,Ei),out(U,Ei))
    end if
  end for
end procedure
```

Figure 10 liveanalysis procedure

Entry	Input Set	Output Set	
	val	val	live
0. assign	∅	x	true
1. assign	∅	y	true
2. assign	∅	i	true
3. repeat	x,i,y,h,n	z,k1,	?
		k2,k3,	?
		k4,y,	?
		i,x	?
4. assign	x,y	z	true
5. assign	h,z	k1	true
6. assign	h,z,k1	k2	true
7. assign	h,z,k2	k3	true
8. assign	h,z,k3	k4	true
9. assign	y,k1,k2,k3,k4	y	true
10. assign	x,h	x	true
11. assign	i	i	true
12. condition	i,n	∅	
13. close			

Figure 11 Live values for the IFT

Language Extensions for the Exposure of Parallelism

In this section, extensions to the high level language are discussed which allow for higher utilization of a data flow machine. The concepts of the "forall" statement [1], the "stream" data type [13,17] and array to scalar functions are introduced to allow more efficient execution. The flow analysis described in the previous section remains basically the same.

The forall statement, depending on its implementation, allows for significant reduction in the order of the computation. The intent of the forall is that the invocations of the body are independent so that, in theory, all may execute in parallel. The syntax of the forall and the corresponding IFT entries are shown in Figure 12.

High level statement	Entries in the IFT

forall forall_cond do
  statement list
end
	forall
	  forall condition
	  entries for statement
	    list
	close

Figure 12 Forall statement and IFT entries

The input and output sets are calculated in the same manner as the iterative for statement. It is assumed that the body of the forall statement obeys the single assignment rule which states that a value may be assigned only once during the execution of the program. Thus, any value used on a right hand side within the body of the forall must be computed outside the forall statement. The only value that can be output from a forall statement is an array.

The implementation of the forall statement is dependent on the underlying data flow architecture. Possible implementations include unwinding of loops by the architecture [5] or by recursion or the use of special hardware functions such as compose and decompose [15].

Loop decomposition, described by Lo [12] and extended by Allan [3], can be used as an optimization technique at compile time to transform some iterative statements to forall statements. Every value used in right context in the body of the loop is examined to determine if it depends on a value computed in a previous iteration. If the iterations are found to be independent, the loop is immediately transformed into a forall statement. Otherwise, attempts are made to break these data dependencies through forward substitution, saving old values of an array in a temporary array, making scalars into arrays and rearranging the code (maintaining the precedence relations that previously existed). If the iterations are still dependent, the loop is decomposed into smaller loops by finding a partition of the statements in the loop such that the precedence relations between the statements are still preserved. If the partitioning is successful, each partition is treated as a single loop and the process is repeated. Loops which cannot be partitioned are executed in the original iterative manner. Figure 13 illustrates a simple decomposition of a loop.

```
i := 1;                     forall i in (1,n) do
while i <= n do               b'(i) := b(i)
  b(i) := a(i) + c(i+1)     end;
  c(i) := b(i+1)            forall i in (1,n) do
end                           b(i) := a(i) + c(i+1)
                            end;
                            forall i in (1,n) do
                              c(i) := b'(i+1)
                            end

(a) before                  (b) after
```

Figure 13 Example of loop decomposition

As a second technique, streams appear to offer some advantages. When a forall is not applicable, streams might still be used to reduce the coefficient of the order of the computation through pipelining sections of the data flow program. Streams, combined with recursion, can result in a reduction in the order of the computation.

A third technique for the higher utilization of a data flow machine is the introduction of certain functions (e.g., sum, product) which map an array or stream to a scalar value. A recursive implementation may be used to reduce the order of a computation from O(n) to O(log2 n), where n is the length of the array or stream.

Code Generation

This section illustrates how the information provided by the data flow analysis may be used in generating code for a highly parallel data flow machine.

The data flow program can be viewed as a directed graph consisting of nodes and edges [8]. Figure 14 shows a data flow graph for the Runge-Kutta program. A node represents a base language operation and an edge represents a data dependency between nodes. The normal firing rules allow a node to execute whenever there is a value on each of its input edges and no tokens on any of its output edges. The value produced by the node is placed on each of the output edges.

Special firing rules exist for merge gate and true and false gate operations, which support conditional execution. A merge gate of the form (T F)

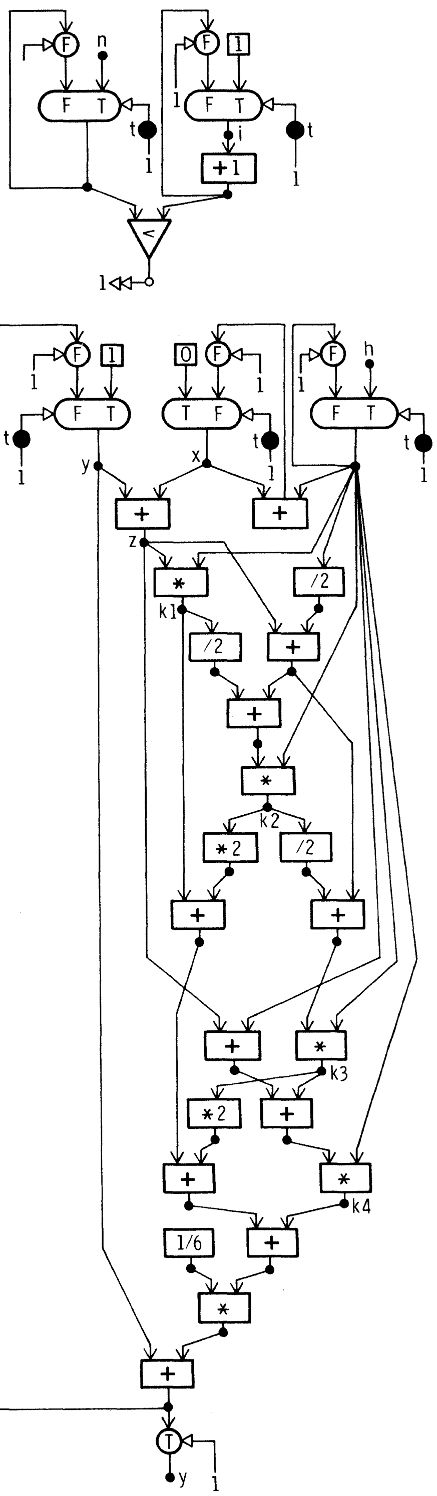


Figure 14 Data flow graph of Runge-Kutta program

takes a data value from its T input edge or F input edge (denoted by open arrow heads) depending on a boolean control value present on its control edge (denoted by a closed arrow head). A true gate of the form Ⓣ allows a data value to be passed from its input edge (open arrow head) to its output edge if a true boolean control value is received on its control edge (closed arrow head). The data value is destroyed if a false boolean control value is received. Analogous firing rules hold for a false gate of the form Ⓕ. Boolean control values are produced by relational nodes.

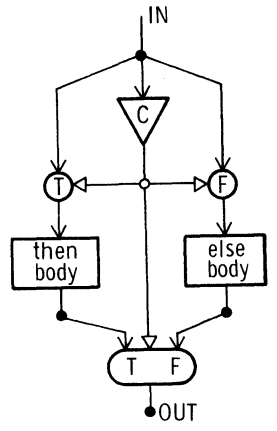
The target language generated by the compiler is a set of instructions for the data flow machine, which is simply a linear representation of the data flow graph. Following the flow analysis described in the previous section, it is conceptually easy to generate the data flow graph. For entries in the IFT, inter-entry dependencies have been established by the use and definition analysis. Intra-entry dependencies are established according to the syntax tree (TREE) of the IFT entry. Generalized code templates for three constructs, found in conventional high level languages, appear in Figure 15. In each of the graphs, the edges labeled IN or OUT indicate the sets of data values that pass into and out of the specified construct. The set of values represented by the IN edge can be found in the input set of the interface entry and the set of values represented by the OUT edge can be found in the output set of the same entry. Figure 14 illustrates the details for a repeat-until construct.

Conclusions

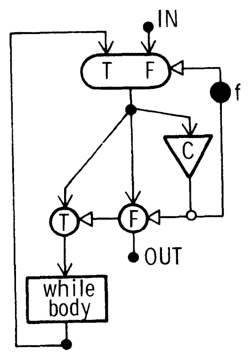
This paper has presented a data flow analysis procedure which is useful in the translation of high level languages to the machine language for a highly parallel data flow processor. This technique could be used for a variety of parallel architectures and is similar to flow analysis techniques used for code optimization on a conventional machine. On one hand, the algorithm presented in this paper is generally simpler than other techniques due to the enforcement of structured programming constructs and the elimination of side effects. On the other hand, the algorithm maintains more data flow information than do other techniques since the primary purpose of the analysis is to generate code for a data flow machine.

A compiler has been implemented and is fully operational using this technique in the generation of code for execution on a simulated data flow machine.

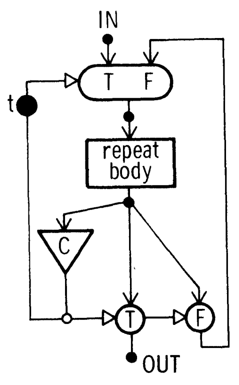
Any parallelism performance data?



(a) if-then-else



(b) while-do



(c) repeat-until

Figure 15 Generalized code templates

References

[1] W. B. Ackerman and J. R. Dennis, VAL--A Value Oriented Algorithmic Language, Preliminary Reference Manual, Laboratory for Computer Science, M.I.T., Cambridge, Mass. (1978).

[2] A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Mass., (1977), pp. 406-477.

[3] S. J. Allan, Reduction of Data Dependencies in High Level Programs, Ph.D. Thesis, Iowa State University, Ames, Iowa, (1979).

[4] F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," Comm. ACM (March, 1976), pp. 137-147.

[5] Arvind and K. P. Gostelow, A Computer Capable of Exchanging Processor Elements for Time, Department of Computer Science, University of California, Irvine, TR-77, (January, 1976).

[6] Arvind, K. P. Gostelow, and W. Plouffe, Programming in a Viable Data Flow Language, Department of Computer Science, University of California, Irvine, TR-89, (August, 1976).

[7] D. D. Chamberlin, Parallel Implementation of a Single-Assignment Language, Ph.D. Thesis, Stanford University, (1971).

[8] J. B. Dennis, First Version of a Data Flow Procedure Language, Project MAC, M.I.T., Cambridge, Mass., Computation Structures Group Memo 93-1, (August, 1974).

[9] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," The 2nd Annual Symposium on Computer Architecture, IEEE, New York, (1975), pp. 126-132.

[10] M. S. Hecht, Flow Analysis of Computer Programs, Elsevier North-Holland, New York, (1977).

[11] M. S. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," SIAM J. Computing, (April, 1975), pp. 519-532.

[12] D. Lo, Transformation of Loop Programs for Parallel Execution, Ph.D. Thesis, The University of Michigan, (1976).

[13] D. Morris and P. C. Treleaven, "A Stream Processing Network," Sigplan Notices (March, 1975), pp. 107-112.

[14] A. E. Oldehoeft, S. Allan, S. Thoreson, C. Retnadhas, and R. J. Zingg, Translation of High Level Programs to Data Flow and Their Execution on a Feedback Interpreter, Department of Computer Science, Iowa State University, Ames, Ia., TR 78-2, (March, 1978).

[15] J. Rumbaugh, "A Data Flow Multiprocessor," IEEE Transactions on Computers (February, 1977), pp. 138-146.

[16] L. G. Tesler and H. J. Enea, "A Language Design for Concurrent Processes," Proceedings of AFIPS, SJCC (1968), pp. 402-408.

[17] K. Weng, Stream-Oriented Computations in Recursive Data Flow Schemas, M.S. Thesis, M.I.T., Cambridge, Mass, (1975).

AN ABSTRACT IMPLEMENTATION FOR CONCURRENT COMPUTATION WITH STREAMS[a]

Jack B. Dennis
Ken K.-S. Weng
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- This paper is a contribution toward developing practical general-purpose computer systems embodying data flow principles. We outline a hardware structure capable of high concurrency and present an abstract model of data flow program execution which could be implemented within the proposed hardware structure. Our abstract model supports a user programming language that includes recursive function modules and provides streams of values for inter-module communication.

Introduction

We present here a conceptual model of program execution that can serve as the functional specification for a distributed or highly concurrent computer system based on data flow principles. The programming language supported by our conceptual model or "abstract implementation" is an applicative or value-oriented language that includes streams of values as a basic programming tool. Streams are attractive because use of streams for communication between program modules leads to programs whose modules have functional semantics and whose overall meaning can be expressed as functional components combined using composition and a fixpoint operator [12] - thus avoiding use of side effects. In the present discussion we only consider determinate programs. The extension of this work to nondeterminate computation is a subject of current research.

Specifically, we introduce a value-oriented language and discuss representation of its semantics by translation into recursive data flow schemas [9]. We sketch an operational semantics (formal interpreter) for these data flow schemas and outline the structure of a hardware system capable of highly concurrent execution of value-oriented programs. A more detailed and complete presentation of this work is given in the thesis of Weng [17].

(a) This research was supported in part by the National Science Foundation under research grant MCS75-04060 A01 and in part by the Lawrence Livermore Laboratory of the University of California under contract 8545403.

A Simple Value-Oriented Language

Our textual language departs from conventional languages in several ways. There is no notion of sequential control flow and there are no explicit primitives for introducing parallelism. The concurrency of a computation is determined by the data dependency within the program rather than by explicit creation of concurrent processes.

The language is value-oriented in the sense that each syntactic unit defines a mathematical function that maps input values into result values: there are no side effects or other spurious interactions in the evaluation of expressions.

The language does not have the notion of memory locations or variables commonly found in conventional sequential programming languages; instead names are used to denote values defined by expressions in much the same way as in mathematics. With value-oriented semantics, it is natural to write programs in a form that exhibits the inherent concurrency of an algorithm. The data types of the language[a] are <u>integer</u>, <u>real</u>, <u>boolean</u>, <u>character-string</u>, <u>structure</u>, and <u>procedure</u>. We shall call these data types *simple data types*. The operations for types <u>integer</u>, <u>real</u>, <u>boolean</u>, and <u>character-string</u> are the usual operations and need no comment. The operations for values of type <u>structure</u> are defined below. The only operation for <u>procedure</u> values is procedure application.

The syntax of the language is given in Fig. 1. A procedure consists of a set of procedure definitions followed by an expression. A procedure definition is of the form

```
P = procedure ( a1:T1,..., am:Tm) yields R1,...,Rn;
        <procedure def>
              .
              .
              .
        <procedure def>
        <expression>
    end P;
```

(a) The language described here is closely related to the language called VAL in development at MIT [3].

```
                    Notation :      { < E > }+ means < E > | < E >, { < E > }+
                                    { < E > } means { < E > }+ | empty

< program > ::= program { < procedure def > } < expression > end
< procedure def > ::= < name > = procedure ( < input list > )
                                    yield < output list >;
                                    { < procedure def > }; < expression >
                                    end < name >
< input list > ::= { < type declaration > }
< type declaration > ::= < name > : < type >
< output list > ::= { < type > }

< expression > ::= < primitive expression >
                 | { < expression > }+
                 | < let-block expression >
                 | < conditional expression >
                 | < application expression >
< let-block expression > ::=
                 let { < type declaration > }; { < name def > }; in < expression > end
< name def > ::= { < name > } = < expression >
< conditional expression > ::=
                 if < expression > then < expression > else < expression > end

< application expression > ::= < name > ( < expression > )
< primitive expression > ::=
                 < expression > < primitive operation > < expression >
                 | < primitive operation > ( < expression > )
                 | < name >
                 | < constant >
< simple data type > ::= integer | real | boolean | character-string | structure
< type > ::= < simple data type > | stream of < simple data type >
```

Figure 1. Syntax of the language

This defines a procedure P that requires m input values $a_1,\ldots,a_m$ of types $T_1,\ldots,T_m$ respectively. The names $a_1,\ldots,a_m$ must be distinct and can appear free in <expression>. The evaluation of the procedure yields an ordered set of values of types $R_1,\ldots,R_n$ resulting from <expression>.

Each expression denotes an ordered set (n-tuple) of values whose *arity* is n. We give a recursive definition of the arity A(E) of each of the five types of expressions as follows:

A(<primitive expression>) = 1

A($\langle exp_1\rangle, \ldots, \langle exp_k\rangle$)
    = A($\langle exp_1\rangle$) + ... + A($\langle exp_k\rangle$)

A(<let-block expression>)
    = A(let <definitions> in <exp> end)
    = A(<exp>)

A(<conditional expression>)
    = A(if <exp> then $\langle exp_t\rangle$ else $\langle exp_f\rangle$ end)
    = A($\langle exp_t\rangle$)
    = A($\langle exp_f\rangle$)

A(<procedure application>)
    = A(<name> (<expression>))
    = the number of elements in the <output list> of procedure <name>.

For a <procedure def> to be correct, the arity of the expression which is its body must match the number of result types specified in its <output list>.

Often it is convenient to introduce names for expressions because they are common subexpressions of larger expressions. The let-block expression is used for introducing names such that each name stands for an expression of arity one. A let-block expression is of the form:

```
let { <type declaration> };
          <name-list_1> = <exp_1>;
                .
                .
                .
          <name-list_k> = <exp_k>;
in <exp> end;
```

The names in type declarations of a let-block are local names meaningful only within the block; these names must be distinct from each other and may appear free in $\langle exp_1 \rangle, \ldots, \langle exp_k \rangle$, and <exp>. Name conflicts in nested let-blocks are resolved by the scope rule that inner definitions take precedence over outer definitions.

We require that the number of names in a name-list be equal to the arity of the expression to the right of the equality sign. The *value* of a name in a name-list is the value of the corresponding expression appearing on the right hand side of the equal sign, and must be of the type specified by the type declaration. The value of a let-block expression is the value of <exp>.

A conditional expression is of the form:

```
if <exp_1> then <exp_2> else <exp_3> end;
```

The expression $\langle exp_1 \rangle$ is a boolean value of arity one. The expressions $\langle exp_2 \rangle$ and $\langle exp_3 \rangle$ have the same arity and the corresponding value in each expression must be of the same type. The value of a conditional expression is the value of $\langle exp_2 \rangle$ if $\langle exp_1 \rangle$ is the boolean value true; otherwise it is the value of $\langle exp_3 \rangle$.

A procedure application expression is of the form:

```
P( <exp> );
```

where the expression <exp> has the same arity as the number of input values required by the procedure P and the type of each value matches that of the input specification. The result of the procedure application is an expression of the arity and types defined by the yield clause of the procedure heading.

As a simple example of a program in our value-oriented language, Fig. 2 shows a procedure that defines a parallel computation of the factorial function.

```
Factorial = procedure ( n : integer )
        yields integer;

    Product = procedure ( n_1 : integer, n_2 : integer )
            yields integer;
                if n_2 =< n_1 then n_1
                else let middle : integer;
                        middle = (n_1 + n_2) quotient 2;
                        in Product( n_1, middle )
                        * Product( middle+1, n_2 ) end
                end
    end Product;

    if n < 0 then error else Product(1, n) end;

end Factorial;
```

Figure 2. An Example Program

Data Structures

For the purpose of the present exposition, we will introduce a simple but very general data structure type. A data structure can be either nil which denotes the structure having no components, or a structure having n component values $v_1, \ldots, v_n$ whose *selector* names are respectively $s_1, \ldots, s_n$. The selectors are either character strings or integers and each selector name must be different from all others in the same data structure. We represent such a structure value by the notation

$(s_1 : v_1, \ldots, s_n : v_n)$.

The operations on data structures are defined below, where d and d' are data structures, s is a selector name, and c is a value of any type:

(1) create ( )
The create operation yields the nil data structure.

(2) append (d, s, c)
The result is a data structure d' which is identical to d except that the s component is c regardless of whether d already contains a component with selector name s.

(3) delete (d, s)
The result is a data structure d' which does not have an s component.

(4) select (d, s)
If d has an s component, the result is the value of that component. Otherwise, the result is the value undefined.

(5) nil-structure (d)
This is a predicate whose value is true if d is nil; otherwise its value is false.

Notice that the effects of

delete (d, s)

and

append (d, s, nil)

are different, since the the delete operation would remove the component (s, d') while the append operation would replace it with (s, nil). It should be mentioned that an array

```
reverse = procedure ( x : structure )
              yields structure;

    if nil-structure ( x )  then x else
              let  left, right : structure;
                       left = reverse( select( x, "r") );
                       right = reverse( select( x, "l") );
              in  append( append
                       ( create( ), "l", left), "r", right)
              end
    end
end reverse;
```

Figure 3. reverse

is simply a data structure whose selector names are all integers.

The data structure operations are illustrated by the recursive procedure "reverse" in Fig. 3, which interchanges the role of selector names l and r in a given data structure of arbitrary depth.

Streams

A stream is a sequence of values, all of the same type, that are passed in succession, one-at-a-time between program modules.

The use of streams of data in programming is an alternative way of expressing computations that have conventionally been expressed as coroutines or a set of cooperating processes. For example, a compiler may be organized into phases which are implemented as a set of coroutines [6].

The operations on values of type stream of T are defined below where s and s' are streams, and c is a value of type T.

(1) [ ]
The result is the empty stream which is the sequence of length zero.

(2) cons (c, s)
The result is a stream s' whose first element is c and whose remaining elements are the elements of the stream s.

(3) first (s)
The result is the value c which is the first element of s. If s is empty, the result is undefined.

(4) rest (s)
The result is the stream left after removing the first element of s. If s = [ ], the result is undefined.

(5) empty (s)
The result is true if s = [ ], and is false otherwise.

```
prime__generator = procedure ( n : integer )
          yields  stream of integer;

     generate = procedure ( i, n : integer )
          yields  stream of integer;
       if i < n then [ ]
       else cons ( i, generate( i+1, n ) ) end;
     end generate;

     sieve = procedure ( s : stream of integer )
          yields stream of integer;
               if empty ( s ) then [ ]
               else  let x : integer,
                        s2, s3 : stream of integer;
                           x, s2 = first ( s ), rest ( s );
                           s3 = delete ( x, s2 );
                           in  cons ( x, sieve( s3 ) ) end;
               end;
     end sieve;

     delete = procedure ( x : integer,
          s : stream of integer )
          yields  stream of integer;
               if empty ( s ) then [ ]
               else  let y : integer,
                        s2, s3 : stream of integer;
                           y, s2 = first ( s ), rest ( s );
                           s3 = delete ( x, s2 );
                           in  if divide ( x, y ) then s3
                                  else cons ( y, s3 ) end;
                        end;
               end;
     end delete;

     sieve (  generate ( 2, n ) );

end prime__generator;
```

Figure 4. A Prime Number Generator

The following identity is satisfied by the stream operations:

```
if  empty( s )  then  s = [ ]
                else  s = cons( first( s ), rest( s ) )
                end
```

The problem of generating all prime numbers less than a given integer n is a good example of the use of streams in constructing a modular program so as to expose many independent actions for concurrent execution. The sieve of

Eratosthenes expressed in our textual language is presented in Fig. 4. The procedure "generate" produces the sequence of successive integers beginning with 2. This stream is processed by "sieve" to remove nonprime elements. Procedure "sieve" operates by taking the first element of its input and removing all multiples of the first element (using "delete") and applying "sieve" recursively to the remaining elements. (The first use of stream concepts for the prime number sieve, as far as we know, was in [16]. It seems the example has been discovered independently by several authors.)

Data flow schemas

A data flow schema is an operational model of concurrent computation. The form of schemas used here derives from the work of Dennis and Fosseen [9] and Dennis [7]. A data flow schema is a directed graph composed of nodes called *actors* and arcs connecting them. An arc pointing to an actor is called an *input arc* of the actor; and an *output arc* is an arc emanating from the actor. Each actor has an ordered set of input arcs and output arcs. There are five types of actors: link, operator, switch, merge and sink. The five types of actors are shown in Fig. 5. An *(m, n) data flow schema* must have m links which do not have input arcs, and n links not having output arcs. These links are respectively called *input links* and *output links* of the (m, n) schema. Further, we require that the schema must be proper in the sense that all other actors must have the required arcs of its actor type, and each arc must be connected at both ends.

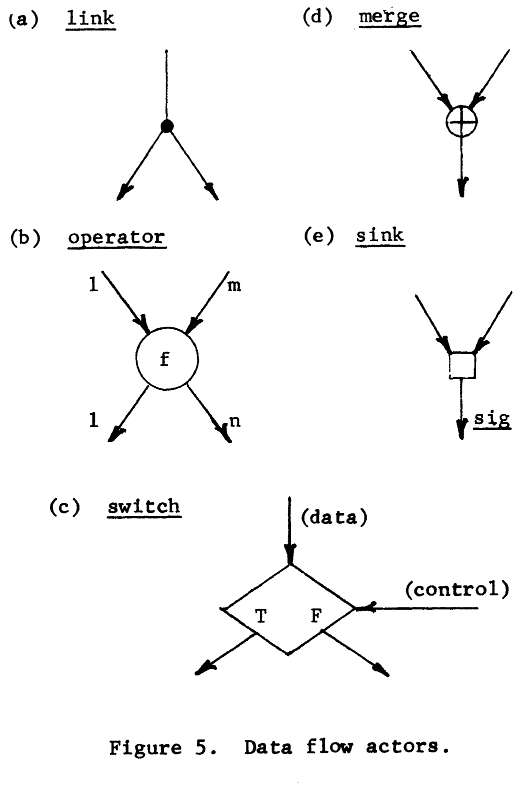


Figure 5. Data flow actors.

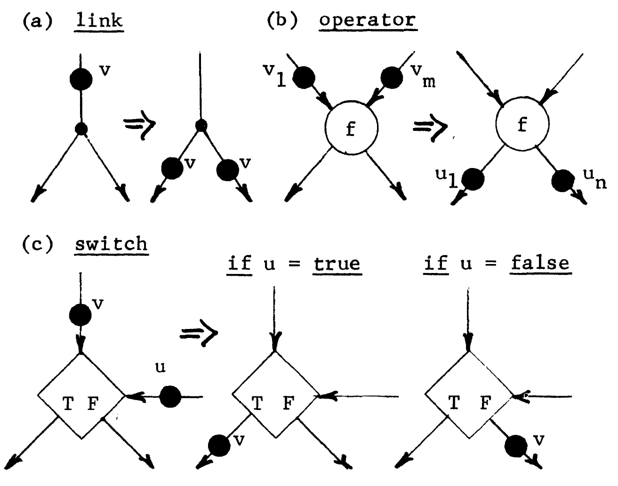


Figure 6. Examples of firing rules.

Stating the operational semantics of data flow schemas requires additional concepts. A *configuration* of a data flow schema is the graph of the schema together with an assignment of labeled *tokens* to some arcs of the graph. An assignment of a token to an arc is represented by the presence of a solid circle on the arc. The label denotes the value carried by the token and may be omitted when the particular value is irrelevant to the discussion. Informally, the presence of a token on an arc means that a value is made available to the actor to which the arc points. For the present, tokens carry values of type integer, real, boolean, structure, or stream.

Firing Rules

Execution of an (m, n) schema advances it from one configuration to another through the *firing* of some actor that is *enabled*. The firing rules for the principal actor types are specified in Fig. 6. A necessary condition for any actor to be enabled is that each output arc does not hold a token. An actor is enabled when a token is present on each input arc -- with the exception of a merge actor. The firing of an actor causes the tokens to be absorbed from the input arcs and completes by placing a token on each of the output arcs. The values of the output tokens are functionally related to the values of the input tokens. A link simply replicates the value received and distributes it to the destination actors indicated by output arcs. The effect of firing an operator is to apply to the inputs $v_1,\dots,v_m$ the function associated with the operation name written inside the operator to yield the outputs $u_1,\dots,u_n$. The switch and merge are used for controlling the flow of tokens. A switch requires a data input and a control input which is a boolean value. The firing of a switch replicates the input token on one of the output arcs according to the boolean control value. The arrival of a token on either input arc enables a merge, and upon firing, a token conveying the same value is placed on the output arc. The behavior of a merge is inherently nondeterminate: when two input tokens reside on the input arcs, the firing rule does not specify in which order the output tokens will be generated. A sink absorbs the input tokens upon firing and places a special token signal on the output arc. The purpose of a sink actor is to absorb unwanted values; the signal output token is necessary for the implementation of schema application to be described.

The set of functions commonly associated with an operator includes the scalar arithmetic operations and constant functions.

Well Formed Data Flow Schemas

Unrestricted use of actors in data flow schemas is undesirable since an arbitrary interconnection of these actors may form a schema which deadlocks or has nondeterminate behavior. Because these properties are undesirable for reliable programming we choose a subclass of schemas which will satisfy the needs of programming.

An *(m, n) well formed* data flow schema is an (m, n) data flow schema formed by any acyclic composition of component data flow schemas, where each component is either a link, a sink, an operator, or a *conditional subschema.*

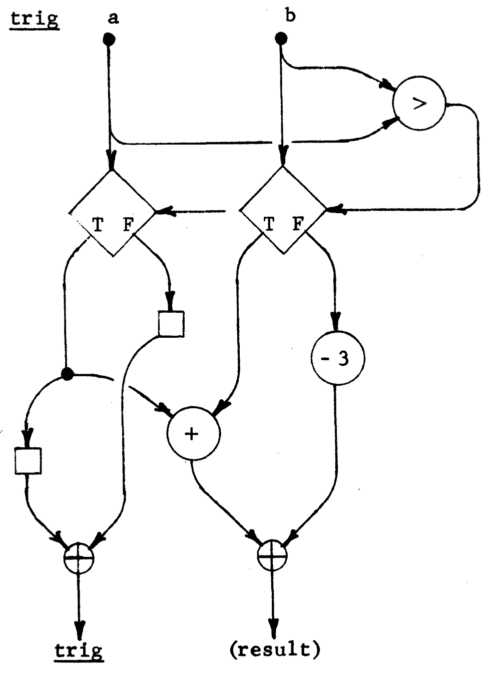


Figure 7. A conditional schema.

Fig. 7 is an example of a conditional schema which computes the value of the expression

if a > b then a + b else b - 3

Here, the trig output provides a *completion signal* indicating that the sink actor has absorbed the unused copy of a. The structure of a conditional schema corresponds in an obvious way to conditional expressions.

The Apply Actor

The class of well formed data flow schemas cannot express program features such as procedures, procedure applications, and iterations. We introduce an actor apply whose meaning is explained in Fig. 8. The first input to an apply actor is a token associated with an (m, n) well formed data flow schema. An apply actor is enabled when a token is present on each input arc. The effect of firing an apply actor is to replace the actor with the specified (m, n) schema as shown in the figure. The (m, n) schema replacing the apply actor may itself contain apply actors, allowing recursion to be expressed.

We have not included structures of data flow schemas which correspond to language constructs such as while loops in Algol 60 or Do statements in Fortran. Such structures necessarily involve cyclic connections of actors which do not correspond to actual data dependencies, and introduce unnecessary delays. Furthermore, the semantics of cyclic schemas is more complicated, since issues of safety and liveness must be dealt with. We choose to support these language features in the equivalent form of recursive application of data flow schemas. This allows simultaneous execution of instances of a data flow schema which correspond to successive iterations of a while loop.

An example of the use of apply actors is given in Fig. 9. This recursive schema implements the "reverse" function stated earlier in Fig. 3. The input link actor labeled trig is an input link whose function is to trigger those actors that generate constants, in this case the create actor that produces the empty data structure.

The apply actor presented requires that all input values be present on the input arcs to become enabled. A language implemented in terms of the apply actor will have "call by value" semantics, that is, the result of application is well defined only when the computations producing arguments to the procedure all terminate. This is in contrast with a more general form of procedure application which allows procedure application to begin even though computation of some arguments is not complete.

Data Flow Processor

The structure of a data flow processor suitable for supporting execution of recursive data flow schemas is shown Fig. 10. It consists of six subsystems: Functional Units, Structure Controller, Execution Controller, the Arbitration and Distribution Networks, and the Packet

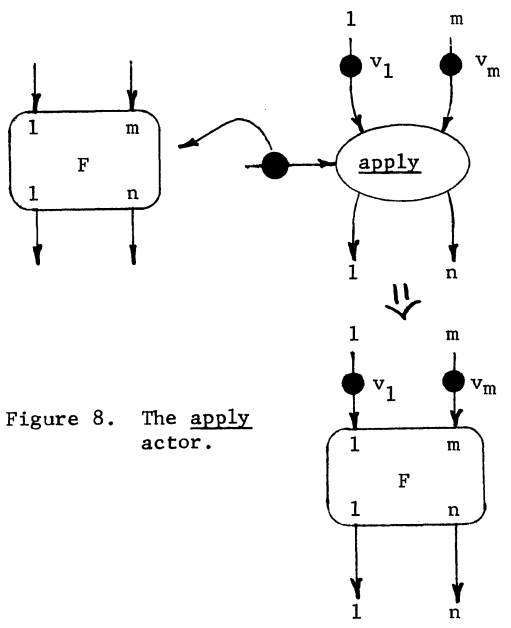


Figure 8. The apply actor.

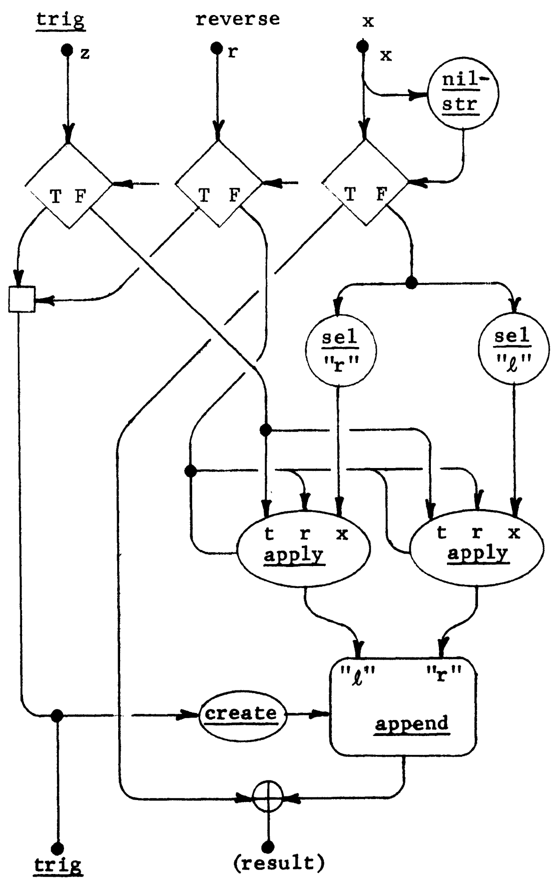


Figure 9. Recursive schema.

Memory. The Execution Controller fetches instructions and operands from the Packet Memory and forms them into operation packets. Each operation packet is passed to the Arbitration Network for transmission to an appropriate Functional Unit if a scalar operation is called for, or to the Structure Controller for the data structure operations create, append, and select. Instruction execution in the Structure Controller and Functional Units generate result packets which are sent through the Distribution Network to the Execution Controller where they will join with other operands to activate their target instructions. How this is done is explained in greater detail in the next section.

The Packet Memory holds the collection of data structures as a collection of *items* each being a one-level data structure having scalar values and unique identifiers of other items as its components [8]. This collection of items represents an acyclic directed graph where each arc corresponds to a unique identifier component of the item representing its origin node. The Packet Memory maintains a reference count for each item and reclaims physical storage space as items become inaccessible.

Data structures held in the Packet Memory have three roles in the execution of data flow schemas: (1) as operands for the data structure operations implemented by the Structure Controller; (2) as *procedure structures* that have as components the instructions of a data flow procedure; and (3) *activation records* which hold operand values for instructions waiting for their enabling condition to be satisfied.

Although the Execution Controller, Structure Controller and the Packet Memory are shown in Fig. 10 as single units, we imagine that each is in fact a collection of many identical units. For example, the Packet Memory subsystem would consist of separate systems, each holding all items whose unique identifiers belong to a well defined part of the address space of unique identifiers. The Execution Controller subsystem would consist of identical modules each of which would serve a distinct subset of procedure activations.

The concept of a Packet Memory System was introduced in [8], and the design issues for these systems and the Structure Controller have been studied in [1, 2].

Implementation of Data Flow Schemas

Procedure Structures

A data flow schema is represented in the machine by a kind of data structure called a *procedure structure* illustrated in Fig. 11a. A procedure structure corresponding to a data flow schema of n actors is a data structure having n components with integer selector names from 1 to n assigned to the actors. Each component, called an *instruction*, is an encoding of an actor and its output arcs.

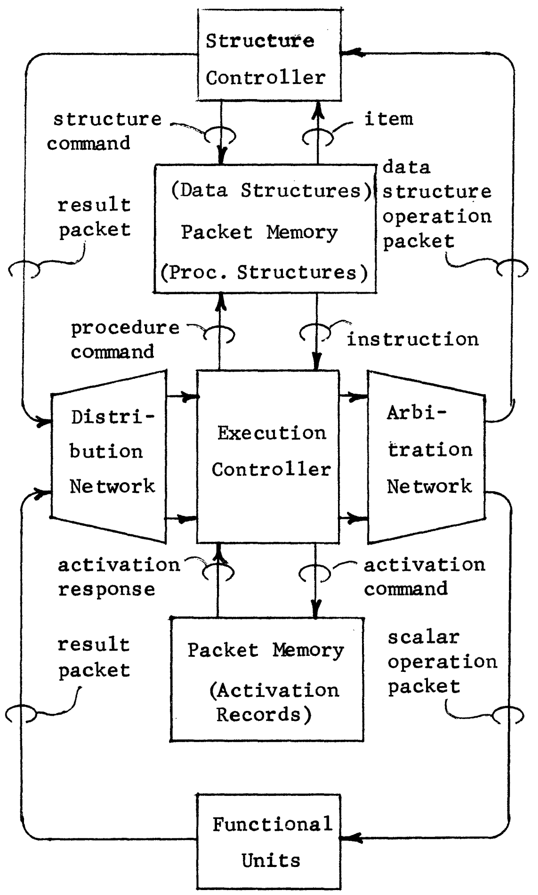


Figure 10. Data flow processor.

The components of an instruction include an *operation* field which defines the function performed by the actor, and *destination* fields D1, ..., Dp corresponding to p output arcs. Each destination field has three subcomponents: the *inst* component is the integer selector name of the destination instruction; the *arc* component is an integer designation of an input arc of the destination; and the *count* component is the number of operand values required by the destination instruction.

Activation Records

Since multiple instances of the same schema may be concurrently active in a computation, each activation (an instance of procedure execution) is represented by a separate *activation record* as shown in Fig. 11b. Each actor in an activation is uniquely identified by the tuple (A, i), where A is a uid allocated for the activation record and i is the integer assigned to the actor in the procedure structure. A token of value v on the k-th input arc of an actor (A, i) corresponds to a result packet that carries the

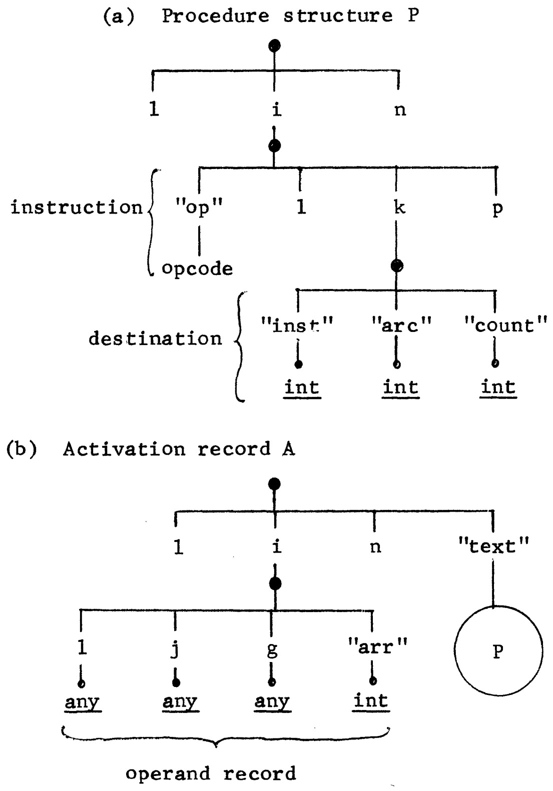


Figure 11. Procedure and activation structures.

information (A, i, k, v, count), where "count" is the number of tokens (operands) required for the enabling of the actor.

Enabling of an actor is detected by checking the number of result packets having arrived at the operand record -- the i component of the activation record A - against the count in the result packet. The detection of enabling is a function of the Execution Controller and the Packet Memory that store activation records. Upon enabling of actor instance (A, i), the instruction of the actor is fetched from the i component of the procedure structure. The following section describes how activation records might be manipulated.

An activation record has components with integer selectors for operand records and an additional "text" component that is the procedure structure for the activation. (In our implementation, this component is shared by other activations of the same schema.) An operand record may have as many integer subcomponents as input arcs of an actor, and also contains an "arrived" subcomponent indicating the number of arrived result packets. Since an activation record stores values of arrived result packets in its components, operations on an activation record modify its components. These operations are defined as follows:

(1) create-activation(P)
This returns the uid of a new activation record having P as its "text" component, but no other components.

(2) insert(A, i, k, v)
The insert operation adds the value v as the k-th operand of the i-th instruction in activation record A. In addition, the "arr" component of the operand record is incremented by one. To handle the first operand value to arrive, a missing "arr" component is interpreted as having the value zero.

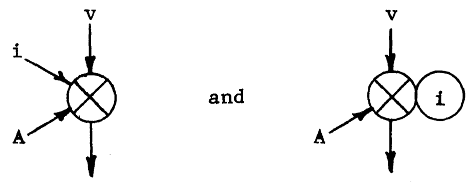
(3) remove(A, i)
This operation releases the i component of A; and is performed by the Execution Controller once it has generated the operation packet for actor instance (A, i).

(4) free(A)
This operation releases the entire activation record A by means of a command packet sent to the Packet Memory.

For each arriving result packet (A, i, k, count, v) the Execution Controller performs the operation insert(A, i, k, v) and tests the updated value of the "arr" component against the "count" field of the result packet. If the values are equal, the instruction is fetched from the Packet Memory and used, together with the operand record, to construct an operation packet which is delivered to the Arbitration Network. The i component of activation record A is then released.

Procedure Activation

Our implementation of the apply actor is illustrated in Fig. 12. The apply actor is replaced by the code diagrammed in Fig. 12b, and the applied graph F is augmented as in Fig. 12c. Here we use the notations



to mean insert (A, i, 1, v). The new actors extr-uid, const-ret and distribute will be explained below.

This implementation assumes the actors in each recursive schema are numbered according to this rule:

(1) Input link actors are numbered 1, ..., m.
(2) The link actors that receive the n-tuple of values resulting from a schema application are numbered J + 1, ..., J + n for some integer J.
(3) A link actor numbered 0 receives a packet (A, J, n) containing the information needed to construct result packets for returning values resulting from procedure execution.
(4) The remaining actors may be numbered arbitrarily.

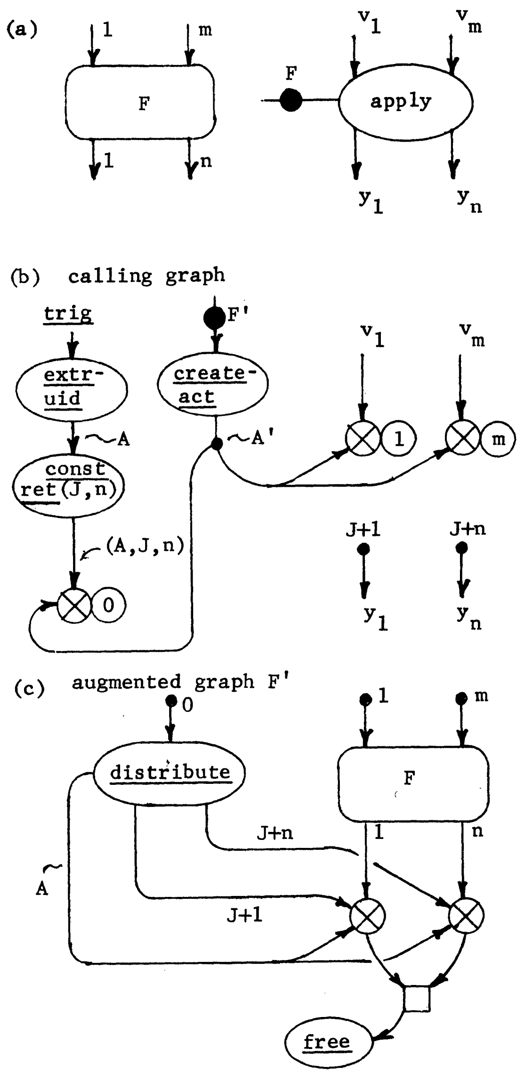


Figure 12. Implementation of apply.

The implementation scheme works as follows: The create-act actor produces the uid A' of a new activation record containing "text" component F' and passes it to the insert actors associated with input value v_1, ..., v_m. These actors cause result packets of the form (A', i, 1, 1, v_i) to be generated which initiate execution of the new activation of F'. At the same time, the extr-uid and const-ref actors form the *return value* (A, J, n) and send it to link 0 of schema F'. Once result values y_1, ..., y_n have been produced, the distribute and insert actors of F' generate result packets of the form (A, J + i, 1, 1, y_i) which deliver result values to the calling schema. The free actor then releases the activation record, and its uid A' is returned to the pool of free uid's managed by the Packet Memory.

Implementation of Stream Actors

In the implementation streams are represented as data structures. A stream is a data structure having an "f" component which is the first element of the stream, and an "r" component which is the data structure representing the rest of the stream. The empty stream is represented by nil. Operations on streams become operations on structure values; thus first(s) and rest(s) are implemented by select(s,"f") and select(s,"r"), respectively.

We wish to make it possible for a stream to be processed by consuming modules while further stream elements are generated concurrently. To provide for this behavior, we must augment our concept of data structures so a data structure may be accessed before it is entirely constructed. We use the concept of *holes* which is based on the work of Henderson [11] who used the term "token". Our idea is related to but different from the idea of "suspensions" discussed by Friedman and Wise [10].

The idea is embodied in the implementation of the cons operation described in Fig. 13. Here the create-hole and write-hole actors are special data structure operators defined as follows:

> A create-hole actor returns a uid H allocated from the data structure address space. The free node is called a hole in that it has two states: *filled* and *unfilled*. In the unfilled state, all data structure operations on the hole are queued except the write-hole operation. Upon completion of the write-hole(H,v) operation, the hole H changes its state to filled and contains the value v. All previously queued and subsequent operations on H are processed without further delay; a subsequent write-hole operation on H is illegal.

To illustrate the concurrency provided by this implementation of streams, consider the recursive schema

s' = cons (v, s)

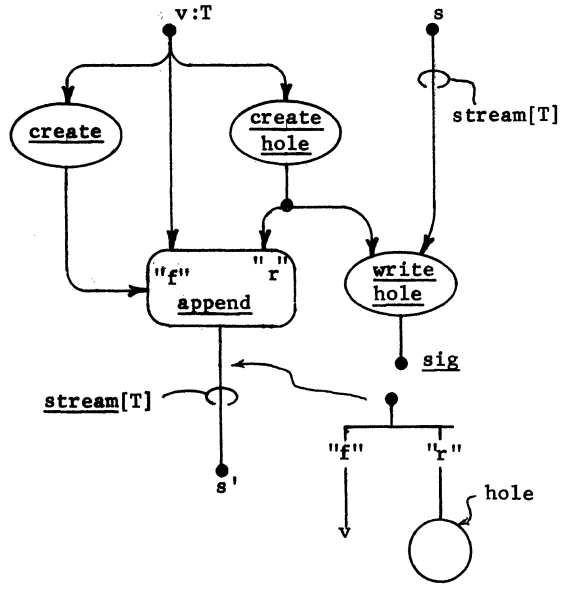


Figure 13. Implementation of cons.

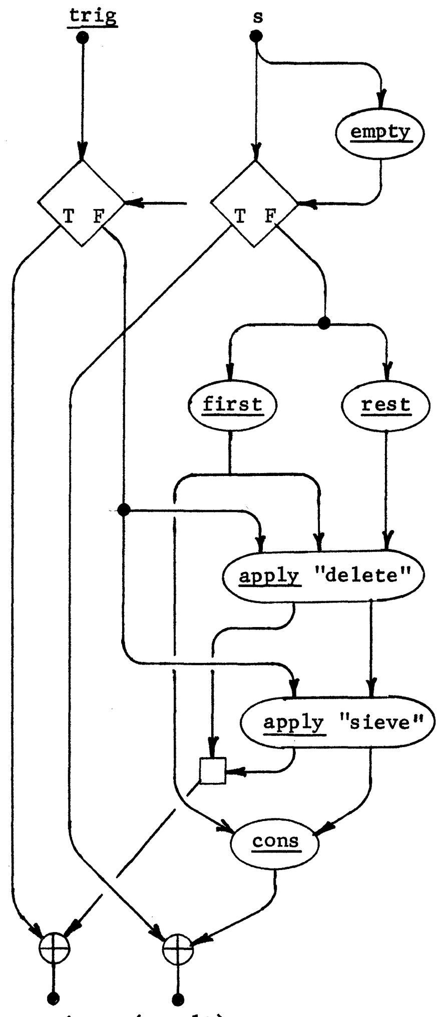


Figure 14. Data flow schema for "sieve".

shown in Fig. 14 for the "sieve" procedure of the prime number generator. Note that the output of the top activation of "sieve" will be a data structure containinng the first element of the result stream and a hole waiting to be filled in with the data structure generated by the recursive activation of "sieve". In this implementation each higher activation of "sieve" may be released as soon as it has completed its work (i.e., its hole has been filled), leaving the remaining work to be finished by deeper activations of the code.

Remarks

The concept of stream has appeared in many forms [5, 12, 14, 15]. One of the earliest papers that discussed streams as a programming feature was an unpublished paper by McIlroy [15]. Despite the conceptual elegance of streams, programming has not yet departed from the sequential notion of coroutines and process synchronization primitives. Recent interest in concurrent programming languages and processors have motivated several other authors to investigate the feasibility of implementation of streams and related concepts of data structures with holes or with suspensions [4, 10, 13].

References

[1] W. B. Ackerman, *A Structure Memory for Data Flow Computers*, Laboratory for Computer Science, Massachusetts Institute of Technology, TR-186, (August, 1977), 125 pp.

[2] W. B. Ackerman, "A Structure Processing Facility for Data Flow Computers," *Proceedings of the 1978 International Conference on Parallel Processing* (August, 1978), pp. 166-172.

[3] W. B. Ackerman, and J. B. Dennis, *VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, Laboratory for Computer Science, Massachusetts Institute of Technology, TR-218, (July, 1979), 80 pp.

[4] Arvind, and K. P. Gostelow, "Some Relationships Between Asynchronous Interpreters of A Dataflow Language," *Formal Description of Programming Concepts: Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts* (August, 1977), pp. 95-119.

[5] W. H. Burge, "Stream Processing Functions," *IBM Journal of Research and Development 19* (January, 1975), pp. 12-25.

[6] M. E. Conway, "Design of a Separable Transition-Diagram Compiler," *Communications of the ACM* 6 (July, 1963), pp. 396-408.

[7] J. B. Dennis, "First Version of a Data Flow Procedure Language", *Programming Symposium: Proceedings, Colleque sur la Programmation, Lecture Notes in Computer Science 19* (October, 1976), pp. 362-376.

[8] J. B. Dennis, "Packet Communication Architecture," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing* (August, 1975), pp. 224-229.

[9] J. B. Dennis, and J. B. Fosseen, *Introduction to Data Flow Schemas*, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Memo 81-1 (September, 1973), 45 pp.

[10] D. P. Friedman, and D. S. Wise, "Aspects of Applicative Programming for Parallel Processing," *IEEE Transactions on Computers C-27* (April, 1978), pp. 289-296.

[11] D. A. Henderson, *The Binding Model: A Semantic Base for Modular Programming Semantics*, Laboratory for Computer Science, Massachusetts Institute of Technology, TR-145 (February, 1975), 282 pp.

[12] G. Kahn, "The Semantics of A Simple Language for Parallel Programming," *Information Processing 74: Proceedings of the IFIP Congress* (August, 1974), pp. 471-475.

[13] R. M. Keller, G. Lindstrom, and S. Patil, "A Loosely-Coupled Applicative Multi-Processing System," *1979 National Computer Conference, AFIPS Conference Proceedings 48* (June, 1979), pp. 613-622.

[14] P. J. Landin, "A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I," *Communication of the ACM 8* (February, 1965), pp. 89-101.

[15] M. D. McIlroy, "Coroutines: Semantics In Search Of A Syntax," Unpublished Paper (1968).

[16] K.-S. Weng, *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science, Massachusetts Institute of Technology, TM-68, (October, 1975), 93 pp.

[17] K.-S. Weng, *An Abstract Implementation for a Generalized Data Flow Language*, Laboratory for Computer Science, Massachusetts Institute of Technology, Technical Report, forthcoming.

TRANSLATION AND OPTIMIZATION OF DATA FLOW PROGRAMS[a]

J. Dean Brock
Lynn B. Montz
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- We present ADFL, an Applicative Data Flow Language with an iterative control abstraction based on tail recursion and an error-handling scheme appropriate to the concurrency of data flow. An algorithm for translating ADFL programs into data flow graphs is described. These graphs may be executed without possibility of deadlock, but with potential loss of some concurrency, on packet communication systems with bounded buffering, such as the Dennis-Misunas data flow computer. Two techniques for optimizing graphs are given and their effect on performance and correctness is analyzed. One is the insertion of identity operators (buffers) into graphs to increase pipelining. The other is the elimination of unneeded acknowledge signals.

Introduction

In a data flow computer, an operation is performed as soon as its operands have been computed. The machine language is an explicit representation of the data dependencies of program operations. Its programs are directed *data flow graphs* whose nodes are called *operators*. The role of operators in a data flow machine is similar to the role of instructions in a von Neumann machine. The execution of an instruction corresponds to the *firing* of an operator. Each operator has several input and output ports. Whenever an operator fires, it absorbs tokens (values) at its input ports and produces tokens at its output ports. Operators have firing rules which determine when they are *enabled* for firing. These firing rules are based on the presence or absence of tokens on the operator's ports.

When operators are joined to form data flow graphs, the links of the graph are directed from operator output ports to operator input ports. A link transports the results produced at an operator output port to an operator input port. Thus, links form the pathways upon which data *flows* as tokens are absorbed and produced by the firing of operators during the execution of a graph.

The data flow graph of an elementary expression resembles its parse tree. The graph for computing the distance function:

$$\mathrm{sqrt}((x1-x2)^2 + (y1-y2)^2)$$

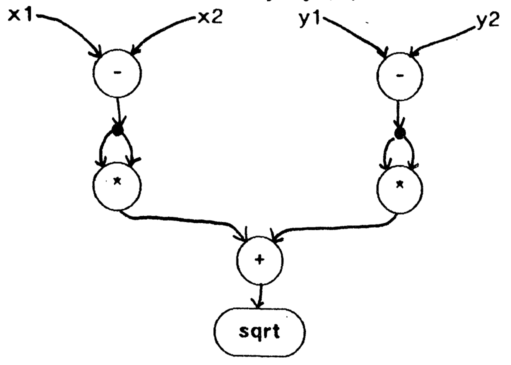
is illustrated in Figure 1. The solid black dot in the figure represents the copy operator which is used to distribute the results of one output port to several input ports. Note how this graph represents the operation dependencies and independencies of the distance function.

Preliminary data flow machine designs have been made by Arvind and Gostelow [2], Davis [5], and Dennis and Misunas [7]. Within these machines, a data flow graph is distributed over a network of processing elements. These elements operate concurrently, constrained only by the operational dependencies of the graph. Thus, a very efficient utilization of the machine's resources appears possible.

ADFL - An Applicative Data Flow Language

Data flow programming languages resemble conventional languages restricted to those features whose ease of translation does not depend on the state of a computation being a single, sequentially manipulated entity. Because the "state" of a data flow graph is distributed for concurrency, *goto*'s, expressions with side effects, and multiple assignments to the same variable are difficult to represent. ADFL, Applicative Data Flow Language, is a simplification of VAL, the value-oriented data flow language being developed by Ackerman and Dennis [1]. A BNF

Figure 1. sqrt((x1-x2)^2 + (y1-y2)^2)



sqrt((x1-x2)^2 + (y1-y2)^2)

(a) This research was supported in part by the Lawrence Livermore Laboratory of the University of California under contract 8545403, in part by the National Science Foundation under research grant MCS75-04060 A01, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661. Part of the research was conducted while Mr. Brock was supported by a National Science Foundation graduate fellowship.

specification of the syntax of ADFL follows:

```
exp ::= id | const | exp , exp | oper(exp) |
        let idlist = exp in exp end |
        if exp then exp else exp end |
        for idlist = exp do iterbody end

iterbody ::= exp | iter exp |
             let idlist = exp in iterbody end |
             if exp then iterbody else iterbody end

id ::= "programming language identifiers"

idlist ::= id { , id }

const ::= "programming language constants"

oper ::= "programming language operators"
```

The most elementary expressions of ADFL are identifiers and constants. Tuples of expressions are also expressions: One such expression is "x, 5". The application of an operator to an expression is an expression. Although, the BNF specification only provides for operator applications in prefix form, such as "+(x, 5)"; applications in infix form, such as "x + 5", are considered acceptable equivalents (sugarings) and will be used in example ADFL programs. In sequential programming languages execution exceptions are generally handled by program interrupts (signals). This solution is inappropriate for data flow since there is no control flow to interrupt. Applied to "exceptional" inputs, data flow operators yield special error values, such as **zero divide** or **pos over**. The documentation of VAL [1] contains a detailed specification of this method of error-handling. For simplicity, only one error value **undef** is used throughout this paper.

Since ADFL is applicative, it provides for the binding, rather than the assignment, of identifiers. Evaluation of the binding expression:

let y, z = x + 5, 6 **in** y * z **end**

implies the evaluation of "y * z" with y equal to "x + 5" and z equal to 6. The result of binding is local: the values of y and z outside the binding expression are unchanged.

ADFL contains a conventional conditional expression, but has an unusual iteration expression. Evaluation of the iteration expression:

for *idlist* = *exp* **do** *iterbody* **end**

is accomplished by first binding the *iteration identifiers*, the elements of *idlist*, to the values of *exp*. Note from the BNF specification of *iterbody*, that the evaluation of the *iteration body* will ultimately result in either an expression or the "application" of a special operator **iter** to an expression. This application of **iter** is actually a tail recursive call of the iteration body with the iteration identifiers bound to the "arguments" of **iter**. The iteration is terminated when the evaluation of the iteration body results in an ordinary, non **iter**, expression. The value of this expression is returned as the value of the iteration expression. The following iteration expression computes the factorial of n:

```
for i, y = 1, 1 do
  if i < n then iter i + 1, y * i else y end
end
```

Syntactic restrictions which ensure that expressions are used only when appropriate in arity and type have been omitted from this discussion. Elsewhere in this volume, Dennis and Weng [8] define arity and type restrictions for a data flow language similar to VAL and, consequently, ADFL. Their language differs from ours in emphasis: They present an abstract interpreter with a dynamic allocation scheme for executing graphs and, accordingly, emphasize procedural control abstractions. We investigate the execution of statically allocated graphs (data flow machine language programs) and, accordingly, emphasize iterative control abstractions.

Translation of ADFL

The translation algorithm of ADFL consists of two functions $\mathcal{T}$, mapping ADFL expressions into their data flow graph implementations, and $\mathcal{T}_I$, mapping ADFL iteration bodies into their implementations. The graph implementing an expression or iteration body has an input port for each free variable of the expression or iteration body. For an expression *exp* which returns *n* values when evaluated, $\mathcal{T}[\![exp]\!]$ has *n* output ports. Recall that evaluation of an iteration body will yield either results to be re-iterated or results to be returned by the containing iteration expression. The graph $\mathcal{T}_I[\![iterbody]\!]$ has an output port **iter?** which signals which possibility has occurred and sets of output ports for each possibility: **I** output ports for values to be *i*terated and **R** output ports for values to be *r*eturned.

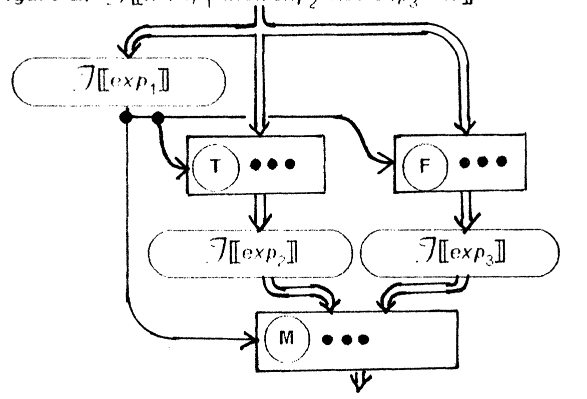
The translation algorithm for ADFL resembles previous translation schemes of Dennis [6] and Weng [11]. A detailed recursive definition of the algorithm over the eleven cases of the BNF specification of the syntax of ADFL has been given by Brock [3]. For brevity, only the cases of the conditional expression, the conditional iteration body, and the iteration expression will be examined in detail. It is assumed that most readers, informed that the graph of Figure 1 may be re-labeled:

$\mathcal{T}[\![$**let** dx,dy = x1-x2,y1-y2 **in** sqrt(dx*dx+dy*dy) **end**$]\!]$

will discover the translation of the eight "trivial" cases.

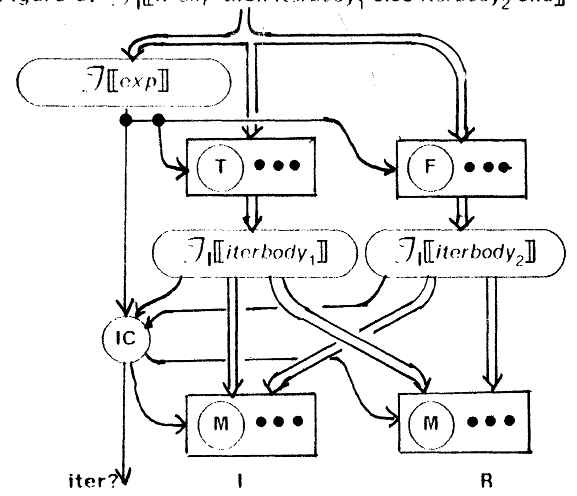
The graph $\mathcal{T}[\![$**if** exp_1 **then** exp_2 **else** exp_3 **end**$]\!]$ is shown in Figure 2. The graph contains three subgraphs, $\mathcal{T}[\![exp_1]\!]$, $\mathcal{T}[\![exp_2]\!]$, and $\mathcal{T}[\![exp_3]\!]$, and several gates. The **T** gate has a control input port (entering its left side), a data input port, and an output port. When the **T** gate fires, it absorbs a token from each input port. If the control token is **true**, the data token is passed to the output port. If the

Figure 2. $\mathcal{T}[\![\mathbf{if}\ exp_1\ \mathbf{then}\ exp_2\ \mathbf{else}\ exp_3\ \mathbf{end}]\!]$



$\mathcal{T}[\![exp_1]\!]$
T
F
$\mathcal{T}[\![exp_2]\!]$
$\mathcal{T}[\![exp_3]\!]$
M

Figure 3. $\mathcal{T}_I[\![\mathbf{if}\ exp\ \mathbf{then}\ iterbody_1\ \mathbf{else}\ iterbody_2\ \mathbf{end}]\!]$



$\mathcal{T}[\![exp]\!]$
T
F
$\mathcal{T}_I[\![iterbody_1]\!]$
$\mathcal{T}_I[\![iterbody_2]\!]$
IC
M
M
iter?
I
R

IC gate firing rules

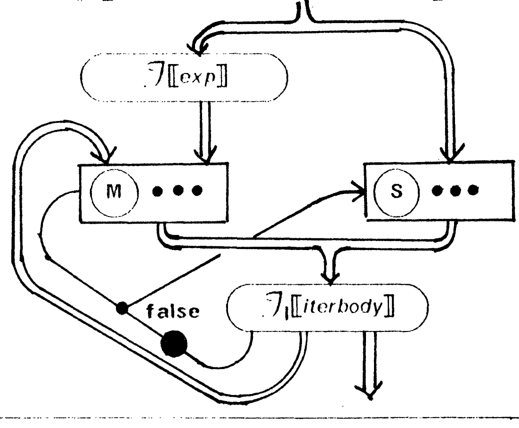
	inputs			*outputs*	
true	**true**	-	**true**	**true**	-
true	**false**	-	**false**	-	**true**
false	-	**true**	**true**	**false**	-
false	-	**false**	**false**	-	**false**
undef	-	-	**false**	-	**undef**

control token is **false**, the data token is simply absorbed. No output token is produced. The F gate is defined analogously. It passes its data token only if its control token is **false**. By passing inputs to $\mathcal{T}[\![exp_2]\!]$, respectively $\mathcal{T}[\![exp_3]\!]$, through T gates, respectively F gates, controlled by the output of $\mathcal{T}[\![exp_1]\!]$, the proper subexpression is "enabled" during data flow evaluation of the conditional expression. The results of $\mathcal{T}[\![exp_2]\!]$ and $\mathcal{T}[\![exp_3]\!]$ are merged by M gates. The M gate has one control input port, two data input ports, and one output port. Its control token selects the data token to be passed. If the control value is the error value **undef**; each T or F gate absorbs a data token and produces no output tokens, and each M gate produces **undef** and absorbs no input tokens. Thus, data flow evaluation of a conditional expression yields a tuple of **undef**'s if the condition is **undef**.

$\mathcal{T}_I[\![\mathbf{if}\ exp\ \mathbf{then}\ iterbody_1\ \mathbf{else}\ iterbody_2\ \mathbf{end}]\!]$, the conditional iteration body graph illustrated in Figure 3, is similar to the conditional expression graph. With T and F gates, the output of the expression subgraph, $\mathcal{T}[\![exp]\!]$, enables one of the iteration body subgraphs, $\mathcal{T}_I[\![iterbody_1]\!]$ and $\mathcal{T}_I[\![iterbody_2]\!]$. The selected subgraph will produce output at either its I or R output ports, according to its **iter?** output: **true**, for I outputs to be iterated; **false**, for R outputs to be returned. Using the output of the expression subgraph and the **iter?** outputs of the iteration body subgraphs, the IC gate calculates three iteration control outputs: the graph **iter?** output and the control tokens for the M gates producing the graph I and R outputs. The table at the bottom of Figure 3 gives the firing rules of the IC gate. Note that, if the output of the expression subgraph is **undef**, the conditional iteration body graph will produce **false** at its **iter?** port, thus announcing termination of iteration, and will produce **undef** at its R output ports.

The graph $\mathcal{T}[\![\mathbf{for}\ idlist = exp\ \mathbf{do}\ iterbody\ \mathbf{end}]\!]$ is shown in Figure 4. This cyclic graph is formed by using M gates to merge the outputs of $\mathcal{T}[\![exp]\!]$ and the I outputs of $\mathcal{T}_I[\![iterbody]\!]$ and by routing the merged outputs into the input ports of $\mathcal{T}_I[\![iterbody]\!]$ labeled by identifiers of *idlist*. The control input port of each M gate is connected to the

Figure 4. $\mathcal{T}[\![\mathbf{for}\ idlist = exp\ \mathbf{do}\ iterbody\ \mathbf{end}]\!]$



iter? output of $\mathcal{T}_I[\![iterbody]\!]$. The connecting arc contains an initial **false** token to ensure that the first data value is selected from $\mathcal{T}[\![exp]\!]$. Thereafter, data tokens are selected according to **iter?**. A **true iter?** token, signalling continued iteration, selects the data tokens of the I output ports. A **false iter?** token, signalling termination, re-initializes the M gates for subsequent iteration expression evaluations. Identifiers which are free in *iterbody* but are not contained in *idlist* are routed through S gates. For **false** control tokens, the S gate absorbs, stores, and outputs its data tokens. For **true** control tokens, it produces its stored value and absorbs no data tokens. Thus, the S gate stores new values when evaluation of the iteration expression begins and produces them at each subsequent iteration step. Like the **M** gate, it is initialized with a **false** control value.

Brock [4] has verified this translation algorithm by proving it to be consistent with a denotational [10] specification of ADFL. In the proof, data flow arcs are assumed to be implemented by infinite (unbounded) queues. The transformations described subsequently will relax this requirement without affecting the correctness of translation.

Transformations of Data Flow Graphs

In proposed data flow machines of the Dennis-Misunas [7] design, operations are held in instruction cells which contain a register for each input arc. These registers are effectively an implementation of data flow arcs as queues of capacity one. The implication of the bounded arcs is that operators must be prevented from producing new tokens until their output arcs are empty. This behavior is ensured by modifying the firing rules so that no operator is enabled if a token is present on *any* of its output arcs.

By performing a transformation, illustrated in Figure 5, which replaces each arc of the graph by an appropriate data/acknowledge arc pair (d/a arc pair), the effect of the modified firing rule can be explicitly built into the graph: The presence of a token indicates that the corresponding data arc is empty. As a consequence, operator firing rules revert to the original format of depending only on the presence of tokens on input (including acknowledge) arcs, where the previous enabling requirement that output arcs be empty has been replaced with the requirement that acknowledge inputs be present.

Montz [9] and Dennis and Misunas [7] have shown that graphs of data flow programs may be executed without deadlock when arcs are implemented as data/acknowledge pairs. Consequently, the correctness of the translation algorithm is not affected by this transformation. However, the implementation is not without cost. Aside from the obvious overhead involved in incorporating acknowledge arcs and tokens, the constraints which they impose on the token flow through the graphs may cause bottlenecks. In response to these issues, Montz [9] has developed optimization techniques specifically aimed at either increasing the throughput by balancing the token flow or decreasing the overhead by removal of unnecessary acknowledge arcs.

Balancing Token Flow

The goal of the optimization to balance token flow through the graph is to increase throughput by modifying the graph to display maximum pipelining. The bottleneck problem, and therefore application of the optimization, arises in acyclic segments of a data flow graph. A clear illustration of the problem and solution is shown in the Figure 6 graph, the implementation of the ADFL expression: $\mathcal{T}[\![$**if** f=1 **then** *f1* **else** *f2* **end**$]\!]$. Although successive sets of inputs should be processed simultaneously, the control structure of the graph dictates that the overlap be very minimal. In order for a second set of values to enter the branches of the conditional, both α and β (Figure 6) must fire a second time presenting the sets of **T** and **F** gates with new control inputs. However, α cannot fire a second time until the **M** gate to which it also sends a control input has fired, to produce an acknowledge. Thus the d/a arc pair connecting α and the **M** gate (shown with slashes in Figure 6) creates a bottleneck whose severity depends on the depth of the computations performed within the branches of the conditional.

Eliminating this behavior so that successive sets of values may pipeline through the graph can be accomplished by inserting identity operators (buffers) along the slashed arc, breaking it into d/a arc segments which consequently

Figure 5. Replacement of one-place buffers with d/a arc pairs

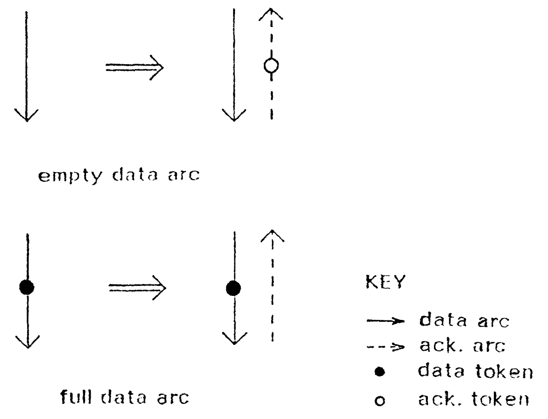
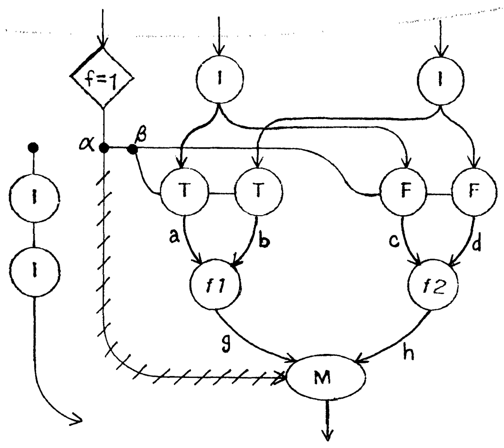


Figure 6. Insertion of buffers for a conditional expression



allow α to fire several times before forcing the **M** gate to fire. For the Figure 6 graph, this is accomplished by replacing the slashed arc with the arc segment shown to its immediate left. To generalize this optimization technique, a determination of the ideal number and location of inserted buffers must be made. This requires an analysis of data flow graph execution.

Though the data flow computer is asynchronous, it can be made to model a synchronous machine by assuming that during any given unit of time all enabled operators must fire and produce a result. This approximates optimal program execution by preventing an enabled operator from remaining enabled and thereby slowing up processing for any length of time.

Referring to Figure 6, we note that each input set to the graph will result in the production of a token on the control (slashed) arc and tokens that will be processed by either *f1* or *f2*. While under the "synchronous machine" assumption the tokens being processed by the functional operators can move one step through the graph during every time unit, the control token on the slashed arc cannot, restricting throughput to an output every fifth time unit. Adding identity operators to equalize buffer capacities achieves maximum pipelining, or equivalently, the optimal throughput of an output every *second* time unit. The algorithm presented below equalizes buffering.

Algorithm to Maximize Pipelining

Starting from each graph input, descend through the graph assigning consecutive numbers to the arcs joining successive sets of operators until a multi-input operator is encountered. Compare the arc numbers on the input arcs of the operator and:

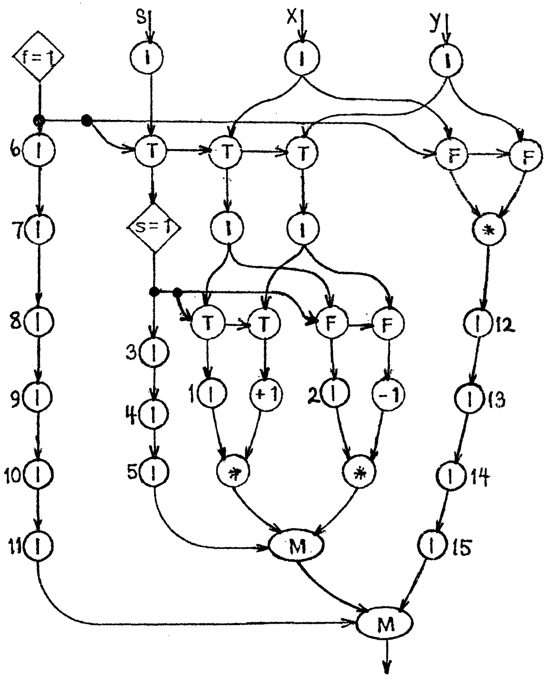
(a) if equal, continue the arc numbering process
(b) if not equal, balance the arcs by inserting identity operators into the lower numbered arcs. Renumber the modified arcs and continue the arc numbering process.

Note that if the operator is an M gate, the comparison and balancing process described above must involve all *three* input arcs, using the highest numbered arc as the goal. Figure 7 shows the result of applying this algorithm to the graph translation of the following program segment:

if f=1 then if s=1 then x*(y+1) else x*(y-1) end
else x*y end

For reference purposes, the added identities have been numbered. Identities I1 and I2 have been added in response to the imbalances which occur when comparing arc numbers on the input arcs to the *multiplication* operators. I3 through I5 are added in response to the comparison of input arcs to the inner M gate. Note that as specified in the algorithm, arc number comparisons involve all three **M** gate input arcs. Finally, operators I6 through I15 are introduced as a result of comparing input arcs to the outer **M** gate.

Figure 7. Example of maximal pipelining



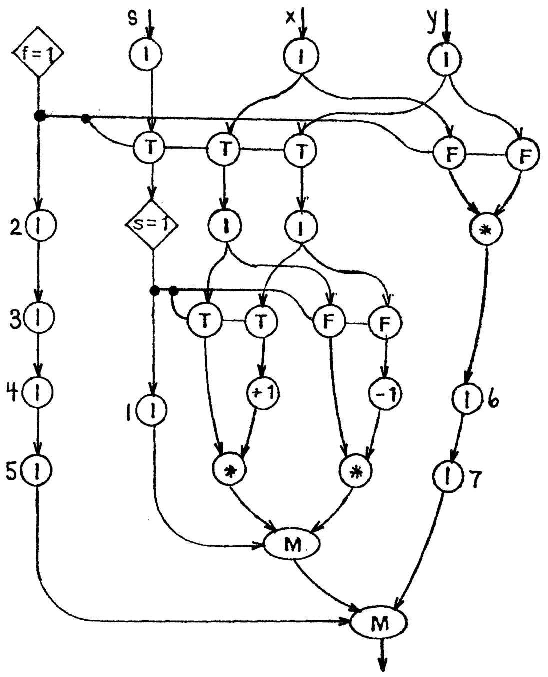
In applying the algorithm to this example, there are several interesting observations to make. Recall from the algorithm, that **M** gate comparisons must involve the two data arcs and the control arc. The algorithm modifies the graph to achieve maximum pipelining by making buffering capacities of the paths through the graph to the control arc and two data arcs the same. However, while each branch of the conditional operates in conjunction with the control arc, the branches themselves are *independent*. Thus, while each branch must pipeline with the control path, they need not necessarily pipeline with each other. If the two conditional paths are of different lengths, the pipelining choices available are to equalize the control path with either the shorter or the longer conditional branch, or to equalize all three. The latter of these, implemented by the algorithm above, achieves best throughput, but has the disadvantage of causing the insertion of additional identity operators in the shorter conditional branch. The other two choices recognize the independence of the two conditional paths and avoid excess buffering, but possibly at the cost of reduced throughput.

A factor not yet considered which interacts with this pipelining choice is the frequency with which graph paths are taken. In Figure 7 each input set can take any of three paths corresponding to the three possible states of f and s. If, for example, the pattern of input sets is such that no one of the three paths is taken twice in a row, identity operators I1 and I2 would be unnecessary and could be removed without decreasing the throughput. Illustrations of this point can be found in Montz [9].

The discussion of trade-offs and options to consider in maximally pipelining data flow graphs, indicates that the advantage of smaller size resulting from a less than maximally pipelined graph may be worth a decrease in throughput. Some key issues influencing the choice might include cost of identity operations, processor utilization, token flow patterns, and width and depth of program. By modifying the pipelining algorithm, we can produce data flow graphs which display *limited pipelining*, meaning that the delay between an operator's firing and receiving appropriate acknowledge signals may be several time units. For example, it is possible to specify that the delay in sending acknowledge signals be no greater than two time units. The change to the algorithm, which involves balancing arcs to within a specified bound, allows a graph to be easily reconfigured to display different degrees of pipelining, and thereby provides a feasible and practical control method of studying varying levels of pipelining in a graph. Though the details of the modified algorithm will not be given, we proceed by briefly comparing the Figure 7 graph with that of Figure 8 which can be produced using a limited pipelining algorithm.

The most striking contrast between the fully pipelined graph and this partially pipelined version is the large reduction in inserted identity operators, from 15 to 7. The question which arises is whether the cost of this reduction is a decrease in performance, where the Figure 7 graph displays the optimum performance by producing an output every *second* time unit. An analysis of several token flow patterns using different successions of input sets shows that the limited pipelining scheme does *not* necessarily degrade the throughput. This can be seen by pipelining three sets of inputs through the Figure 8 graph assuming that they respectively follow the paths indicated by the f-s values: true-true, false, and true-false.

Figure 8. Example of limited pipelining



Once an actual data flow machine is available, a study of the number of inserted identity operators vs. throughput trade-off should provide insight into the direction to take concerning optimization. This information in combination with a particular application should indicate other optimization possibilities; for instance, concentrating on only the main source of bottleneck within a graph. For the conditional construct this point appears to be the control arc to the **M** gate. Modifications of the pipelining algorithm could also be weighed more realistically as alternative approaches.

A final point to note in the consideration of this pipelining optimization strategy is that conditional constructs and general compositions of operators turn out to be fairly representative of the type of graphs for which this optimization is applicable. In fact, this optimization approach is basically inappropriate for an iterative process whose function is to modify and recycle a single set of inputs at a time (although subgraphs within an iteration may be pipelined). Thus an alternative optimization which aims to minimize the number of acknowledges in a graph by eliminating those which are unnecessary has been developed.

Eliminating Acknowledges

This optimization technique aims at decreasing overhead by removing acknowledge arcs which are not necessary to maintaining safe operation. This *safety* requirement is equivalent to guaranteeing that at most one token will reside on any arc of a data flow graph at any time. An examination of various ADFL constructs leads to the identification of arc pairs which are candidates for *acknowledge arc removal*. The strategy will be to develop a rule specifying the requirements for acknowledge arc removal for each candidate arc pair identified in the construct. By recursively applying the resulting set of rules to the data flow graph translation of an ADFL program, acknowledge arc removal for all candidate arc pairs can be determined.

To illustrate the analysis and formulate the desired rules, we begin by considering the data flow graph translation of the general conditional construct shown in Figure 6. As in the preceding section, the discussion centers on the arc pair connecting α and the **M** gate. However, while overcoming the restricting behavior of this arc pair was the focus of that optimization aimed at increasing pipelining, the restriction is an advantage to the process of eliminating acknowledges. Specifically, α, which *cannot* fire a second time until it receives an acknowledge from the **M** gate, guarantees that a second input set will not be within the branches of the conditional until processing of the preceding set has completed. Each input set (which will be processed by either *f1* or *f2*) places a token on the controlling arc of the **M** gate and a data token on each of the arcs labeled either **a** and **b**, *or* **c** and **d**, depending respectively on whether the control token was *true* or *false*. Assuming that *f1* and *f2* are well-formed, an output should appear on arc g (assuming the control token was *true*) within finite time, with the impossibility of a second token appearing on arc g, or of *any* token appearing on arc h until the **M** gate has fired. This firing simultaneously processes the token on arc g and sends an acknowledge token to α, consequent to which a successive input set may enter a branch of the conditional. This behavior guarantees that the acknowledge arc of the arc pair denoted by g can be safely removed. By an analogous argument we can remove the acknowledge arc of the arc pair labeled h.

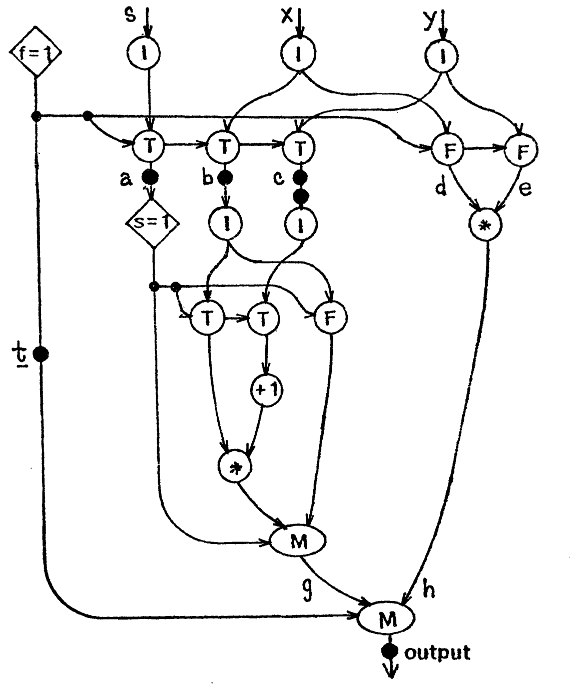
Using similar reasoning one might be tempted to remove the acknowledge arcs from arc pairs a, b, c, and d under the assumption that once a set of tokens has entered a branch of the conditional, the tokens must be used by the appropriate function to produce the corresponding output. However, a consideration of the Figure 9 data flow graph will show that removal of acknowledge arcs for these arc pairs is dependent on the subgraphs represented by *f1* and *f2*.

The Figure 9 data flow graph is a translation of the following ADFL program segment:

if f=1 **then if** s=1 **then** x*(y+1) **else** x **end**
else x*y **end**

Assume that the outer decision operator evaluates to *true* and that of the inner conditional construct previously represented by *f1*, evaluates to *false*. The important point to note is that an output can be produced using only the tokens on arcs *a* and *b*. The token on arc c need not

Figure 9. Unsafe token configuration resulting from removal of c's acknowledge arc



propagate through the graph, and may in fact still be on the arc when a successive set of values arrives. Removal of c's acknowledge arc would make it possible to reach the *unsafe* token configuration shown in Figure 9. This example shows that the necessity of acknowledge arcs for arc pairs a through e is dependent on whether or not their values are guaranteed to be used in producing the outputs of their appropriate subgraph (*f1* or *f2*). An analysis of the subgraphs in Figure 9 reveals that tokens arriving on arcs *a*, *b*, *d*, and *e* *must* be used to produce their corresponding output, while the need of a token arriving on arc c is dependent on the outcome of the inner decision operator. Therefore, we must leave *c*'s acknowledge arc, but can remove those of arc pairs *a*, *b*, *d*, and e.

This analysis, specific to the conditional construct, results in designating all input arc pairs to the *f1* or *f2* subgraphs subject to rule C1 with regard to acknowledge arc removal:

> C1: The acknowledge arc of an input arc pair to a subgraph may be removed if any token arriving on the arc *must* be used in producing the output of the subgraph.

This form of analysis must be recursively applied to subgraphs in determining acknowledge arc removal for both inner constructs and outer arc pairs. It is interesting to note that this rule could be applied at the source level by taking the intersection of variables appearing in the **then** and **else** clauses. Variables found in the intersection would be guaranteed to be used in producing the output, and in graph form would not require acknowledge arcs.

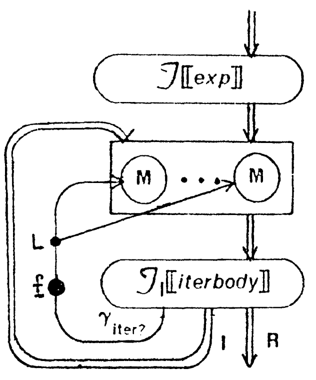
Referring to Figure 6, the arc pairs presenting inputs to the T and F gates have not yet been discussed with regard to acknowledge arc removal. Since the only way to guarantee the absence of a token on any of these data arcs is via the presence of a token on the corresponding acknowledge arc, these acknowledge arcs must remain. A final point concerns the initially discussed control arc connecting α with the **M** gate which may not need an acknowledge arc. The control arc of the inner conditional construct of Figure 9 is an example of such an occurrence which can be characterized by rule C2:

> C2: The acknowledge arc of the control arc connecting α and the **M** gate of a conditional construct can be removed if the acknowledge arc of the output arc pair of the **M** gate has been removed.

Developing a complete recursive algorithm to determine acknowledge arc removal in data flow graphs requires this type of analysis for each ADFL construct.

As a second example, we briefly examine the iteration construct shown in Figure 10 to identify candidate arc pairs for acknowledge arc removal. The arc labeled $\gamma_{\text{Iter?}}$, the control output of the iteration, provides the controlling value for the sequence of M gates handling the presentation of

Figure 10. $\mathcal{T}$[[**for** *idlist* = *exp* **do** *iterbody* **end**]]



successive sets of inputs to the iteration body. Since the $\gamma_{\text{iter?}}$ value is dependent on at least some of the M gate inputs, a number of them must fire before a second $\gamma_{\text{iter?}}$ value is produced. This necessarily implies the firing of the copy operator, "L", to present the M gates with new control inputs needed to re-enable them, ensuring that the $\gamma_{\text{iter?}}$ output arc from the iteration body to L must be empty for a successive $\gamma_{\text{iter?}}$ value to be produced. Consequently, the $\gamma_{\text{iter?}}$ arc needs no acknowledge. No such guarantee can be made for the arcs between the copy operator and M gates, acknowledges for which can be conditionally removed subject to rule T1:

T1: The acknowledge arc for an arc pair between operator L and the sequence of M gates can be removed if its data value *must* be used in producing the $\gamma_{\text{iter?}}$ value.

The output arc of the iteration body labeled I represents the arc pairs for the iteration variables. The analysis for these arcs is more complex and is governed by the following rule:

T2: The acknowledge arc of an I (iteration) arc pair can be removed if *either*

(1) The iteration body cannot emit a value on that output arc until it has absorbed the corresponding input value on the corresponding input arc.

(2) The $\gamma_{\text{iter?}}$ value depends on the corresponding input arc.

Examples involving the iteration construct, as well as an expanded discussion of these rules and an analysis of the remaining arcs can be found in Montz [9].

Conclusions

We have described a data flow language, an algorithm for translating its programs into data flow graphs, and two techniques for optimizing these graphs for execution with data flow machines of the Dennis-Misunas [7] design. While the two optimization methods have been presented as isolated techniques, they must be integrated into a single procedure for application to a given program.

We have not compared the costs of operation of the Dennis-Misunas [7] computer design with that of the Arvind-Gostelow [2] design, which avoids conflicts through the use of tagged values rather than acknowledge tokens.

Acknowledgments

We wish to thank Jack Dennis for supervising the theses [3, 9] on which this paper is based and Bill Ackerman for reading and criticizing drafts of this paper.

References

[1] W. B. Ackerman, and J. B. Dennis, *VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, Laboratory for Computer Science, Massachusetts Institute of Technology, TR-218, (July, 1979), 80 pp.

[2] Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time", *Information Processing 77: Proceedings of IFIP Congress 77* (August, 1977), pp. 849-853.

[3] J. D. Brock, *Operational Semantics of a Data Flow Language*, Laboratory for Computer Science, Massachusetts Institute of Technology, TM-120, (December, 1978), 55 pp.

[4] J. D. Brock, *Consistent Semantics for a Data Flow Language*, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Memo 172, (January, 1979), 30 pp.

[5] A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proceedings of the Fifth Annual Symposium on Computer Architecture, Computer Architecture News* 6 (April, 1978), pp. 210-215.

[6] J. B. Dennis, "First Version of a Data Flow Procedure Language," *Programming Symposium: Proceedings, Colleque sur la Programmation, Lecture Notes in Computer Science 19* (October, 1976), pp. 362-376.

[7] J. B. Dennis, and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *The Second Annual Symposium on Computer Architecture: Conference Proceedings* (January, 1975), pp 126-132.

[8] J. B. Dennis, and K.-S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," *Proceedings of the 1979 International Conference on Parallel Processing*, (August, 1979).

[9] L. B. Montz, *Safety and Optimization Transformations for Data Flow Programs*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, S. M. Thesis in preparation.

[10] R. D. Tennent, "The Denotational Semantics of Programming Languages," *Communications of the ACM 19* (August, 1976), pp. 437-453.

[11] K.-S. Weng, *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science, Massachusetts Institute of Technology, TM-68, (October, 1975), 93 pp.

LIMITING MULTIPROCESSOR PERFORMANCE ANALYSIS

Russell F. Vaughan and Mark S. Anastas
Boeing Aerospace Company
P.O. Box 3999
Seattle, Washington 98124

Abstract -- This paper describes an analysis of the major sources of overhead in multiprocessor systems with emphasis on performance equations for large systems. A model is developed for studying the relative contributions of these sources of overhead. The traditionally treated problem of memory contention is shown to be containable within bounds with limit equations provided. Software control table lockout on the other hand is shown to be beyond containment in large systems such that an upper limit on performance exists. Effective methods of reducing lockout overhead are explored. Control program efficiency is shown to be the only means of achieving very large multiprocessor systems which are efficient. It is shown also that if such efficiency could be obtained in a centralized control mechanism (by hardware or other means), there are no other immediate theoretical problems associated with increasing multiprocessor size.

Introduction

There are known limitations to single processor approaches to increasing general purpose computer throughput capabilities [8]; moreover, requirements for increased throughput seem more general and insatiable than ever. The advent of inexpensive microprocessors has emphasized the necessity for an effective multiprocessing technology capable of effectively combining many processors to obtain significant throughput. The cost advantages of multi-microprocessors over high speed main frame processors provide a natural motivation for re-evaluating the problems previously encountered in large MIMD multiprocessing systems. It is therefore the limiting performance behavior where many processors are involved that is the central theme of this paper.

The theoretical problems associated with deadlock avoidance and synchronizing concurrent processes have been solved. [3],[9],[13] The practical problems however, which are encountered when implementing large multiprocessing systems have seemed unavoidable. To address these practical issues, a general parameterized model of the major overhead contributions in multiprocessing systems is presented. Descriptions of the individual overhead contributions modeled separately are found in the literature, but not integrated mathematical models as presented here. Nor has the emphasis of these other models been on performance expectations in the limit as system size increases. The model described in this paper relates the three major contributions to overhead in multiprocessing systems to the desired application program processing requirements in order to assess potential performance capabilities. A diagramatic illustration of the modeled sources of overhead is provided in Figure 1. These are the following:

1. System Control. The multiprocessor executive control program execution time requirements.

2. Control Table Lockout. To provide coordinated control, common queues are required which imply critical sections in the control program which accesses these queues.

3. Memory Contention. Common physical memory for multiple processors requires the possibility of multiple processors converging on the same physical memory module, in which case a processor may have to wait until other processors' access requests have been serviced.

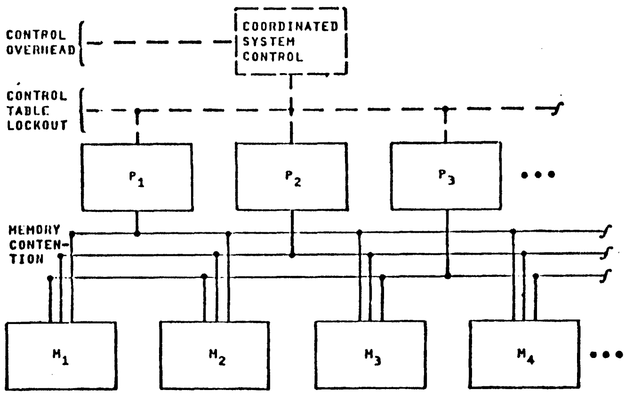


FIGURE 1: MODELED SOURCES OF OVERHEAD

To obtain a comprehensive model of multiprocessing overhead, without inappropriate complexity, a hierarchical model has been developed. The levels and states in this hierachy are the obvious ones. Figure 2 is a state diagram of the time expenditure states at the top level in this model of a multiprocessor system. These states are: P, the normal processor operations associated with instruction sequencing and performing the instructions in its repertoire, and C, the memory delays which may include sequences awaiting memory contention resolution. In order for this model to be valid, both the spatial and temporal distributions of memory access requests must be constant and independent of the changing occupations of the processor. These assumptions are characteristic of current multiprocessing. (One of the

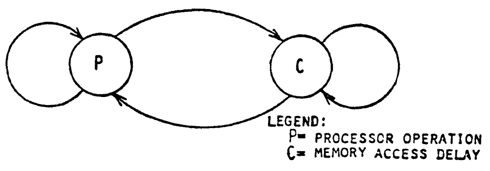


FIGURE 2: TIME EXPENDITURE STATES

design trades considered further on investigates potential advantages resulting from changing the temporal distribution.)

Time overhead (throughput) is the multiprocessing concern here. Other aspects of multiprocessing including memory and peripheral sizing have been modeled in reference [6]. These other aspects are very important in a system, and should be optimized to obtain the best performance for any given configuration. But they are not the major obstacles to a viable multiprocessing capability.

Processor Time Expenditure Model

The time utilization characteristics of the various activities that can be assigned to the processor are modeled here. In a multiprocessor system, it is expected that for some of these activities the amount of time expended may be dependent upon the number of processors, N. (This definition of N will be assumed throughout the rest of this paper.) The P state of the processor shown in Figure 2 can be modeled in more detail as shown in the state diagram of Figure 3. The four

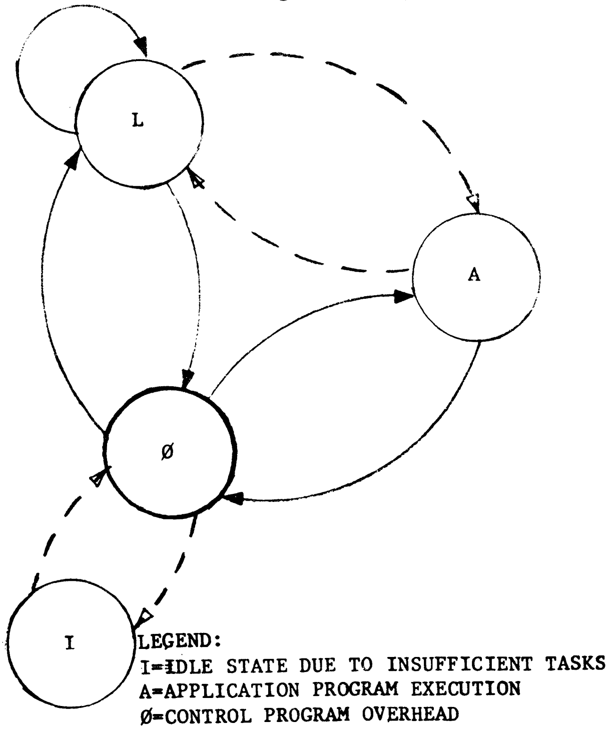

L
A
Ø
I
LEGEND:
I=IDLE STATE DUE TO INSUFFICIENT TASKS
A=APPLICATION PROGRAM EXECUTION
Ø=CONTROL PROGRAM OVERHEAD
L=CONTROL TABLE LOCKOUT STATE

FIGURE 3: SECOND LEVEL: TIME EXPENDITURE SUBSTATES WITHIN THE PROCESSOR TIME EXPENDITURE STATE, P

states in this diagram are the following:

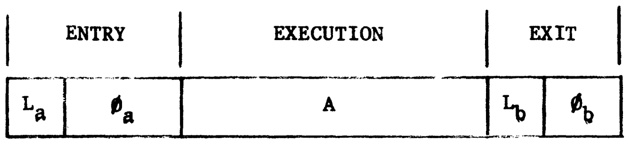
1. Idle state, awaiting an eligible application program task,

2. Application task execution,

3. Control program execution, and

4. Control table lockout.

In order to get performance predictions independent of the software configuration, it has been assumed that the idle state will be null. We are only interested here in performance degradation not attributable to insufficient jobs to go around. (Utilization considerations will be discussed later on however.) It is also assumed that lockout will only be experienced as a part of the control program execution, and is therefore called control table lockout. Critical sections in the application program are assumed to be resolved by task eligibility considerations handled by the control program. To resolve such conflicts in the application programs is not the direction of high performance multiprocessing, since excluding the parallel execution of such programs improves throughput. The timeline in Figure 4 shows the phasing among the remaining three states.



A = APPLICATION PROGRAM EXECUTION TIME

$\emptyset = \emptyset_a + \emptyset_b$ = MULTIPROCESSING EXECUTIVE OVERHEAD

$L = L_a + L_b$ = LOCKOUT OVERHEAD FOR ACCESSING CONTROL INFORMATION COMMON TO MULTIPLE PROCESSORS

FIGURE 4: TASK TIMELINE BASIS FOR PROCESSOR OVERHEAD

Each of the three remaining processor time expenditures is modeled very simply in the following. An equilibrium situation is assumed among the states, so that the numbers of processors entering and leaving each state are approximately equal. The level of sophistication could obviously be increased appreciably in these models, but it has been found that performance predictions are relatively insensitive to such improvements. The simpler models are easier to describe and understand, and fit existing multiprocessor performance data very adequately.

Application Program Task Execution

The model of application task execution involves a constant execution time requirement, A, for all tasks with a single queue/dispatch/exit control program request overhead. The model is still valid for programs making multiple requests so long as the ratio of application to control program execution time, $\rho \equiv A/\emptyset$, is a contant. This ratio is used extensively later on in the analytical derivation of performance; it is called the individual processor efficiency. It is affected only by the control program overhead per application task, defined so as to exclude the effects of lockout induced by multiple processors.

Control Program Execution

The execution time of the control program is assumed to be broken into J partitions. These partitions are assumed to be mutually exclusive critical sections with equal execution frequency as well as execution time, $\emptyset_j$.

$$\emptyset \equiv \sum_{j=1}^{J} \emptyset_j = J\emptyset_j$$

The control program is assumed to require the same constant total amount of execution time, $\emptyset$, for each task. It is also assumed that its execution time is independent of the number of processors in the system. The latter of these assumptions supposes that queues are implemented with multiple pointers such that the lengths of queues do not result in a commensurable amount of searching to process linked task lists. This seems to be a unilateral approach to sophisticated control programs appropriate to multiprocessing.

Control Table Lockout

Coordination of the activities of many processors to achieve a single computational objective requires the control program to have common task queues for exploiting the parallel aspects of individual application programs. It is assumed that control table lockout occurs at entry to each of the J control program partitions, each of which is comprised of a mutually exclusive critical section. The total amount of lost time due to this control table lockout will be:

$L = \sum_{j=1}^{J} L_j$, where L_j is the amount of lockout

attributed to the jth critical section.

In order to derive an expression from which a value can be computed for the overhead L, we will define N_j as the number of processors waiting and/or executing the jth critical section in the control program. From this definition it can be seen that the amount of lockout time a processor will experience before entering the jth critical section will be $L_j = N_j\emptyset_j = N_j \frac{\emptyset}{J}$. N_j can be determined as the probability P_j of an individual processor being in this jth state, times the number of possible competing processors, N-1 in this case. The probability P_j can be determined as the proportion of time spent in the jth state to the total amount of time spent by each processor.

$$P_j = \frac{\emptyset_j+L_j}{A+\emptyset+L} = \frac{(1+N_j)}{J(\rho+1+N_j)}$$

Thus, since $N_j = P_j\cdot(N-1)$, we obtain a second order equation for N_j:

$$N_j = \frac{(1+N_j)\cdot(N-1)}{J(\rho+1+N_j)}$$

The formal solution to this equation is:

$$N_j = \frac{N-1}{2J} - \frac{\rho+1}{2} + \sqrt{\left[\frac{N-1}{2J} - \frac{\rho+1}{2}\right]^2 + \frac{N-1}{J}}$$

For J=1, we obtain:

$$L = \emptyset\left[\frac{N-\rho}{2} + \sqrt{\left(\frac{N-\rho}{2}\right)^2 + \rho} \;\; -1\right]$$

The expected number of locked out processors, N_l is plotted in Figure 5 for various values of ρ.

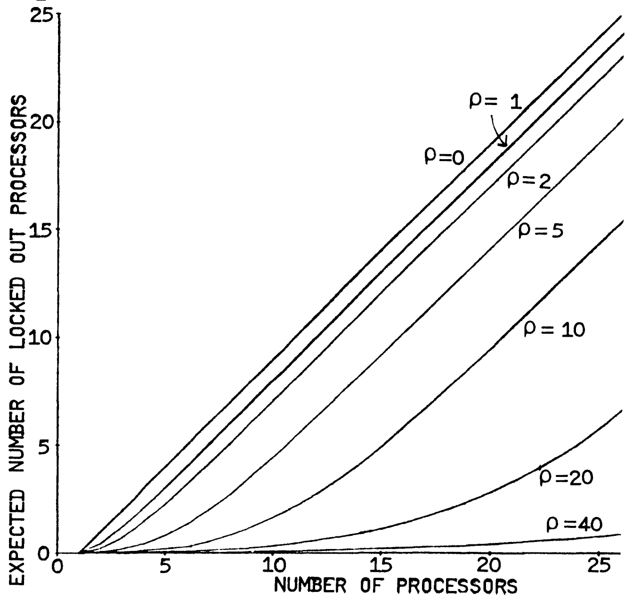


FIGURE 5: IMPACT OF PROCESSOR EFFICIENCY ON LOCKOUT OVERHEAD

These curves are in agreement with Madnick [15] in spite of a very different derivation. The significance of increasing the effective partitions in the control program will be discussed further on.

Combining Processor Time Expenditures

The objective of the processor activity modeling has been to obtain insight into the relative amount of time spent by each processor in its A, $\emptyset$ and L states. Equivalently we are interested in knowing the total number of processors in the configuration occupied by each activity. This assessment can be obtained by establishing the ratios of time spent in each activity to the unit of a processor's time. By defining X_A, $X_\emptyset$, and X_L as the respective ratios for the A, $\emptyset$, and L activity states, it can be seen that:

$$X_A + X_\emptyset + X_L = \frac{A}{A+\emptyset+L} + \frac{\emptyset}{A+\emptyset+L} + \frac{L}{A+\emptyset+L} = 1$$

Furthermore, the equivalent number of processors involved in each activity per unit time N_A, $N_\emptyset$, and N_L can be determined as:

$$N_A = X_A\cdot N, \; N_\emptyset = X_\emptyset\cdot N, \; N_L = X_L\cdot N$$

In order to establish these relative contributions, we can substitute in the results obtained previously. The unit of processor time for J=1 is thus seen to be:

$$U \equiv \rho+1+L/\phi = \left[\frac{N+\rho}{2} + \sqrt{\left(\frac{N-\rho}{2}\right)^2 + \rho} \right]$$

$$N_A = \rho \cdot N \cdot U^{-1}$$

$$N_O = N \cdot U^{-1}$$

$$N_L = (U-\rho-1) \cdot N \cdot U^{-1}$$

Memory Contention Delay Model

There are various memory/processor interconnection schemes that can be employed for access arbitration including multiport controllers and crossbar switches as described by Enslow [10] which effect the logical interconnecting paths shown in Figure 1. Specific configuration dependencies such as processor clock phasing, memory address interleaving, processor to memory speed ratios, and processor memory request duty cycle are discussed in reference [17]. The mathematical modeling of the performance to be expected of configurations incorporating such dependencies is addressed here.

In a general multiprocessor configuration with M memories and N processors, we are concerned with the percentage of time that the processors spend waiting for a memory to service their requests [2],[12].

General Model of Synchronous Interleaved Memory

To simplify the model we have assumed equal likelihood of a processor accessing any of the memories on a given request. Address interleaving makes that a realistic assumption. In addition, it has been assumed that each processor synchronously makes a memory access each cycle; this is a worst case situation tending to make the resulting performance predictions pessimistic rather than optimistically unrealistic.

We will begin by defining the probability, $P_S(i)$ of exactly i processors converging on single memories anywhere in the system on a given access:

$$P_S(i) = \sum_{j=1}^{\lfloor \frac{N}{i} \rfloor} p_S(i,j),$$ where $\lfloor x \rfloor$ is the largest integer less than or equal to x, and $p_S(i,j)$ is the probability that there are j instances of exactly i processors converging on single memories in the system. (For a detailed treatment of probability theory, refer to Feller [11].) To proceed, we will consider the conditional probabilities $p_p(i,j)$, and $p_m(i,j)$ which are respectively the probabilities of a processor and a memory being involved in an i-way convergence of processors on memories if there are j instances of such convergence in the system. Under the random accessing and equivalence between processors assumptions that we have made:

$p_p(i,j)=i\cdot\frac{j}{N}$, since ixj of the N processors are involved.

$p_m(i,j)=\frac{j}{M}$, since j of the M memories are involved.

Now the unconditional probabilities of processors and memories being involved in i-way convergence situations can be determined as:

$$P_p(i) = \sum_{j=1}^{\lfloor \frac{N}{i} \rfloor} p_p(i,j) \cdot p_s(i,j) = \frac{i}{N} \sum_{j=1}^{\lfloor \frac{N}{i} \rfloor} p_s(i,j) \cdot j$$

$$P_m(i) = \sum_{j=1}^{\lfloor \frac{N}{i} \rfloor} p_m(i,j) \cdot p_s(i,j) = \frac{1}{M} \sum_{j=1}^{\lfloor \frac{N}{i} \rfloor} p_s(i,j) \cdot j$$

And therefore: $P_p(i) = i \frac{M}{N} P_m(i)$

Modeling Memory Response Time

So far we have only been dealing with the probabilities of processor/memory convergence, whereas what we are really interested in is contention situations where processor time is lost. We therefore assume that there is some number, k (not necessarily unity, but for convenience a positive integer) of processors whose requests can be accommodated by each memory module without any of the contending processors experiencing delays. k is the ratio of processor request time over memory response time. A new conditional probability, $P_R(i)$ can therefore be defined which is the probability that a processor involved in an i-way convergence situation will actually experience contention:

$$P_R(i) = \frac{(i-k)}{i}, \text{ for } i>k; \; P_R(i) = 0, \text{ otherwise.}$$

Then the probability a processor will experience memory contention due to i-way convergency situations is:

$$P_C(i) = P_R(i) \cdot P_P(i)$$

$$P_C(i) = (i-k) \cdot \frac{M}{N} \cdot P_m(i), \text{ for } i>k;$$

$$P_C(i) = 0, \text{ otherwise.}$$

The total probability of a processor experiencing memory contention P_C, can be computed as:

$$P_C = \sum_{i=k+1}^{N} P_C(i),$$ since contention can only occur when i>k. Therefore we have:

$$P_C = \frac{M}{N} \sum_{i=k+1}^{N} P_m(i) \cdot (i-k)$$

Approximating the Distribution Function

We are left then with the requirement for obtaining a distribution function $P_m(i)$. Many such models of processor queueing on individual memories have been advanced [2],[7]. It has been shown that little accuracy advantage accrues from selecting the more sophisticated models involving Markov chains. This is particularly applicable for the configurations discussed in this article where memory contention is shown to be small, since we are primarily interested in configurations for which $M \geq N$ and $k \geq 1$. Bhandarkar [5] has shown percentage errors of less than 5 percent in all cases for the model assumed here.

The model that we have selected is the binomial approximation of Strecker [16] which was found to "work well in all cases" by Baskett and Smith [4] and with more accuracy for $M \geq N$ by Bhandarkar [5]. This model is precisely valid for the initial allocation of processors to memories under the assumptions made previously.

According to this model, the probability that exactly i processors converge on a given memory module on a given cycle is:

$$P_m(i) = \binom{N}{i}\left(\frac{1}{M}\right)^i\left(1-\frac{1}{M}\right)^{N-i},$$

where $\binom{N}{i} \equiv \frac{N!}{i!(N-i)!}$

Therefore, according to this model:

$$P_c = \frac{M}{N}\sum_{i=k+1}^{N} \frac{N!(i-k)}{i!(N-i)!}\left(\frac{1}{M}\right)^i\left(1-\frac{1}{M}\right)^{N-i}$$

The form of P_c as a function of the number of memory modules is shown for N=20 processors in Figure 6. The impact of varying the relative

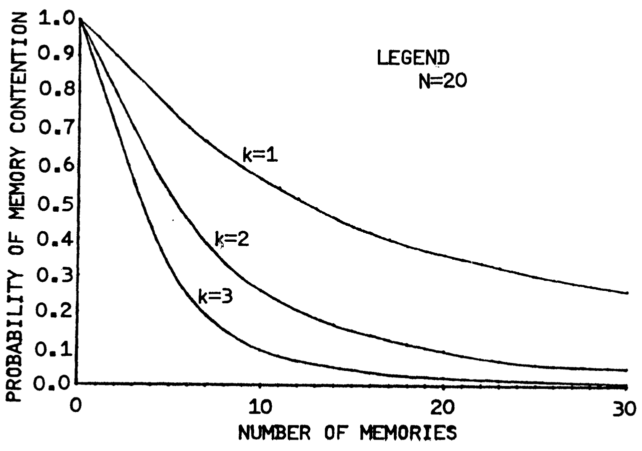


FIGURE 6: IMPACT OF RATIO OF PROCESSOR REQUEST TO MEMORY RESPONSE TIMES

speed, k of memory access and processor request logic is illustrated in the figure, applying respectively for k=1, 2 and 3.

It should be noted that all of the convergence and contention probabilities are functions of M, N, and k, specifically $P_c = P_c(M, N, k)$. The probability distribution functions P_p and P_m are functions of M and N as well as i, e.g., $P_m(i) = P_m(i, M, N)$.

Limiting Behavior of "Square" Systems

It is interesting to note that memory contention decreases very rapidly with M until the numbers of memories and processors are approximately equal (M=N), and very slowly thereafter. We will refer to systems for which M=N as "square" multiprocessors, and define the notation: $P_c(M{=}N,\emptyset,k) = P_c(M,M,k) = P_c(N,N,k)$. To understand the significance of configuring multiprocessors with approximately equal numbers of memory modules and processors, consider the limiting values of $P_c(M,N,k)$ as M and N become large. The limits of the summation can be changed to obtain:

$$P_c = \frac{M}{N}\sum_{i=0}^{N}(i-k)\,P_m(i) - \frac{M}{N}\sum_{i=0}^{k}(i-k)\,P_m(i)$$

Then noticing that $\sum_{i=0}^{N} P_m(i,M,N)=1$, we obtain:

$$P_c = \frac{M}{N}\sum_{i=0}^{N} i\,P_m(i) - k\frac{M}{N} - \frac{M}{N}\sum_{i=0}^{k}(i-k)\,P_m(i)$$

To obtain a limit for P_c, we have substituted M=N into $P_m(i,M,N)$ and used the limit

$$1/e = \lim_{N\to\infty}\left(1-\frac{1}{N}\right)^N.$$

Then for "square" systems:

$$\lim_{N\to\infty} P_c(M{=}N,k) = 1-k+\frac{1}{e}\sum_{i=o}^{k}\frac{(k-i)}{i!}$$

The limting values for k=1,2, and 3 are shown in Table I.

TABLE I: LIMITING MEMORY CONTENTION PROBABILITIES

Relative Speed (Memory to Processor, k)	Asymmetry Ratio (Numbers of Processors to Memories, η)	Limiting Contention Probability Limit $P_c(M,N)$) $M,N\to\infty$
1	1	0.368
2	1	0.104
3	1	0.023
1	1/2	0.213
1	1/3	0.150

Incorporating Access Duty Cycle

In real systems there is typically not exactly one memory access per processor per request cycle, and the processors are not synchronized relative to whether they actually access memory on a given cycle. There are two typical processor characteristics which are responsible.

1. Processor operations do not typically require an access on every cycle of the instruction. Statistically, somewhat less than half of the TI 9900 microprocessor machine cycles require a memory access, for example.

2. Some processors implement a cache memory scheme for look-ahead memory accessing to reduce the average wait time in the processor. This reduces the number of cycles for which the processor makes memory accesses, but substantially increases the number of accesses outstanding when they are made.

These (in general combined) phenomena establish an effective, although statistically varying memory access duty cycle. These characteristics of real systems cannot be modeled by varying the memory to processor speed ratio, k. However, at least where large numbers of processors are assumed, and approximately constant access duty cycle, d can be expected which will alter the apparent number of processors actually making memory accesses at any particular cycle to an equilibrium value for large systems of $N= d . N'$. Real "square" systems would then be characterized by the model as "rectangular" systems of dimensions $N= \eta . M$, where η is the apparent asymmetry ratio.

Limiting Behavior of "Rectangular" Systems

It is interesting to consider memory contention effects when system size is increased in congruent rectangular form. Just as was the case for "square" systems, it can be seen that for large "rectangular" systems the contention probabilities level off to approximately constant values. Chang, Kuck and Lawrie [8] derived an expression for the limit from the memory's viewpoint (the probabilitiy of a memory rather than a processor being involved in a contention situation). The results do not incorporate the speed ratio, k.

Limiting processor contention in large "rectangular" systems can be derived using the same approach as described previously for "square" systems.

$$\underset{M\to\infty}{\text{Limit}}\ P_c\ (N= \eta \cdot M,k) = 1 - \frac{k}{\eta} + \frac{1}{e} \sum_{i=0}^{k} \frac{(k-i)\ \eta^{i-1}}{i!}$$

Accuracy considerations relying on Bhandarkar's [4] data suggest $\eta \leq 1$ as the primary domain of usefulness for this equation. The limits for k=1 and for asymmetry values $\eta = 1$, 1/2 and 1/3 are shown in Table I. The asymptotic approach to these limits is shown in Figure 7.

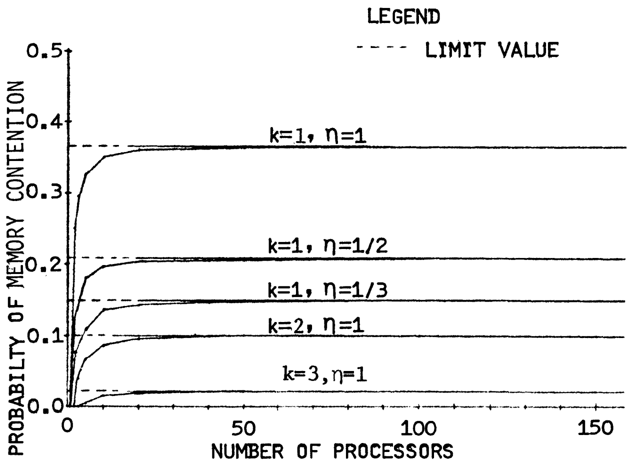


FIGURE 7: ASYMPTOTIC APPROACH TO LIMITING MEMORY CONTENTION PROBABILITY VALUES

Combining Processor and Memory Contention Overhead

In the previous accounting of processor time expenditures, there were only three categories corresponding to the three processor states of application program, control program and control table lockout. It must now be acknowledged that not all of the time spent in these three states is correctly attributed to these causes, since memory contention takes a proportional amount of time from each. By this assumption, we have: $C=(A+\emptyset+L) \cdot P_c$. Thus, if we define the respective primed quantities to represent the time in each state exclusive of memory contention, we have: $A+\emptyset+L=A'+\emptyset'+L'+C$ and therefore: $A'+\emptyset'+L'=(A+\emptyset+L)(1-P_c)$ and the respective number of processors in a multiprocessor configuration expended in the various states are the following:

$$N'_A = N_A\ (1 - P_c)$$

$$N'_\emptyset = N_\emptyset\ (1 - P_c)$$

$$N'_L = N_L\ (1 - P_c)$$

$$N_C = (N_A + N_\emptyset + N_L) \cdot P_C = N \cdot P_C$$

The form of N_C is independent of N_A, $N_\emptyset$, and N_L. N_C increases linearly with increasing system size for congruent rectangular increases, with the slope depending upon the relative speed of the memories and processors and the asymmetry ratio. This phenomenon is shown in Figure 8. The dashed line represents the extrapolation from data presented by Bhandarkar [5] which resulted from a more accurate Markov chain model for k=1, η=1.

The form of the other three expected numbers of processors N'_A, $N'_\emptyset$ and N'_L can be obtained by substitution from previously obtained solutions for N_A, $N_\emptyset$, N_L and P_C. It should be clear that

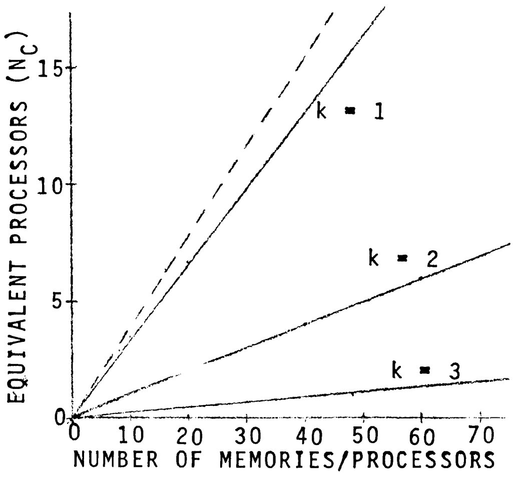


FIGURE 8: N_C FOR "SQUARE" SYSTEMS

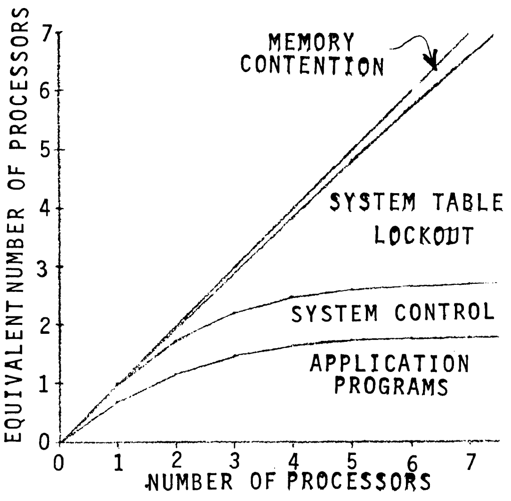


FIGURE 9: OVERHEAD CONTRIBUTIONS FOR $\rho = 2, k = 1, \eta = 1, M = 50$

N'_A provides a desirable measure of throughput in multiprocessors. It provides the effective number of processors being applied to the application programs.

To understand the importance of individual processor efficiency on multiprocessor throughput performance, it is interesting to look at the form of N'_A (ρ):

$$N'_A = \frac{\rho \cdot N \cdot (1-P_C)}{\frac{N+\rho}{2} + \sqrt{\left(\frac{N-\rho}{2}\right)^2 + \rho}}$$

For large N there is an asymptotic approach to a limiting throughput, τ , and this limit is:

$$\tau \equiv \lim_{N \to \infty} N'_A = \rho \cdot (1-P_C)$$

The trailing factor may approach a limit as well, since in general P_c is a function of N. Thus, the control program efficiency not only determines the utilization per processor, but also the maximum achievable throughput of the entire machine. In Figure 9 (which represents state of the art capabilities in large scale multiprocessor systems) there is a maximum achievable return (even with P_e=0) of two equivalent processors applied to application programs. By adding any number of processors beyond 4, the most that will be gained is 0.35 equivalent processors applied to application programs.

The previous equation also indicates the impact of memory contention on maximum performance. Memory access efficiency, ρ_M, the probability of not experiencing contention on an access can be defined as follows: $\rho_M = 1-P_C$. Then maximum throughput, τ , for the whole system is equal to the product of the efficiencies of an individual processor, ρ_p, computed with no contention or lockout, and ρ_M of the memory accesses:

$$\tau = \rho_p \cdot \rho_M.$$

Design Trades in Multiprocessors

It is significant that in the example shown in Figure 9, memory contention is not responsible for the reduced efficiency of processors as a function of their increased number. This is not to say that memory contention cannot be a very significant overhead factor, but rather that it is a problem which has been solved by the existing multiprocessing technology. In the example, memory contention is reduced to insignificance by the large number of memory modules (M>>N). Another method which solves the memory contention problem, which is particularly appropriate in microprocessor systems, is increasing the relative speed of the memories. These solutions are appropriate respectively to large mainframe configurations requiring a large memory base to perform their normal operations, and to microprocessor-based systems for which it is not a stringent requirement to obtain relatively fast memories.

Reducing Memory Contention

There are of course many configurations for which memory contention appears to be very significant. In the solid lines in Figure 10, the situation previously presented in Figure 9 has been modified to include only 5 rather than 50 memory modules. In this example, there is actu-

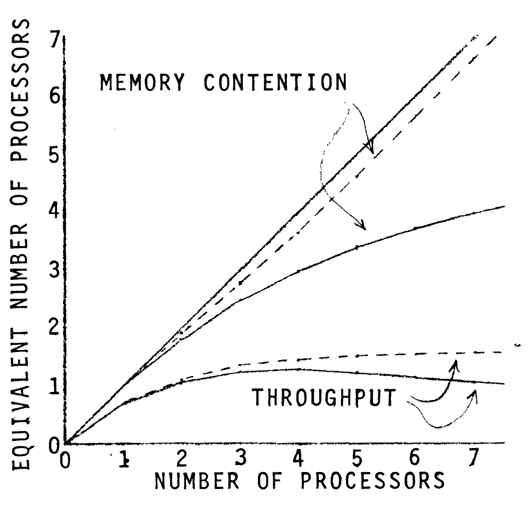


FIGURE 10: REDUCING MEMORY CONTENTION

ally a negative improvement in application program throughput for more than 4 processors. The reason for this negative return can be seen to be attributable to the increasing number of locked out processors. These processors are assumed to access semaphores in main memory and thereby contribute heavily to memory contention and are not productive even when successful. This phenomenon can be eliminated by assuming the semaphores are stored in a special purpose memory dedicated to semaphore control. In this case the ratio of the numbers of processors in the three processor states independent of contention are the same. The effective number of processors competing for memory is reduced, however, to N_A+N_O. By estimating P_C for 5 memories and N_A+N_O processors, we obtain the revised overhead plots shown as dotted lines. The marginal gain in performance for few processors can be seen. Memory contention has been effectively reduced, but the advantage has largely been taken up by increased lockout and system control overhead. This example illustrates the very important point that memory contention can be reduced to insignificance without a commensurable return in throughput. See also Flores [12] for a similar conclusion. Memory contention is not the peril of multiprocessing.

Reducing Processor Lockout

It is clear from the preceding discussion that lockout is the primary contributor to multiprocessor inefficiency for large numbers of processors. Let us therefore consider various means by which it can be reduced. The starting point of course is the consideration of the assumptions that went into the model of control table lockout. The primary assumption was that the control tables are locked out throughout the execution of the control program. Thus, the approaches in attempting to resolve the control table lockout problems are:

1. Design a control program employing a more limited use of lockout,

2. Reduce the execution time of the critical sections in the control program, and

3. Partition the control program into many separate rather than a single common critical section.

The relative effectiveness of the various methods of reducing lockout overhead ultimately depends upon the design of the control program itself. There are upper limits for each of these methods. The amount of processing power released to application programs as the result of improvements in these areas will be discussed below. For few processors (small N) the advantage of reducing the length of critical sections or increasing the number of partitions is negligible, whereas an improvement in control program overhead is an immediate advantage even for few processors. For large N the improvement in performance has the same form for reducing extent of critical sections and improving efficiency.

It should be apparent that these three solutions have direct analogs in the reduction of memory contention which are respectively: Reducing accesses to common memory, increasing the relative speed of memory response logic, and increasing the number of independent memory modules which can be accessed. Solutions incorporating the three approaches to lockout are illustrated in the following discussion, with the improvements all being relative to the system whose performance characteristics were shown in Figure 9. Line A in Figure 11 represents this baseline system's throughput performance.

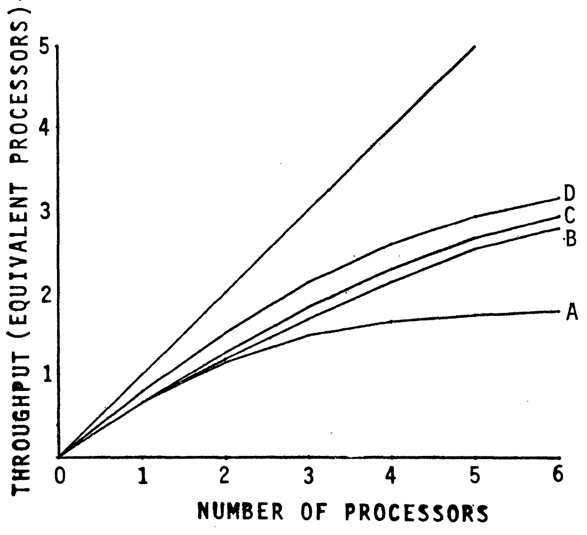


FIGURE 11: COMPARISON OF APPROACHES TO REDUCING CONTROL TABLE LOCKOUT

Limiting Control Program Lockout. It is not actually necessary for the entire control program to be locked out such that only one processor can be executing it at any one time. In Figure 11, line B, the expected performance is shown for a system whose control program need be locked out only half of the time. The data for the figure was obtained using a different value of ρ for lockout than for determining the proportion of useful work performed. $\rho_{LOCKOUT}$ can be computed as the total amount of time the processor spends in non-locked-out processing states divided by the amount of processing time spent in states for which lockout is required. In the current model this can be expressed as:

$\rho_{LOCKOUT} = \frac{A+\emptyset(1-Z)}{Z\emptyset} = \frac{\rho+1}{Z} - 1$, where Z is the proportion of the control program requiring lockout. In Figure 11, line B, $\rho=2$, Z=0.5, and therefore $\rho_{LOCKOUT} = 5$.

Control Program Efficiency. An obviously effective method of improving multiprocessor throughput is by directly decreasing the execution time of the critical section portions of the control program. Figure 11, line D illustrates the performance to be expected if the efficiency of the control program were improved by a factor of 2. In this case $\rho=4$ instead of $\rho=2$.

Partitioning the Control Program. The lockout which is necessary in control programs does not necessarily lock out all of the critical sections in the program. Earlier, an equation was developed for lockout assuming there were J partitions of the control program with independent critical sections. This equation was used to obtain the performance indicated in Figure 11, line C, for J=2. In reference [18] it was also suggested that a small number (2, 3 or 4) or partitions significantly improve efficiency. It should be obvious that the limiting number of partitions that could be incorporated is not a large number however.

Increasing Individual Processor Efficiency.

The level at which application programs interface with the control program has the same impact on efficiency as does the overhead involved in the control program. If the execution time of the typical application program task is increased such that the number of executable instructions is doubled, the same efficiency advantages will accrue as if the overhead of the control program were reduced to one-half its original value. One must be careful in this regard, however, since the utilization of processors can be significantly reduced. Utilization was ignored in this article by assuming that there are no processors in the idle state. (See Figure 3.) The job control languages of batch processing systems largely determine the task level. This is a critical issue particularly in mainfram multiprocessing, but one which is beyond the scope of the current article.

Kuck [14] investigated the potential for breaking up general problems into parallel segments to obtain commensurable speedup. The inherent parallelism was shown to be roughly proportional to the size of the application program, if the program units which are dispatched are taken to a low enough level. This is in contrast to what was formerly thought to have been an order of log relationship [14]. Thus, there is potential in the programs themselves for solution by parallel arrays of slow processors to obtain very high throughput. But this level would reduce the effective value of A by orders of magnitude which in turn reduces ρ (and with it feasibility) by orders of magnitude. And thus, methods which artifically increase ρ do not attack the multi-microprocessor program.

The Future of Multiprocessing

It has been demonstrated that the high leverage design considerations in multiprocessing at this time are control table lockout and the control program overhead. Hardware support for the multiprocessor executive is the obvious place to look for help, since the improvement required to realize large arrays of processors is orders of magnitude rather than simple multiples.

Let us consider the potential of such solutions to determine whether there are other theoretical problems. Figure 12, line A illustrates the system described originally in Figure 9, but assuming an individual processor efficiency of $\rho = 100$. In this configuration memory contention becomes appreciable after about 10 or 20 processors, and the maximum achievable throughput is seen to be about 40 processors. But as shown in Figure 12, line B, the asymptotic limit can be more than doubled by increasing the relative speed of the memory response logic. In this case k=3 rather than k=1.

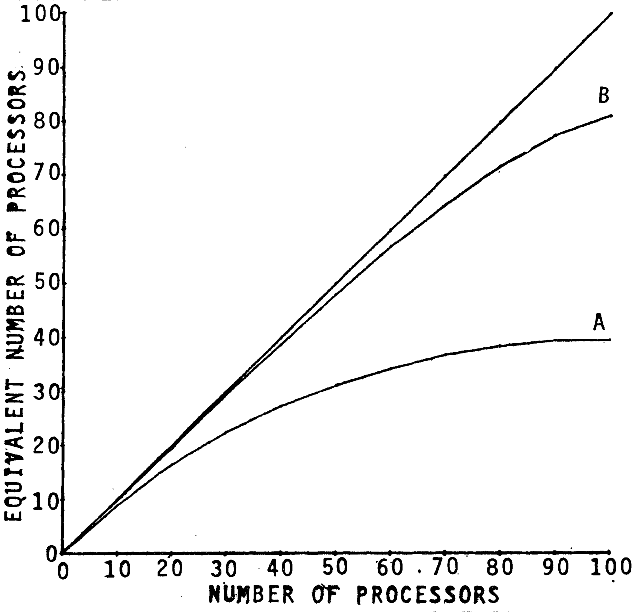


FIGURE 12: IMPROVING THROUGHPUT IN HYPOTHETICAL SYSTEMS

In going to such high throughput systems, however, there would be requirements for commensurably larger numbers of memory modules. Figure 13, line A, illustrates the situation for ρ =100 with "square" multiprocessors (M=N) and k=1. The improvement in contention with increasing memory response time can be seen in lines B and C respectively for k=2 and k=3.

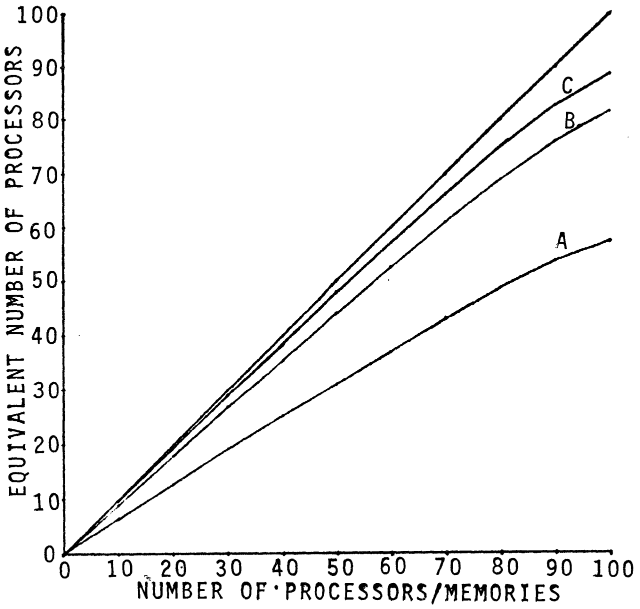


FIGURE 13: HYPOTHETICAL SYSTEMS WITH THE SAME NUMBER OF PROCESSORS AND MEMORIES

Conclusions

It has been shown that at least analytically there are no size limitations to conventional multiprocessing approaches which are beyond the current state of the art except control program efficiency. Hardware seems to be the only effective way of significantly increasing this parameter. Exploring methods of increasing hardware support for the control programs is therefore the most likely avenue to extending the limits for multiprocessor throughput performance. A companion paper discusses such an approach for which multiprocessor control can be made extremely efficient [1].

References

[1] Anastas, M. S. and Vaughan, R. F., "Parallel Transition Machines", Proc. of 1979 Int. Conf. on Parallel Processing (Aug. 1979)

[2] Asher, J. R. and Skinner, C. E., "Effects of Storage Contention on System Performance", IBM Sys. Jour., No. 4 (1969) pp 319-333

[3] Baer, J. L., "A Survey of Some Theoretical Aspects of Multiprocessing", ACM Comp. Surveys, Vol. 5, No. 1 (Mar. 1973) pp 31-80

[4] 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

[5] Bhandarkar, D. P., "Analysis of Memory Interference in Multiprocessors", IEEE Trans. Comp., Vol. C-24, No. 9 (Sept. 1975) pp 897-908

[6] Boyse, J. W. and Warn, D. R., "A Straightforward Model for Computer Performance Prediction", ACM Comp. Surveys, Vol. 7, No. 2 (June 1975) pp 73-93

[7] Burnett, G. J. and Coffman, E. G., "A Combination Problem Related to Interleaved Memory Systems", Jour. ACM, Vol. 20, No. 1 (Jan. 1973) pp 39-45

[8] Chang, D. Y. Kuck, D. J., and Lawrie, D. H., "On the Effective Bandwidth of Parallel Memories", IEEE Trans. Comp., Vol. C-26, No. 5 (May 1977) pp 480-490

[9] Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control", Comm. ACM, Vol. 8, No. 9 (Sept. 1965) pp 569

[10] Enslow, P. H., "Multiprocessor Organization - A Survey", ACM Comp. Surveys, Vol. 9, No. 1 (Mar. 1977) pp 103-129

[11] Feller, W., An Introduction to Probability Theory and Its Applications, Vol. I, Wiley, New York (1968)

[12] Flores, I., "Derivation of a Waiting-Time Factor for a Multiple-Bank Memory", Jour. ACM, Vol. 11, No. 3 (July 1964) pp 265-282

[13] Habermann, A. N., "Synchronization of Communicating Processes", Comm. ACM, Vol. 15, No. 3 (Mar. 1972) pp 171-176

[14] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", ACM Comp. Surveys, Vol. 9, No. 1 (Mar. 1977) pp 29-57

[15] Madnick, S. E., "Multi-Processor Software Lockout", Procs. - 1968 ACM Nat. Conf., pp 19-24

[16] Strecker, W. D., "Analysis of the Instruction Execution Rate in Certain Computer Structures", PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, 1970

[17] Vaughan, R. F. and Anastas, M. S., "Microprocessor Based Transition Machines", Proc. of COMPCON FALL '79 (Sept. 1979)

[18] Wulf, W. and Bell, C. G., "C.mmp-A Multi-Mini-Processor", 1972 Fall Joint Comput. Conf. AFIPS Conf. Proc., Vol. 41 Part II, Washington, DC, Spartan, 1972, pp 767-777

AN INTRODUCTION TO THE MODELLING OF PARALLEL SYSTEMS WITH DYNAMIC STRUCTURE

Jack C. Wileden
Department of Computer and Information Science
University of Massachusetts at Amherst
Amherst, Massachusetts 01003

Abstract. Contemporary complex software systems are frequently organized as collections of interacting parallel processes in which processes are created and destroyed or patterns of process interaction are altered during system execution. However, existing formal models of parallel computation generally assume a fixed set of processes and/or fixed patterns of process interaction. This report discusses a new modelling scheme capable of representing parallel systems with dynamic structure. An example is presented illustrating the model's use in studying correctness of process interaction in complex software systems whose structure may change during system execution.

Introduction

An active area of interest in computer science within the last few years has been the area of software reliability. A great deal of effort has been directed toward discovering design methods and analysis techniques applicable to the production of correctly-working software systems. An important basis underlying much of this effort in software reliability has been work in the formal modelling of software systems.

The research described here is aimed at the development of a formal modelling scheme applicable to one particular class of complex software systems. This class, which we refer to as systems with dynamic structure, consists of systems of parallel processes in which processes may be created and/or destroyed, and in which patterns of process interaction may vary, over the course of system execution. Before considering the modelling scheme for parallel systems with dynamic structure, we outline in the next two sections of this report some background and motivation for this work.

Complex Software Systems

It is possible for any sequential computer program to be arbitrarily complicated if it is sufficiently large or poorly designed. However, as used in this report the term 'complex software system' refers to a large system in which concurrent or parallel activity is a significant aspect of system behavior. Most modern operating systems or data base management systems would be considered complex software systems according to these criteria. Such complex software systems have the additional attribute of being difficult to understand due to their large size and concurrent activity, and are therefore also difficult to design and analyze.

To facilitate the understanding of any complex system, the standard approach as discussed by Simon [21] is to decompose the system into an organized collection of smaller, simpler parts. Hopefully the system can then be understood in terms of these smaller, simpler parts and the interactions among them. In the specific case of complex software systems, this approach translates into decomposition into processes [6] or modules [16]. The result may be referred to as a system of cooperating sequential processes [4] or a system of interacting parallel processes. These equivalent terms emphasize different aspects of the resulting system. The former stresses the sequential nature of the individual processes whose cooperation results in the overall system behavior, while the latter underscores the fact that the component processes are operating in parallel, with their interaction determining the system's performance.

Decomposition of a complex software system into a system of interacting parallel processes is a useful step toward understanding complex software systems, but it by no means completely solves the problem. The parts are indeed simpler, but the interactions among them remain quite complicated. This fact is witnessed by the appearance in the literature of incorrect solutions to comparatively simple interaction situations (e.g., [7]). It is all the more difficult to attain understanding of a complex software system with dynamic structure, where processes may be created and/or destroyed and interprocess communication paths (process interaction patterns) may be altered during system execution. It is for this reason that formal modelling schemes may be useful in designing and analyzing complex software systems.

Software Reliability and Formal Models

In the domain of sequential programs, the utility of formal models as a foundation for software reliability work is quite evident. For example, the widely known structured programming method of Dijkstra [3], Wirth [24] and others, draws upon the formal work of Bohm and Jacopini [2]. Proof techniques applied to program correctness by Floyd [5], Manna [14] and others are closely related to the program schema model which was defined by Ianov [20].

In contrast to the situation regarding the domain of sequential programs, the realm of parallel programming (i.e., complex software systems) has yet to settle upon any accepted approaches to software reliability.[a] Nonetheless, the influence of formal models can be seen here also. Campbell and Habermann's path expression work has been closely tied to Petri nets [13]. Keller [11] has recently introduced a verification methodology for parallel systems based upon a formal model which he has defined, which is in turn an extension of Petri's. The DREAM system for design and analysis of complex systems [19, 23], has as its basis Riddle's formal modelling scheme for complex software systems [18].

Given the evident utility of formal modelling schemes for work in software reliability, it would appear useful to have a formal model for parallel systems with dynamic structure. However, the existing formal models of parallel computation, such as Petri nets [17], parallel computation graphs [8, 1], parallel program schemata [9, 10], and PPML/MTEs [18] all apply only to essentially static structure situations and are therefore of little use in modelling parallel systems with dynamic structure. It is for this reason that we are developing a formal modelling scheme for parallel systems with dynamic structure.

Modelling Parallel Systems with Dynamic Structure

A formal model for parallel systems with dynamic structure attempts to represent systems whose structure changes over time. In developing such a modelling scheme, we have taken a particular view of parallel systems with dynamic structure. Since our main interest is in modelling complex software systems, we consider the systems which are to be modelled as being composed of interacting parallel processes. The modelling scheme focuses upon representing dynamic structure and process interaction; in particular, internal computations of the individual processes are abstracted (i.e., not explicitly represented) in the modelling scheme. Parallelism in the model is represented by an arbitrary interleaving of strings of indivisible events, a fairly standard technique.

Two major components are required for a modelling scheme for parallel systems with dynamic structure. The scheme must provide a means for describing the possible processes which might appear (be instantiated) in the system during the course of its operation. Descriptions of potential processes must describe both their (potential) behavior and the possible communication paths by which they may interact with other processes in the system. The scheme must also allow for the description of the system's configuration at a given time, and for describing changes in that configuration. A configuration description should include a representation of the currently active (instantiated) processes, the current process interaction configuration and the current states of the individual processes and of interprocess communication. The remainder of this report outlines the major features of a modelling scheme for parallel systems with dynamic structure which includes both of these components and also presents a simple example illustrating the use of the modelling scheme. Complete details on the modelling scheme and additional examples of its use may be found in [22].

[a] Examples of the directions which have been proposed to date include [12], [15], [11] and [18].

Elements of a Modelling Scheme

The behavior and communication possibilities for potential processes in the dynamic process modelling scheme are described using process templates. Each process template represents a class of potential processes; at any given time the modelled system may include zero or more instantiations of any particular process class. The process template is described using a simple abstract programming language called the Dynamic Modelling Language or DYMOL. DYMOL contains a sufficient set of control constructs, including conditional branching instructions which allow for testing the current system configuration, the last message received by the process or the result of some computation internal to the process. The latter is essentially a non-deterministic branch, since internal process computation is not explicitly represented in the modelling scheme. DYMOL also includes an instruction for setting the contents of the distinguished process storage location known as the buffer. Each process has, implicitly, a buffer which is the source for its message transmission to other processes and the sink for its message reception from other processes. In addition to these, DYMOL includes the six instructions listed below:

```
CREATE      <process class id> <process ref var>
DESTROY     <process id expr>
ESTABLISH   <port name>  <port name>
CLOSE       <port name>  <port name>
SEND        <port id>
RECEIVE     <port id>
```

The first two commands, CREATE and DESTROY, bring about changes in the process structure of the modelled system. The CREATE command, when executed, causes a new process of the class specified by its first parameter to be added to the system. The unique process identifier assigned to the newly created process as part of the creation operation is returned to the creating process in the result variable specified as the second CREATE command parameter. A unique process identifier is formed according to the following BNF production:

```
<process id> ::= <process class id> <integer>
```

Integers are not allowed to appear in process class identifiers, which are specified as part of each process template. The DESTROY command, when executed, causes the specific process named by its parameter to be deleted from the system. A process identifier expression can be any of the three possibilities given by the following BNF production:

```
<process id expr> ::= <process id> |
  <process ref var> | ME
```

since any of these evaluates to a process identifier.

The ESTABLISH and CLOSE commands are used to alter the interprocess communication configuration of the modelled system, while the SEND and RECEIVE commands are used for actual interprocess communication. Each process template specifies a set of ports through which processes of the given process class may communicate with other processes in the system. These ports are implicitly declared by the <port id> parameters of the SEND and RECEIVE commands. Thus, each instantiation of a particular process class has the same set of local port identifiers. The ESTABLISH command, when executed, causes the specified ports of two particular processes to be connected by a message buffering channel called a link. The particular ports which are to be connected are specified by their port names which are unique identifiers formed as indicated by the following BNF production:

```
<port name> ::= <process id expr> . <port id>
```

The special DYMOL process identifier expression ME can be used to indicate that the port named is a port of the process executing the ESTABLISH command. The CLOSE command, when executed, causes the link connecting the two specified ports to be deleted from the system. The SEND command, when executed, transmits the message currently contained in the process' buffer out through the specified port. Messages in the modelling scheme are represented by a finite set of message classes. Since the SEND command takes a port identifier for its parameter, a sending process need not and cannot specify the receiving process for the message. Any process connected to the port through which the message is sent may receive the transmitted message. (Of course, the sender may be able to control the destination of a message using the ESTABLISH command.) The RECEIVE command, when executed, causes a message from one of the links connected to the specified port to be brought into the receiver's buffer. The link from which the message is to be received is non-deterministically chosen from those currently containing messages, and the message received is non-deterministically chosen from among those currently in the selected link. If none of the links connected to the specified port currently contains a message, the process executing the RECEIVE is suspended until a message is available, at which time it may receive the message and continue execution.

Figures 1 and 2 are example process templates. The two process classes represented, SYNCH and TASK, are isomorphic at the level of abstraction of the modelling scheme. They will be distinguished in the example to be presented below. The statement labels found in the DYMOL process template descriptions are not used for branching, but rather are needed for the specification of process state, as discussed below. DO FOREVER is an infinitely looping control construct, while BEGIN, END and the semicolon separator syntax are algolic. The graphical representation of the process templates is a useful device for presenting the models; note the distinction between inbound and outbound ports and also the boxes representing links.

Having a means for describing potential processes, we now require a means for describing the system's configuration and changes to that configuration. In the dynamic process modelling scheme this is done primarily through use of the configuration matrix, C. C is a matrix whose indices are the set of unique process identifiers for active (instantiated) processes in the system. The entries of C describe the current interprocess communication configuration of the system; each entry containing a set of ordered pairs of port identifiers. The appearance of the ordered pair (x,y) in C(a,b) indicates that port x of process a is currently connected to port y of process b. Thus the configuration matrix summarizes the current configuration of the system at any point in time, its indices providing a list of the active processes in the system and its contents indicating the current interprocess communication linkages.

```
 SYNCH:

SY0:  DO FOREVER
        BEGIN
SY1:      RECEIVE MUTEX_V;
SY2:      SEND MUTEX_P
        END.
```

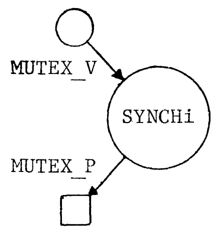


Figure 1

```
 TASK:
TA0:  DO FOREVER
        BEGIN
TA1:      RECEIVE IN;
TA2:      SEND OUT
        END.
```

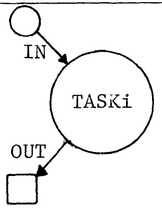


Figure 2

The configuration matrix C may be altered by execution of the DYMOL commands CREATE, DESTROY, ESTABLISH and CLOSE. In fact, it is these changes to C which define the semantics of the four DYMOL commands. Execution of the CREATE command causes a new row and column to be added to C, representing the newly created process. Thus, suppose that the current configuration matrix were:

	SCHED1	SYNCH1
SCHED1	∅	∅
SYNCH1	∅	∅

Then, execution of the DYMOL command:

```
CREATE    TASK    TVAR
```

might result in the following configuration matrix:

	SCHED1	SYNCH1	TASK1
SCHED1	∅	∅	∅
SYNCH1	∅	∅	∅
TASK1	∅	∅	∅

All the entries in the newly-added row and column are initially null, indicating that no inter-process communication connections yet exist between the newly-created process and any of the other processes.

Execution of the ESTABLISH command causes an ordered pair to be added to an entry in the configuration matrix. Thus, if the next DYMOL command to be executed were to be either of the equivalent pair:(b)

```
ESTABLISH    SYNCH1.MUTEX_P    TASK.IN
```

or

```
ESTABLISH    SYNCH1.MUTEX_P    TVAR.IN
```

the resulting configuration matrix would be:(c)

	SCHED1	SYNCH1	TASK1
SCHED1	∅	∅	∅
SYNCH1	∅	∅	(MUTEX_P,IN)
TASK1	∅	∅	∅

Thus, the new entry in C(SYNCH1,TASK1) indicates that a communication link exists between the two processes SYNCH1 and TASK1 through ports MUTEX_P and IN, respectively. This link provides a means for SYNCH1 to send messages to TASK1, but the null entry in C(TASK1,SYNCH1) indicates that TASK1 has no capability of sending messages to SYNCH1 at this point.

(b)The two commands given are equivalent only due to the assumption that one of them is the next command executed, so that the process reference variable TVAR still contains the process identifier TASK1.

(c)To simplify notation, set brackets around singleton sets are omitted throughout this report.

Execution of the CLOSE command causes an ordered pair to be deleted from an entry in the configuration matrix. Thus, if the next DYMOL command to be executed were to be either of the equivalent pair:

```
CLOSE    SYNCH1.MUTEX_P    TASK1.IN
```

or

```
CLOSE    SYNCH1.MUTEX_P    TVAR.IN
```

the resulting configuration matrix would be:

	SCHED1	SYNCH1	TASK1
SCHED1	∅	∅	∅
SYNCH1	∅	∅	∅
TASK1	∅	∅	∅

The new null entry in C(SYNCH1,TASK1) indicates that SYNCH1 can no longer send messages to TASK1. However, any messages which it may have already transmitted to TASK1 prior to execution of the CLOSE command remain available for TASK1 to receive at a later time.

Execution of the DESTROY command causes a row and column to be deleted from the configuration matrix.(d) Thus, if the next DYMOL command to be executed were to be either of the equivalent pair:

```
DESTROY    TASK1
```

or

```
DESTROY    TVAR
```

the resulting configuration matrix would be:

	SCHED1	SYNCH1
SCHED1	∅	∅
SYNCH1	∅	∅

Given the modelling scheme outlined above, a model in the scheme consists of a set of process templates, an initial instantaneous configuration (the notion of instantaneous configuration will be defined shortly) and (optionally) a set of terminal instaneous configurations. More formally, a model M = (P,x<0>,T) where:

P = a finite set of process templates defined by abstract programs in the Dynamic Modelling Language

x<0> = an instantaneous configuration, called the initial instantaneous configuration, and

(d)This is a slight simplification of the actual semantics for DESTROY as given in [22].

T = an optional set of instantaneous configurations, called the terminal instantaneous configurations set

Thus to model a parallel system with dynamic structure using the dynamic process modelling scheme, one would describe the set of potentially active processes using the abstract programming language, specify an initial system configuration and perhaps specify any terminal configurations of interest (e.g., a configuration representing a system deadlock). The modelling scheme can thus be used in a purely descriptive fashion, which could prove useful in the design of complex software systems with dynamic structure. hoping!)

An instantaneous configuration for a model M is a complete description of the model at some instant of time during its execution. Thus, an instantaneous configuration must specify the current set of active processes and the state (i.e., location counter values and message buffer contents) of each, the current process interaction configuration and the current state of each link currently in the model. In formal terms[(e)], an instantaneous configuration x = (C,A,Q,L) where:

C = current configuration matrix

A = current set of active processes (i.e., the indices of C)

Q = current state of processes in A (i.e., location counter and message buffers of processes in A)

L = current state of links

Using the notion of instantaneous configuration, it is possible to define a dynamic process computation step y<i,j> = x<i>x<j> where x<i> and x<j> are instantaneous configurations for a model M and x<j> is the result of the occurrence of a single primitive operation (e.g., execution of one process command or transfer of one message) with M in instantaneous configuration x<i>. Then a computation for a model M may be defined as z = y<0,1>·y<1,2>· ... ·y<t-1,t> = x<0>x<1>x<2>...x<t-1>x<t>, i.e., a series of computation steps beginning with the initial instantaneous configuration of M and ending with a terminal instantaneous configuration (i.e., x<t>$\in$ T) or with any instantaneous configuration of M if T is not specified. Finally, the set of all possible computations for M may be defined as Z<M> = $\{z | z$ is a computation for M$\}$.

A Simple Example

The use of the dynamic process modelling scheme and an indication of its potential utility for software reliability work can be illustrated by the following simple example. The scheme is used to model a set of tasks, created and destroyed by a scheduler, all accessing a shared resource requiring exclusive access. An example of such a situation might be an on-line reservation system composed of a number of reservation-making tasks accessing a common data base, where the number of tasks varies with system load.

(e) This definition and those following it in the remainder of this section are somewhat simplified versions of the definitions given in [22].

The process templates for the tasks and for the process which will synchronize the tasks' access to the shared resource were presented in figures 2 and 1, respectively. Figure 3 is the template for the scheduler process, which alternately creates new tasks, connecting them to the synchronizer, and destroys tasks. The WHILE INTERNAL TEST construct models the unelaborated internal process computation which determines when tasks are to be created or destroyed, represented in the modelling scheme as non-deterministic, indefinite iterations. The FOR SOME construct selects a value for TVAR; it is evident that the resulting value must be a process identifier for a TASK process.

Given these three process templates and an initial configuration matrix, C<0>, of the following form:

$$\begin{array}{c|c|c|}
 & \text{SCHED1} & \text{SYNCH1} \\ \hline
\text{SCHED1} & \emptyset & \emptyset \\ \hline
\text{SYNCH1} & \emptyset & \emptyset \\ \hline
\end{array}$$

then the model M may be represented formally and graphically as shown in figure 4. By convention, the entries in the current process state tuple, Q, are presented in the same order in which the processes are listed in A. Each entry is itself a tuple in which the current process location counter value, expressed by statement label, appears first, the current buffer contents appear last, and the values of any internal process variables appear in keyworded form between these two. Note the message SEM which is initially located in the link connected to the MUTEX_P port of SYNCH1.

```
 SCHED:

SC0:  WHILE INTERNAL TEST DO
        BEGIN
          WHILE INTERNAL TEST DO
            BEGIN
              CREATE TASK TVAR;
              ESTABLISH TVAR.OUT SYNCH1.MUTEX_V;
              ESTABLISH SYNCH1.MUTEX_P TVAR.IN
            END;

SC2:      WHILE INTERNAL TEST DO
            BEGIN
              FOR SOME TVAR ∈ A - {SCHED1,SYNCH} DO
                BEGIN
SC1:              DESTROY TVAR
                END
            END
        END.
```

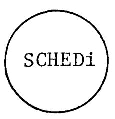


Figure 3

$M = (P, x\langle 0\rangle, T)$
where
$P = \{SCHED, SYNCH, TASK\}$
$x\langle 0\rangle = (C\langle 0\rangle, A\langle 0\rangle, Q\langle 0\rangle, L\langle 0\rangle)$
$T = \{(C,A,Q,L) \mid q_{SYNCH1}=(-,\emptyset),\ \forall i\ q_{TASKi}=(-,\emptyset)$
and $L=(\emptyset,\ldots,\emptyset)\}$
$A\langle 0\rangle = \{SCHED1, SYNCH1\}$
$Q\langle 0\rangle = ((SC0, TVAR = \emptyset, \emptyset), (SY0, \emptyset))$
$L\langle 0\rangle = (\langle SEM\rangle)$

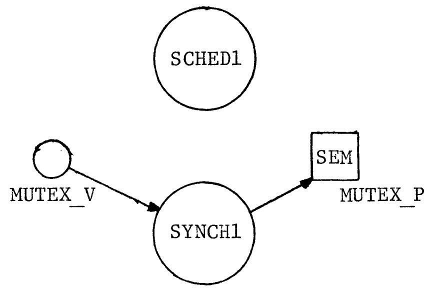


Figure 4

A possible instantaneous configuration, x<i>, for M[(f)] is presented formally in figure 5 and graphically in figure 6. Three TASK processes are currently in the system, each connected to the SYNCH1 process. Some other TASK processes may have been destroyed by SCHED1, although this cannot be determined from the present configuration since (aside from those which the modeller assigns to processes present in the initial instantaneous configuration) unique process identifiers are generated randomly as part of the creation operation and do not necessarily appear in numerical order.

A possible computation step from x<i> is shown in figure 7. Only the process state tuple, Q, and the link state tuple, L, are changed as a result of this computation step, which results in the message SEM being transmitted back to the SYNCH1 process. This may be viewed as the relinquishing of the shared resource by TASK7.

An alternative possible computation step from x<i> is shown in figure 8. In this instance, SCHED1 has destroyed the task process TASK7, as evidenced by the appropriate changes to C, A, and Q. The new instantaneous configuration, x<i+1> resulting from this computation step is an element of the set of terminal instantaneous configurations, T, which corresponds to the set of instantaneous configurations yielding task deadlock. While the overall system is still capable of executing additional computation steps, including the creation and destruction of tasks, no task process can possibly progress since the SEM message was destroyed with TASK7 and the modelled system is incapable of generating any new messages. Thus, all task processes will wait indefinitely at their RECEIVE instructions (TA1). The appearance of a computation resulting in task deadlock would most likely indicate to a software system designer that the modelled system corresponds to an incorrect design. The example thus suggests the potential utility of the dynamic process modelling scheme as a software design tool.

Finally, a corrected version of the model M, based upon the revised scheduling process SCHED' illustrated in figure 9, is presented in figure 10. The model M' incorporates the necessary synchronization of the scheduler and the task processes to prevent the destruction of a task which is currently holding the SEM message (i.e., a task which is in its critical region). For the model M', there is no computation z which results in an instantaneous configuration x<t> in T', and thus the revised model corresponds to a design which is free from task deadlock. (A proof of this assertion may be found in [22].)

In Conclusion

The dynamic process modelling scheme which we have described here was developed as a basis for formulating design methods and analysis techniques applicable to complex software systems with dynamic structure. To that end, we are currently incorporating its concepts and constructs into the DREAM software design aid system. The modelling scheme has also been used to investigate decidability issues for dynamically-structured parallel systems, to define and study subclasses of these systems, and as a vehicle for considering the necessity and modelling power of various representational constructs. At the same time, a non-procedural, expression-based behavioral representation technique, called constrained expressions, has been developed and an effective procedure has been defined for deriving constrained expressions from certain subclasses of parallel systems with dynamic structure. It is hoped that continued research along these various dimensions will lead to an improved understanding of parallel systems with dynamic structure and to genuinely useful tools for designers of complex, dynamically-structured software systems.

Acknowledgements

The work reported here was supported in part by grants from Sycor, Inc. and the Horace H. Rackham School of Graduate Studies of the University of Michigan. The author is deeply indebted to Professor William E. Riddle who inspired the research described here and to Professors W. C. Rounds, L.K. Flanigan and K. B. Irani for their contributions to this work. Ms. Lorraine Morin's patient and skillful typing of the manuscript is also sincerely appreciated.

[(f)] One of the numerous sequences of computation steps for M which could lead to x<i> is enumerated in [22].

x<i> = (C<i>, A<i>, Q<i>, L<i>)

where

C<i> =

	SCHED1	SYNCH1	TASK2	TASK7	TASK10
SCHED1	∅	∅	∅	∅	∅
SYNCH1	∅	∅	(MUTEX_P,IN)	(MUTEX_P,IN)	(MUTEX_P,IN)
TASK2	∅	(OUT,MUTEX_V)	∅	∅	∅
TASK7	∅	(OUT,MUTEX_V)	∅	∅	∅
TASK10	∅	(OUT,MUTEX_V)	∅	∅	∅

A<i> = {SCHED1, SYNCH1, TASK2, TASK7, TASK10}

Q<i> = ((SC1, TVAR=TASK7, ∅), (SY1, ∅), (TA1, ∅), (TA2, SEM), (TA1, ∅))

L<i> = (∅, ∅, ∅, ∅)

Figure 5

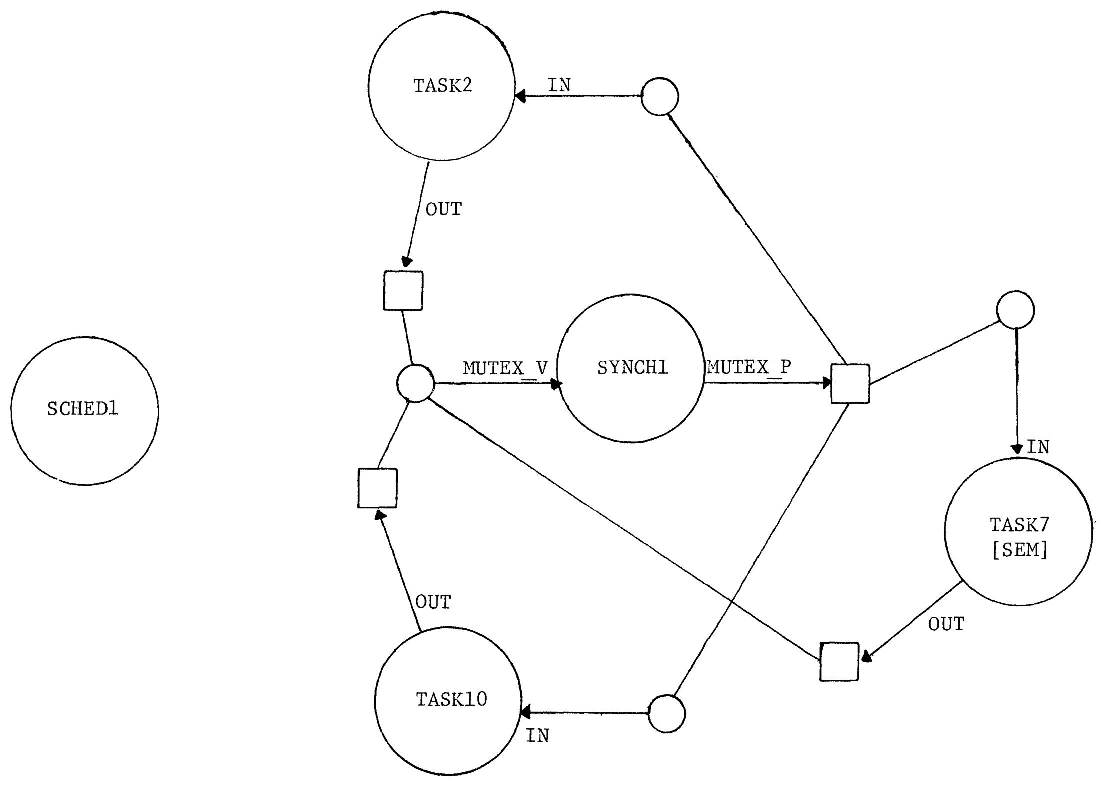


Figure 6

y<i, i+1> = x<i>x<i+1>

where

x<i+1> = (C<i>, A<i>, Q<i+1>, L<i+1>)

Q<i+1> = ((SC1, TVAR=TASK7, ∅), (SY1, ∅), (TA1, ∅), (TA1, ∅), (TA1, ∅))

L<i+1> = (∅, ∅, <SEM>, ∅)

Figure 7

y<i, i+1> = x<i>x<i+1>

where

x<i+1> = (C<i+1>, A<i+1>, Q<i+1, L<i>)

C<i+1> =

	SCHED1	SYNCH1	TASK2	TASK10
SCHED1	∅	∅	∅	∅
SYNCH1	∅	∅	(MUTEX_P,IN)	(MUTEX_P,IN)
TASK2	∅	(OUT,MUTEX_V)	∅	∅
TASK10	∅	(OUT,MUTEX_V)	∅	∅

A<i+1> = {SCHED1, SYNCH1, TASK2, TASK10}

Q<i+1> = ((SC2, TVAR = TASK7, ∅), (SY1, ∅), (TA1, ∅), (TA1, ∅))

Figure 8

```
  SCHED':
SC0:  WHILE INTERNAL TEST DO
        BEGIN
          WHILE INTERNAL TEST DO
            BEGIN
              CREATE TASK TVAR;
              ESTABLISH TVAR.OUT SYNCH1.MUTEX_V;
              ESTABLISH SYNCH1.MUTEX_P TVAR.IN
            END;
SC2:      WHILE INTERNAL TEST DO
            BEGIN
              FOR SOME TVAR ∈ A - {SCHED1, SYNCH1} DO
               BEGIN
                 RECEIVE FREEZE;
SC1:             DESTROY TVAR;
                 SEND THAW
               END
            END
        END.
```

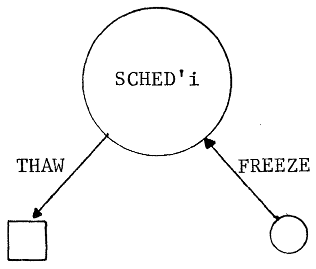


Figure 9

M' = (P', x<0>', T')

where

P' = {SCHED', SYNCH, TASK}

x<0>' = (C<0>', A<0>', Q<0>, L<0>)

T' = {(C, A, Q, L) | $q_{SCHED'1}$=(-,∅), q_{SYNCH1}=(-,∅), ∀i q_{TASKi}=(-,∅) and L = (∅,...,∅)}

C<0>' =

	SCHED'1	SYNCH1
SCHED'1	∅	(THAW,MUTEX_V)
SYNCH1	(MUTEX_P,FREEZE)	∅

A<0>' = {SCHED'1, SYNCH1}

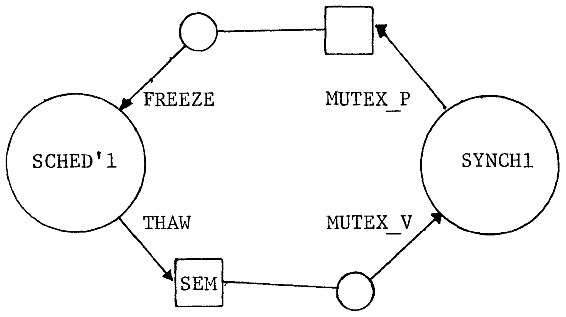


Figure 10

References

[1] D. A. Adams, A Computation Model with Data-flow Sequencing, Computer Science Dept. Stanford University, Tech. Rep. CS-117 (Dec. 1968).

[2] C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Communications of the ACM (May 1966), pp. 366-371.

[3] O. -J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press, (1972).

[4] E. W. Dijkstra, "Cooperating Sequential Processes," in F. Genuys (ed.), Programming Languages, Academic Press, (1968), pp. 43-112.

[5] R. W. Floyd, "Assigning Meaning to Programs," Proceedings of Symposia in Applied Mathematics, Mathematical Aspects of Computer Science (1967), pp. 19-32.

[6] J. J. Horning and B. Randall, "Process Structuring," Computing Surveys (March 1973), pp. 5-30.

[7] H. Hyman, "Comment on a Problem in Concurrent Programming Control," Communications of the ACM (January 1966), p. 45.

[8] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing," SIAM Journal of Applied Math. (November 1966), p. 1390-1411.

[9] R. M. Karp and R. E. Miller, "Parallel Program Schemata: A Mathematical Model for Parallel Computation," IEEE Conference Record of 1967 Eighth Annual Symposium on Switching and Automata Theory, 1967.

[10] R. M. Karp and R. E. Miller, "Parallel Program Schemata," Journal of Computer and System Sciences (1969) 147-195.

[11] R. M. Keller, "Formal Verification of Parallel Programs," Communications of the ACM (July 1976), pp. 371-384.

[12] H. C. Lauer, Correctness in Operating Systems, Dept. of Computer Science, Carnegie-Mellon University (September 1972).

[13] P. E. Lauer and R. H. Campbell, A Description of Path Expressions by Petri Nets, Computing Laboratory, University of Newcastle-Upon-Tyne, TR-64 (May, 1974).

[14] Z. Manna, Mathematical Theory of Computation McGraw-Hill, (1974).

[15] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach, "Communications of the ACM (May 1976), pp. 279-285.

[16] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM (December 1972), pp. 1053-1058.

[17] C. A. Petri, "Communication with Automata," Supplement 1 to Technical Report RADC-TR-377, Vol. 1, Griffiss Air Force Base (1966).

[18] W. E. Riddle, An Approach to Software System Modelling, Behavior Specification and Analysis, Dept. of Computer and Communication Sciences, University of Michigan, RSSM/25 (May, 1977).

[19] W. E. Riddle, J. Wileden, J. Sayler, A. Segal and A. Stavely, "Behavior Modelling During Software Design," IEEE Transactions on Software Engineering (July 1978), pp. 283-292.

[20] J. S. Rutledge, "On Ianov's Program Schemata," Journal of the ACM (January 1964), pp. 1-9.

[21] H. A. Simon, "The Architecture of Complexity," Proceedings of the American Philosophical Society (December 1962), pp. 467-482.

[22] J. C. Wileden, Modelling Parallel Systems with Dynamic Structure, Dept. of Computer and Information Science, University of Massachusetts, COINS Tech. Rep. 78-4 (January, 1978).

[23] J. C. Wileden, "DREAM - An Approach to Designing Large Scale, Concurrent Software Systems," Proceedings of ACM National Conference, (October, 1979).

[24] N. Wirth, "Program Development by Stepwise Refinement," Communications of the ACM (April 1971), pp. 221-227.

A Generalized Cluster Structure for Large Multi-microcomputer Systems *

Shyue B. Wu and Ming T. Liu

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210, USA

Summary

This paper presents a generalized cluster structure for interconnecting a large number of microcomputers, where each microcomputer (microprocessor plus memory) constitutes a node of the cluster. It turns out that four of the popular cluster structures (viz., hypercube [4], hierarchy [2], star [1] and tree [3]) are all special cases of the generalized cluster structure characterized by different parameter values. Through the use of the generalized cluster structure, a unique tool for analyzing a variety of interconnection structures are also discussed.

The generalized cluster structure consists of three components and can be completely described by two functions. The three components are the computation node, the switching node and the path; and the two functions are the interconnection function and the switching function.

A computation node (CN) is a microcomputer which contributes computing power to the system. A switching node (SN) is a microcomputer which performs switching functions. A path is a medium over which messages are passed. The medium may be a dedicated link or a time-shared bus. It is viewed as a bus (BUS) in this paper. An interconnection function (IF) specifies a way of interconnecting buses. Hence it characterizes a topological structure. A switching function (SF) can be circuit switching, mesage switching or packet switching.

Let N, I, S, B, E, F, G, M and L be the parameters of a generalized cluster structure, all taking on integer values except for M and L. Then the whole system is organized into N levels of subclusters ($N \geq 1$). Several elements of each component (I, S, B elements respectively for CN, SN, BUS) are initially interconnected to form a basic subcluster. Then M subclusters at the same level and several additional elements of each component (E, F, G elements respectively for CN, SN, BUS) are interconnected to form a subcluster or portion thereof (L) at one higher level.

Formally, a generalized N-level cluster structure is a 5-tuple, (CN, SN, BUS, IF, SF), where, for i = 1, 2 --- N,

- CN : A set of CNs = [Ci]
- SN : A set of SNs = [Si]
- BUS : A set of level-i buses.
- IF : A set of interconnection functions = [Bi]
  - Bi = specification of a level-i bus $= f(C_i, S_i) \subseteq [C_i, S_i] \subseteq [C_n, S_n]$
  - $|B_i|$ = no of nodes on a level-i bus
- SF : A set of switching functions

* Research reported herein was supported in part by NSF-MCS-77-23496 and by AFOSR-77-3400.

Using different parameter values of the generalized cluster structure, we can specify different structures. The parameter values of four special cluster structures (viz., hypercube, hierarchy, star and tree) are tabulated in Table 1.

A. Components

	CN		SN		BUS		MEMBER	
	I	E	S	F	B	G	M	L
1.	K	0	0*	K	1	1	K	1
2.	1+K	1	1$	1	K	K	K	1
3.	K	1	0*	1	1	2/K	(K-1)/K	1/K
4.	K	0	1	1	1	1	K	1

B. Interconnection Functions

	B1	$\|B_1\|$	Bi+1	$\|B_{i+1}\|$
1.	I	K	Ci	K
2.	I	2	Ci,F	2
3.	I	K	Ci	K
4.	I,S	1+K	Si,F	1+K

1 = Hypercube; 2 = Hierarchy; 3 = Star; 4 = Tree
* One of I CNs takes care of the control and communication over each of B buses.
$ One I CNs is viewed as an SN to take care of path control communication and switching function over K buses.

Table 1: Parameter Values of 4 Special Structures

Linear complexity and easy extensibility are two important characteristics of the generalized cluster structure. That is, the complexity (thus the cost) of the system increases linearly with the power of the system (or the number of computation nodes) and the system can be easily extended to an arbitrary number of nodes.

The generalized cluster structure can cover a variety of interconnection structures. A unique tool for analyzing and comparing different structures has been obtained [5]. Traffic congestion and message delay have been two serious problems in all structures supporting indirect interprocessor communication. Traffic congestion can occur when a node or a path is overloaded, whereas message delay is a result of indirect interprocessor communication.

An analytical model based on the probability of utilization is proposed to consider traffic congestion and message delay problems. This model accepts a given set of interprocessor communication traffic and outputs a set of analysis results. The intra-cluster communication traffic is assumed to be equally distributed and the inter-cluster communication traffic is assumed to be symmetrically distributed. The interprocessor communication traffic is expressed by the degree of local cooperation, which is defined to be the frequency that a node issuses a message to another node at the same level of subcluster.

Traffic congestion is determined by the probability of utilization of system components. The theoretical upper bound of how much interprocessor communication traffic a structure can support is the maximum interprocessor communication traffic without having theoretical traffic congestion. The maximum interprocessor communication traffic a structure can have is expressed by the minimum degree of local cooperation. This optimal value (called OPTLC) depends on interconnection structure and system size. Figure 1 shows the OPTLC values vs different system sizes of the four popular cluster structures.

Message delay is measured by the number of buses needed for passing a message. It depends on which level of subcluster the source and destination nodes of the message belong to. That is, the interconnection distance between the source and the destination. Average message delay is the average value of all message delay. It depends on the quantity and the distribution of message traffic. Giving the optimal message traffic derived from Figure 1, the average message delay (OPTDAVG) vs different system sizes of the four popular cluster structures are plotted in Figure 2.

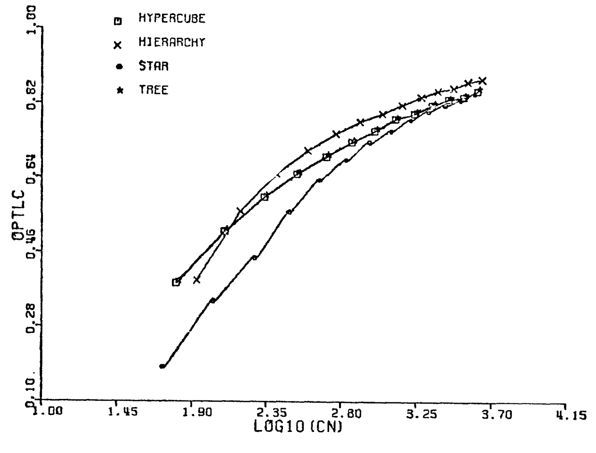


Figure 1 : OPTLC vs System Sizes

The lower the OPTLC value is, the better the structure is, since the system will have a greater degree of extensibility. OPTDAVG tells us the average message delay and the smaller the OPTDAVG is, the better the structure is. These two values could give us an idea about system performance and application restriction. In interactive and real-time applications, these could be very helpful.

The generalized cluster structure has been demonstrated as a useful tool for interconnection system design. Selecting a particular interconnection structure depends on system applications and restrictions. The results from our analysis are qualitative rather than quantitative, thus one must be careful in interpreting the results.

Referrences

1. A. Despain and D. A. Patterson, "X-Tree: A Tree Structured Multi-processor Computer Architecture", Proc. of the Fifth Annual Symposium on Computer Architecture, April 1978, PP. 144-151.

2. J. A. Harris and D. R. Smith, "Hierarchical Multiprocessor Organization", Proc. of the Fourth Annual Symposium on Computer Architecture, 1977 , PP. 41-48.

3. R. M. Keller et al., "A Loosely-coupled Applicative Multi-processing System", Proc. 1979 NCC, AFIPS, Vol. 48, 1979. PP. 613-622.

4. L. D. Wittie, "Efficient Message Routing in Mega-Micro-Computer Networks", Proc. of the Third Annual Symposium on Computer Architecture, January 1976, PP. 136-140.

5. S. B. Wu and M. T. Liu, "Optimal Interconnection Design for Distributed Processing Systems", in preparation.

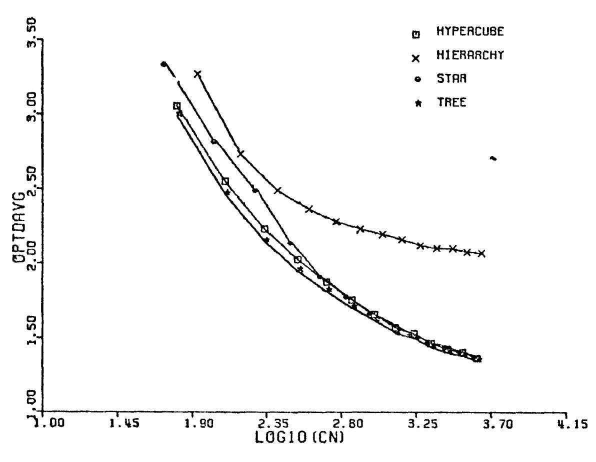


Figure 2 : Message Delay vs System Sizes

PARALLEL TRANSITION MACHINES

Mark S. Anastas and Russell F. Vaughan
Boeing Aerospace Company
P.O. Box 3999
Seattle, Washington 98124

Abstract -- The architecture for a general purpose parallel computer called a Parallel Transition Machine is derived from an abstract theoretical model of parallel computation. Its development is pursued to a design description involving hardware block diagrams. Applicability across a broad spectrum of computational requirements is suggested by the modular extendability of the hardware units. It is apparent that computational problems can be broken down in a design hierarchy where each level executes in a virtual Transition Machine dynamically assignable to a free hardware unit.

Introduction

There are completely general models of parallel computation [8], but there are currently no machine architectures suitable for efficient execution of parallel programs generated in accordance with one of these models. The parallel computer architectures which do exist are special purpose devices appropriate only within a restricted domain of any general parallel computational model. For example, the restricted homogeneous parallelism afforded by an associative or an array processor can only perform identical operations on multiple data sets concurrently.

This paper describes a family of computational machine architectures which implements a general model of parallelism. The conceptual model of parallel computation described by Keller [8] under the nomenclature of transition systems has been accepted here as the basis for a more detailed model of a machine architecture. This Parallel Transition Machine model provides a machine architecture in which transition systems can be executed. The details of this architecture are defined to a level where development can proceed. A prototype system is currently under development; some of the detail design issues addressed by this development are discussed in reference [3].

The architecture can be characterized as a multiprocessor with a separate System Controller as shown in Figure 1. Interrupts, I/O controllers, and other special purpose processors can be integrated into this model of Transition Machines, and introduce no significant developmental problems. The System Controller is in essence a functional equivalent (implemented in hardware) of a multiprocessor executive for transition systems. The operation of the System Controller is effected by a series of logical operations on fixed data constructs descriptive of the conditions under which the various computations become eligible. It effects the system transitions by performing matrix operations on a system status vector to obtain a procedure eligibility vector. The procedure eligibility vector provides a basis for task assignments to the processors; completion of the assignments results in a modified system status vector.

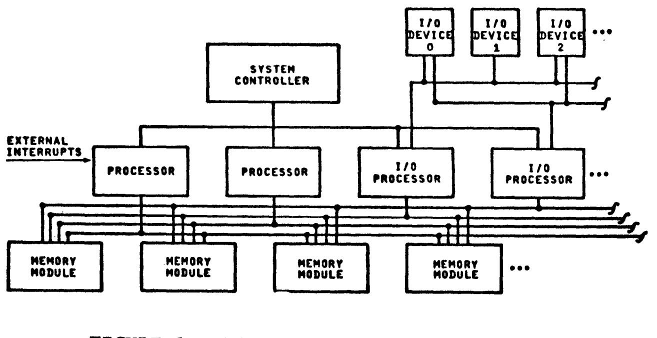


FIGURE 1: PARALLEL TRANSITION MACHINE

The development cost of System Controllers is small relative to the cost of the multiprocessors which are controlled by them. System transitions can be effected in a fraction of the time that is currently required for straightforward software multiprogramming executives. This supports an approximately linear extendibility of throughput in large array multiprocessors. To implement the equivalent system control matrix operations in a software multiprocessing executive would be infeasible due to the high overhead as shown in reference [3].

The application of Parallel Transition Machines to large systems is extremely promising, and the feasibility of configuring arrays of coordinated microprocessors seems evident [14]. But large systems introduce commensurate challenges; for example, an operating system and linkage editor of considerable complexity are required to implement the overlaying of partitioned matrices.

Parallel Transition Machines also require programming structures which are not traditional. These non-traditional structures provide the advantages of highly structured programs which result in enhanced software productivity [1]. It is possible, however, to develop only the translator for a suitable existing compiler so that traditional program structures could be translated to run on Parallel Transition Machines.

Abstract Parallel Computation Model

Parallel Transition Machines are based on a particular abstract model of parallel computation, selected because of its generality. It is transition systems $(Q, \rightarrow)$, where Q is the set of possible system states and $\rightarrow$ is the set of transitions between states as described by Keller. A named transition system is a triple $(Q, \rightarrow, \Sigma)$. The components correspond respectively to the set of possible system states $(q_1, q_2, q_3 \ldots)$, a set of transitions between states $(\rightarrow_1, \rightarrow_2, \rightarrow_3 \ldots)$, and a set of names ($\sigma_1$, σ_2, σ_3...) associated with groups of individually programmed transitions between states [8]. Since there is a one-to-one correspondence between the indices on sigma and the names themselves, the indices will be used to indicate the names: i implies σ_i, and $I \equiv \{i\}$ implies Σ. The index $i \in I$ is associated with a group of system transitions described by the statement:

when $R_i(\xi)$ do $\xi' = \psi_i(\xi)$

The symbols in this statement are defined as follows:

i	=	the index of the group of transitions whose common feature is that they all result in the data transformation indicated by the function ψ_i.
ξ	=	the set of all data items in the system.
$R_i(\xi)$	=	the subset of satisfied propositions on the data set, ξ which are essential to defining the appropriateness, and therefore constitute the enabling predicate, for transitioning as determined by performing the data transformation $\psi_i(\xi)$.
$\psi_i(\xi)$	=	the programmed functional data transformation, associated with the group of system transitions indicated by i, which operates on the data set, ξ and results in a revised data set ξ'.

The group i can be associated with a procedure (including preamble) that can be written by a programmer to effect the data transformation, ψ_i on the data set ξ when the appropriate set of conditions R_i is satisfied on that data set. (Although obviously not the intent in Keller's work, it has been demonstrated that program requirements can be implemented to advantage in this manner [1].) In a parallel computation step, multiple sets of conditions, R_i can be satisfied simultaneously such that multiple transitions can proceed in parallel. The R_i are enabling predicates that indicate the requisite status of propositions on the data set ξ which properly enable the function ψ_i. Relevant propositions that have been defined on data elements $e_k \varepsilon \xi$ are the following:

1. the data element e_k is available/not available for use in subsequent computations,

2. the data element e_k satisfies/does-not satisfy a specified condition relative to some constant or other data element $e_{k'}$ (for example, $e_k < e_{k'}$), and

3. the data element e_k can/cannot be updated.

This paper deals exclusively with valid parallel programs; these programs will not exhibit race conditions, and therefore procedures which read and write the same data element will have a predetermined execution order specified by their respective enabling predicates. The properties of determinacy, commutativity and persistence are described by Keller [9]. These and other properties of valid parallel programs are also discussed in references [5], [6], [8], and [11].

Transition Machine Model

As an organizational basis for implementing the architectural model of parallel computation, we have defined a set of constructs and the logical matrix operations on these constructs which effect the system control functions for a Parallel Transition Machine. The constructs and logic are exemplified in Figure 2.

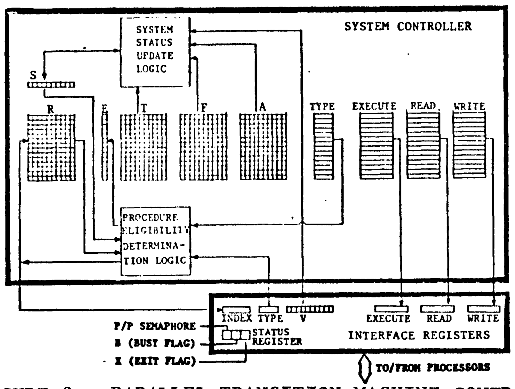


FIGURE 2: PARALLEL TRANSITION MACHINE CONTROL CONSTRUCTS AND LOGIC

Eligibility Determination

Definition 1. The system status vector, S is a set of binary status indications for a set of propositions concerning the data set ξ such that for every possible proposition on the set there is an associated status indication, S_j in S if and only if the proposition on the data set is relevant to enabling some procedure in the system. (In hierarchical implementations discussed further on, conditions which are relevant to every procedure at a given level will also be excluded.) $S_j = 1$ if the associated proposition on the data set is met, $S_j = 0$ otherwise.

For convenience we will use the phrase "data condition" in referring to a "proposition on the data set" throughout the remainder of this paper.

Definition 2. The system eligibility vector, E is a set of binary status indications for the set of predicates R_i, such that for each predicate R_i there is an associated status indication, E_i in E indicating whether R_i is currently satisfied, enabling the associated procedure. $E_i = 1$ indicates the associated predicate is satisfied; $E_i = 0$ otherwise.

Definition 3. A data condition associated with S_j is relevant to enabling procedure i if and only if the data condition whose status is indicated by S_j is included in the predicate R_i.

Proposition 1. The predicate, R_i can be represented as a set of binary relevance indications associated (and in conjunction) with each of the data conditions whose status is maintained in S. This proposition follows directly from the previous definitions.

Definition 4. The relevance matrix, R is comprised of binary relevance indications, r_{ij} indicating the relevance of a data condition j to enabling procedure i. Relevance is indicated by $r_{ij} = 0$, irrelevance by $r_{ij} = 1$.

Definition 5. The logical dot product, of a matrix M (with dimension IxJ) and a vector, W (a vector of dimension J) is defined as the vector, $P = M \cdot W$, with dimension I, where

$$P_i = \bigwedge_{j=1}^{J} M_{ij} \vee W_j$$

In this equation (and throughout this paper) the following symbol definitions apply:

$$\bigwedge_{n=1}^{N} x_n \equiv x_1 \wedge x_2 \wedge \cdot \cdot \wedge x_n .$$

$\wedge \equiv$ logical "AND",
$\vee \equiv$ logical "OR".

Proposition 2. The system eligibility vector, E can be computed appropriate to a given state of the system by generating the logical data product of the relevance matrix, R and the system status vector, S.

Proof:

From definition 5 is follows that:

$$\left[R \cdot S\right]_i = \bigwedge_{j=1}^{J} r_{ij} \vee S_j$$

From definitions 4 and 1 it follows that $r_{ij} \vee S_j = 1$ if and only if data condition j is either met or irrelevant to enabling procedure i. Then by proposition 1 it follows that $\left[R \cdot S\right]_i = 1$ if and only if all data conditions of the predicate R_i are satisfied. Thus, $\left[R \cdot S\right]_i = E_i$ by definition 2, and it is proved that $E = R \cdot S$ as proposed.

There is now a prescription for determining procedure eligibilities based on system status and the procedures' data conditional requirements. What remains to be shown is the computation of the new system status vector appropriate to having completed a given procedure.

System Status Update

Since Keller did not address actual implementations of named transition systems, it was not incumbent upon his work to address representation and the associated maintenance of state. In this paper however, we posit that there are J data conditions (propositions concerning the data set) whose status (true or false) will provide sufficient information concerning the state of the system to effect any and all of the named transitions defined for the system. But the status indications for these data conditions maintained in the S vector are not a part of the data set ξ associated with the transformations $\psi_i(\xi)$. Therefore, they must be updated separately in order that changes of state be reflected in the system status vector.

There are several possible implications on the status of a data condition at the completion of a procedure which implements the data transformation. They are as follows:

1. The data condition's status remains unaffected by the procedure running to completion.

2. The data condition's status is satisfied whenever the procedure runs to completion.

3. The data condition's status is negated whenever the procedure runs to completion.

4. The data condition's status is determined dynamically during the execution of the procedure.

The fixed constructs which are implemented to effect system status modifications are described below:

Definition 6. The jth element, t_{ij} of the true condition vector, T_i is a binary status indication associated with procedure i and the data condition, j such that $t_{ij} = 1$ implies the data condition j is either satisfied or unchanged by the completion of procedure i.

Definition 7. The jth element, f_{ij} of the false condition vector, F_i is a binary status indication associated with the procedure i and the data condition, j. The element $f_{ij} = 1$ implies the data condition j is either negated or unchanged by the completion of procedure i.

Definition 8. The variable condition update vector, V is a set of binary status indications which can be set dynamically by a procedure running in a sequential processor. The component V_j is set to 1 by the procedure to indicate that data condition j is satisfied or V_j is set to 0 to indicate data condition j is not satisfied. For elements in S that are not to be dynamically updated, the associated element in the V vector can be set to either 0 or 1.

Proposition 3. The four possible implications on change in system status following completion of procedure i can be computed according to the formula:

$$S_{new} = (S_{old} \wedge T_i) \vee (T_i \wedge \overline{F}_i) \vee (\overline{F}_i \wedge V_i)$$

where the bar indicates the logical NOT operation.

Proof: The proof follows directly from the definitions of the associated vectors as shown in Table I.

TABLE I SYSTEM STATUS UPDATE POSSIBILITIES

t_{ij}	f_{ij}	S_{jNEW}	Implications to System Status
1	1	S_{jOLD}	unchanged
1	0	1	set true
0	1	0	set false
0	0	V_j	set variably

It should be noted that there are many forms which definitions 6 through 8 and proposition 3 could have taken. The expression which we have used has the advantage of restricting the range of V such that a procedure can dynamically modify only conditions for which it is authorized.

Proposition 4. The rangeof V is restricted such that V can modify only a specific subset of the data conditions, j. This subset is determined by T_i and F_i for procedure i such that S_j is determined by V_j if and only if $t_{ij} = 0$ and $f_{ij} = 0$.

Proof: The implied new values of S_j for the various values of t_{ij} and f_{ij} from proposition 3 are shown in Table I from which the proposition follows directly.

It should be noted that there are also implied modifications to system status at entry to a procedure; these modifications are to prohibit the same transition from being attempted in other processors by denying subsequent update access to relevant portions of ξ when $\xi' = \psi_i(\xi)$ has been initiated.

In order to accommodate exclusive data access, another construct must be added to negate availability of data which is to be updated by a currently activated procedure. The update is required to insure that read/write conflicts do not arise between procedures whose execution is not "indivisible". A discussion of the concept and definition of indivisibility can be found in references [9] and [11]. To implement this update, a vector A_i has been defined which is associated with each procedure, i to specify the status update implied on entry to that procedure.

Definition 9. The vector A_i is a set of binary status conditions a_{ij}, where the index j is associated with the data conditions whose status is maintained in S. $a_{ij} = 1$ if and only if the jth data condition is a mutually exclusive data availability condition required at entry to procedure i; $a_{ij} = 0$ otherwise.

Proposition 5. Modifying the system status vector according to the formula $S_{NEW} = S_{OLD} \wedge \overline{A}_i$ prior to entry is sufficient to effect contemporaneous access protection for procedure i.

The proof of this proposition follows immediately from definitions 1, 4 and 9 and proposition 2 if there are no procedures activated prior to activating procedure i which are affected by or affect these mutually exclusive data availability conditions. If such procedures are currently active, procedure i would not have become eligible. (Refer to Keller [9] for definitions of commutativity and persistence as they relate to valid parallel programs.)

Proposition 6. If $\overline{A}_i$ is identical to the ith row in R for all i, then all procedures with any entry conditions in common must execute sequentially.

The proof of this proposition follows as a special case of proposition 5.

Proposition 7. Modifying the system status vector according to the formula $S_{NEW} = S_{OLD} \vee A_i$ restores S to its original value.

Proof: The proof of this proposition follows directly from definition 9 and proposition 5 if there are no changes to S between entry and exit of the ith procedure. When there are other procedures initiated or terminated in the interval, the proof holds because no procedures can proceed in parallel if they are affected by or affect the same data availability condition covered by A_i. (Refer to Keller [9] for definitions of commutativity and persistence.) Therefore, for every condition for which $a_{ij} = 0$ there will have been no intermediate change to S_j and the proof is completed.

Proposition 8. The change in system status following completion of procedure i can be computed according to the formula:

$$S_{NEW} = ((S_{OLD} \vee A_i) \wedge T_i) \vee (T_i \wedge \overline{F}_i) \vee (\overline{F}_i \wedge V_i)$$

The proof follows directly from the proofs of propositions 3 and 7.

It has been shown in reference [14] that interrupts can be integrated into the model in a near conventional manner. Externally activated procedures are defined for them which can never become eligible based upon their R_i vector, but which have an associated system status update identical to internally activated procedures, when they exit. This updated system status will then activate appropriate interrupt processing procedures.

Procedure Activation

The emphasis of the preceding definitions and propositions has been to create a basis for determining the eligibility of the individual data transformations which comprise the computation, as well as to maintain a current system status vector. In effect we have a sufficient basis for the determination of the "When $R_i(\xi)$". This does not however include a sufficient basis for the activation of the data transformations, i.e., the "DO $\xi' = \psi_i(\xi)$". As a basis for this activation procedure, it will suffice to maintain a triple of descriptors for each procedure ($READ_i$, $WRITE_i$ and $EXECUTE_i$) in the System Controller. These descriptors designate respectively the elements of the data base, ξ which are to be read, the elements of the data base, ξ' which are to be written, and the starting address of the executable procedure, ψ_i which implements the data transformation. The appropriate descriptor triple can be transferred to the interface registers to effect activation of the procedure in the requesting processor as shown in Figure 2.

The addressing structure of the application programs which implement the transformations is shown in Figure 3. The EXECUTE register value

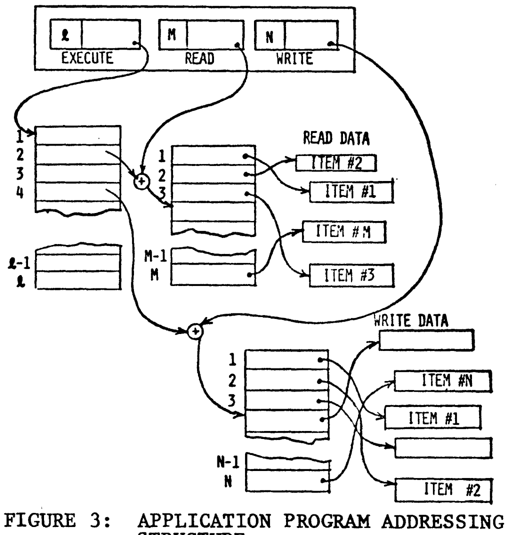


FIGURE 3: APPLICATION PROGRAM ADDRESSING STRUCTURE

transferred to the processor when the procedure is activated specifies the initial program counter value to be used. Data accesses by the program must be implemented with displacements relative to pointer packets whose starting addresses are indicated by either the READ or WRITE descriptor register value. This displacement specifies a particular descriptor value which in turn points to the data item being referenced. This scheme accommodates unique arguments to re-enterable programs as well as providing a basis for containment in multilevel secure systems [15].

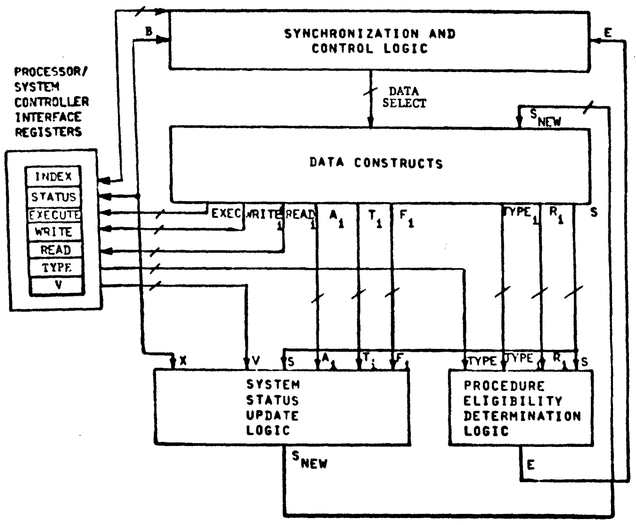
Processor Type Accommodations

The architecture model has been extended to include heterogeneous processor types. This is effected by maintaining a processor type designation for each procedure, $TYPE_i$. The procedure eligibility determination includes an evaluation of the equivalence between the type of the requesting processor and the type designation of the eligible procedure. Thus, if the defined procedure requires an I/O activity, an I/O controller would be specified as a requirement for the procedure. Having incorporated this approach into the model allows procedures to specify a special processor type such as floating point processors, vector instruction set processor, byte or word oriented processor, or just a specific processor model if several are multiprocessed in the same configuration.

The data construct, TYPE in Figure 2 is defined to accommodate this capability. In addition, each processor must have its own type identification available to the eligibility determining logic in the interface registers.

System Controller Hardware Organization

The System Controller is the device that is designed to contain the fixed data constructs for each procedure, performs the logic to determine procedure eligibility and system status updates, and assigns activities to processors. The device described in this section is a specific design based on the architecture model which has just been described. This design is applicable to either single or multiple processor systems. Figure 4 is a functional block diagram of this device. The content and function of the major blocks in the diagram are described below:



SYNCHRONIZATION AND CONTROL LOGIC
B
E
PROCESSOR/ SYSTEM CONTROLLER INTERFACE REGISTERS
DATA SELECT
S_{NEW}
DATA CONSTRUCTS
INDEX
STATUS
EXECUTE
WRITE
READ
TYPE
V
EXEC
WRITE
READ
A_i
T_i
F_i
$TYPE_i$
R_i
S
X
V
S
TYPE
SYSTEM STATUS UPDATE LOGIC
PROCEDURE ELIGIBILITY DETERMINATION LOGIC
S_{NEW}
E

FIGURE 4: FUNCTIONAL BLOCK DIAGRAM OF THE SYSTEM CONTROLLER

Processor/System Controller Interface

The interface block contains all data and control registers accessible to the processors. The structure and use of these registers are as follows:

STATUS a 3-bit read/write register whose bits are labeled P/P, B, and X, and which contain the following synchronization, protocol and mode request information.

P/P is a 1-bit binary semaphore used to prevent multiple processors from accessing the System Controller interface registers simultaneously. The P/P semaphore is set when a processor is accessing the System Controller and it is reset when no processor is currently accessing the System Controller.

B is used to prevent the processors from accessing the System Controller while it is busy servicing a request. When a processor makes a request to the System Controller, it waits until B is reset, sets X to the appropriate value, and sets B true. This activates the System Controller which resets B when the request has been serviced.

X is used to notify the System Controller of the type of service being requested. X is set (true) by the processors when the service requested is the result of a procedure exiting and it is reset when an activity is requested. X is only required in multiple processor implementations.

TYPE is a register used to contain the processor type identification. The System Controller uses this register to determine the next eligible procedure whose identification is to be loaded into INDEX. TYPE contains the processor category appropriate to the processor making the request. The System Controller returns the index of the next eligible procedure, whose type matches the value in the TYPE register.

INDEX is a register used to contain the identification of either the assigned procedure or the procedure currently being exited. As the fulfillment of processor activity requests, the System Controller loads INDEX with the index of the next eligible procedure whose type matches the value contained in the TYPE register, or INDEX is loaded with a 0 if no procedures of the appropriate processor type are eligible. When a procedure exits, the System Controller assumes INDEX contains the associated procedure index.

EXECUTE contains the entry point of the procedure whose index is contained in INDEX. (Refer to Figure 3.) EXECUTE is loaded by the System Controller as the result of the activity request. EXECUTE is unused when an exit is requested.

READ contains an indirect pointer to the global data item(s) accessible to the associated procedure in a read capacity. (Refer to Figure 3.) READ is loaded by the System Controller as the result of the activity request. READ is unused when an exit is requested.

WRITE contains an indirect pointer to the global data item(s) accessible to the associated procedure in a write capacity. (Refer to Figure 3.) WRITE is loaded by the System Controller as the result of the activity request. WRITE is unused when an exit is requested.

V contains the variable status update vector loaded by the processors upon exit from a procedure. This vector allows a procedure to return variable data condition status to the system status vector. Notice that his allows the task to modify only selected data elements since any attempt to modify unauthorized data will be masked out by the T and F vectors, stored internally to the System Controller.

By allowing the processors to access only the data and status registers defined above, all system control logic is localized to the System Controller. This also prevents the processors from accessing unauthorized programs, data, or control information, providing a natural basis for implementing secure systems. The security aspects of Transition Machines are discussed in reference [15].

Data Constructs

The data block is comprised of memory modules which contain the data structures required to control the system transitions, as shown in Figure 2. These include the EXECUTE, READ, WRITE, and TYPE arrays and the T,F,A, and R matrices as defined previously. The data block receives one input, the data select bus which addresses each of the EXECUTE, READ, WRITE, TYPE, T,F,R, and A constructs concurrently, causing the element indexed in each of these memories to be output on its associated data bus.

It is assumed that there is a load capability which will allow the programmer to change the content of these memory modules. The content must necessarily change during program development or in real-time in large systems where the matrices will be overlayed dynamically during the execution of the system (analogous to overlaying the task control blocks by a conventional operating system). For dedicated special purpose applications, however, these constructs could be fixed and put into read-only memory. The load procedures are system-dependent and are therefore not a subject of this paper. An implementation applicable to large general-purpose systems is discussed further on and is the subject of continuing research. The sizing of these memory modules is addressed further on in this paper also.

Fixed Transition Logic

The fixed transition logic block contains the combinational logic necessary to update the system status vector and to determine procedure eligibility. This block requires the T,F,V, and A vectors for the procedure currently being assigned or exited in order to generate the new system status vector, S. The logic expression for the new S vector generated as the result of either an activation or procedure exit request are shown in Figure 5. This diagram uses the

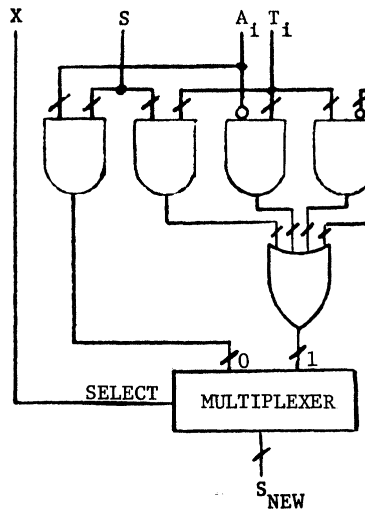


FIGURE 5: SYSTEM STATUS UPDATE LOGIC

convention defined in Figure 6 of using a "/" on a line to indicate multiple lines treated identically.

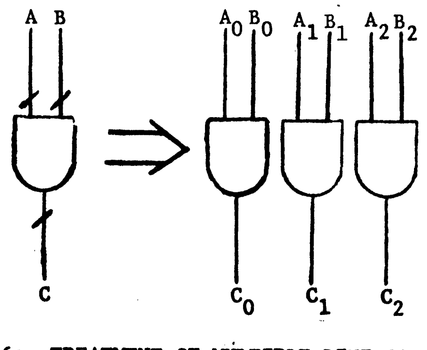


FIGURE 6: TREATMENT OF MULTIPLE LINE GATE INPUTS

The combinational logic also combines the current S vector with successive rows from the R matrix and compares the successive elements of the type array with the TYPE register to determine the eligibility of each procedure. A single output bit, E_i is provided as an input to the synchronization and control logic. The logic to obtain E_i is shown in Figure 7.

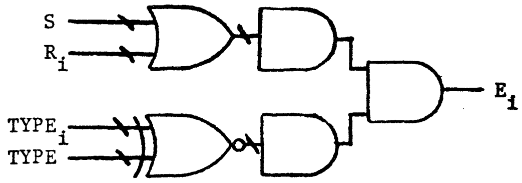


FIGURE 7: PROCEDURE ELIGIBILITY DETERMINATION LOGIC

The System Controller design described here assumes the eligibility vector (E) is computed one element at a time. This need not be the case. An associative memory can be used to generate the entire E vector in parallel which will result in much faster transition speeds but requires more hardware support.

Synchronization and Control Logic

The synchronization and control logic synchronizes the operation of all the components of the System Controller. The control logic operates as shown in Figure 8. The System Controller waits

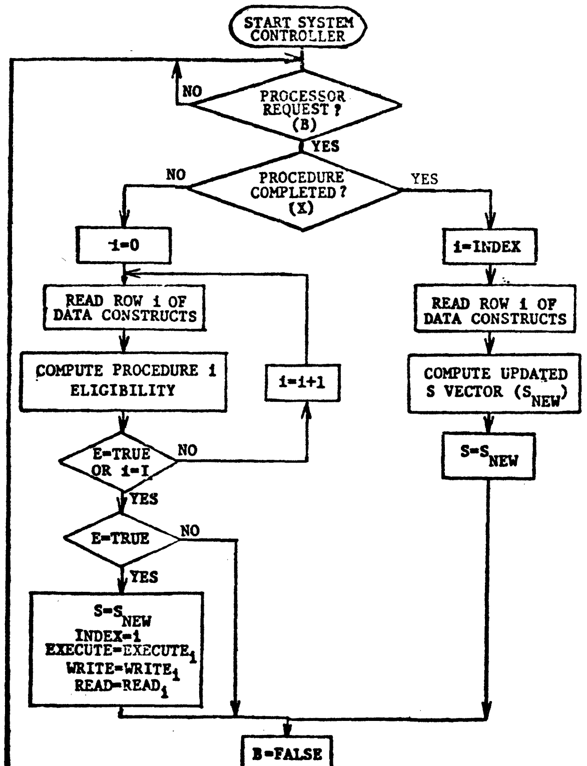


FIGURE 8: SYNCHRONIZATION AND CONTROL LOGIC

in an idle state until its execution is initiated by a processor request (i.e., B set). If X is set, the System Controller initiates a system status update and if X is not set, a procedure eligibility determination and assignment is initiated.

When a procedure exit request is initiated, the non-zero procedure index provided by the processor is used as the address selection value on the data select bus. This in turn causes the appropriate array elements to be output on each of the memory data buses. The fixed transition logic then updates the system status vector to effect the update implied by the procedure exit. The B indicator in the STATUS register is then reset to indicate the request has been serviced.

The procedure activation request causes successive rows of the R matrix and the TYPE array to be output on their respective data buses. As each row is output, the fixed transition logic generates the next element of E. INDEX is loaded with either the procedure index for the first eligible procedure or with a zero signifying that no procedure is currently eligible. If INDEX is non-zero, the EXECUTE, READ, and WRITE pointers associated with the indexed procedure are transferred to their respective interface registers. The fixed transition logic then generates the new S vector appropriate to the protection of the assigned procedure. At this point, a complete entry transition has been effected. The System Controller busy indicator, B is then reset to allow the processors access to the interface registers again.

The detailed processor logic and System Controller interface and internal logic are provided in the syntactical expressions of Figure 9 and 10. These figures have didactic value as indicative of a source language structure applicable to application programs to be run on Transition Machines.

Performance Characteristics

A throughput performance model was developed which predicted the throughput capabilities of parallel transition Machines [13]. This model was made general enough to include multiprocessors with software executive control mechanisms. The three major contributions to system overhead that were examined are memory contention, the overhead associated with the central control mechanism, and control mechanism lockout experienced while waiting for a request to be serviced. Memory contention contributions were assumed to arise even from processors which are determining their next application program assignment (i.e., currently executing the executive program in the case of a conventional system or waiting for the System Controller to service the outstanding procedure entry request in Transition Machines). Since Parallel Transition Machines can be designed to be exempt from this contribution to memory contention, measured performance should be better than predicted by the model in this regard.

The significant performance parameter was shown to be $\rho \equiv A/\emptyset$, where $\emptyset$ is the characteristic System Controller overhead per application procedure (eligibility determination and system status update), and A is the characteristic application procedure execution time requirement. The value of ρ determines the number of processors that can be effectively combined in a tightly coupled mode of operation as described in the reference.

```
When Exit false (Entry)
 When test and set of P/P semaphore true
  When B false (System Controller not busy)
   Begin: Assignment Processing
    Store TYPE
    Store X false (entry)
    Set B true (activate System Controller)
    When B false (System Controller not busy)
     If INDEX≠0 then
        Load INDEX
        Load READ
        Load WRITE
        Load EXECUTE
        Clear P/P semaphore
        Transfer Control to EXECUTE
        Set Exit true
     Else
        Clear P/P semaphore
   End: Assignment Processing
When Exit true
 When test and set of P/P semaphore true
  When B false (System Controller not busy)
   Begin: Exit Processing
    Store INDEX
    Store V
    Set X true (exit)
    Set B true (activate System Controller)
    Set Exit false
    Clear P/P semaphore
   End: Exit Processing
```

FIGURE 9: PROCESSOR CONTROL LOGIC

```
When B true
 Begin: System Controller logic
  If X true (exit) then
   i = INDEX
   $S_{NEW} = ((S_{OLD} \vee A_i) \wedge T_i) \vee (\overline{F}_i \wedge V) \vee (T_i \wedge \overline{F}_i))$
  Else (entry)
   Clear i
   While $E_i = 0$ and $i < I$
    Increment i
    $E_i = \bigwedge_{j=1}^{J} (s_j \vee r_{ij}) \wedge (TYPE \odot TYPE_i)$
   If $E_i = 0$ then
    INDEX = 0
   Else
    INDEX = i
    READ = $READ_i$
    WRITE = $WRITE_i$
    EXECUTE = $EXECUTE_i$
    $S_{NEW} = (S_{OLD} \wedge \overline{A}_i)$
  Set B false
 End: System Controller Logic
```

FIGURE 10: SYSTEM CONTROLLER LOGIC

The overhead, $\emptyset$ is very dependent upon the component technology used in the development of the System Controller, A is dependent upon the speed of the individual processors. Current component technology will support $\emptyset \leq 1$ microsecond. With microprocessors, $A \geq 100$ microseconds is very conservative. This yields $\rho = 100$, which indicates according to the model that on the order of 100 microprocessors could be combined in a tightly coupled mode of operation controlled by a single System Controller with a proportionate throughput capability.

Lockout is the primary reason for the flattening of the throughput curve as a function of the number of processors. To avoid this problem in Parallel Transition Machines, multiple system Controllers (including separate interface registers) can be incorporated so that if a processor is locked out of one System Controller it can attempt to acquire another, etc. Thus, in a batch type system, many disjoint computations could be running contemporaneously across all processors.

A given computation will be characterized by some maximum and average numbers of concurrent execution paths. (See for example Kuck [10].) The average concurrency will determine the processor utilization realizable during the execution of a given computation. Thus, in general where many processors are included in a configuration, there is a requirement for concurrency of active computations in order to achieve efficient utilization of processors. This applies particularly where a large system is being used for many small computations.

The upper limit on throughput in Parallel Transition Machines is thus determined by other than system control considerations. Physical module interconnection schemes now become the limiting factors. High speed buses and processor/memory groupings [12] are likely approaches to extending these limits.

System Controller Memory Sizing

In what has been presented so far, there has been the implicit assumption that all of the constructs associated with relevant data conditions and programmed procedures for an entire system can be accommodated in the data constructs memory modules in the System Controller. In a prototype system currently in the development stages, these constructs are contained in a single module and are allocated dimensions of 32 conditions by 32 procedures. Analytic studies and simulations have indicated that as a rule of thumb there are one and one-half times as many conditions required to control a tightly coupled network of procedures as the number of procedures involved. This would indicate that optimum memory utilization would be more probable for System Controllers characterized by such dimensional ratios.

How large these memories should be made to support general applications is a key issue. There is no real problem in sizing these arbitrarily large, but application systems have a way of outgrowing single physical modules. Methods have therefore been investigated which extend the logical capacity of the System Controller in modular incremental units to justify the development of a standard System Controller applicable across a broad spectrum of system sizes.

The most direct method of extending capacity is to directly increase the dimensions of the memory in the System Controller. The design described in the previous section can be expanded to accommodate more data conditions (i.e., horizontal expansion of the arrays) or more procedures (i.e., vertical expansion of the arrays) by cascading multiple sets of the standard component blocks which are then connected to a common processor/System Controller interface. This facilitates the construction of an arbitrarily large System Controller by the interconnection of many standard System Controller components each of fixed size. This cascading of component System Controllers is shown in Figure 11 for horizontal expansion. A

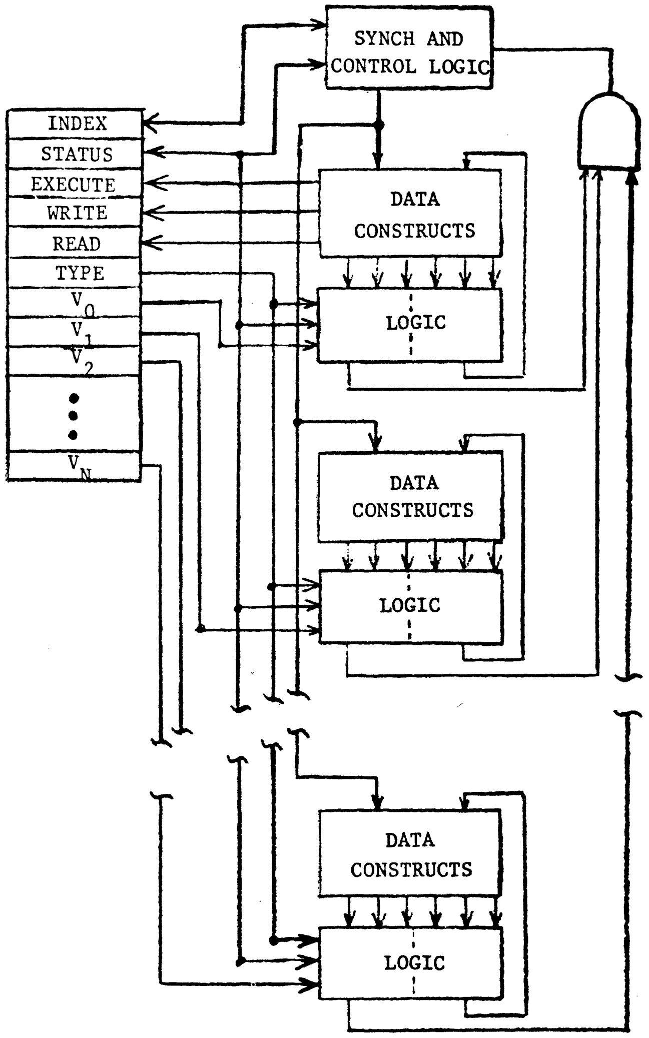


FIGURE 11: MODULAR HORIZONTAL EXPANSION OF THE SYSTEM CONTROLLER

similar approach is applicable to vertical expansion, where a master System Controller is used to multiplex between vertical segments, each of which is controlled by its own System Controller. These methods of modular expansion do not increase worst case transition times and can be applied in a structured physical hierarchy.

Virtual Transition Machines

Memory size problems are not new to computing, and the resolution of former problems encountered with insufficient main memory can be applied directly to the System Controller memories. The concept of virtual memory and commensurable virtual machines is particularly germain to Transition Machines. The segmentation and paging of the data constructs required by Transition Machines can be an integral part of a hierarchical application program design approach. The approach encompasses the design and implementation of systems as a complete transition system at each level in a hierarchy. This approach incorporates the capability of associating an indentured R matrix with a row in a higher level matrix. Conditions appropriate to each matrix level include only those which are relevant to the procedures (or immediate further indentured matrices) at this level. This top-down recursion can proceed in the extreme all the way down to where the procedures become the instruction set of processors or even indivisible operators. This instruction set can then be restricted to exclude branching type instructions. In fact, even going up one level, the Parallel Transition Machines can be used to implement a completely general system in conventional processors without requiring a branch type (GOTO) instruction in the application domain of the processors. The implications to software productivity are discussed in reference [1].

To implement the logical extension of the data constructs by partitioning the R, T, F and A matrices, additional data constructs and algorithms will be required to effect the dynamic real-time loading and overlaying of procedures. The required constructs are the following:

1. System Controller identification

2. Relevance matrix identification

3. Indication of whether a procedures or an indentured matrix is associated with each row in an R matrix.

4. Controller active indication for each System Controller

5. Logical parent identification (relevance matrix identification of parent)

6. Physical parent identification (System Controller identification of parent)

These data constructs are felt to be sufficient to implement a global operating system which results in a virtual implementation of the total system relevance matrix. Such an operating system is the subject of current and anticipated future research.

Conclusions

It has been demonstrated that even though there are no completely general parallel computation architectures commercially available, such computers are nonetheless realizable. Parallel Transition Machines which meet these specifications are defined to a level where credibility is established. A prototype of such a machine is currently in the developmental stages at the Boeing Aerospace Company. A design has been presented in this paper which is extremely flexible to meeting a wide range of implementation variations. Such machines appear to have considerable advantages over current machine architectures in several areas. These area include: Multiprocessing throughput, software productivity, and ADP security.

It is left to the future to develop the compilers, linkage editors and overlaying operating systems appropriate to the application of such computers.

References

[1] Anastas, M. S. and Vaughan, R. V., "Direct Architectural Implementation of a Requirements-Oriented Computer Structure", (submitted for publication in 1979)

[2] Anderson, G. A., and Jensen, E. D., "Computer Interconnection Structures: Taxonomy, Charactistics and Examples", ACM Comp. Surveys 7,4 (Dec. 1975), pp 197-213

[3] Bell, C. G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, N.Y. (1971)

[4] Conway, M., "A Multiprocessor System Design", Proc. AFIPS 1963 Fall Jt. Computer Conf., Spartan Books, Baltimore, Md. (1963) pp 139-146

[5] Dijkstra, E. W., "Cooperating Sequential Processes", In Programming Languages, F. Genuys (Ed.), Academic Press (1968) pp 43-112

[6] Habermann, A. N., "Synchronization of Communicating Processes", Comm. ACM 15,3 (Mar. 1972) pp 171-184

[7] Jensen, E. D., Thurber, K. J., and Schneider, G. M., "A Review of Systematic Methods in Distributed Processor Interconnection", Proc. IEEE Int. Conf. on Communications (1976)

[8] Keller, R. M., "Formal Verification of Parallel Programs", Comm. ACM 18,7 (July 1976), pp 371-384

[9] Keller, R. M., "Parallel Program Schemata and Maximal Parallelism", J.ACM 20,3 (July 1973), pp 514-537

[10] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", ACM Comp. Surveys 9,1 (Mar. 1977) pp 29-59

[11] Lipton, R. J., "Reduction: A Method of Proving Properties of Parallel Programs", Comm. ACM 18,12 (Dec. 1975) pp 717-721

[12] Swan, R. J. Bechtolsheim, A., Lai, K. and Ousterhout, J. R., "The Implementation of the Cm* Multi-Microprocessor", AFIPS Conf. Proc. Vol. 46, Nat. Computer Conf. (1977) pp 645-655

[13] Vaughan, R. F. and Anastas, M. S., "Limiting Multiprocessor Performance Analysis", Proc. 1979 Int. Conf. on Parallel Processing, (Aug. 1979)

[14] Vaughan, R. F. and Anastas, M. S., "Microprocessor-Based Transition Machines", Proc. COMPCON FALL '79 (Sept. 1979)

[15] Vaughan, R. F. and Anastas, M. S., "Preliminary Analyses to Obtain an Expanded Model and Preferred Implementation of Verifiably Secure ADP System", Boeing Doc. D180-25090-1 (1979)

PERFORMANCE EVALUATION AND RESOURCE OPTIMIZATION OF MULTIPLE SIMD COMPUTER ORGANIZATIONS

Kai Hwang and Lionel M. Ni
School of Electrical Engineering
Purdue University
W. Lafayette, Indiana 47907

Abstract -- Multiprocessor computer systems, which can be used to execute multiple number of SIMD vector jobs, are modeled and analyzed in this paper. Such a parallel computer organization has usually multiple Control Units (CU's) sharing a pool of dynamically allocated Processing Elements (PE's). We characterize the MSIMD machine as a finite-storage multi-server M/M/K/ℓ queueing family where the number of active servers, K, is an upper bounded random variable and ℓ is the maximal queue length. Analytic results using conditional probabilities are obtained under equilibrium conditions. These results can be readily applied to evaluate the performance of multiple SIMD machines. The system performance is measured by the resource utilization factors CU's and PE's and by the average job response time. Given a fixed number of CU's and a prespecified workload, systematic procedures are given to determine the optimal size of the resource pool of PE's and the sufficient queue capacity for MSIMD operations.

I. INTRODUCTION

This paper presents the results of an analytic study to evaluate the performance of shared-resource Multiple Single Instruction stream and Multiple Data streams (MSIMD) machine organizations. An MSIMD machine is composed of more than one Control Units (CU's) sharing a pool of finite number of Processing Elements (PE's) through a interconnection switching network. To facilitate generalized discussions, we assume that the system has m CU's and r PE's interconnected through a full crossbar switching network. Furthermore, sufficient large local memories and I/O facilities are attached with each CU and each PE in the system. (Fig.1).

Each CU is required to be allocated with a subset of PE's for the execution of a given SIMD job (a vector process). The word "job" will be used here to denote some identifiable piece of work that is logically independent of all other jobs. Thus, the work on one job is never dependent on the work of another; the only way the jobs interact with each other is by their independent needs for the same resources. Several parallel machines capable of executing multiple SIMD jobs have been proposed in the past. The original ILLIAC IV design was for four CU's [1]. Senzig [15] studied an array processor with multiple CU's. Radoy and Lipovski [13] proposed multiple array processing with switched multiple instructions and multiple data streams. In the MAP system, Nutt [10]-[12] presented a series of studies on MSIMD machines including operating system strategies and performance evaluations. Hwang and Ni proposed a dynamically allocated multiple SIMD/SISD architecture [5]. Recently, a reconfigurable multiprocessor system, called PM^4, was proposed by a research team at Purdue University for pattern recognition and image processing applications [2]. PM^4 is designed for performing MIMD and MSIMD operations through a dynamic reconfiguration approach.

*This research was supported in part by NSF Grant MCS-78-18906.

In this study, we will not partition the set of PE's into fixed subsets. Instead, we assume a completely accessible crossbar switch which can establish all the possible interconnections between CU's and PE's. This is necessary for dynamic partitioning of the PE set as required in our analysis. The subset of PE's allocated to a given CU may vary in size according to the job requirement. The time required to allocate PE's to the designate CU is considered part of the CU service time. Various types of queueing models have been proposed to study the performance of computer systems by a number of authors [3,4,14]. This paper presents the first attempt to model MSIMD machines in an analytic fashion.Numerical plottings of the analytic results are presented with commentaries and comparisons.

An analytical solution is obtained by modeling the MSIMD machines as an M/M/K/ℓ queueing family, where M and M represent respectively the exponential interarrival time and the exponential service time. The number of active servers (CU's), K, is a random variable and ℓ is the maximum queue length. The MSIMD system performance is measured by the utilization of CU's and of PE's and by the average response time for jobs with arbitrarily distributed vector sizes. Resource organization on optimal PE-pool size and method to determine the sufficient buffer size for specified wordload distribution are also presented.

II. THE QUEUEING MODEL FOR MSIMD MACHINES

The MSIMD machine organizations illustrated in Fig.1 can be modeled by the queueing network in Fig.2. Specified below are the general characteristics of the model for MSIMD machine organizations and queueing disciplines to be used in our study.

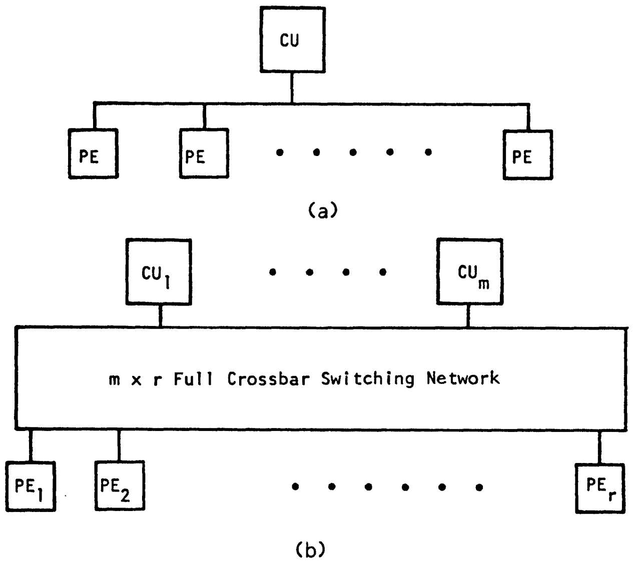


Fig. 1 (a) SIMD machine organization.
(b) Multiple SIMD (MSIMD) machine organization

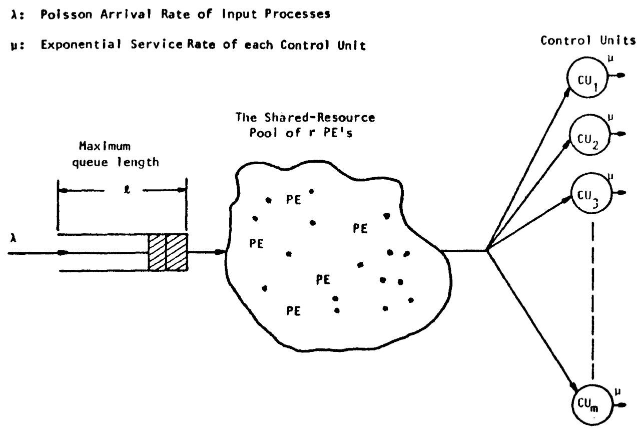

λ: Poisson Arrival Rate of Input Processes
μ: Exponential Service Rate of each Control Unit
Control Units
The Shared-Resource Pool of r PE's
Maximum queue length
ℓ
λ
PE
CU_1
CU_2
CU_3
CU_m
μ

Fig. 2 The MSIMD model with m Control Units (CU's) and a shared-resource pool of r Processing Elements (PE's).

1. The Servers (CU's).

There are m identical servers called the Control Units (CU's). The service time of each CU is assumed to be exponentially distributed with service rate μ. Each CU can supervise the execution of one SIMD job at a time and at most m SIMD jobs can be executed simultaneously in the system. All the servers are uniprogrammed and operate independently except sharing the same pool of PE's under the control of an executive system which may reside in a separate manager processor.

2. The Shared Resource Pool (PE's).

There are r identical Processing Elements (PE's) shared by all the CU's. Each PE can be only allocated to one CU at one time through the m x r crossbar switching network. Note that m additional crosspoints are needed per each additional PE. We call this the switching overhead per each PE. All the PE's are independent and uniprogrammed to accept one assignment at a time.

3. The Job Arrival Process.

The arrival of the SIMD jobs assumes a Poisson distribution. We shall write

$$P_r\{M(t) = n\} = \frac{e^{-\lambda t}(\lambda t)^n}{n!} \qquad (1)$$

as the probability that there will be n SIMD jobs arriving in time period t, with the mean arrival rate λ.

4. The SIMD Job Characteristics.

Let $\{X_i, i=1,2,\ldots\}$ be a sequence of positive, integer-valued, independent random variables with a common distribution F. The PE demand, X_i represents the maximum number of PE's required for the i-th SIMD job. The probability distribution F of X_i is called the PE-demand distribution. The values of X_i are confined within the range [1,r] where r corresponds to the maximum number of available PE's.

The PE-demand distribution F does not follow a specific pattern in reality. Nutt [12] has assumed a normal distribution for F. Our analytic results are obtained based on arbitrary distribution of F. Analytic results are numerically plotted with respect to a "truncated" normal distribution for F. For real SIMD job, the vector size may be greater than the maximum pool size r. Under such circumstances, the long vector instruction must be partitioned into several subvector instructions, chained together to fit the SIMD job requirement. The average PE demand is defined by

$$\eta = E[X_i] = \sum_{k=1}^{r} k \cdot P_r\{X=k\} \qquad (2)$$

5. The Maximum Queue Capacity (ℓ).

For an infinite queue, every arriving SIMD job is allowed to wait in the queue until service can be provided. In practical multiprocessor design, this assumption is unrealistic due to the finite capacity of buffer area used. We shall assume "ℓ" to be the maximum number of jobs allowed in the queue. When the queue is full, the new jobs are turned away. With a finite-storage queue, the Actual Job Arrival Rate, λ_a, is less than the Real Job Arrival Rate, λ, corresponding to an infinite queue.

6. The Queueing Discipline.

The queueing discipline specifies the order in whcih the multiple SIMD jobs are to be executed by multiple CU's. Let us first define the states of a CU. A job is assigned to a CU, (that CU enters a busy state) provided a sufficient number of PE's are available to form the task force. Until the task force can be adequately formed, the unassigned CU's are still available for assignment and in the waiting state.

The First-Come-First-Serve (FCFS) discipline decides the order of service strictly by the order of job arrivals. It is the simplest scheduling policy to implement. However, it has the disadvantage of having an SIMD job, which demands a large number of PE's from the pool, blocking in the front of the queue waiting for sufficient task force, even the remaining available PE's in the pool may satisfy other small jobs in the queue.

The Least-PE-Demand-Serve-First (LDSF) discipline can be used to alleviate the drawbacks of the FCFS discipline. The insertions of successive jobs must maintain a decreasing PE demand order in the queue such that less-PE-demanding jobs are always placed ahead of the more-PE-demanding jobs. Once a job is assigned to a CU, it cannot be ejected until the service is completed (that is nonpreemptive).

III. SYSTEM PERFORMANCE MEASURES

Before we evaluate the performance of the MSIMD machine organization, we have to define the states of the system. The random number, N(t), of SIMD jobs residing in the system at time t can be divided into two parts

$$N(t) = N_q(t) + N_s(t) \qquad (3)$$

where $N_q(t)$ and $N_s(t)$ are nonnegative, integer-valued random variables representing at time t, respectively, the number of jobs waiting in the queue and the number of jobs which are currently under execution. The system is in state E_{ij} at time t, when $N_q(t) = i$ and $N_s(t) = j$ where $0 \leq i \leq \ell$, $0 \leq j \leq m$. The probability that the system is in state E_{ij} at time t is denoted by $P_{ij}(t)$. Note that $P_{ij}(t) = 0$ for $i>\ell$ or $j>m$.

After the system becomes stabilized, i.e. being operational for a long time period, the influences of the initial conditions and transient responses will be damped out and the number of jobs in the system becomes time-independent. The system is then entering the so-called "steady state". The limiting probability of $P_{ij}(t)$ is simply denoted by

$$P_{ij} \triangleq \lim_{t\to\infty} P_{ij}(t) \qquad (4)$$

Furthermore, in the steady state, we have the time-invariant random variables $N \triangleq \lim_{t\to\infty} N(t) = \lim_{t\to\infty} (N_q(t) + N_s(t)) = N_q + N_s$.

Four system measures are defined below to evaluate the performance of MSIMD machines. The internal efficiency of an MSIMD system is primarily determined by its utilization factors of the CU's and PE's. The external performance of the system is indicated by the mean response time of user's jobs. All these measures are evaluated under the steady-state condition.

1. The CU utilization (ρ_{CU})

The ratio of the expected number of CU's in busy state to the total number of CU's in the system defines the CU utilization factor

$$\rho_{CU} = \frac{1}{m} E[N_s] = \frac{1}{m} \sum_{i=0}^{\ell} \sum_{j=1}^{m} j \cdot P_{ij} \qquad (5)$$

2. The PE utilization (ρ_{PE})

Not all the PE's allocated to a CU will be busy all the time. For example, when a control or scalar instruction is executed in the CU directly, none of the allocated PE's will be called. In a masked vector operation, portion of the allocated PE's may be masked out. In both cases, the disabled PE's may be left idling even they have been allocated to the CU. For simplicity in analysis, we shall consider a PE to be utilized once it is allocated to a CU, regardless whether the PE is actually executing a broadcasted instruction from the CU or it is left idling during the instruction cycle.

$$\rho_{PE} = \frac{1}{r} \sum_{i=0}^{\ell} \sum_{j=1}^{m} \eta_j \cdot P_{ij} \qquad (6)$$

where η_j is the average number of PE's required for j SIMD jobs.

3. The Total System Utilization (ρ)

The overall system resource utilization can be measured as a weighted sum of the two utilization factors ρ_{CU} and ρ_{PE} divided by their maximum values.

$$\rho = (\eta \cdot m \cdot \rho_{CU} + r \cdot \rho_{PE}) / (\eta \cdot m + r) \qquad (7)$$

, where the weighting factors m and r correspond to the multiplicity of CU's and PE's in the system and η is the average PE-demand in an arbitrary SIMD job. The reason that the CU's are weighted higher is due to their higher hardware complexity and software capability. Whenever a CU is busy, an average of η PE's will be drafted for doing the job.

4. The Average Job Response Time (W)

By Little's formula [8], we can define the Average Job Response Time, W, as the ratio of two parameters

$$W = L/\lambda_a = (\sum_{i=1}^{\ell}\sum_{j=1}^{m} i \cdot P_{ij} + m \cdot \rho_{cu})/\lambda \cdot (1 - \sum_{k=1}^{m} P_{\ell k}) \qquad (8)$$

, where L is the average number of jobs in the system including both jobs waiting in the queue and those currently under execution and λ_a is the actual job arrival rate. $P_{\ell k}$ is the probability that the queue is full (no additional arriving jobs are allowed to enter a full queue) when k SIMD jobs are being executed in the system. These performance measures, ρ_{CU}, ρ_{PE}, ρ and W, will be plotted in section V with numerical examples.

IV. THE ANALYSIS AND PROBABILISTIC RESULTS

Let S_k be a random variable representing the total number of PE's required for k independent SIMD jobs. We can write S_k in terms of a sequence of random variables X_i

$$S_k = \sum_{i=1}^{k} X_i \qquad \text{where } 1 \le k \le m \tag{9}$$

Let F_k be the probability distribution of S_k. The expected value of S_k is denoted by $\eta_k = E[S_k]$. F_k can be written as the k-fold convolution of F with respect to itself k times, where * is the convolution operator in the r-domain. We shall write $F_k(r) = P_r\{S_k \le r\}$, which equals F_k evaluated at point r.

$$F_k = \underbrace{F * F * \cdots * F}_{k \text{ times}} \tag{10}$$

Since all random variables X_i are independent with identical distribution, the expected value η_k can be written as $\eta_k = E[S_k] = k \cdot E[X_i] = k \cdot \eta$.

Given the PE-pool size r. Let K(r) be the random variable representing the number of active servers (CU's) in the system. Having k active servers means the system can have at most k CU's in busy state. The new jobs will waiting in queue only when all the k CU's are busy.

The probability of having k active servers is denoted by $\alpha_k = P_r\{K(r) = k\}$. The expected number of active servers is denoted by $\alpha = E[K(r)]$. The number K(r) is greater than or equal to k, if and only if k SIMD jobs can be executed simultaneously; that is $K(r) \ge k \leftrightarrow S_k \le r$ for all $1 \le k \le m$.

Lemma 1:

With a given pool of r PE's we can write

$$\alpha_k = \begin{cases} F_k(r) - F_{k+1}(r) & 1 \le k \le m-1 \\ F_m(r) & k=m \\ 0 & k>m \end{cases} \tag{11}$$

Proof:

$$\alpha_k = P_r\{K(r) = k\} = P_r\{S_k \le r, S_{k+1} > r\}$$
$$= P_r\{S_k \le r\} - P_r\{S_{k+1} \le r\}$$
$$= F_k(r) - F_{k+1}(r), \text{ for } 1 \le k \le m-1.$$

$\alpha_m = P_r\{K(r) = m\} = P_r\{S_m \le r\} = F_m(r)$, for k = m.

$\alpha_k = 0$,since there are only m CU's, for k > m.

$$\sum_{k=1}^{\infty} \alpha_k = \sum_{k=1}^{m-1} \alpha_k + \alpha_m = \sum_{k=1}^{m-1} [F_k(r) - F_{k+1}(r)]$$
$$+ F_m(r) = F_1(r) = 1$$

Q.E.D.

Lemma 2:

The expected number α of active servers can be evaluated by

$$\alpha = \sum_{n=1}^{m} F_n(r) \tag{12}$$

Proof:

$$\alpha = E[K(r)] = \sum_{k=1}^{m} k \cdot \alpha_k$$
$$= \alpha_1 + 2\alpha_2 + \cdots + (m-1)\alpha_{m-1} + m \cdot \alpha_m$$
$$= (\alpha_1 + \alpha_2 + \cdots + \alpha_m) + (\alpha_2 + \alpha_3 + \cdots + \alpha_m)$$
$$+ \cdots + (\alpha_{m-1} + \alpha_m) + \alpha_m$$
$$= \sum_{n=1}^{m} P_r\{K(r) \ge n\} = \sum_{n=1}^{m} P_r\{S_n \le r\}$$
$$= \sum_{n=1}^{m} F_n(r)$$

Q.E.D.

For a fixed number of k active servers, the system can be described by a finite storage M/M/k/ℓ queue. We can view the collection of m M/M/k/ℓ queues for k = 1,2,...,m as an M/M/K/ℓ queueing family, where K is a random variable ranging from 1 to m as defined above. The analytic results obtained below are averaged over m possible M/M/k/ℓ queues in the M/M/K/ℓ queueing family. These results can be used only as an approximated solution to the MSIMD system in a statistical sense. Simulation results reported in [9] verifies that this approximated solution is indeed very close from below to what can be obtained from the simulation study using either disciplines.

Using the theorem on total probability, the limiting probability P_{ij} can be evaluated as

$$P_{ij} = P_r\{N_q = i, N_s = j\}$$
$$= \sum_{k=1}^{m} P_r\{N_q = i, N_s = j \mid K(r) = k\} \cdot \alpha_k \tag{13}$$

where $P_r\{N_q = i, N_s = j \mid K(r) = k\}$ is the conditional probability of the system being in state E_{ij} given that there are only k active servers.

We define $u = \lambda/\mu$ the traffic intensity and Z_k

the probability of an empty system with all k active servers idling and an empty queue, that is $Z_k = P_r\{N_q = 0, N_s = 0 \mid K(r) = k\}$.

The feasible state space of an $M/M/k/\ell$ queue is $\{E_{00}, E_{01}, \ldots, E_{0k}, E_{1k}, E_{2k}, \ldots, E_{\ell k}\}$. Then $P_r\{N_q = i, N_s = j \mid K(r) = k\}$ represents the steady state probability of the system in state E_{ij}. With finite queue length, the system approaches the steady state for various values of traffic intensity. Proofs of the following theorems can be found in [9].

Theorem 1:

$P_r\{N_q = i, N_s = j \mid K(r) = k\}$

$$= \begin{cases} \frac{u^j}{j!} Z_k & , i=0 \text{ and } 0 \le j \le k \\ \frac{u^k}{k!} (\frac{u}{k})^i Z_k & , j=k \text{ and } 1 \le i \le \ell \\ 0 & , \text{otherwise} \end{cases} \quad (14)$$

$$\text{where} \quad Z_k = \left[\sum_{n=0}^{k} (\frac{u}{n!})^n + \frac{u^k}{k!} \sum_{n=1}^{\ell} (\frac{u}{k})^n \right]^{-1} \quad (15)$$

Theorem 2.

The limiting probability P_{ij} defined in Eq.5 can be evaluated by

$$P_{00} = \sum_{k=1}^{m} Z_k \cdot \alpha_k$$

$$P_{0j} = \sum_{k=j}^{m} \frac{u^j}{j!} Z_k \cdot \alpha_k \qquad 1 \le j \le m \quad (16)$$

$$P_{ij} = \frac{u^j}{j!} (\frac{u}{j})^i Z_j \cdot \alpha_j \qquad 1 \le i \le \ell,\ 1 \le j \le m$$

Theorem 3.

The CU utilization factor ρ_{CU} and the PE utilization factor ρ_{PE} can be written as

$$\rho_{CU} = \frac{u}{m} \cdot \left[1 - \sum_{k=1}^{m} Z_k \cdot \alpha_k \frac{u^k}{k!} (\frac{u}{k})^\ell \right] \quad (17)$$

$$\rho_{PE} = \frac{m \cdot \eta}{r} \cdot \rho_{CU} \quad (18)$$

Corollary:

The system utilization ρ can be evaluated by

$$\rho = \frac{2m\eta\ \rho_{CU}}{m \cdot \eta + r}$$

$$= 2u \cdot \eta \left[1 - \sum_{k=1}^{m} Z_k \cdot \alpha_k \frac{u^k}{k!} (\frac{u}{k})^\ell \right] \Big/ (m \cdot \eta + r) \quad (19)$$

Lemma 3:

The average queue length L_q can be evaluated by

$$L_q = \sum_{j=1}^{m} \frac{u^j \cdot Z_j (u/j)}{j!(1-u/j)^2} \left[1 - (\frac{u}{j})^{\ell+1} - (\ell+1)(\frac{u}{j})^\ell (1 - \frac{u}{j}) \right] \cdot \alpha_j \quad (20)$$

The following theorem shows that the average job response time W equals the sum of the average job waiting time, L_q/λ_a, and the average job service time, $1/\mu$.

Theorem 4

$$W = \frac{L_q}{\lambda} \cdot \left[1 - \sum_{k=1}^{m} \frac{u^k}{k!} \cdot (\frac{u}{k})^\ell \cdot Z_k \cdot \alpha_k \right]^{-1} + \frac{1}{\mu} \quad (21)$$

V. NUMERICAL EXAMPLES AND COMPARISONS

Results obtained from the analytic study are displayed and compared by numerical examples in this section. The influences of the input workload characteristics are demonstrated. Analyses given in previous section are independent of the PE-demand distribution F. We assume a "truncated" normal distribution for the PE resource pool as demanded by the user's jobs. Such truncated normal distribution can be formally defined by

$$P_r\{X = k\} = \begin{cases} \frac{1}{N_X \sqrt{2\pi\sigma_n^2}} \text{EXP} \left[- \frac{(k - \eta_n)^2}{2\sigma_n^2} \right], \\ \qquad \text{for } 1 \le k \le r \\ 0, \text{ otherwise} \end{cases} \quad (22)$$

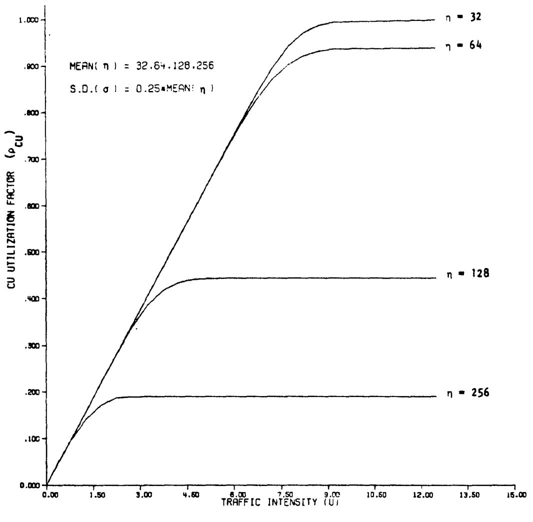
where $N_X = \sum_{k=1}^{r} \frac{1}{\sqrt{2\pi\sigma_n^2}} \text{EXP} \left[- \frac{(k - \eta_n)^2}{2\sigma_n^2} \right]$ is the normalizing factor.

The conventional normal distribution with mean η_n and standard deviation σ_n is denoted by $N(\eta_n, \sigma_n)$. The truncated normal distribution of $N(\eta_n, \sigma_n)$ in range $[1, r]$ is denoted

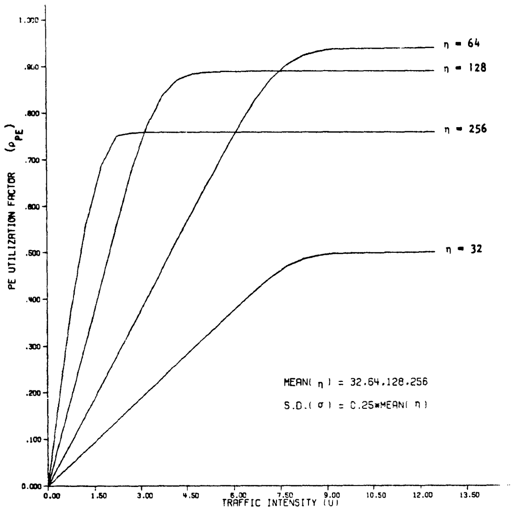
$N_t(\eta,\sigma) = N_{[1,r]}(\eta_n,\sigma_n)$. Since the truncation is not symmetric with respect to the mean η_n, η may be either greater than η_n, when $\eta_n \leq (r-1)/2$ or less than η_n when otherwise. The standard deviation σ is always less than σ_n after the truncation. The difference between η and η_n is proportional to the magnitude of σ_n. For example, if $\sigma_n = \eta_n/4$ for η_n ranging from 32 to 256 with r=512, η_n becomes very close to η. By Fourier transformation, the convolution operator in Eq.10 can be replaced by multiplication in the frequency domain. With a given PE-demand distribution F, the n-fold convolution F_n can be calculated by using Discrete Fourier Transform (DFT) pair.

The effect on the utilization factors by different values of η is shown in Figs.3(a,b,c,d). We considered four different values of η ranging from 32 to 256 with $\sigma = \eta/4$. Note that when $\sigma = \eta/4$ then $\eta_n \doteqdot \eta$. Fig.3.a shows that for each η, ρ_{CU} increases when u increases. With sufficient large u, ρ_{CU} will become saturated (flat curves as shown). The smallest value of u leading to the saturated flat utilization is called the

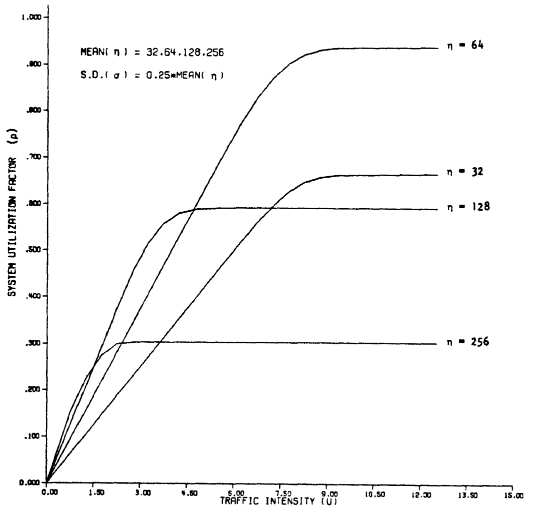
(a)



(b)



(c)



(d)

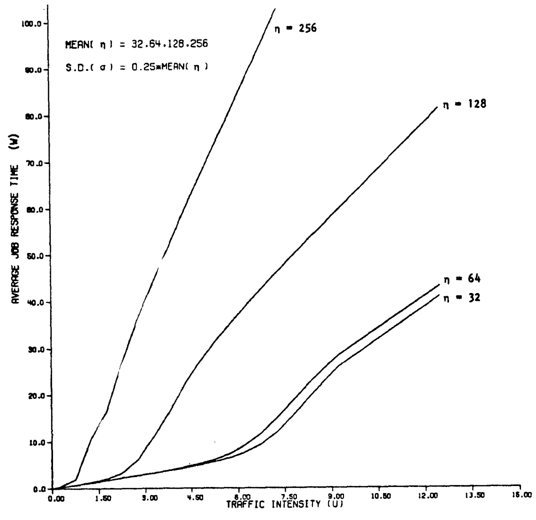


Fig. 3 The system performance measures, (a) ρ_{CU}, (b) ρ_{PE}, (c) ρ, (d) W, versus the traffic intensity u, for different values of distributions $N(\eta,\sigma)$.

saturation point u_s. From Eqs.18 and 19, we know that the saturation points for ρ_{PE} and ρ coincide with ρ_{CU}. Fig.3.a shows that the saturated ρ_{CU} decreases when η increases. This is obvious because, with small η, most of the CU's are allocated with enough PE's to perform the SIMD jobs. For the deterministic case ($\sigma = 0$), the number of active servers $\alpha = \min(m, r/\eta)$. The system can then be modeled as an $M/M/\alpha/\ell$ queue with $u_s = \alpha$. When $\sigma \neq 0$, because of the variation in PE demand, the saturated ρ_{CU} is less than the corresponding deterministic case. When σ is small, the saturation point u_s is close to the expected number of active servers α. For example, with $(\eta,\sigma) = (128,0)$, the saturated $\rho_{CU} = 0.50$ for ($\alpha = u_s = 4$); and with $(\eta,\sigma) = (128,32)$, the saturated $\rho_{CU} = 0.44$ for ($\alpha = 3.56$, $u_s = 3.9$). Poor CU utilization may be caused by high PE demand or by low traffic intensity.

Fig.3.b shows that ρ_{PE} increases when η increases for low intensity u. For saturated case, ρ_{PE} increases to maximum value and then drops down, when η increases. The maximal value on ρ_{PE} occurs near the point $\eta = \frac{r}{m}$. When $\eta << \frac{r}{m}$, there are still many idling PE's, even all CU's are busy. If $\eta >> \frac{r}{m}$, the number of idling PE's can hardly satisfy the large PE-demanding SIMD jobs. The poor PE utilization is primarily caused by low traffic intensity or η being far away from the ratio $\frac{r}{m}$.

The system utilization ρ, is the weighted sum of ρ_{CU} and ρ_{PE} as shown in Eq.7. Fig.3.c shows the system utilization ρ in terms of different η's. The system utilization is high only when both the ρ_{CU} and ρ_{PE} are high. The highest system utilization occurs at η=64 in the numeric example shown. When η approaches the value $\frac{r}{m}$ from either side of the curve, utilization ρ improves significantly as shown. Fig.3.d shows that the average job response time W increases with u. W increases also with η because the larger the PE-demand, the less the chance that the job will be allocated with enough number of PE's. The above results has been reinforced with extensive simulation results against the FCFC and the LDSF queueing disciplines as reported in [9].

VI. RESOURCE OPTIMIZATION METHODS

In this section, we present analytic methods to optimize the organization of resources in an MSIMD system with respect to given workload distributions and desired performance levels. Cost-effective machine organization is essential to parallel processing. The first problem deals with the determination of optimal number of PE's required in the pool given a fixed number of CU's and a known workload. The second problem decides the sufficient size of the finite-storage job queue (or the sufficient capacity of the buffer area).

A. Optimization of the Resource Pool of PE's

The optimal design of an MSIMD system should achieve high system utilization at low system cost. Methods to evaluate the total system utilization ρ has been presented in previous sections. The total system cost of an MSIMD system can be estimated as a quantity which is linearly proportional to the sum $m+\omega\cdot r$, where ω is the cost ratio of one PE (including the attached local memory and the crossbar switching overhead) to one CU (including all the attached memory and I/O facilities). Now we are ready to define a Resource Utilization and Cost Ratio (RUCR) using ρ defined in Eq.7.

$$\text{RUCR} = \frac{\rho}{\omega\cdot r+m} = \frac{(r\cdot\rho_{PE} + \eta\cdot m\cdot\rho_{CU})/(r + \eta\cdot m)}{\omega\cdot r+m} \qquad (23)$$

According to Eqs.17,18 and 19, both ρ_{PE} and ρ_{CU} are functions of six variables, (r,m,u,ℓ,η,σ). Therefore, we can write RUCR = RUCR $(r,m,u,\ell,\eta,\sigma,\omega)$ as a function of seven variables. Given fixed values of m,l,η,u,σ, and ω, one can plot the RUCR(r) as shown in Fig.4. The optimal size of the resource pool, say r_o PE's, is determined by finding the maximum value of RUCR(r),

$$\text{RUCR}(r_o) = \text{Max}\{\text{RUCR}(r)\,|\,r \geq 1\} \qquad (24)$$

What has demonstrated in Fig.4 corresponds to an MSIMD system with known parameters $(m,\ell,\eta,\sigma,\omega) = (8,20,64,16,1/16)$. A family of five curves were obtained for RUCR(r) with respect to five representative values of the traffic intensity u

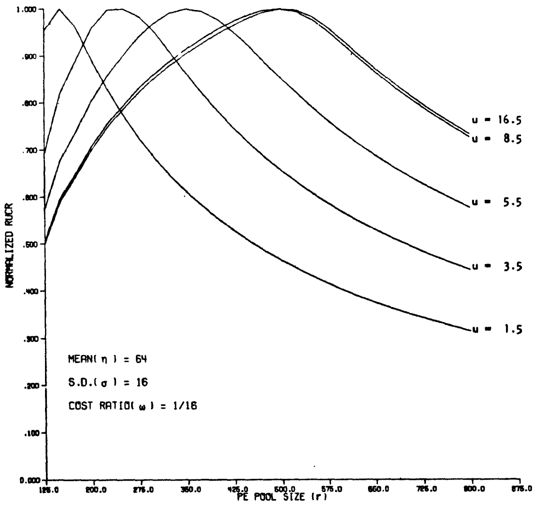


Fig. 4 The Resource-Utilization-and-Cost-Ratio (RUCR) versus the PE-pool size (r) for five different traffic intensities (u) with fixed ratio ω = 1/16.

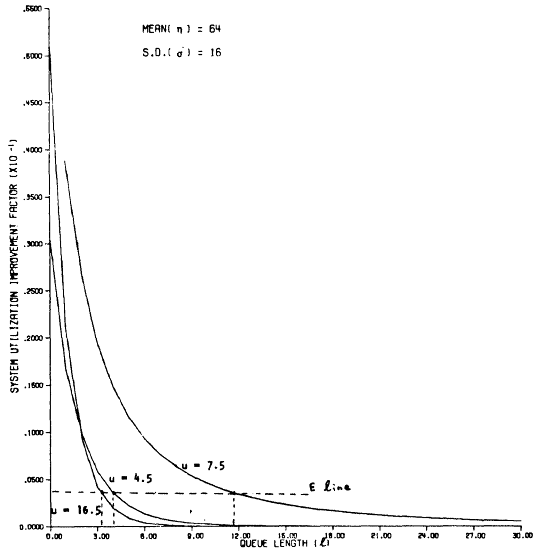


Fig. 5 The Relative-Utilization-Improvement-Factor (RUIF) versus the queue length (ℓ) for three different traffic intensities (u).

ranging from low (u = 1.5) to saturation (u $\geq$ 8.5). Normalized curves with respect to the maximal value of each RUCR(r) plot are shown. The optimal choice of r_o corresponds to the peak of each curve. These peak values of r_o in Fig.4 increases with increasing values of the traffic intensity u. However, once reaching saturation, the optimal size r_o becomes upper bounded. Obviously, the optimal choice of r_o is affected by the cost ratio ω used in Eq.23. With fixed traffic intensity u, the $r_o(\omega)$ decreases rapidly for small values of the cost ratio ω. The decreasing of $r_o(\omega)$ becomes much slower for cost ratios greater than 1/16.

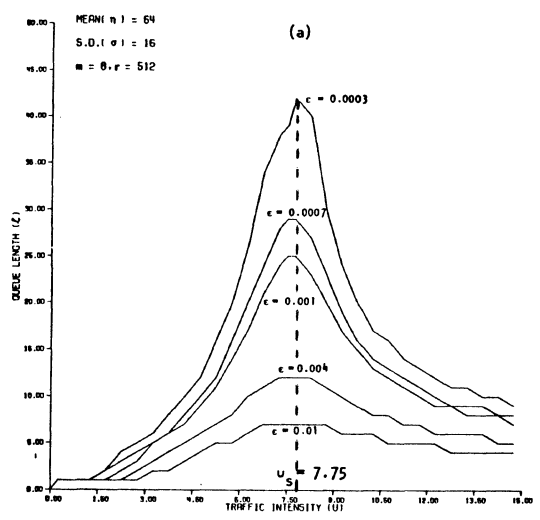
B. Determination of Sufficient Queue Length

With a finite-length queue, the new arriving jobs will be turned away when the queue is filled up to full capacity. In general, the longer the queue is, the better the resource utilization should be expected. However, for sufficient large queue length, such improvement may become negligible. In practice, most computers have finite buffer areas for the input queues. The effective queue length can be determined by using the following Relative Utilization Improvement Factor (RUIF).

$$\text{RUIF}(\ell) = \frac{\rho(\ell+1) - \rho(\ell)}{\rho(\ell)} \qquad (25)$$

, where the numerator represents the magnitude of utilization improvement in an MSIMD system with two consecutive queue lengths.

Given a very small number ε, the sufficient queue length can be determined by finding the smallest queue length, ℓ_s, such that $\text{RUIF}(\ell_s) \leq \varepsilon$. In other words, $\text{RUIF}(\ell_s-k) > \varepsilon$ for all $k \geq 1$. The family of RUIF(ℓ) curves plotted in Fig.5 shows that the relative improvement RUIF(ℓ) decreases rapidly with increasing queue length. If one draw a horizontal line in Fig.5, say an ε-line, the sufficient queue length, ℓ_s, for various traffic intensities are immediately revealed at all the intersections points. When the intensity u approaches saturation, the sufficient queue length turns out to be upper bounded by a constant.



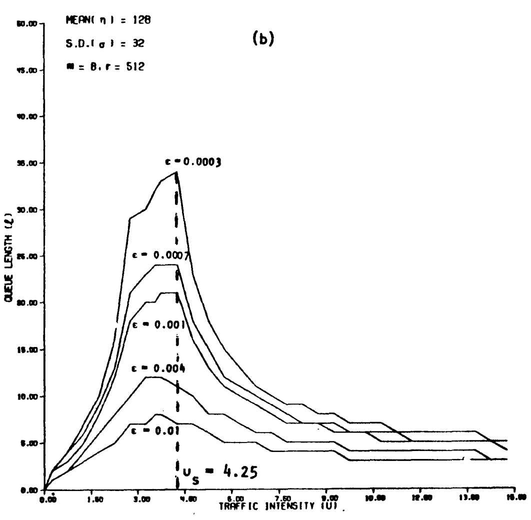


Fig. 6 The sufficient queue length (ℓ_s) versus the traffic intensity (u) for five RUIF bounds (ε). (a) $(n,\sigma) = (64,16)$ and $u_s = 7.75$ (b) $(n,\sigma) = (128,32)$ and $u_s = 4.25$

This upper bound can be further illustrated by the curves of sufficient-queue-length (ℓ_s) versus the traffic-intensity (u) shown in Fig.6. Five curves are shown corresponding to five representative ϵ values. The peak of each ϵ-curve marks the maximum sufficient queue length, ℓ_m required, that is $\ell_s(u) \leq \ell_m$ for all u. The larger the ϵ is valued, the shorter the maximal queue length is required. Theoretically speaking, the peak approach infinity when ϵ goes to zero. With reasonably small ϵ, the ℓ_m is still finite as shown. The system parameters given in Part (a) and Part (b) of Fig.6 are essentially the same except with different η of PE-demands. It is interesting to point out that the maximum sufficient queue lengths $L_m(\epsilon)$ for various ϵ values occur at the same traffice intensity, when u approaches the saturation point u_s. With two different pairs of $(\eta,\sigma) = (64,16)$, and $(128,32)$ in Parts (a) and (b), respectively, the maximal sufficient queue lengths are determined at the saturation points $u_s = 7.75$ and $u_s = 4.25$ in Fig.6 respectively. Note that the sufficient queue lengths decreases rapidly on both sides of the critical traffic intensities which produce all the peak values. With small u, there is no need to use a long queue, because the queue can hardly filled up. When u is very large even with a short queue, the queue is hardly emptied. Thus, the server can always get a job from the queue. Theoretically, ℓ can be zero when u goes infinity. The above procedures can be used to determine both the maximal sufficient queue length (the peaks) and the sufficient queue length for any given traffic intensity. However, only the maximal sufficient queue length will be used in practical design problems.

VII. CONCLUDING REMARKS

We have demonstrated how to use the proposed queueing model for evaluating the performance of shared-resource MSIMD computer organizations. The utilization factors for CU's, PE's and the system and the average job response time are used to estimate the system throughput with respect to a given workload distribution. Our analytic results will aid the designers of MSIMD system to optimize the size of the PE resource pool and to determine the sufficient queue length. Direct simulation study of MSIMD machines for parallel vector processing has been given in Ref. [9].

The CU and PE utilization factors can be further upgraded if multiprogramming is built into each CU and each PE. Queueing discipline other than FCFS and LDSF can be also considered, such as the Shortest-Job-Serve-First (SJSF) discipline. The system degradation due to access conflicts and I/O overhead were not considered in our study. With sufficient large local memories in CU's and in PE's, the influence due to page faults or memory access conflicts should have some effect on our results but not significantly. An alternative approach using two-dimensional Markov chains for the analysis of MSIMD machine is currently under further research by the authors for parallel vector processing in multiprocessor systems.

REFERENCES

[1] Barnes, G. H. et al., "The Illiac IV Computer", IEEE Trans. Comput., Vol. C-17, pp. 746-757, Aug. 1968.

[2] Briggs, F.A., Fu, K.S., Hwang, K. and Patel, J., "PM^4: A Reconfigurable Multiprocessor System for Pattern Recognition and Image Processing," Proc. of NCC, AFIPS, pp. 255-266, June 1979.

[3] Chandy, K.M. and Sauer, C.H., "Approximate Methods for Analyzing Queueing Network Models of Computer Systems," Computing Surveys, Vol. 10, No. 3, pp. 281-317, Sept. 1978.

[4] Chow, Y-C, and W. H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," IEEE Trans. Comp., Vol. C-28, No. 5, May 1979, pp. 354-361.

[5] Hwang, K. and Ni, L.M., "Modeling and Analysis of Multiple SIMD/SISD Computer System," Proc. of the Third International Computer Symposium, Taiwan, China, Dec. 1978.

[6] Kleinrock, L., Queueing System, Vol. 1: Theory, John Wiley & Sons, 1975.

[7] Lipsky, L. and Church, J. D., "Applications of a Queueing Network Model for a Computer System," Computing Surveys, Vol. 9, No. 3, Sept. 1977, pp. 205-221.

[8] Little, J.D.C., "A Proof of the Queueing Formula: $L = \lambda W$," Operations Res. 9(3), pp. 383-387, 1961.

[9] Ni, L.M. and Hwang, K., "Modeling and Analysis of Shared-Resource Multiple SIMD Computer Organization," Tech. Report TR-EE-79-22, Purdue University, W. Lafayette, Indiana 47907, May 1979.

[10] Nutt, G.J., "Memory and Bus Conflict in an Array Processor," IEEE Trans. Comput., Vol. C-26, pp. 514-521, June 1977.

[11] Nutt, G.J., "The Architecture of a Multi Associative Processor," Univ. Colorado, Dept. Comput. Sci., Tech. Rep. CU-CS-070-75, June 1975.

[12] Nutt, G.J., "A Parallel Processing Operating System Comparison," IEEE Trans. SE. Vol. SE-3, No. 6, pp. 467-475, Nov. 1977.

[13] Radoy, C.H. and Lipovski, G.J., "Switched Multiple Instruction, Multiple Data Stream Processing," Proc. 2nd Annu. Symp. Computer Architecture, pp. 183-187, 1974.

[14] Regis, R. C., "Multiserver Queueing Models of Multiprocessing System," IEEE Trans. Comput., Vol. C-22, No. 8, Aut. 1973, pp. 736-745.

[15] Senzig, D. N., "Observation on High-Performance Machines," Proc. of FJCC, Vol. 31, 1967, pp. 791-799.

COMPUTATION STRUCTURES REFLECTED IN GENERAL PURPOSE AND SPECIAL PURPOSE MULTI-MICROPROCESSOR SYSTEMS

W. Händler, H. Schreiber, and V. Sigmund
Universität Erlangen-Nürnberg
Institut für Mathematische Maschinen
und Datenverarbeitung III
Martensstraße 3, 8520 Erlangen
Federal Republic of Germany

Abstract -- Recent multi-microprocessor projects have included general purpose as well as special purpose systems. Both types achieve high performance by using parallelism. But the design considerations of the structure of general and spec. purpose computers are yet closer related. Even if we take analog devices as the oldest special purpose machines into account, the essential structure of computations performed by both machine types is the same. This results from the most natural consideration of computations as motions in real space and time. Two computer architecture projects at the University of Erlangen-Nürnberg are described and discussed from this point of view: a general purpose parallel/associative hierarchical array of processors, EGPA [1], and a system of plug-in computer modules for special application space-sharing systems, DIRMU, both currently under construction.

1. Introduction

There are a number of recent projects in computer architecture whose common origin is the progress in the technology of large-scale integrated semiconductor devices of the 1970's. The designers try to connect tens, hundreds or even thousands of processors and other computer modules into single high-performance systems. Some of these systems are intended to be general purpose computers, others for special applications.

We are working on two such projects at the Institut für Mathematische Maschinen und Datenverarbeitung of the University of Erlangen-Nürnberg: a general purpose parallel/associative hierarchical array of processors EGPA (Erlanger General Purpose Array) [1], [2], and a system of plug-in computer modules for special application systems, DIRMU (Desinvolvimento e Implementação de Redes de Multiprocessadores) [3], [4]. The latter is a joint project with the Universidade Estadual de Campinas, S.P., Brazil.

For both general purpose and special purpose computers, an extensive use of parallelism is the usual way to achieve high performance at a given stage of circuit technology. But special purpose machines have evolved from a quite different starting point than general purpose computers. The oldest special purpose machines are analog devices. Are there still other common structural features between analog and general purpose digital computers in addition to the (occasional) use of parallelism? It is interesting to read a fifteen or twenty years old book on analog computers and to see that these were not only highly parallel but also, quite naturally, "data driven".

[1] supported by the Bundesministerium für Forschung u. Technologie (German Federal Ministry of Research and Technology), contract No. DV 4.906-081207O

If we consider computations as motions in real space and time - as explained in section 2 - we see that the traditional differentiation between the infinite data sets and continuous functions executed by analog function units on one hand, and the finite data sets and discrete functions executed by digital function units on the other hand is not as significant as the fact that the essential structure of computations performed by both machine types is the same. The conceptual barrier between analog and general-purpose digital computers loses importance with the steady progress of large-scale integration and the corresponding shift from time-sharing to space-sharing systems, where processing space is shared between tasks executed at the same time. This becomes evident if we compare the automatic patching of analog function units in hybrid computers (cf. Mawson [5]) and the dynamic reconfigurable or varistructured digital computers suggested recently (cf. Miller and Cocke [6], Arnold and Page [7], Lipovski [8], Lipovski and Tripathi [9], Kartashev and Kartashev [10]). Multi-microprocessor DDA-systems (digital differential analyzers) dedicated to space-sharing solution of differential equations (cf. Korn [11], Kempken and Ameling [12]) are another example.

In sections 3 and 4 we briefly describe our computer architecture projects EGPA and DIRMU and discuss the design decisions related to the structure of computations. The present EGPA-system consists of five microprogrammable 32-bit processors AEG 80-60 configured in a way that allows multiple modes of parallel and associative processing. The pilot implementation of DIRMU modules uses the 16-bit microprocessors Intel 8086 and multiport memories. The first application is a space-sharing configuration for a learning classification process used in pattern recognition systems (cf. Rohrer [13]).

2. The structure of computations

To compute means to obtain for some arguments x the appropriate results y:

$$x \longmapsto y,$$

or, otherwise stated, to execute some function (operation, map) $f : X \rightarrow Y : x \mapsto y$. If we happen to find some natural process executing this function in real space and time, we are finished. This is a very simple case of analog computation.

Examples of such processes are motions of wheels and levers in mechanical calculators and analog devices, or motions of electrical signals in operation units of an analog computer. We note that the execution of a boolean operation such as AND : $\{0,1\} \times \{0,1\} \rightarrow \{0,1\}$ in switching circuits of a digital computer also relies on the physical analogue of motions of electrical signals through the corresponding gate. This is only a special case of analog computation where the argument and the value set are very small.

If we do not find any natural analogue for the execution of our desired function f we have to search for other functions executed in an analogue manner which, appropriately composited, give us the function f. Fig. 1 shows an example of the general case of compound functions.

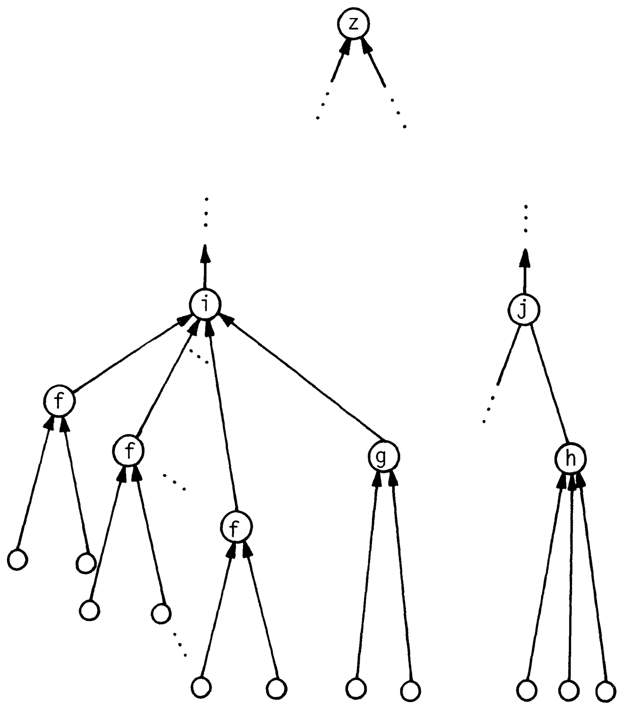
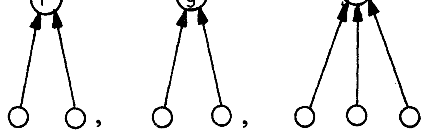


Figure 1: Example of compound functions

Directed graphs such as Fig. 1 are mathematical tools which help us to see what happens when we compute. Compound functions are well known in mathematics. One can prove that all graphs of the type shown in Fig. 1 can be composited of the "primitive" graphs such as



by using only two sorts of composition: parallel (essentially putting several graphs "horizontally" next to each other, such as "g" and "h", or as the multiple copies of "f" in Fig. 1), and in series (essentially putting two graphs one above another and connecting the nodes). The related mathematical structures are called "algebraic theories", cf. Lawvere [14], or Arbib and Give'on [15] (especially Theorem 2.4, p. 340), [16]; for the relevance of algebraic theories for the whole of computer science cf. for example Goguen, Thatcher, Wagner and Wright [17]. In [18], [19], a calculus of expressions for the classification of computer architectures (ECS: Erlanger Classification Scheme) has been proposed that reveals the same structure (cf. also a companion paper [20]).

Now, speaking about the structure of computations we mean first of all precisely the directed graphs such as Fig. 1. If we agree that "to compute" means to execute some functions, then any computation whatever, analog or digital, can be represented by directed graphs such as Fig. 1. This structure occurs at several levels in a given application. At the switching circuit level, the operations marking the nodes of Fig. 1 would be AND, OR, NOT, NAND, etc.; at the instruction level and the subroutine level each operation actually presents a composition and union of operations of the level one below, and so on. We call this the successive interpretation (cf. [21]).

The extent to which the structure of compound functions is spatially reflected in real machines differs from level to level, and also from one specific machine to another since the "computation space" is related to the hardware cost. Fully spatial expansion appears often at the switching circuit level, where we get combinational circuits as a direct isomorphic image of the corresponding graph Fig. 1. At the instruction level, the classical Princeton-type computer expands in space only one "programmable" node of Fig. 1 in its ALU, where the contents of the instruction register defines the performed operation f, g, h, ... The execution of compound operations has to be composited in the time coordinate. In contrast to this, Illiac IV, Cyber 76, and others expand at least certain special cases of compound operations spatially, such as cartesian power ("SIMD-machines") or cartesian product ("MIMD-machines"). Some more recent attempts in this direction are seen in the "data-flow" or "data-driven" and "single assignment" computers (cf. Dennis and Misunas [22], Rumbaugh [23], Arwind and Gostellov [24], Plas et al. [25]). Operation units of an analog computer are connected in a way that fully reflects the structure of compound operations in space.

We can generally speak of the space/time trade-off in all machines. Parallel, (instruction-) pipeline, data-flow, single-assignment and similar computers use more computation space (i.e. operation units) at the instruction level than conventional computers, in order to reduce computation time. Arwind and Gostellov explicitly speak of exchanging processors for time [24]. This is also a quite usual trade-off in analog computers (cf. Adler and Neidhold [26], Chapter 2.2.2). At the bit-operation level of digital computers, a well-known example from the beginner's logic design course is the parallel adder that requires more computation space but less computation time than the sequential adder. More advanced examples are arithmetic pipelines in the Cray 1 or Star 100 and its successors Cyber 203, 205, where the structure of compound operations such as floating addition, floating multiplication is fully expanded in space. Similar space/time trade-offs apply not only to hard-machines, but to soft-machines (programs), too. Any experienced programmer knows that, in general, to make a program faster costs additional program lines, and vice versa.

We note that the above considerations of computations and their structure are not at all new. In fact, they are so old and simple that some researchers seem to ignore them, in speaking of actors, tokens and firing.

3. General purpose space-sharing systems: EGPA

Our considerations regarding the structure of computations hold also at the program level of the software and the corresponding PMS level of the hardware (cf. Bell and Newell [27]), as well as the related space/time trade-off. The PMS level of computer design became especially attractive in the 1970's, as the number of multi-microprocessor projects shows. We discuss the main design considerations of our projects at this level in the following.

Fig. 1 here shows the operations (tasks) f,g, h, ... defined by different user program modules and their composition while they are executed. The usual single-processor general purpose computer essentially reflects only one "programmable" node of Fig. 1 in its processor, if we neglect I/O. The execution of compound operations such as Fig. 1 has to be composited in the time coordinate; one speaks of time-sharing. By space-sharing we mean the case where the execution of compound operations is spatially expanded in the machine, as mentioned in the last section. In [28], the notion "macro-pipelining" was used in a similar context, with a general (non-linear) pipeline ("data-flow") at the PMS-level. Parallelism and linear pipelining at the PMS-level are only the expansion of special cases of compound operations in processing space: the castesian power or product, and the function composition f $\circ$ g (f followed by g).

The main problem in the use of space-sharing in general purpose systems is that the structure of compound operations in user programs changes from application to application. One possible solution is to provide for flexible interconnections between processors that can change dynamically. This is the approach taken in various projects of dynamically reconfigurable systems. The well-known drawback is the price of the interconnection structure and its complexity which grows with the square of the number of processors. The complexity again causes additional problems in the computer organization.

An alternative solution would reflect some approximation to the "universal" structure of compound operations in fixed interconnections, and then provide for dynamic projections of the actual application structures into the fixed structure. Fig. 1 suggests that local interconnections between processors should be sufficient in the most cases. The structure of the pool of locally interconnected processors should take into account that cartesian power composition of operations - i.e. several occurences of the same operation to be executed on different data elements ordered in arrays - appear in many applications. The dynamic projection of the applications into the processing space requires control overhead. To be consequent in space-sharing, the processing capacity for this task should be provided by extra processors.

Fig. 2 shows the EGPA configuration suggested in [2] that satisfies these requirements. A

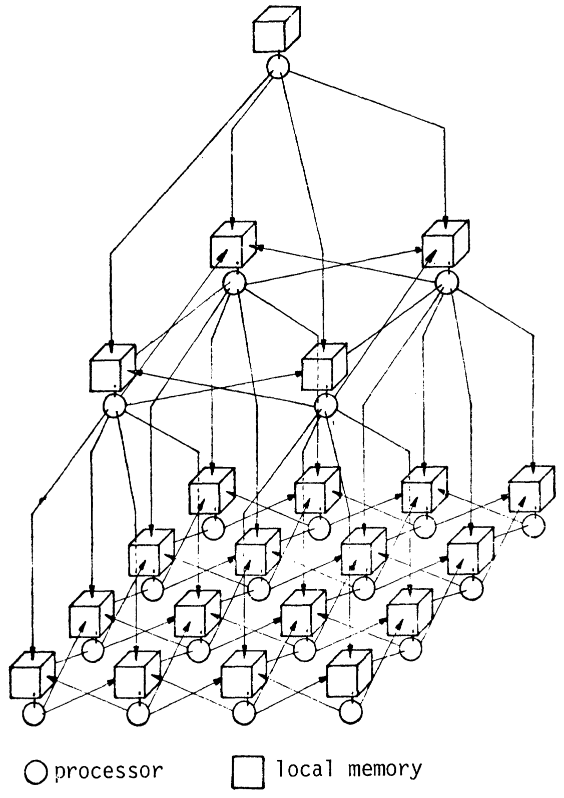


Figure 2: EGPA configuration

regular array of processors, each having its local memory block, is interconnected through multiport memory access from direct neighbours. The memory blocks store application programs for the processors and also serve as buffers for data flow which is implemented simply by exchanging of addresses between neigbours. The pyramid over this array is the space-sharing implementation of the control hierarchy. Local interconnections through multiport memories are well suited here, too, so that the whole system is arbitrarily expandible. There is no need for special interconnection modules, all main components are homogeneous and standard.

This configuration allows the following three processing modes in the array controlled by the pyramid:

- normal multiprocessing of independent tasks
- cartesian power processing of (multiple copies of) a task ("PMS-parallel")
- general space-sharing processing projected into the array ("general PMS-pipeline")

The use of microprogrammable processors and specific microprogrammed instructions for "vertical processing" - a method of implementation of associative processing on conventional hardware (cf. [29], [30], [31]) - allows a further three modes of parallel associative processing which are combinations of the above modes with vertical processing (cf [2]).

The present EGPA system built in cooperation with AEG-Telefunken Konstanz consists of the "smallest pyramid", i.e. of a four processor array and a fifth processor as the control. The aim of this pilot implementation is to prove the viability of the concept and to gain experience with space-sharing systems across the related disciplines of computer science, as the following list of some design and implementation subactivities of the pilot project shows:

hardware:

1) interprocessor interrupts
2) support I/O-module for vertical processing
3) hardware measurement monitor

firmware:

4) instructions for vertical-associative processing

operating systems:

5) multiprocessing time-sharing operating system for the pilot system

performance analysis:

6) measurements and analysis of system performance (cf. companion paper [32]).

programming languages:

7) Fortran-extension for vertical-associative processing

application software:

8) analysis of application algorithms with regard to their suitability for EGPA processing modes
9) implementation of selected applications

The pilot system uses the microprogrammable 32-bit processors AEG 80-60. The local memories of the processors of the array consist of 64K bytes each, and the local memory of the control processor has 256K bytes capacity. The system is due to be operable in March 1980.

4. Special purpose space-sharing systems: DIRMU

Considerations regarding the structure of computations and the related space/time trade-off (cf. section 2) at the PMS-level of computer design are yet more fruitful in the case of special applications. A machine is dedicated to a particular computation structure, so that high-performance special-purpose computers have been able since the 1960's to expand this structure spatially to the extent that it was economically feasible (cf. Enslow [33]). The economic considerations have changed rapidly with the advent of LSI. Let us assume a user with a particular application, where a directed graph such as Fig. 1 shows the composition of operations (tasks) defined by user program modules. As an alternative to the use of an (expensive) general purpose computer, where the program modules would be executed in the time-sharing mode, the user can now purchase cheap microprocessor, memory, and support chips, join them together in a way described by the graph of compound operations, and execute the tasks in the space-sharing mode.

However, the design of a large special-purpose system using LSI components is a fairly complicated and time-consuming task that has been performed only by teams of specialists from computer manufacturers or universities. Even conventional microprocessor system development for simple control applications today requires knowledge and experience in electronics, computer organization, and machine language programming from the designer, in addition to the knowledge of the desired application. If we want to make the above alternative directly available to the user, then it should not be more difficult to configure his space-sharing system than to link program segments together in a time-sharing environment. Another comparable task is the configuring of analog operating units at the patch-board of an analog computer. We remember that all computations, digital as well as analog, possess the same essential structure.

Thus the aim of the DIRMU project is to design a system kit of plug-in LSI moduls for configuring user-definable high performance special purpose systems [3]. The principal shape of the DIRMU module should be no surprise after the above explanation. It corresponds to a single node of a graph such as Fig. 1. Fig. 3 shows a simplified notation for the module consisting of a processing unit, local memory for program storage and data buffering, and connections to other nodes. A complete space-sharing DIRMU system for pattern recognition, our first application described farther below, is shown in Fig. 4.

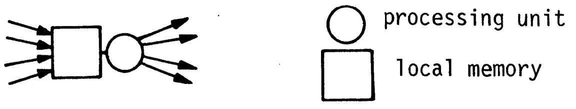


Figure 3: DIRMU module

At this place we relate the DIRMU project to other similar projects. The micromodules of Cooper [34] should decrease the fabrication and debugging costs of logic circuits in the design of parallel and pipeline computers. Computer modules of Fuller, Siewiorek and Swan [35], and of Kartashev and Kartashev [10], as well as commercially available bit-slice microprocessors must appeal primarily to

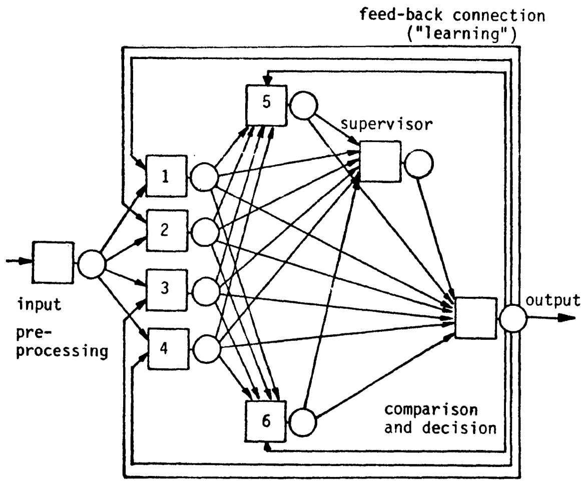


Figure 4: Example of automatic classification (pattern recognition)

computer architects. The main objective of the DIRMU modules is their easy configurability, in order to aid the end user directly.

Architecture of DIRMU modules.

Considerations of desired system properties have led to the following requirements for the modules [3]:

- microprogrammable processor, 16-bit wordlength, interrupt capability
- microprogram memory consisting of a read-only part for the implementation of the basic instruction set, and a read/write part for the implementation of special-purpose user definable instructions
- local working memory
- provisions for easy reconfigurability of the system, e.g. through the use of multiport memory access for local working memory of each module
- provisions for connection of two modules to form a single module of wordlength 32 bit, if an enhanced processing power of some modules of the system is required (similarly to bit-slice microprocessors)
- each module having I/O-ports for communication with the outside world.

The microprogrammability will make modules more flexible for adaptation to special processing modes (e.g. non-numeric processing, associative processing, cf. [29], [30], [31]).

Pilot implementation of the modules.

In order to gain initial experience and results as soon as possible, it was decided to build pilot modules using standard microprocessor components. The 16-bit microprocessor Intel 8086 seemed to be the best momentarily achievable compromise. It receives good hardware and software support from the manufacturers and offers multiprocessor capabilities. Microprogrammability and some other features from the original catalogue of requirements for the module had to be postponed.

One of the most important properties of our pilot modules had to be an easy configurability of their internal structure, because of their experimental character. We have therefore chosen the system design kit Intel SDK-86 as the basic building element for our modules. It offers the microprocessor 8086 together with all support circuits needed for the "minimum-mode" system configuration (Intel) including 2K bytes RAM, parallel and serial I/O-interfaces, hexadecimal keypad and display, 8K bytes ROM with system monitor for keypad/display or serial-I/O machine code programming (teletype or CRT) and for serial program loading. A large area of the board is left for user's custom extensions.

The structure of our module is shown in Fig. 5. It is a dual bus system with a processor bus controlled by the CPU and a memory bus controlled by a specifically designed multiport controller.

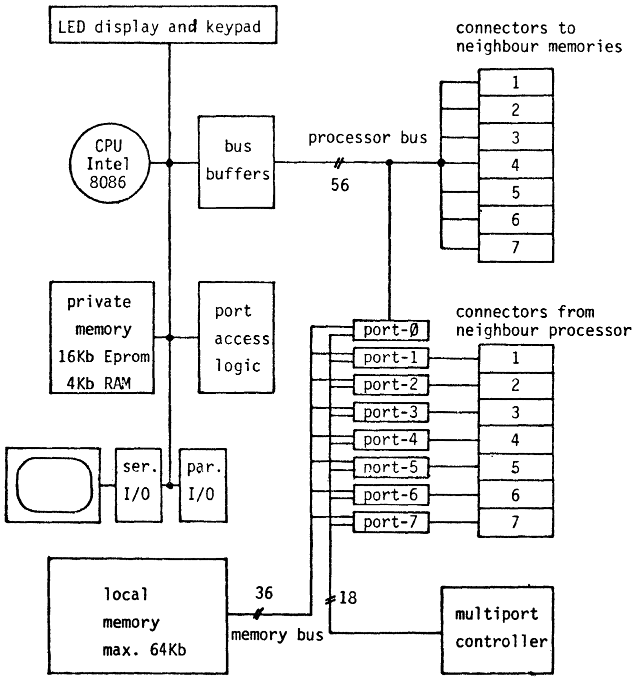


Figure 5: pilot DIRMU module

Processor part.

The processor part consists of the 8086 with the clock and a modified wait-state generator of the SDK-86. The control bus has been extended through the use of the bus controller Intel 8288 to the "maximum-mode" bus (Intel) that is necessary for multiprocessor operation. The module's private memory consists of 16K bytes EPROM and 4K bytes RAM for systm initialisation and monitor routines and tables. The original keypad and display of the SDK-86 serve as a useful maintenance console of the pilot module for the inspection of register and memory contents, and for single-step execution during system testing and debugging. A CRT display terminal can be connected through the serial I/O-port to any module for the same purpose. Only the latter feature will be provided in the final DIRMU modules. The three parallel I/O-ports of the SDK-86 remain without changes. They allow interfacing a module to a mass storage device or a host computer.

The processor requests access to its local working memory as well as to the working memories of its neighbours via bus buffers. The additional control signals needed to control the memory access that are not available in the 8086, are generated within the port-access control logic. These are the handshake signals REQ-i, RDY-i and ETRANS necessary to coordinate and synchronize memory access via the port i. The LOCK signal of the 8086 to the multiport controller is needed when consecutive memory cycles are requested. 32 lines of the processor bus are used for separate 16-bit data and addresses. The port signals REQ-i are generated by decoding the uppermost four address bits of the 8086. Only eight access ports are used at present.

Memory part.

A multiport read/write memory with eight memory access ports is used as a local working memory of each module. The port number zero serves for the access of the corresponding local processor, ports number 1 to 7 are used for the connections to the neighbour modules as required by the operation structure of the application (cf. section 2). The working memory stores the application program code defining the operation of the module and also serves for data exchange. Word (16 bit) or byte data transmissions through the selected port and the corresponding memory bus is possible for data exchange. The access is controlled on a request/grant basis by a specifically designed multiport controller.

ting processors are forced into 8086 wait states. The port access logic of the served processor answers with an end-of-transfer signal ETRANS as soon as the memory access of the processor has finished, and the multiport controller grants memory access to the next processor selected by the priority arbitration logic. If the request has been accompanied by the LOCK signal of the processor - which can be forced by the corresponding prefix in front of any 8086 instruction-, consecutive memory accesses are granted to this processor as long as its LOCK is active. The priority arbitration logic serves multiple simultaneous requests on the round-robin strategy. If only one access at a time is requested, the multiport access control remains invisible for the accessing processor.

The 1M byte address space of the 8086 suggests the partitioning in 16 blocks of 64K bytes each, where one block is used for the processor's private EPROM and RAM and up to 15 blocks can be used for the working memories of the modules accessible by one processor, including its local memory. In our pilot implementation only 8 blocks of working memories are accessible by one processor leaving nearly half the address space unused. Any working memory of maximally 64K bytes capacity can be accessed through one of its eight memory ports by different processors. The 20 bit addressing capacity would allow modified configurations with local memories up to 128K bytes each or with increased number of access ports of each processor.

Programming considerations.

We have called the local read/write memory block of a DIRMU module as shown in Figures 3 and 4 (and accessed through the port Ø in Fig. 5) the working memory of the module, since it serves not only as a module's buffer for data flow through the system but also for the storage of the application program code defining the operation of the module. This program code can be loaded by a serial loader program residing in the private EPROM of the processor through the serial I/O-port.

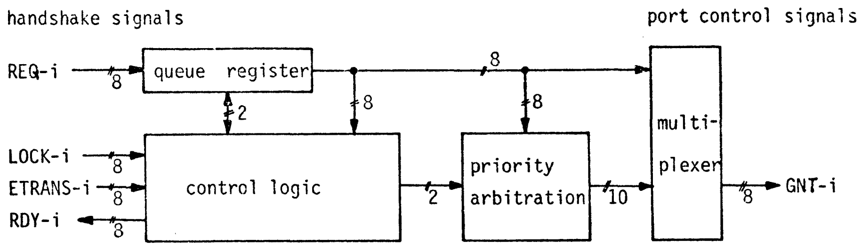


Figure 6: block structure of the multiport controller

Fig. 6 shows the block structure of the controller. Multiple simultaneous requests to memory are saved in a queue register. The control logic activates the priority arbitration part that decides which request will be granted. The grant signal GNT-i of the controller activates the corresponding memory port (cf. Fig. 5) and at the same time a ready signal RDY-i is sent to the processor that receives memory access. Other reques-

Alternatively, a space-shared bootstrapping could be applied, where the application programs of all modules are loaded through the fixed system input (cf. Fig. 4) and spread over the system. If several modules perform the same operation (task) (e. g. moduled 1,2,3 and 4 in Fig. 4), they can use reentrant code located in their common direct successor.

A draw-back of the above solution is that instruction fetch of a processor has to compete with the memory access of the neighbour modules to its local working memory. The port access logic of the processor part and the multiport controller of the module cause a delay that will require at least one 8086 wait state even in case of a single access request at a time. We consider therefore also the solution in which a larger private memory of the processor stores the application code. EPROMs containing the code could be directly plugged into the processor board.

Applications and Outlook.

The DIRMU system is aimed as a modular system kit for user specified space-sharing implementation of any application. The system configurability at the PMS level suggests first of all such applications where a clean operation structure at the corresponding software level can be recognized easily (cf. section 2).

Fig. 4 shows a space-sharing configuration for a learging classification process used in pattern recognition systems (cf. Rohrer [13]). It consists of pattern input and preprocessing followed by four parallel classifiers 1,2,3 and 4. Their intemediate results are passed to a further classifier pair 5,6. Measurements of distances between given templates and the submitted pattern allow comparison and decision making for a possible next interative step in which case a learning feedback effect on the classifiers applies. These phases are performed and controlled by the next two modules in Fig. 4. The model has already been programmed and tested on a PDP 15 and found suitable for a space-sharing implementation [13], [3].

The pilot DIRMU system for this application is being built at our institute. Other applications include the feedback solution of ordinary differential equations (cf. Korn [11]) and the relaxation method for the solution of partial differential equatinos. In parallel to this pilot DIRMU project, design of an ultimate LSI DIRMU-module has begun. The following features are considered in addition to the design objectives mentioned at the beginning of this section:

- special instructions for data exchange such as broadcast and collect as well as instructions for processor intercommunications
- module synchronization/communication phase as a part of the basic system cycle containing the usual instruction fetch/execute cycle of each processor
- failure tolerant system provisions.

Acknowledgement.

We are obliged to Mr. R.K. Bell for valuable advice and useful comments on this paper. We thank the referees for suggestions that helped to improve the final version.

References

[1] Händler, W., "A unified associative and von-Neumann processor EGPP and EGPP-array", in Feng T. (ed.) Parallel Processing, Proc. of the Sagamore Computer Conference 1974, Lect. Notes in Comp. Science 24, Springer, Berlin (1975), pp. 97-99

[2] Händler, W., Hofmann, F., Schneider, H.J., "A general purpose array with a broad spectrum of application", in Händler, W. (ed.), Computer Architecture, Workshop of the GI, Erlangen, May 1975, Springer, Berlin (1976), pp. 311-335

[3] Händler, W. and Rohrer, H., "Gedanken zu einem Rechner-Baukasten-System", to appear in Elektronische Rechenanlagen (1979)

[4] Schreiber, H. and Sigmund, V., "Functionally structured computer architecture using plug-in microprocessing modules", Proc. of the Intern. Microcomputers, Minicomputers, Microprocessors '79 Conference, Geneve (June 1979)

[5] Mawson, J.B., "Pacer 600 autopatch computing system", in Feilmeier, M. (ed.), Parallel Computers - Parallel Mathematics, Proc. of the IMACS-GI-Symposium, March 1977, Munich, North-Holland, Amsterdam (1977), pp. 147-160

[6] Miller, R.E. and Cocke, J., "Configurable computers: A new class of general purpose machines", in Ershov, A.P. and Nepomniaschy, V.A. (eds.), Internat. Symposium on Theoretical Programming (1972), Lecture Notes in Computer Science 5, Springer, Berlin (1974), pp. 285-298

[7] Arnold, R.G. and Page, E.W., "A hierarchical, restructurable multi-microprocessor architecture", Proc. of the 3rd Annual Symposium on Computer Architecture, IEEE, New York (1975)

[8] Lipovski, G.J., "On a varistructured array of microprocessors", IEEE Trans. Computers C-26 (1977), pp. 125-138

[9] Lipovski, G.J., Tripathi, A., "A reconfigurable varistructured array processor", Proc. of the 1977 Internat. Conf. on Parallel Processing, IEEE, New York (1977), pp. 165-174

[10] Kartashev, S.I., Kartashev, S.P., " A microprocessor with modular control as a universal building block for complex computers", in Microcomputer Architecture, Euromicro Symp., North-Holland (1977), pp. 85-91

[11] Korn, G., "Back to parallel computations: Proposal for a completely new on-line simulation system using standard minicomputers for low-cost multiprocessing", Simulation (Aug. 1972), pp. 37-45

[12] Kempken, E., Ameling, W., "Parallel DDA based on universal microprogrammable computing units", cf. [5], pp. 139-142

[13] Rohrer, H., A supervised network on adaptive automata for pattern recognition, Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, (1975)/8, pp. 1-41

[14] Lawvere, F.W., "Functional semantics of algebraic theories", Proc. Nat. Acad. Sci. (USA), 50 (1963), pp. 869-872

[15] Arbib, M.A. and Give'on, Y., "Algebra automata I: Parallel Programming as a prolegomena to the categorical approach", Information and Control 12 (1968), pp. 331-345

[16] Give'on Y. and Arbib, M.A., "Algebra automata II: The categorical framework for dynamic analysis", Information and Control 12 (1968), pp. 346-370

[17] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B., A junction between computer science and category theory, IBM Research Reports, Yorktown Heights (1973)

[18] Händler, W., "On classification schemes for computers in the post-von-Neumann-era", in: Siefkes, D. (ed.), GI-4. Jahrestagung, Berlin, Oct. 1974, Lect. Notes in Comp. Science 24, Springer Verlag, Berlin (1975), pp. 439-452

[19] Händler, W., "Impact of classification schemes on computer architecture", cf. [9], pp. 7-15

[20] Händler, W., Sigmund, V., "Complexity measures of computer structures", this conference.

[21] Sigmund, V., "Parallel compiled interpretation", cf. [9], pp. 16-25

[22] Dennis, J.B., Misunas, D.P., "A preliminary architecture for a basic data-flow processor", Proc. of the 2nd Annual Symposium on Computer Architecture, IEEE, New York (1974), pp. 126-132

[23] Rumbaugh, J., "A data flow multiprocessor", Proc. of the 1975 Sagamore Comp. Conf. on Parallel Proc., IEEE, New York (1975), pp. 220-223

[24] Arwind and Gostellov, K.P., "A computer capable of exchanging processors for time", in Gilchrist, B. (ed.), Information Processing 77, North-Holland, Amsterdam (1977), pp. 849-853

[25] Plas, A., et al., "LAU system architecture: a parallel data-driven processor based on single assignment", Proc. of the 1976 Intern. Conf. on Parallel Processing, IEEE, New York (1976), pp. 293-302

[26] Adler, H., Neidhold, G., Elektronische Analog- und Hybridrechner, VEB Deutscher Verlag der Wissenschaften, Berlin (1974)

[27] Bell, C.G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, New York (1971)

[28] Händler, W., "The concept of macro-pipelining with high availability", Elektr. Rechenanlagen 15 (1973), pp. 269-274

[29] Händler, W., Rechenwerk einer digitalen Rechenmaschine, Patentschrift 1157009, Telefunken Patentverwertungsgesellschaft mbH, Ulm/Donau (Sept. 1961)

[30] Händler, W., Prozessor mit Mikroprogrammsteuerung einer digitalen Rechenanlage , Patent No. P2419241.4-53, Deutsches Patentamt (April 1974)

[31] Händler, W., "Unconventional computation by conventional equipment", Proceedings of the Defence Research Group Seminar on the Design and Evaluation of Information Systems, Athens (1974), pp. 351-371, reprinted in Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg 7 (1974)/2

[32] Herzog, U., Hoffmann, W. and Kleinöder, W., "Performance modelling and evaluation for hierarchically organized multiprocessor computer systems", this conference

[33] Enslow, P.H., Multiprocessors and Parallel Processing, Wiley, New York (1974)

[34] Cooper, R.M., "Micromodules: Microprogrammable building blocks for hardware development", 1st Annual Symposium on Computer Architecture 1973, IEEE, New York (1973), pp. 221-226

[35] Fuller, S.H., Siewiorek, D.P., Swan, R.J., "Computer modules: An architecture for large digital modules", cf. [34], pp. 231-237

PERFORMANCE MODELING AND EVALUATION FOR HIERARCHICALLY ORGANIZED MULTIPROCESSOR COMPUTER SYSTEMS[*]

U. Herzog, W. Hoffmann, W. Kleinöder
Institute of Mathematical Machines
and Data Processing (III)
University of Erlangen-Nürnberg, Germany

Abstract -- We first overview the field of traffic theory, related problems, its methodology and its importance: performance modeling and evaluation is needed from the initial conception of a system architecture to the daily operation of a computer system.

Secondly, we show how to model and evaluate the performance of hierarchically organized multiprocessor computer systems. Most important: the flow of information depends on the hardware/software structure and the structure of the application programs, as well. We therefore classify application programs, discuss their implementation on hierarchically organized multiprocessor systems and describe the flow of information by a new class of queuing models.

We finally evaluate some interesting examples, the results of which are obtained by exact or close approximate solutions.

1. Introduction

The field of traffic theory is to describe, to analyze and to optimize the flow of data and control information in computer systems. Performance characteristics, such as utilization, throughput and response time give information about the efficiency of the hardware/software structure and allow the detection of bottlenecks. Computer performance evaluation is necessary from the initial conception of a system architecture to the daily operation of a computer installation.

This paper deals with the performance modeling and evaluation for hierarchically organized multiprocessor computer systems: There are several layers of computers, a connecting network between and within these layers, and a strict organization of the overall system. Hierarchical structures are transparent since we may concentrate coordination problems while distributing independent user tasks.

Typical examples are the EGPA-project [3], the multiprocessor system at SUNY [4], the Siemens system SMS [12], MOPPS [15], X-TREE [2] and others.

We first introduce traffic theory and discuss typical problems we are interested in.

Secondly, we show how to model and analyze hierarchically organized multiprocessor computer systems. As we shall see: The temporal sequence of events is determined not only by the structure and operating mode of a multiprocessor system. Rather, it is heavily influenced by the internal structure of the application programs to be run on the system.

We therefore classify the variety of application programs, consider their implementation on a given computer system and develop the corresponding models.

The quality of performance statements heavily depends on the accuracy of our models: The more careful our modeling technique, the more reliable are the results.

We therefore have to take into consideration the particulars of hierarchically organized multiprocessor systems such as synchronization, data transfers, code and data sharing. Introducing a new class of queuing systems we show how to study the impact of these problems on the effiency of a system.

2. General Remarks on Performance Modeling and Analysis (Traffic Theory)

The field of traffic theory for computer systems is determined by the following three points

- the investigation of service and transportation processes within a single processor, between several processors within the hierarchy, and between the hierarchy and I/O-facilities,
- the definition and evaluation of characteristic performance values, such as utilization, throughput and response time,
- the detection and removal of bottlenecks related to the structure and operating mode of the system.

In doing so the overall objective is to design optimal economic structures and operating modes for

- prescribed performance under normal conditions, and
- prescribed minimum performance under heavy load and failure conditions.

Because our investigations were initiated and influenced by the EGPA-project, a 1.7 Million Dollar project at the University of Erlangen, we outline next the EGPA-architecture and related traffic problems.

2.1 The EGPA-pyramid [3]

The EGPA (Erlangen General Purpose Array) consists of a rectangular array of processors

[*] This work was supported by the BMFT, the German Ministry of Research and Technology under the EGPA-contract.

(A-processors) connected via multiport memories. Each processor may access its "own" memory, and the memories of its four neighbours in the north, west, south and east direction (figure 1). The edges of the rectangle are connected to form a toroid. In addition to the array-processors, there are processors dedicated to the transfer of data between the array-processors and the peripherals.

At present, each of these processors, called boundary (B)-processors, is allocated to a 2 x 2 square of the array. In addition there is a single processor, called control (C)-processor, which has a supervisory function. The pyramid structure is shown in figure 2. A pilot-pyramid is being implemented now.*

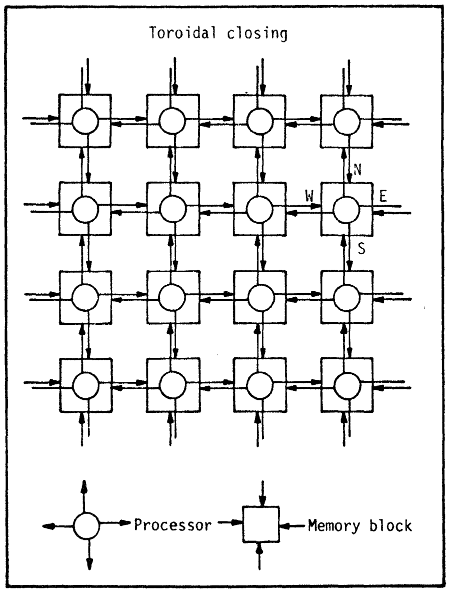


Fig. 1: Erlangen General Purpose Array (EGPA)

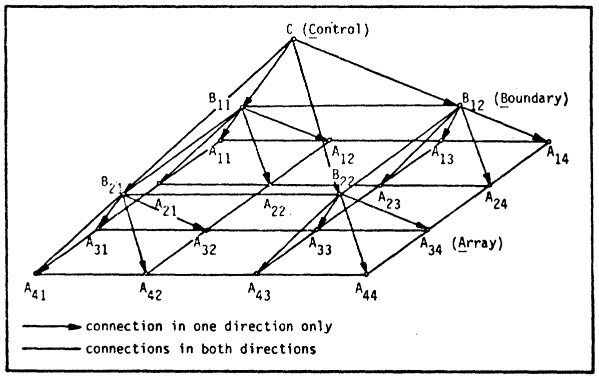


Fig. 2: Cellular structure of EGPA

* The EGPA-Project is supported by the BMFT, the German Ministry of Research and Technology.

In analyzing the performance we may distinguish between several levels of traffic problems. Dependent on the point of view we distinguish between the global behaviour of the system and the detailed description of specific subproblems.

2.2 Macroscopic behaviour and related performance characteristics

As stated above, the overall objective is to find optimal economic structures and operating modes for a prescribed workload of the computer system. We therefore have to investigate the efficiency of each individual component (processors, memories, ...), the interconnection scheme (communication between components), and the efficiency of the global operating system (distribution of user task, assignment of resources, ...), as well.

Typical performance characteristics which give insight into these global problems are

- system throughput
- response time (mean and distribution function)
- utilization of processors (idle state, user state, monitor state)
- mean number of processors working simultaneously
- overlap (concurrency) of resources
- global synchronization of user tasks
- utilization of storage units and channels
- page traffic, etc.

2.3 Microscopic behaviour and related performance measures

Here we investigate the flow of control information and data at a much more detailed level.

First we may be interested in a detailed analysis at the user program level:

- response time for individual application programs
- percentage of overhead, data transfers, application processing
- degree of parallelism.

Secondly, we are interested in a detailed analysis at the task level:

- number of tasks, interarrival and execution times
- number of task interruptions and related overhead
- degree of parallelism
- influence of the global and local operating systems (coordination, synchronization, etc.)

Thirdly, we are interested in a detailed analysis of data transfers between several layers of the multiprocessor system and background storage, as well:

- number of blocks, interarrival times and block length,
- queue length and traffic bottlenecks,
- proper design of buffers, etc.

Last but not least a very important area is the investigation of memory interference and related conflicts, especially when several processors may access the same storage unit.

2.4 Performance analysis

There are three important methods available for the investigation of traffic flow [7]:

1) The measurement of real system behaviour by special software and hardware components,
2) The simulation of the structure and operating modes by means of special programs or simulation languages,
3) The mathematical methods which allow the exact or approximate investigation of the traffic flow in components or in the overall system.

The main point of measurement is that we obtain real data of existing systems, and we obtain results to check our initial assumptions of simulation and mathematical methods. However, measurements which do represent the actual system behaviour are rather expensive and time consuming. And what about new system structures and operating modes, not yet implemented?

The main advantage of simulation compared to the mathematical methods is twofold: A very detailed modeling and evaluation of the actual traffic flow is possible. And new problems may be investigated in relatively short and well defined time intervals. However, the problem of system engineers is not only the analysis but also the synthesis of computer systems. And real optimization at reasonable expense is only possible by means of the mathematical methods.

This summary shows that each of these methods has its advantages and disadvantages, as well. We therefore may find all three methods in various stages of the design process of a computer system.

Up to now we discussed the scope of traffic theory in general and by means of examples. Let us complete these remarks in summarizing the main steps of the traffic theory:

1) Critical analysis of existing computer systems and existing tools for performance evaluation
2) Modeling of computer systems and its hardware/ software components
3) Analysis of the performance by means of simulation and exact or approximate mathematical tools
4) Validation of the above models by system-measurements
5) Synthesis of optimal components and an optimal overall design.

As mentioned above, we focus our attention to the problem of modeling. The other subjects, however, are briefly touched and related publications cited.

3. Modeling of Hierarchically Organized Multiprocessor Computer Systems

3.1 Methodology

The flow of information in a multiprocessor computer system depends on the hardware components, the interconnection scheme and the operating system. It is, however, heavily influenced also by the internal structure of the algorithms implemented in the application programs.

Our modeling methodology therefore includes the following steps:

1) Classification: we classify the variety of application programs using directed graphs.
2) Implementation: we consider the implementation of these application programs on hierarchically organized multiprocessor computer systems taking into consideration the operating mode (monoprogramming and/or multiprogramming).
3) Modeling and evaluation: we develop and analyze the corresponding queuing models taking into account the particulars of the multiprocessor system (hardware/software structure) and the application programs, as well.
4) Characteristic workload: application programs in pure form (one type only) may occur in special purpose multiprocessor systems. For general purpose systems, however, a mixture of all program structures is typical. We therefore have to unify several models (combined model) and choose the model parameters according to a characteristic workload.

In the following sections we outline step one, two and three. Although we have not investigated yet the combined model, the analysis seems to be straight forward. From our point of view it seems to be more difficult to find a characteristic workload and to choose the model parameters accordingly (cf. final section).

3.2 Classification of application programs

Following the work of Adams [1] and others, we describe a program by means of a directed graph, the nodes representing subtasks (well defined functions or sets of functions), the edges showing interdependencies and representing data buffers (unlimited FIFO-queues). Nodes (subtasks) are performed if and only if each input edge to this node contains at least one datum.

• Type-1-program structure

The type-1-program consists of a loop which may be passed several times, cf. figure 3. Within that loop n independent subtasks can be distinguished (there may exist some pre- and postprocessing). A new loop-cycle may be started if and only if all n independent subtasks are completed.

Problems are often of this type: algorithms for the solution of linear-algebraic or partial differential equation systems, optimization procedures, simulations including subruns for the purpose of estimating confidence intervals, problems of picture processing, etc. etc.

Validation example: in order to validate our performance modeling and evaluation technique (cf. also section 4) we investigated carefully the travelling salesman problem, a special optimization procedure:

- A salesman has to visit each of s given cities once and only once. What tour should he choose in order to minimize the total tour cost?
- We can solve this problem by so-called 3-optimal tours [14]:

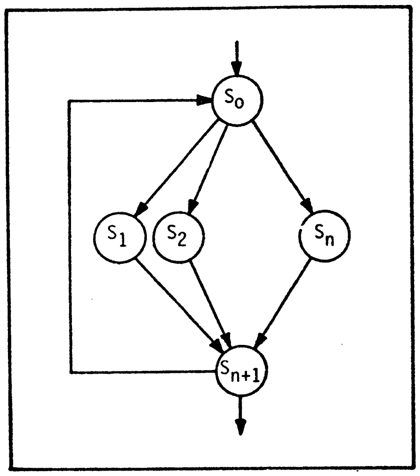


Fig. 3: Type-1-program structure

Starting from a randomly chosen initial tour the algorithm tries to get an improvement by replacing three links by another combination of three links. This basic operation will be done until no more improvement can be achieved. Because a 3-optimal tour is not optimal, we produce some 3-optimal tours and take the best tour as solution of the problem.

Obviously, this algorithm may be implemented as a type 1-program structure: first we may generate n (randomly chosen) initial tours (subtask s_0). Starting with these initial tours we search in parallel and independently for 3-optimal tours (subtask s_i, $1 \leq i \leq n$). Having finished all parallel subtasks we compare these 3-optimal tours and select the best one (subtask s_{n+1}).

• Type-2-program structure

Here, the program also consists of a loop. However, the n subtasks influence each other in some way (figure 4 shows one possibility) rather than being completely independent. Compared to type-1-program structure there are not only global but also local synchronizations necessary.

• Type-3-program structure

For many application programs the degree of parallelism varies rather than being constant. Such program structures are classified as type-3-programs, an example of which is shown in fig. 5.

• Type-4-program structure

These program structures consist of completely independent tasks $t_1, \ldots, t_n$ (cf. fig. 6a). In addition there may be some preprocessing t_0 and a feedback loop (fig. 6b). However, no synchronization is necessary between $t_1, \ldots, t_n$ since they are completely independent.

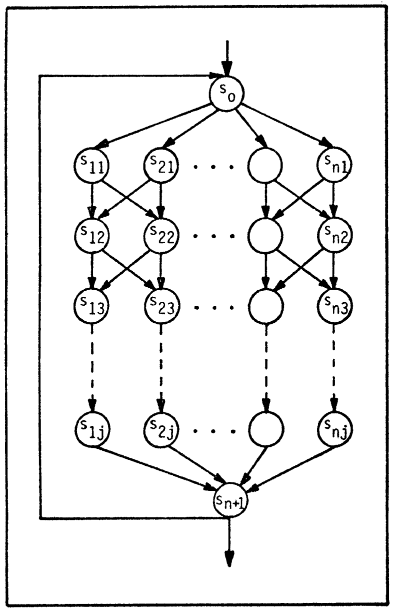


Fig. 4: Type-2-program structure

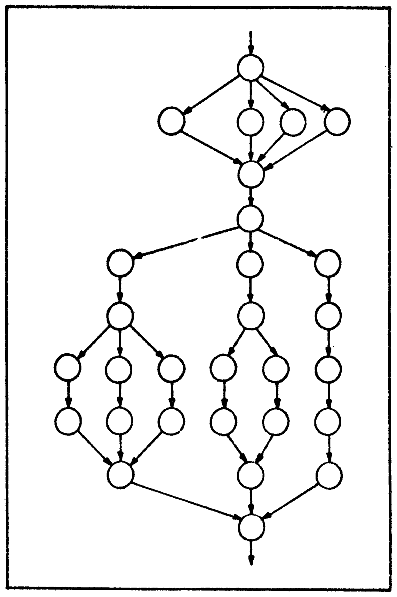


Fig. 5: Structure of an ALGOL-program with varying parallelism [13]

Remark:

It should be noted that the travelling salesman problem may also be implemented as a type-4-program. Then, however, subtasks $t_1, \ldots, t_n$ correspond to the parallel but independent search for one 3-optimal tour, i.e. each subtask tries to improve a rotation of the best tour known at that moment.

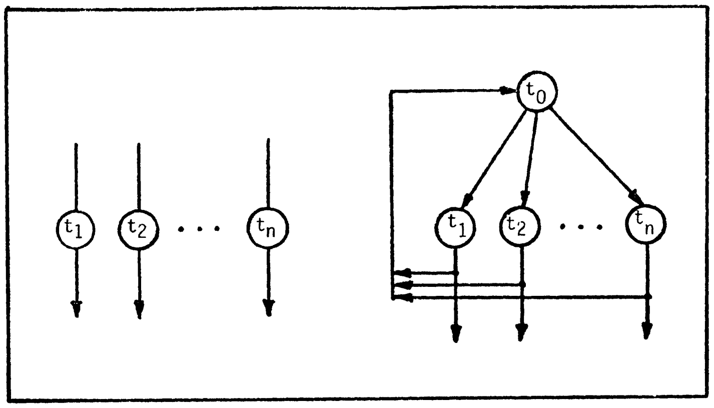


Fig. 6: Type-4-program structures

3.3 Implementation and modeling for a two level hierarchy

• Type-1-program structure

Type-1-programs may be implemented very efficiently on a hierarchically organized multiprocessor system with two levels (cf. figure 7):

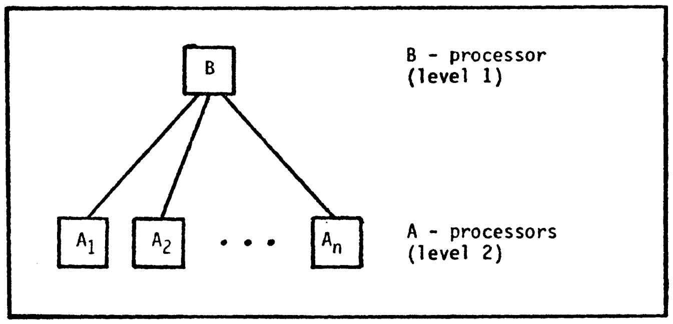


Fig. 7: Hierarchically organized multiprocessor system with two levels

- At first the B-processor starts the program.
- Then the B-processor initiates the execution of n independent subtasks by the A-processors.
- Having completed its subtask, each A-processor has to inform the B-processor.
- Postprocessing and preparation of a new loop-cycle by the B-processor is only possible when all subtasks are completed (i.e. synchronization is necessary).

The corresponding timing-diagram is shown in figure 8. And, again, it is worth-while to interprete it for the travelling salesman problem.

A queuing model which allows us to describe and analyze the traffic flow including the above synchronization problem is shown in figure 9 (synchronization is shown symbolically by a horizontal bracket).

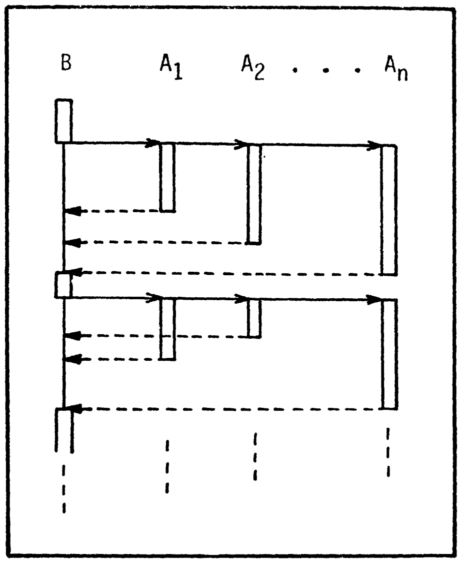


Fig. 8: Execution of a type-1-program on a two-level hierarchy

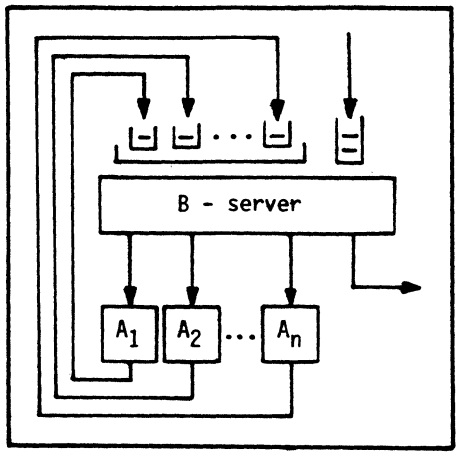


Fig. 9: Queuing model for type-1-programs and monoprogramming

Monoprogramming

For reasons of simplicity and transparency we first assume monoprogramming for both B- and A-processors:

- Newly arriving demands (source programs) are buffered in the input queue.
- If the "inner" system is empty the first demand is processed by the B-processor.
- The B-processor generates n independent sub-demands and distributes them simultaneously among all n A-servers (more sophisticated transfers, cf. section 3.5).
- After completion, each sub-demand is buffered in the corresponding input queue of the B-server. If all n sub-demands are buffered, they are removed simultaneously (symbolized by $\llcorner\!\_\_\!\lrcorner$) from the n parallel input queues and processed in one step.
- After completion there are two possibilities: the (complete) demand leaves the system or n new sub-demands are generated simultaneously and a new loop-cycle is started.

Multiprogramming

If we allow multiprogramming for both B- and A-processors several programs may be interleaved. In principle, the corresponding queuing model is similar to that of monoprogramming. However, queues may build up in front of the A-processors, too.

Mixed multi- and monoprogramming

Multiprogramming allows us to increase system throughput. However, for reason of simplicity and transparency of the operating system there is a trend to introduce monoprogramming again. For many applications a mixture of both seems to be an efficient solution: multiprogramming for the B-processor and monoprogramming for the A-processors. Figure 10 shows the corresponding queuing model and is rather self-explanatory:

Assume a number m of independent demands $(t_1,\dots,t_i,\dots,t_m)$ to be served sequentially (!) on the B-server. After completion each task t_i $(i = 1,\dots,m)$ generates n_i independent subtasks to be processed by the reserved A-processors A_{i1} to A_{in_i}. Task t_i may be resumed if and only if all n_i subtasks have been completed by the A-processors. If the B-server is busy, complete demands wait in front of the server and are served in the order of arrival (FIFO).

Note, synchronization is only necessary between sub-demands belonging together, an important fact for analysis.

· Other program structures

We illustrated the process of modeling by means of type-1-program structures. It is straight forward to develop the queuing models for the other program types and a mixture of several program types, too. The variety of these queuing models is presented in [8], two more examples are shown in fig. 11. We rather deal now with three- and multi-level-hierarchies.

3.4 Performance modeling for three- and multi-level-hierarchies

As before we do have to analyze the structure of the application programs to be run on the hierarchy and we do have to decide how to implement the algorithms (a detailed description how to implement the travelling salesman problem may be found in [9, 10], numerical results are shown in section 4.3).

In modeling the traffic flow we have to consider the following most important point:

- It is rather unwise and probably unsuccessful to describe and analyze the complicated flow of information and all interdependencies by a single global queuing model.

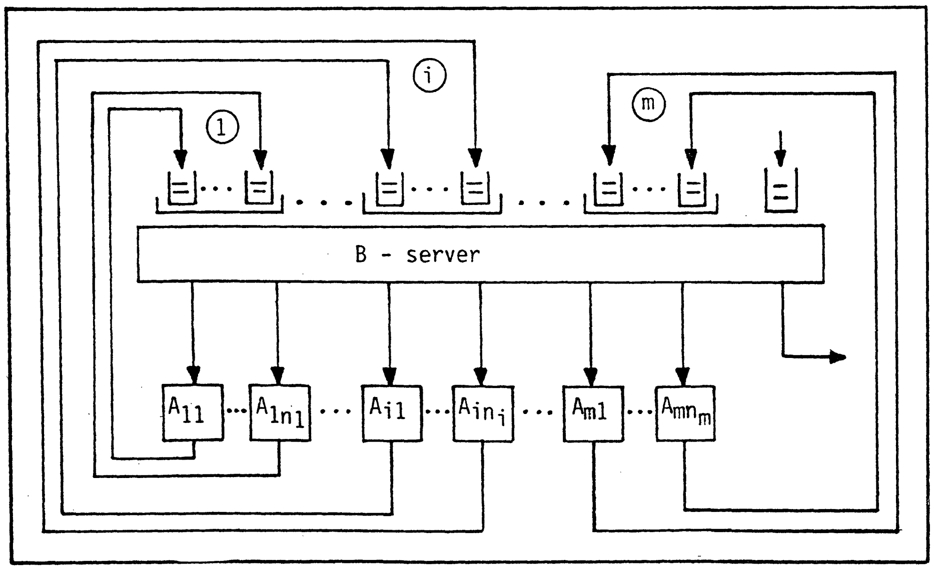


Fig. 10: Queuing model for mixed mode

An efficient and often successfully applied technique (often referred to as hierarchical modeling technique [7, 11]) is nothing else but a systematically structured top-down modeling and a bottom-up evaluation.

- We decompose the total system in hierarchically structured subsystems of manageable size and develop the corresponding models: a macro level model, several medium level models, etc. etc.
- We analyze these models individually either by simulation or mathematical tools.
- Starting at the lowest modeling level we determine "local" performance values, embed them into the next higher modeling level, etc. etc., up to highest level, the macro level model.

Figure 11 illustrates this technique, applied to a three-level hierarchically organized multiprocessor system (numerical results and their validation are discussed in section 4.3):

First, we investigate the interactions between the top processor C and all subsystems SS1, ...,SSn and develop the macro-level model. Obviously, the duration of service by these subsystems is not yet known. We therefore have to go "one level deeper", investigate the interactions within each subsystem (one B- and several A-processors) and develop the corresponding models (which may be of the same or different type), etc. etc.*

In evaluating this hierarchy of models we have to proceed in the other direction: we start with the lowest level models, determine "local" performance values which reflect the duration of service at this level, and embed these results in the next higher level of models, etc. etc. Finally we reach the top level and obtain global performance values such as throughput and response-time of the total system.

3.5 Refinement of models

Up to now we have neglected some important features of many real computer systems in our examples:

- the initialization of subprocesses often takes a considerable amount of time
- application programs and data have to be transferred between foreground and background memories
- synchronization between subtasks causes interrupts and overhead for the top processor

etc. etc.

It is possible to include these phenomena in our modeling technique, too, as demonstated next by one example.

• Initiation of subprocesses

Up to now we have assumed negligible time intervals for the initiation of subprocesses. In

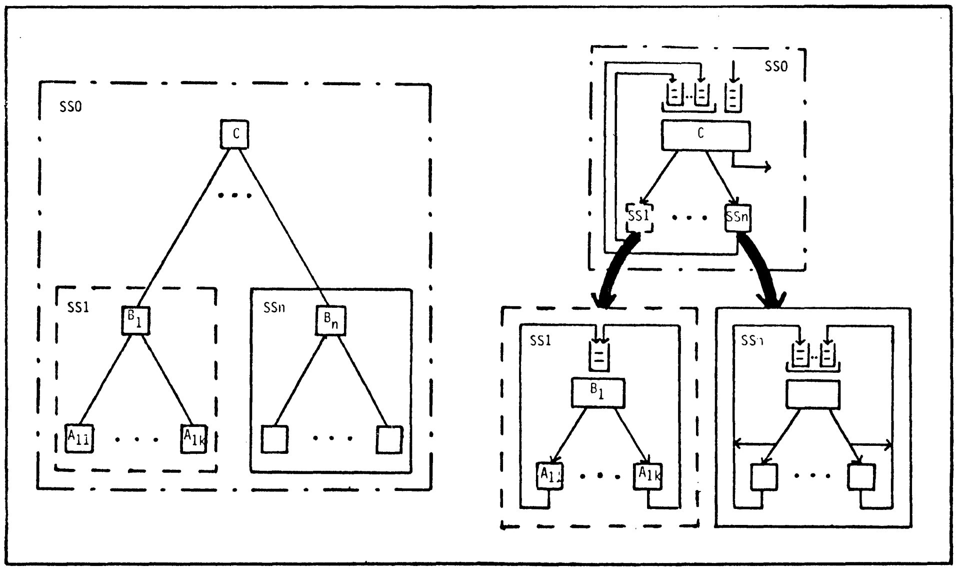


Fig. 11: Modeling of a three-level hierarchically organized computer system by one macro-level model (SSO) and n medium level models (SS1,...,SSn) (SS1 corresponds to a type-4-program structure while SSn allows us to model type-3-program structures).

practice, however, there may be a considerable amount of time necessary for the preparation and transfer of control information, user programs and data.

From our experience with the EGPA-project [17], these details may be captured by the following operations (cf. timing diagram, figure 12):

* Structured submodels for each individual processor are well known from literature [11].

- After some common preprocessing the B-processor initiates the first subprocess: code and user data are loaded to the memory of processor A_1; additionally there may be an individual preprocessing for this subprocess necessary, handshaking, etc.
- Processor A_1 may now start to process its subtask and the above procedure is repeated for processors A_2, A_3 to A_n.

The corresponding queuing model is the same model as in the case without individual phases for starting. However, we now interpret the behaviour of the B-server in another way:

In the simple model, the B-server removes simultaneously from all input queues one subdemand and processes them in one step. Now, the B-processor also removes from each queue one subdemand, but it serves them sequentially. And there is some overlap between the B-processor and the A-processor to be started next.

Interarrival and service times are interpreted as random variables with a given distribution function. The usefulness and validity of such an approach has often been proven (cf. [11]) and efficient methods are available to determine type and parameters of distribution functions [16, 18].

All models for two-level hierarchies presented above have been investigated under various assumptions [10, 19, 20].

- First we analyzed the models under Markovian assumptions, i.e. for reason of simplicity we assumed exponentially distributed service times for both, B- and A-servers (the mean value for different A-servers may vary).
- Since service times are most often non-exponentially distributed we then relaxed these assumptions and analyzed the models for Erlangian, hyperexponential and even generally distributed service times.

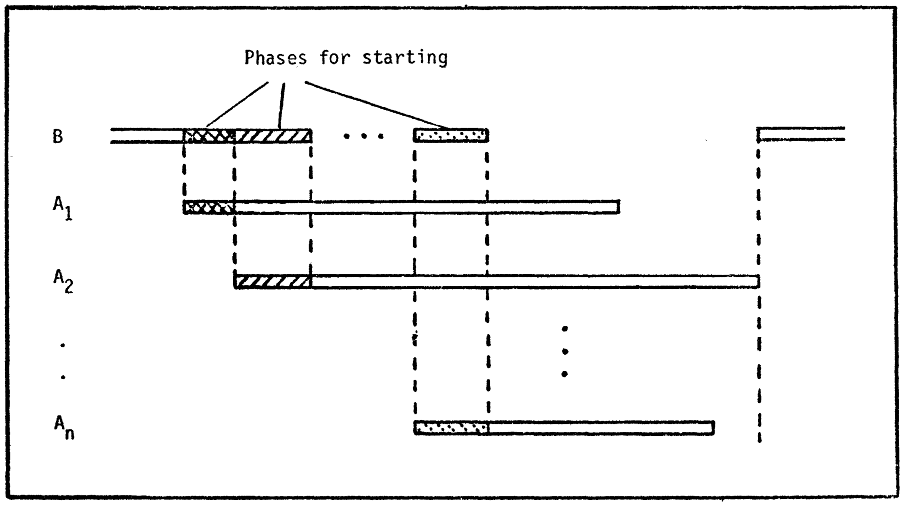


Fig. 12: Timing diagram with phases for starting an A-processor

4. Analysis of Some Models and Numerical Results

4.1 Remarks on the probabilistic analysis

The flow of information within computer systems is determined by

- the transportation of data and control information between subsystems
- the processing of data by processors and control units
- the blocking and waiting at various locations within the computer system.

The sequence of events of such transportation-, waiting- and processing-times is at least partially deterministic although the interdependencies are rather complex. From the point of view of a subsystem, however, the sequence of events seems to be random. In other words, the sequence of events may be described by means of stochastic processes:

In analyzing the performance we usually assumed stationarity, determined then state probabilities and characteristic performance values such as

- system throughput
- server utilization individual for each server
- mean numbers of A-servers working simultaneously
- mean numbers of demands and/or sub-demands waiting in front of servers (queue length)
- mean synchronization time, i.e. mean time between the moment when all related sub-demands start and the last one is completed
- mean cycle time for complete demands, the sum of synchronization time and processing time at the B-server
- distribution functions for both the synchronization and cycle time.

4.2 The model for type-1-program structures and mixed multi- and monoprogramming

Be given a model structure according to figure 10, the service discipline of which being described in section 3.3. For reasons of clarity service times for both B- and A-servers are assumed to be exponentially distributed with

μ	: service rate for the B-server
λ_{ij}	: service rate for the A_{ij}-server, $i \in \{1,2,\dots,m\}$ $j \in \{1,2,\dots,n_i\}$
m	: number of complete demands D_i (competing for the B-server), i.e. degree of multiprogramming
n_i	: number of parallel sub-demands belonging to demand D_i.

Analysis is performed under equilibrium conditions, i.e. stationarity is assumed.

· Decomposition: Recall, again, that synchronization is only necessary between sub-demands S_{ij}, $j \in \{1,2,\dots,n_i\}$ together. So, if we are able to analyze all individual synchronization processes, the overall behaviour is described exactly by the following mode, known from the model for type-4-programs (cf. figure 11, model for SS1):

B-server:	service times exponentially distributed with rates μ as in the complete model.
A-server:	service times generally distributed with different service rates λ'_i <u>according to the d.f. of the synchronization times $F_i(t)$ and its mean $E_i[T_S]$, $i \in \{1,2,\dots,m\}$</u> (definition cf. 4.1). No queues in front of the A-servers.

We therefore attack first the (individual) synchronization problem also being the solution for the monoprogramming model shown in figure 9. Then, in a second step, we briefly outline the solution for the model for type-4-programs, a generalization of the so-called repairman-model.

• Synchronization: Let D_i be the complete demand the synchronization time T_S of which has to be determined. Processing of all sub-demands S_{ij}, $j \in \{1,2,\dots,n_i\}$, starts at the same instant. Suppose all processing times are exponentially distributed with uniform service rates $\lambda_{i1} = \lambda_{in_i} = \lambda_i$ (solution for different rates, cf. [5, 10]).

Now, T_S is the maximum of n_i service times $T_{A_{ij}}$, $j \in \{1,2,\dots,n_i\}$ (cf. figure 13), and therefore the distribution function $F_i(t)$ of the synchronization time is the product of the distribution functions $F_{ij}(t)$ of the service times $T_{A_{ij}}$ $(1 \leq j \leq n_i)$.

If $F_{ij}(t) = 1 - e^{-\lambda_i t}$ for $1 \leq j \leq n_i$

we obtain

$$F_i(t) = P(T_S \leq t) = \prod_{j=1}^{n_i} F_{ij}(t) = (1 - e^{-\lambda_i t})^{n_i}$$
$$= \sum_{k=0}^{n_i} \binom{n_i}{k_i} (-1)^k \cdot e^{-k\lambda_i t}$$

with mean and variance

$$E_i(T_S) = \frac{1}{\lambda_i} \sum_{k=1}^{n_i} \frac{1}{k}$$

$$VAR_i(T_S) = \frac{1}{\lambda_i^2} \sum_{k=1}^{n_i} \frac{1}{k^2}$$

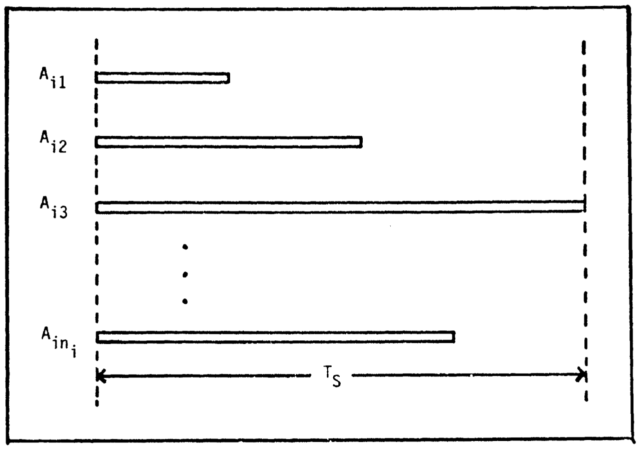


Fig. 13: The synchronization time T_S

• Overall behaviour: Since the synchronization problem is solved now the overall behaviour is completely determined by the model for type-4-programs, presented in fig. 11, SS1 with no queuing in front of the A-servers and where the service rate of these m A-servers is

$$\lambda'_i = \frac{1}{E_i[T_S]} \quad i \in \{1,2,\dots,m\}.$$

We first analyzed this model under Markovian assumptions: system states were described by a (m + 1)-dimensional vector, stationarity was assumed and the explicit solution derived for the stateprobabilities, a generalization of the well known M/M/1/m-solution.

Secondly, we proved the solution also being valid in case of general service time distributions for the A-servers. The detailed analysis may be found in [10], some characteristic performance values are:

Utilization of the B-processor Y_B

$$Y_B = \frac{1}{h}\Big(\sum_{j=1}^{m} \sum_{i_1,\dots,i_j \in \{1,2,\dots,m\}} \mu^{m-j} \prod_{i=i_1,\dots,i_j} \lambda'_i \Big)$$

Utilization of each A-processor Y_A

$$Y_A = \frac{1}{h}\Big(\mu^m + \sum_{j=1}^{m-1} \frac{m-j}{m} \mu^{m-j} \cdot \sum_{i_1,\dots,i_j \in \{1,\dots,m\}} \prod_{i=i_1,\dots,i_j} \lambda'_i\Big)$$

Mean cycle time t_z for task D_i

$$t_z(i) = \frac{1}{\lambda_i'} + \frac{\mu^{m-1} + \sum_{j=2}^{m} j\,\mu^{m-j} \sum_{i_1,\ldots,i_{j-1} \in \{1,\ldots,m\} \setminus \{i\}} \prod_{k=i_1,\ldots,i_{j-1}} \lambda_k'}{\mu\left(\mu^{m-1} - \sum_{j=2}^{m} \mu^{m-j} \sum_{i_1,\ldots,i_{j-1} \in \{1,\ldots,m\} \setminus \{i\}} \prod_{k=i_1,\ldots,i_{j-1}} \lambda_k'\right)}$$

in twos different

with

$$h = \mu^m + \sum_{j=1}^{m} \mu^{m-j} \sum_{i_1,\ldots,i_j \in \{1,\ldots,m\}} \prod_{i=i_1,\ldots,i_j} \lambda_i'$$

4.3 Numerical results for two-level hierarchies and a three-level hierarchy

Figure 14 shows the mean cycle time t_z for complete demands on a two-level hierarchy as a function of:

1) the program (and therefore model) structure: we may have one demand with four sub-demands to be synchronized (upper curve), we may have two independent demands with two sub-demands each (middle), or we may habe four independent demands (lower curve).
2) the mean service time $1/\mu$ of the B-server

Furthermore, it is assumed that the service rate λ is equal to one for all sub-demands. In order to compare the results for the same load per cycle of the B-server we introduced different scales. The diagram shows clearly how the type of program influences the cycle time.

Figure 15 shows a three-level multiprocessor computer system and the flow of information if we run the travelling salesman problem on such a configuration (cf. also section 3.4).

We developed three approximate procedures by decomposing the system in several subsystems. Some results are shown in figure 16 and compared with a step-by-step simulation of the complete 3-optimal-tour-algorithm. In other words: we simulate the algorithm as it would be performed on a real three-level multiprocessor system (since the algorithm starts from a randomly chosen initial tour, the execution time varies even for the same number n of cities: we therefore performed the algorithm ten times for each problem).

Comparisons show that the very simple procedure 1 (expon. assumptions) tends to overestimate the execution time. Procedure 2 (Erlangian d.f. with constant variance) yields, for a small number of cities, results which are somewhat too optimistic. Finally, the most sophisticated procedure 3 (Erlangian d.f., variance properly adjusted) yields always accurate results.

5. Summary and outlook

The intension of this paper is twofold:

- First, we tried to overview the field of traffic theory, related problems, its methodology and its importance: performance modeling and evaluation is needed from the initial conception of a system architecture to the daily operation of a computer system!
- Secondly, we showed how to model and evaluate the performance of hierarchically organized multiprocessor computer systems. Most important: the flow of information depends on the hardware/software structure and the structure of the application programs, as well!

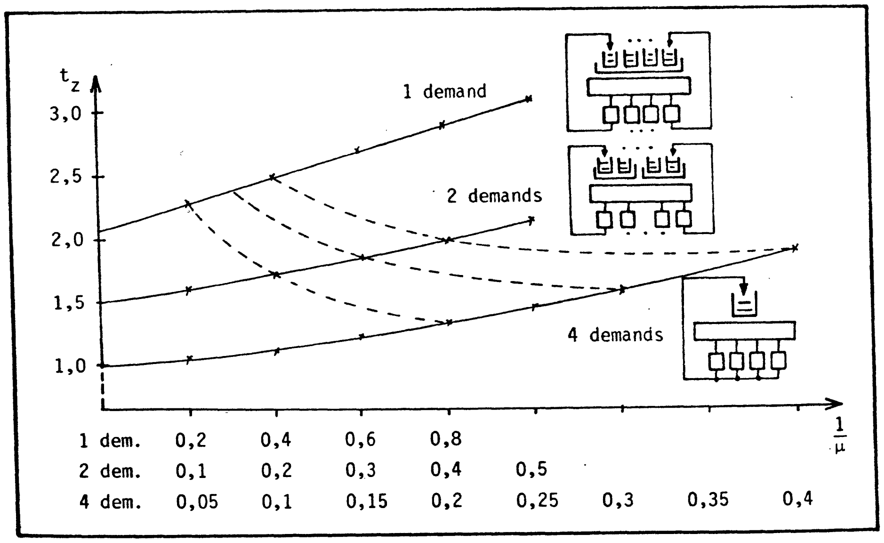


Fig. 14: Examples for the mean cycle time t_z on a two-level hierarchy

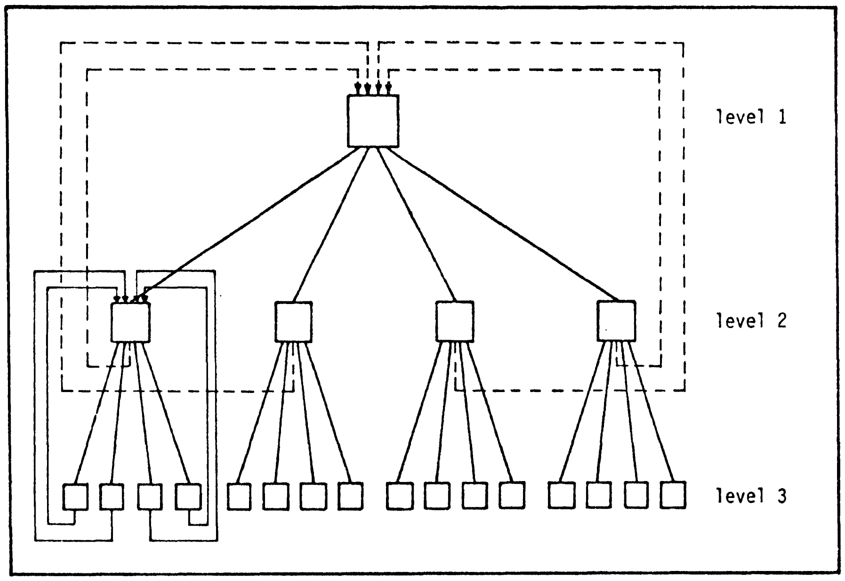


Fig. 15: Three-level multiprocessor computer structure

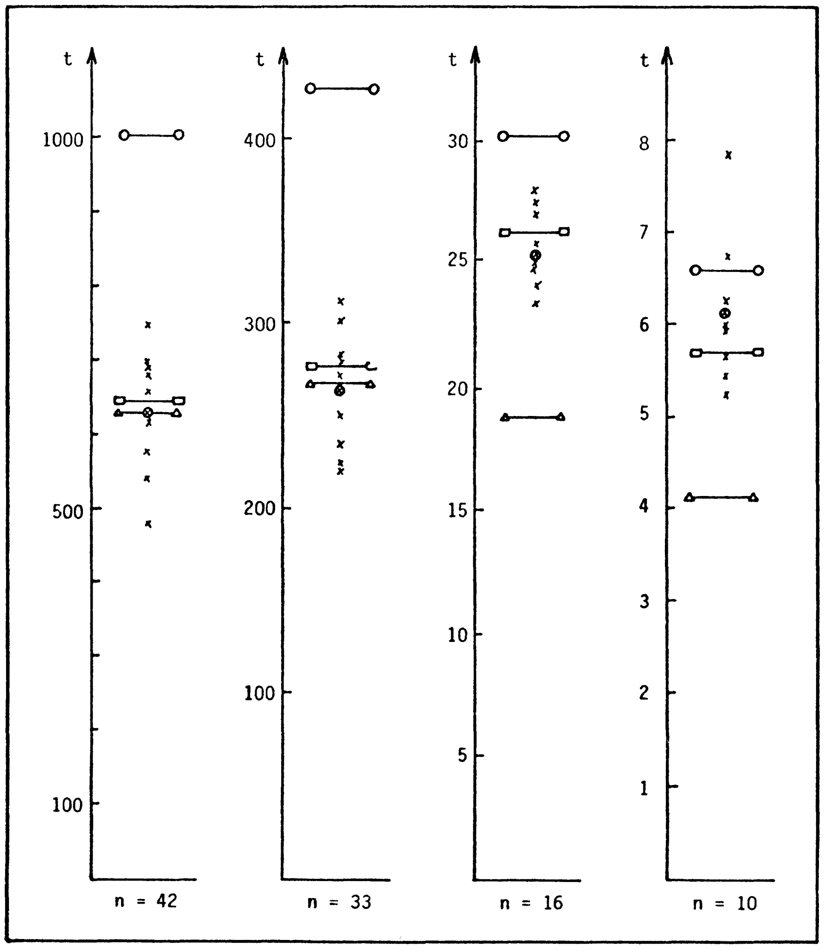


Fig. 16: Execution time t for the travelling salesman algorithm on a three-level hierarchical multiprocessor computer system. Comparison between results from a step-by-step simulation of the algorithm (simulations x, mean value ⊗) and three approximate solutions (proc. 1 o—o, proc. 2 Δ—Δ, proc. 3 □—□). Results are shown for n = 10, 16, 33 and 42 cities (cf. text).

Work is going on. At the moment our research goes into two directions:

- Beside general distribution functions for service times you may find in our models exponential assumptions,too. We are trying to extend our results by introducing more general distributions.
- Secondly, we try to refine our models: transfer times, individual initiation of subprocesses and priorities are some of the most important examples.

From these remarks you may see that there are still many important and challenging problems to solve in the area of performance modeling and evaluation for multiprocessor computer systems.

References

[1] ADAMS, D.A.: A Model for parallel computations. In: Parallel Processor Systems, Technologies and Applications, Hobbs et al.(eds.), Spartan Books, New York, Washington (1970)

[2] DESPAIN, A.M.; PATTERSON, D.: X-Tree, A Tree Structured Multi-Processor Computer Architecture. SIGARCH Newsletter, Computer Architecture News, Vol. 6, No. 7 (1978), pp. 144-151

[3] HÄNDLER, W.; HOFMANN, F.; SCHNEIDER, H.J.: A General Purpose Array with a Broad Spectrum of Applications. In: Computer Architecture (Händler ed.), Informatik-Fachberichte 4, Springer-Verlag (1975), pp. 311-335

[4] HARRIS, J.A.; SMITH, D.R.: Hierarchical Multiprocessor Organizations. Computer Architecture News, Vol. 5, No. 7 (1977), pp. 41-48

[5] HERZOG, U.: Verteilungsfunktion der Zykluszeit für das Synchronisationsmodell mit beliebiger Zahl von A-Prozessoren und einem Gesamtauftrag. University of Erlangen-Nürnberg, unpublished memo (April 1978)

[6] HERZOG, U.: Synchronisations- und Zykluszeit für Typ-2-Programmstrukturen bei Monoprogrammierung, offene und geschlossene Warteschlangenmodelle. University of Erlangen-Nürnberg, unpublished memo (July 1978)

[7] HERZOG, U.: Modeling of Data Networks. Invited Paper at ITC (Intern. Teletraffic Congree)Seminar on "Modeling of Stored Program Controlled Exchanges and Data Networks". Delft University of Technology (Oct. 1977), to be published by North Holland

[8] HERZOG, U.; HOFFMANN, W.: Synchronization Problems in Hierarchically Organized Multiprocessor Computer Systems. Proc. of 4th Intern. Symposium on Modeling and Performance Evaluation, Wien (Feb. 1979)

[9] HOFFMANN, W.: Queuing models for parallel processing and their application to a hierarchically organized multiprocessing system. Proceedings of First European Conference on Parallel and Distributed Processing, Toulouse, France (Feb. 1979)

[10] HOFFMANN, W.: Warteschlangenmodelle für die Parallelverarbeitung Ph.-D.-Thesis. University of Erlangen-Nürnberg (1978)

[11] KOBAYASHI, H.: Modeling and analysis, an introduction to system performance evaluation methodology. Addison-Wesley (1978)

[12] KOPP, H.: Numerical Weather Forecast with the Multi-Microprocessor System SMS 201. In: Parallel Computers - Parallel Mathematics. Feilmeier (ed.), IMACS, North Holland Publishing Company (1977), pp. 265-268

[13] NETT, E.: On further applications of the HU-algorithm to scheduling problems. Proc. 1976 Int. Conf. on Parallel Processing, Enslow (ed.), Wayne-State-University, IEEE-ACM (1976), pp. 317-325

[14] SHEN LIN: Computer Solutions of the Travelling Salesman Problem. Bell system technical journal, 44 (1965), pp. 2245-2269

[15] SHIMOR, A.; WALLACH, Y.: A Multibus-Oriented Parallel Processor System. IEEE Trans., Vol. IECI - 25 (1978), pp. 137-141

[16] BUX, W.; HERZOG, U.: The phase concept: approximation of measured data and performance analysis. Proc. IFIP-Symposium on Computer Performance Modeling, Measurement and Evaluation, Eds. Chandy and Reiser, North Holland (1977), pp. 23-28

[17] KLEINÖDER, W.: Verwaltungszeiten. University of Erlangen-Nürnberg, unpublished memo (Jan. 1979)

[18] LAZOWSKA, E.: Selecting parameter values for servers of the phase type. Proc. 4th International Symposium on Modeling and Performance Evaluation of Computer Systems, Wien (Feb. 1979)

[19] HERZOG, U.: Performance Characteristics For Hierarchically Organized Multiprocessor Computer Systems With Generally Distributed Processing Times. Proc. of the Ninth Intern. Teletraffic Congress, Torremolinos, Spain (Oct. 1979)

[20] HOFFMANN, W.; KLEINÖDER, W.: Warteschlangenmodelle für Mehrprozessorsysteme: Maximale Bedienungszeit von l parallel arbeitenden Bedienungsstationen. Proceedings of GI-9. Jahrestagung, Bonn (Oct. 1979), to be published by Springer.

Performance bounds for a certain class of parallel processing

Norman L. Soong, Sperry Univac

A major reason for the exploration of parallel computation is to improve computing system performance. A family of absolute mathematical upper bounds for system performance improvement is discovered and presented in this paper. Of all the upper bounds, there is a least upper bound, l.u.b. The mathematically derived l.u.b. is then applied to a set of empirically obtained performance improvement data. The fit is more than casual. The result is also compared to a similar one obtained at the Stanford University.

Processes, processors, and tasks are the fundamental concepts of this paper. A process is a progression of computing machine operations. A processor is an entity capable of carrying out operations. Represented by a collection of processes, a task is mapped onto a collection of processors to be computed. Four basic assumptions are fundamental to this study:

* MIMD organization,
* identical individual processors,
* inexhaustive supply of processors,
* unit operation execution time.

The term speedup (Sp) is related to task performance gain. Sp is defined as the ratio of the time needed to compute the task on a uni-processor system to that of a multi-processor system. Let 'p' be the maximun number of utilized processors to compute this task, then Sp=T1/Tp is commonly adopted as the definition of performance gain.

Let one unit of work be defined as the accomplishment of one processor operating for one unit of time, then the ith-work partition (Ai) can be introduced as the percentage of work done by i-processors for this task. It can be shown that

$$Tp = \Sigma \ Ai * T1 / i$$

is the expression for Tp. Substitute Tp into Sp, the fundamental relation for Sp is shown as

$$Sp = 1 / (\Sigma \ Ai / i) \quad \text{for } i=1,2,...p.$$

Theorem 1 $Sp \leq (1/p)*(p!/\Pi Ai)^{1/p}$ where p is an integer, and $Ai \neq 0$ for all i.

Theorem 2 Let $F(Ai)=(1/p)*(p!/\Pi Ai)^{1/p}$ for integer p, then the optimal minimal function of F(Ai) <u>is</u> such that all Ai's are equal and subject to the constraint of $\Sigma Ai = 1$.

The significance of this paper is the discovery of a class of mathematical upper bounds, theorem 1, and the identification of the l.u.b. of this class, theorem 2. To verify the validity of this l.u.b., it is applied to a set of empirical data. This set of data is derived from approximately 200 programs. According to their appllications, they are organized into seven groups to illustrate their average speedup characteristics. Six groups fall under the l.u.b. and one is 5 percent above the l.u.b.

A similar effort has been carried out at the Stanford University. A definite inequality is derived instead of the semi-definite inequality of theorem 1, and this approach has led to a family of not-so-sharp upper bounds. Two data points are above the Stanford University bound by a significant margin of twenty percent and sixty-one percent respectively.

The mathematical derivation of the l.u.b. consists of a formal application of the geometric inequality to a speedup definition. If the l.u.b. has any intrinsic meaning at all, then it must be contained originally in the speedup definitions. A study of programs in the time-processor product space suggests that the parallelism extracted at the statement level is the prime contributor to performance improvement. Another significant evidence is offered by the reasonable agreement with the available data points. These evidences suggest that the l.u.b. bounds the maximum <u>ensemble</u> performance improvement extractable from ordinary programs by exploring the statementwise parallelism contained in a program. An optimizing FORTRAN compiler is a typical example of this type of automatic performance extractor.

Emphasized in the above claim is the implication that the bound of Theorem 2 is an <u>ensemble</u> bound. Both the Sp-definition and the available data points are averaged <u>ensemble</u> quantities. It is possible to have it running on a machine a single program or even a class of programs, whose performance can be improved beyond the l.u.b. Yet the contrary is also true that there exist programs and classes of programs, of which little improvement can be made to their performances. It is the average of all those situations that is the quantity called <u>ensemble</u> speedup, or <u>ensemble</u> performance improvement. This is the quantity that is bounded by the l.u.b. The effectiveness of the l.u.b. is related to the number of data points in the experiment. The fit will improve with more data points.

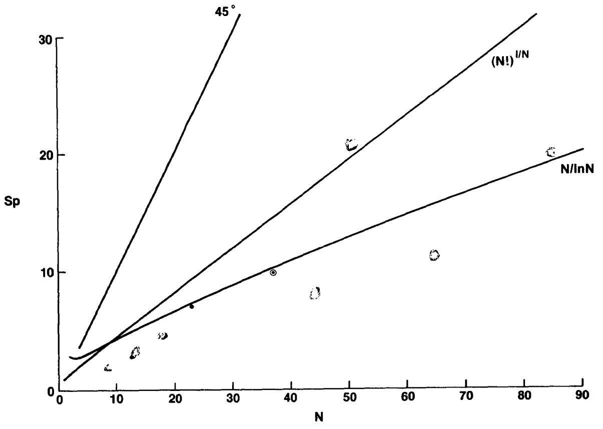


Figure. The Ensemble Performance Bound

A BLOCK-ORIENTED SPARSE EQUATION SOLVER FOR THE CRAY-1

D. A. Calahan
Department of Electrical and Computer Engineering
University of Michigan
Ann Arbor, MI. 48109

Abstract

This paper investigates the impact of a current vector processor architecture on the algorithms of a sparse direct equation solver. Timing models of a block-oriented solver implemented on the CRAY-1 allow an evaluation of the overhead of a general solver on such a processor. It is shown that a general solver can outperform even dedicated solvers written conventionally in Cray assembly language.

Sparse Equation Solvers

General sparse equation solvers are complicated mathematical software packages which solve directly (contrast iteratively) simultaneous linear equations having an arbitrary off-pivot sparsity structure [1] - [2]. This structure is described either by a linked list [3] or a bit map [4]. Such solvers have been used for more than a decade as kernels in (1) the implicit solution of algebraic and ordinary differential equations associated with lumped physical systems such as electronic circuits, electrical power systems, and 3-D mechanisms, and (2) the solution of discretized partial differential equations (PDE's) such as arise in oil reservoir simulation.

This paper examines the characteristics and applications of such solvers implemented on a memory-hierarchical vector processor, the CRAY-1. Because the architecture is fixed, a "bottom-up" approach is used, wherein the performance of appropriate numerical kernels on the CRAY-1 is studied first. These kernels are shown to have a strong preference for block matrix structure. This single fact profoundly affects the development of the solver and its applications, discussed in the remainder of the paper.

Because this study is directed toward a specific processor, it is first well to summarize three general issues which implementation of such a code on any vector processor are likely to involve.

(a) Algorithmically, vectors arise from either local matrix density (coupling of variables) or from symmetries in globally decoupled parts of the problem structure. The former will be considered in this paper; the latter is discussed in [5].

(b) The coding of such a solver will profoundly affect its efficiency; this coding in turn is intimately related to the data flow of the processor, in part due to the linked list or bit processing. This implies a need for coding in a low level language and a consequent high degree of machine dependency. The CRAY-1 was chosen for this study due to its superior scalar and short vector performance. Linked lists rather than bit maps were adopted because of the CRAY-1's limited bit controllable vector operations.

(c) The performance evaluation involves construction of a timing model so that the price paid for using a vector processor and a general equation-solving code can be evaluated by a potential user. In this study, the general solver will be shown to execute faster than codes written for specific matrix structures and conventionally coded in CAL (Cray Assembly Language).

Block Equation Solvers

Kernels of Sparse Solvers. The proposed general sparse equation solver operates on linked list descriptions of the matrix structure. Because vector operations utilizing linked lists (termed "gathering" and "scattering") commonly proceed at 1/5 to 1/10 the speed of linearly-indexed array computation, it is important for efficient vector operation that the list not point to a single non-zero element. Assuming significant local matrix coupling, the list should point to either

(a) a dense segment of a row or column, or, more generally,

(b) a dense block

of the matrix. With sufficient numeric computation resulting from this density, the index list processing can be overlapped or at least overwhelmed by the numeric computation.

Consider the factorization of a matrix A into upper and lower triangular forms U and L respectively. The numeric kernel is commonly of the form

$$D_i \leftarrow D_i - B_i C_i \tag{1}$$

where D_i, B_i, and C_i are submatrices from the overall sparse systems matrix. The accumulation of the column segments of (a) will be termed a line-vector operation of the form

$$A_{i:j,k} \leftarrow A_{i:j,k} - u_{\ell,k} L_{i:j,\ell} \tag{2}$$

where $A_{i:j,k}$ and $L_{i:j,\ell}$ are dense column segments of the matrix A and the triangular factor L, extending from row i to row j, and $u_{\ell,k}$ is an element of U (Figure 1(a)). A block-vector operation can be written (Figure 1(b)).

$$A_{i:j,p:q} \leftarrow A_{k:j,p:q} - L_{i:j,k:\ell}\, U_{k:\ell,p:q} \tag{3}$$

with similar subscript notation to indicate initial and final row and column indices.

An approximate timing model for the line-vector accumulation loop on the CRAY-1 has been found by simulation to be

$$\text{MFLOPS} = 53.3 \left(\frac{1}{1 + 31.3/\bar{\ell}} \right) \qquad (4)$$

where $\bar{\ell}$ is the average length of a dense column segment encountered during the accumulation. This model, valid when all vectors are longer than 14, achieves a maximum rate of 35.8 for $\bar{\ell}$ = 64, the maximum vector length on the CRAY-1.

The mathematical model for the vector block accumulation of (3) is given in Figure 2 as a function of block dimensions. The asymptotic rate of this kernel is 151 MFLOPS for q = 64 (the vector length) and p and q large.

The ratio of asymptotic execution rates (151/35.8 = 4.2) can be traced directly to the memory bandwidth required to implement the line-vector accumulation of (2), where an element of $A_{i:j,k}$ and $L_{i:j,\ell}$ must be loaded and $A_{i:j,k}$ stored for each and multiply. With a single memory path transferring one word in 12.5 nsec, the low asymptotic rate of $2(80 \times 10^6)/3 = 53.3$ MFLOPS of (4) follows.

Block Solver Characteristics. The above preference for block processing is unfortunate in the sense that such blocking is not unique and little is known of "optimal" methods of blocking a general sparse matrix. The equation formulation and the "fill" production [6] produce considerably overlapping of block structures, often masking the original problem structure. An example blocked structure is shown in Figure 3. Thus, a block solver must either be given the block structure or await the development of blocking algorithms. This paper considers only the development of the solver, in part because general blocking methods will undoubtedly be dependent on characteristics of the solver.

A general block-oriented equation solver for a vector processor, although conceptually derived from previous single-element processing codes, portends to have several new important applications.

(a) Common matrix structures are naturally blocked by local coupling and variables of nodes. The speed of vector processors encourage such coupling, especially to speed up the convergence of global iterative methods. For example, block relaxation methods are being considered to replace single-variable relaxation [7].

(b) A memory-hierarchial processor with a limited vector length requires the partitioning of large dense systems to reduce data flow between hierarchies--the source of difficulties in the line-vector accumulation above. Although specialized partitioned codes can be developed for each general class of dense matrix (full, band, profile, block tridiagonal), the coding in a low level language for a vector processor is sufficiently complicated that, at this writing, only partitioned full matrix codes have been developed for the CRAY-1 [8].

If one observes that partitioning is a symbolic rather than numeric process, it is reasonable to propose a solver which requires the user to construct only a set of block descriptors from the matrix structure. These descriptors would then guide the numeric solution. The descriptors must be adequate to describe common block structures and typical numeric storage schemes, but limited in number so that they do not seriously interfere with the numeric kernels, on which the processing speed of the solver depends.

Algorithm Organization

The algorithm can be divided into two conceptual and organizational levels: (1) the global level, where general blocking rules are established and a general block-oriented solution algorithm is presented, and (2) the local level where sub-block structure forces tradeoffs between generality and speed.

Global Blocking. It is proposed that the blocking be performed on the LU map of the matrix since it is the structure of L and U rather than the structure of A which is important at each stage of the solution process. Further, it is proposed that the blocking be based on the size of diagonal blocks. In the general sparse case, these blocks are determined by scanning the matrix diagonal (in one direction) for the largest full, square blocks. This yields a unique diagonal blocking. Row and column strips are defined throughout the matrix based on these diagonal block partitions. Off-diagonal blocks are then defined only within the intersection of row and column strips. Figure 3 shows an example of this blocking.

With the blocks constrained to lie within such strips, it is possible to specify the block solution algorithm. In Figure 4, the factorization steps are given for a blocked matrix of the form

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n_b} \\ A_{21} & A_{22} & & \\ \vdots & & \ddots & \vdots \\ A_{n_b 1} & & & A_{n_b n_b} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{n_b} \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{n_b} \end{bmatrix} \qquad (5)$$

where the A_{ii} are square dense blocks and the sparsity of the $A_{ij} (i \neq j)$ is as yet unconstrained.

Local blocking. Define diagonal-based column and row strips (DBCS and DBRS, respectively) as those portions of column and row strips extending from each diagonal to the southern and eastern boundaries of the matrix, respectively. These strips contain all of the A_{ij} and A_{ii} blocks of Figure 4; it is their sparsities which are now of concern in the general sparse case.

An examination of the substitution step shows that any nonzero position in either blocks A_{ij} or A_{ki} will result in propagation or "raining" of nonzero positions to the eastern or southern boundary (respectively) of that block. This property is observed in the L and U map of Figure 3. This effect (a) fixes the contour along at least one of the sides of each block, and (b) requires the maximum length of any block along its DBCS or DBRS to be at its eastern or southern boundary, respectively (Figure 5).

Beyond this restriction, an arbitrary substructure of L and U is possible. To accommodate this generality in a solver requires block contour descriptors and, more importantly, slows the multiplication kernel of Figure 4. A compromise adopted for this study was to assume each subblock extends across its DBRS or DBCS as shown in Figure 5; however, any number of blocks may lie within an intersection of a row and column strip. This assumption is valid for common types of block structures or when blocking dense matrices. However, when special equation ordering is used to avoid fill [9], nonzeros may have to be added to extend blocks across the DBRS and DBCS; these are shown as darkened areas in Figure 3.

With the above considerations in mind, the following are proposed as descriptors of DBRS blocks.

(b1) the row position of the (1,1) block position;

(b2) the number of columns in the block; the number of rows is fixed by the diagonal block size;

(b3) the column strip number containing the block; this descriptor can be determined in a preprocessor step; it initiates the scan to locate A_{jk} in the multiplication step;

(b4) the address of the (1,1) block position in the packed matrix numeric array; this permits an arbitrary numeric block storage;

(b5) the address increment in the numeric array between the (i,j) and the (i,j+1) block position; this eliminates the need for contiguously-stored block columns.

These descriptors can completely describe the sparsity of the DBRS, under the above assumptions. A similar set of descriptors is used for the blocks in the DBCS with the terms "row" and "column" interchanged, except (b5) which remains the column address increment to represent the column storage of all blocks.

These descriptors are stored in a list in the processing order of the blocks. From the multiplication step of Figure 4, this order begins at the first DBCS, then the first DBRS, then the second DBCS, etc., always beginning at the diagonal block. This is often termed Crout processing order.

Storage Considerations

Whereas the symbolic pointers (b1-b3) are related to the order of block processing, the numeric pointers (b4-b5) are intended to allow an arbitrary block numeric storage. The result is that common compressed storage schemes can be accommodated; also, these pointers permit certain efficiencies to be taken in the blocking.

(a) Reducing the number of blocks. The restriction $q \leq 64$ in the numeric kernel of Figure 2 requires that row strips not exceed 64 in width. This is a universal restriction. Other of the previously mentioned global blocking restrictions were imposed to maintain the integrity of the numeric processing with arbitrary placement of blocks in the numeric storage. However, certain common storage schemes allow violation of these rules in the interest of efficiency. For example, a full matrix stored in column-ordered form in a numeric array could be partitioned as in Figure 6, with no column strip extending above the diagonal block. A single block multiply such as S*T, if directed to be accumulated into P, would, in fact, be accumulated into blocks P-Q-R. One only has to insure that descriptor (b5)--the address increment between columns of a block--is the pseudo-row dimension of the numeric array representing the entire matrix. Of course, this freedom to violate a global blocking rule involves the risk of improper accumulation across several blocks or indeed into void areas of a sparse matrix. For example, no check is made whether the user has accounted for all the block fill in a sparse matrix in setting up the block descriptor list.

(b) Compressed format. In Figure 7 (b)-(c), several standard formats of block tridiagonal matrices are given. Storage I is similar to that of banded matrices, where the diagonals of the matrix are stored as rows of a two dimensional compressed array. This storage is achieved simply by reducing descriptor (b5) for all blocks to one less than the row dimension of the entire matrix array; a rectangular block contour then is skewed into a parallelogram as shown. Figure 7 (d) shows that the numeric descriptors are different for the two storage schemes, but the symbolic pointers remain the same.

Performance Evaluation

Figure 4 depicts the major loops of the factorization program and yields, with Table 1, the corresponding timings of the overhead and kernel computations. These were developed using a timing simulator [10].

From these timings, it is possible to develop expressions for the solution time, execution rate, and the fraction of time--termed the efficiency η-- in the numeric kernels for any particular matrix structure.

For example, consider solution of a block tridiagonal matrix with the general solver. The execution time and efficiency are determined from Figure 4 to be, for n_b nxn blocks,

$$T_F = 345 + (n_b-1)(620 + T_f + T_s + T_m) + T_f$$

where T_f, T_s, and T_m are defined in Table 1, and

$$\eta = \frac{1}{1 + \gamma}$$

where

$$\gamma = \frac{345 + (n_b - 1)(620)}{T_f + (n_b - 1)(T_f + T_s + T_m)}$$

For a large number of blocks, the execution rate and the efficiency become simply

$$\text{MFLOPS} = \frac{80(28n^3 - 9n^2 + 5n)}{6(620 + T_f + T_s + T_m)}$$

$$\eta = \frac{1}{1 + 620/(T_f + T_s + T_m)} \qquad (6)$$

The efficiency of (6) is calculated in Table 2, where reasonable efficiencies are shown for $n_b > 2$.

An interesting aspect of the general solver is that significant effort can be justified in development of the numeric kernels. In [11], for example, it is shown that $\cong$ 50% speedup can be achieved in these kernels for small matrices (without penalty for large matrices) by avoiding vector chaining--through which the CRAY-1 normally achieves concurrency--and utilizing the vector registers instead as a dual cache memory which seeks to keep the floating point pipelines continuously busy. One can then view the speedup of these unconventional assembly language kernels as an asset which compensates for the overhead of list processing, the price for generality. An "adjusted efficiency" can then be obtained by multiplying the efficiencies of Table 2 by the kernel speedups. The product is the speedup of the general code over a code written for a block tridiagonal matrix with conventional assembly language kernels. These speedups are shown in parentheses in Table 2, for block sizes of four and eight, and range from 1.07 to 1.3.

From recent comparisons of Fortran and conventional assembly language codes [12], it is clear that speed comparisons of specialized Fortran-coded solvers and the general solver would result in an even greater advantage for the general solver. Thus, a user faced with programming a Fortran solver for a specific block structure would be well advised to consider use of this general solver requiring only a block descriptor input.

Conclusions

The goal of developing an efficient general equation solver for a memory-hierarchical vector processor appeared initially to yield the negative result that a block structure was necessary. While challenging to algorithmists traditionally interested in problems with random sparsity, this restriction risks application to a relatively small class of problems.

However, the application of a general block-oriented solver to regularly-blocked matrices was found appealing due to (1) the difficulty of specialized coding in a low level language and (2) the necessity for blocking large dense matrices in a memory-hierarchical environment. Indeed, it was shown that in at least some cases a general solver offered a speedup over a conventionally-coded specialized solver.

Extensions of the solver are planned to process band and profile matrices by developing a small library of specialized numeric kernels within the general solver. The solution efficiency of finite element and finite difference problems also needs study, although this will have to await a preprocessor which automatically generates a blocked LU map from the original matrix structure.

Acknowledgment

The author acknowledges the programming assistance of S. Arya, and the insight provided by a CRAY-1 simulator developed by D. A. Orbits. This work was sponsored jointly by the Mathematical and Information Sciences Directorate of the Air Force Office of Scientific Research, and by the Air Force Flight Dynamics Laboratory, Wright Patterson AFB under Grant 75-2812

References

[1]. W. F. Tinney, J. W. Walker, "Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization," Proc. IEEE, vol. 55, November, 1967, pp. 1801-1809.

[2]. R. P. Tewarson, "Solution of a System of Simultaneous Linear Equations with a Sparse Coefficient Matrix by Elimination Methods," BIT, vol. 7, 1967, pp. 226-239.

[3]. F. G. Gustavson, "Some Basic Techniques for Solving Sparse Systems of Linear Equations," Sparse Matrices and Their Applications, D. J. Rose and R. W. Willoughby Eds., Plenum Press, 1972, pp. 41-52.

[4]. G. von Fuchs, J. R. Roy, and E. Schrem, "Hypermatrix Solution of Large Sets of Symmetric Positive Definite Linear equations," Comput. Math. Appl. Mech. Engring., vol 1, 1972, pp. 197-216.

[5]. D. A. Calahan, "Vectorized Sparse Elimination," Proc. SCIE Meeting on Impact of Advanced Systems on Scientific Computation, Livermore, CA, Sept. 12-13, 1979.

[6]. D. A. Calahan, Computer-Aided Network Design, McGraw-Hill, 1972.

[7]. D. Boley, B. L. Buzbee, and S. V. Parter, "On Block Relaxation Techniques," Report #318, Computer Sciences Dept., University of Wisconsin, Madison, June, 1978.

[8]. R. Fong, T. L. Jordan, Some Linear Algebraic Algorithms and Their Performance on the CRAY-1, Report LA-6774, Los Alamos Scientific Laboratory, June, 1977.

[9]. A. George, "Numerical Experiments Using Dissection Methods to Solve n by n Grid Problems," SIAM J. Numer. Anal., vol. 14, April, 1977, pp. 161-179.

[10]. D. A. Orbits, A CRAY-1 Timing Simulator, Report #118, Systems Engineering Laboratory, University of Michigan, September, 1978.

[11]. W. G. Ames, et al, Sparse Matrix and Other High Performance Algorithms for the CRAY-1, Report #124, Systems Engineering Laboratory, University of Michigan, January, 1979.

[12]. J. J. Dongarra, Linpack Working Note #11: Some Linpack Timings on the CRAY-1, Report #LA-7389-M, Los Alamos Scientific Laboratory, August, 1978.

Square block size	Factorization Kernel M_f/T_f	Substitution Kernel M_s/T_s	Multiplication Kernel M_m/T_m
1	1.1/.073	.5/.15	2.7/.11
2	2.5/.20	3.0/.39	7.6/.16
3	6.9/.28	5.7/.63	17/.32
4	8.5/.42	12/.76	26/.40
6	18/.69	21/1.5	43/.78
8	23/1.1	30/2.6	60/1.4
12	45/2.0	46/5.8	84/3.3
16	60/3.5	58/1.1	98/6.7
32	94/18	89/58	124/42
64	123/112	117/358	141/298

Table 1. [Execution rate (MFLOPS)]/[execution time (kiloclocks)] of three numeric kernels.

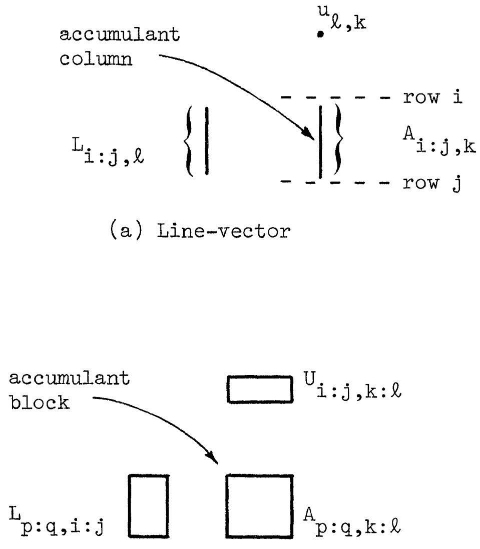


Figure 1. Accumulation models

Block size	MFLOPS	η
1	.33	.35
2	1.9	.55
3	5.0	.66
4	10.	.72(1.07)
6	21.	.83
8	32.	.90(1.3)
12	54.	.85
16	69.	.97
32	102.	.995
64	126.	.999

Table 2. Estimated performance of general block sparse solver on the CRAY-1 for a large system of block tridiagonal equations.

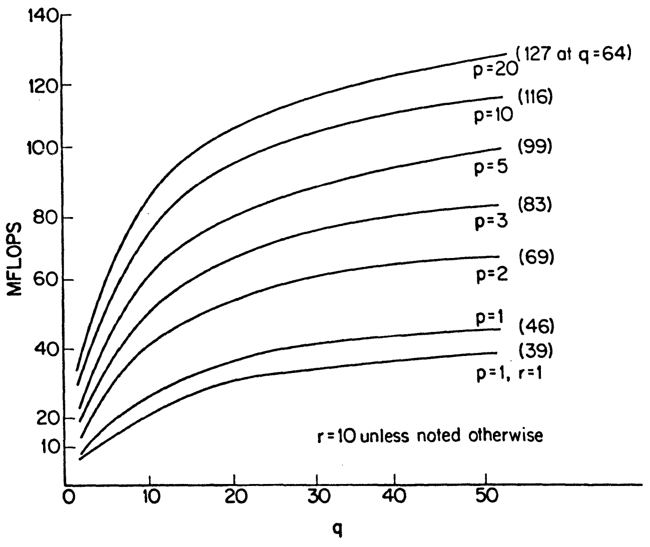


Figure 2. Execution rate of CRAY-1 for multiplication and accumulation of two qxp and pxr matrices. q is limited to 64 by vector length restriction

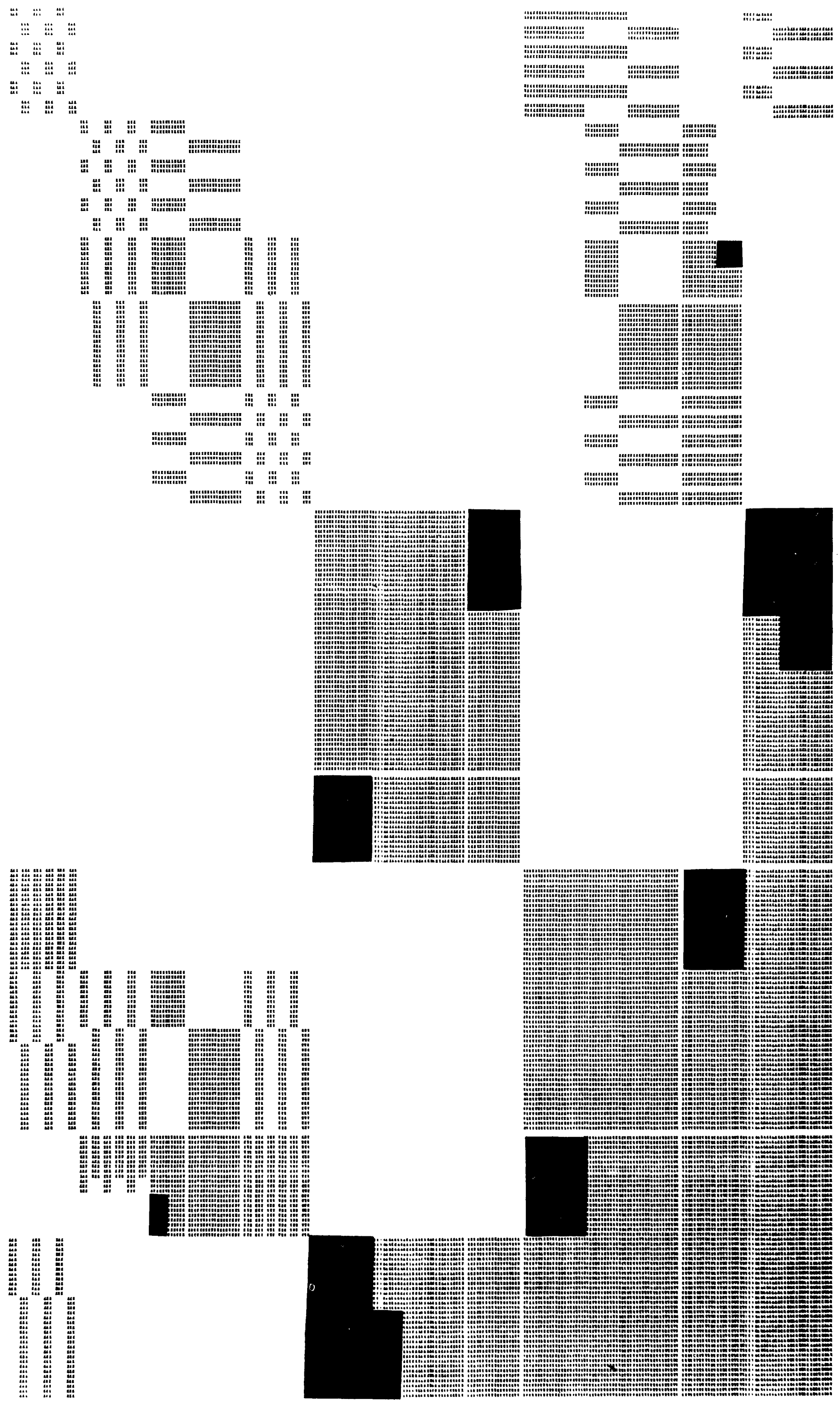
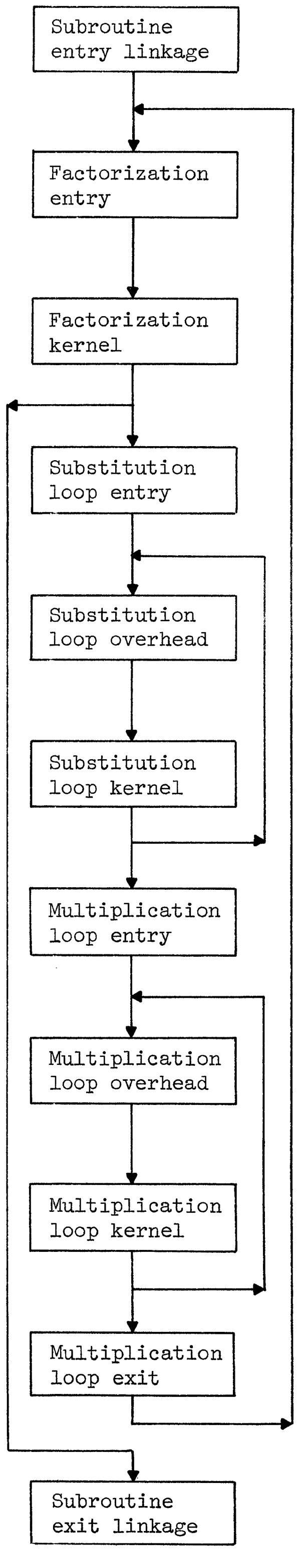


Figure 3. Block structure of dissected finite element matrix; boundary conditions account for some irregularities in block structure. Blackened regions indicate where non-zeros were added to conform to blocking rules.

Algorithm

```
Factorization

For i = 1,2, ... $n_b$

        Factor $A_{ii} = L_{ii}U_{ii}$

Substitution
    For j = i + 1, ... $n_b$

        $A_{ij} \leftarrow L_{ii}^{-1}A_{ij}$

    For k = i + 1, ... $n_b$
        $A_{ki} \leftarrow A_{ki}\,U_{ii}^{-1}$

Multiplication

    For j = i + 1, ... $n_b$

        For k = j, ... $n_b$

            $A_{kj} \leftarrow A_{kj} - A_{ki}\,A_{ij}$

    For j = i + 1, ... $n_b$
        For k = j + 1, ... $n_b$

            $A_{jk} \leftarrow A_{jk} - A_{ji}\,A_{ik}$
```



Flow Chart		Time (clocks)
Subroutine entry linkage	--	115
Factorization entry	--	105
Factorization kernel	--	See Table 1
Substitution loop entry	--	30
Substitution loop overhead	--	35
Substitution loop kernel	--	See Table 1
Multiplication loop entry	--	270
Multiplication loop overhead	--	130
Multiplication loop kernel	--	See Table 1
Multiplication loop exit	--	50
Subroutine exit linkage	--	125

Figure 4. Algorithm and timing for factorization portion of sparse solver code.

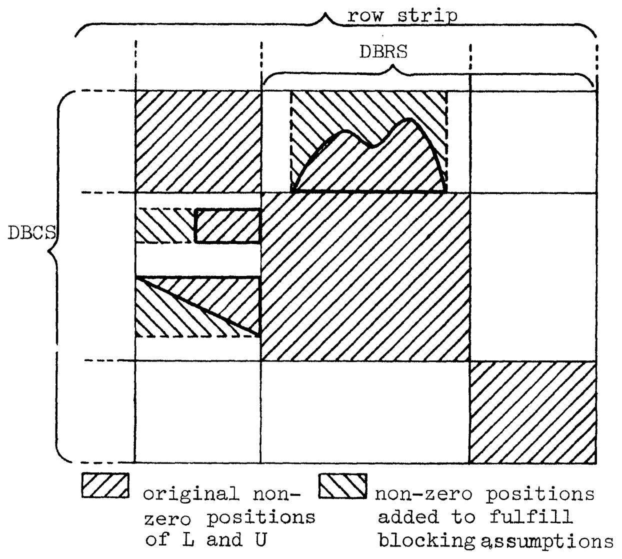

row strip
DBRS
DBCS
original non-zero positions of L and U
non-zero positions added to fulfill blocking assumptions

Figure 5. Local blocking assumptions

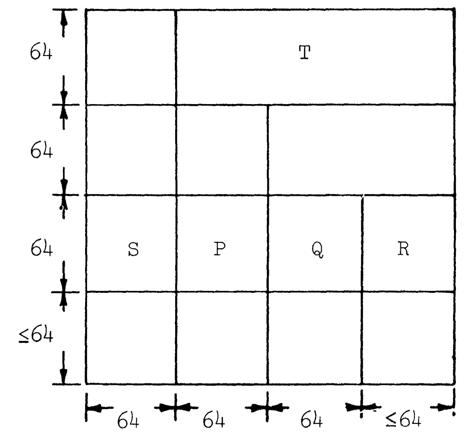
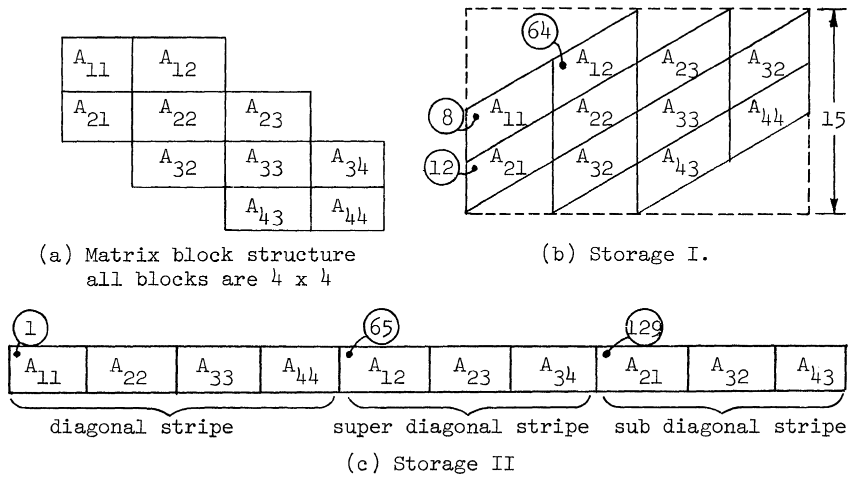


Figure 6. Simplified blocking of full matrix



Block	Storage	Symbolic descriptors			Numeric descriptors	
		b1	b1	b3	b4	b5
A_{11}	I	1	4	1	8	14
	II	1	4	1	1	4
A_{21}	I	5	4	2	12	14
	II	5	4	2	129	4
A_{12}	I	5	4	2	64	14
	II	5	4	2	65	4

(d) Descriptors

Figure 7. Block tridiagonal matrix and two storage descriptions. Circled numbers refer to numeric array addresses.

VLSI ARCHITECTURES FOR MATRIX COMPUTATIONS

by

Dr. Ellis Horowitz

Computer Science department

University of Southern California

Los Angeles, California 90007

6/13/79

Abstract The continuing revolution in the fabrication of integrated circuits has brought us to the point where soon one million devices will be placed on a single chip. However, this miracle of microminiaturization has not been followed with revolutionary new computer designs. In this paper we discuss how the potential of VLSI can be exploited in one particular domain, the processing of matrix computations. There are two major goals of any VLSI design which must be met for such a chip to be profitable. These are regularity of structure in the circuit layout and maximum overlap of computation during processing. In this paper we show how a single set of locally connected processors can be used to achieve a maximum overlap of computation for all of the following problems: (i) a matrix times a vector, (ii) LU - decomposition, and (iii) back substitution. Also we lay the groundwork for analyzing the time complexity of parallel algorithms executed by VLSI circuitry. Finally we introduce some high level programming language notation for expressing the execution of algorithms on multiprocessors.

Key Words: VLSI, large scale integrated circuits, matrix operations, parallel algorithms.

This research has been supported by the National Science Foundation under grant number MCS78-01798

We begin by considering the problem of multiplying a matrix times a vector. We assume the matrix is an n x n with elements $a_{i,j}$ and a vector $x = x_1,\ldots,x_n$. We imagine a set of processors which contain three registers: R_a, R_b, and R_c. Each processor has three inputs and three outputs, one each to each register. As this circuit is to be placed on a single chip the number of external pins is limited and for this circuit we assume only two input pins and three output pins. A partial view of this circuit is drawn below. One sees two rows of n processors each. Arrows indicate register connections between processors. The fact that all the connections are local is extremely important for VLSI design, see [Mead78]. Register R_a connections are not needed for this problem so they are not drawn.

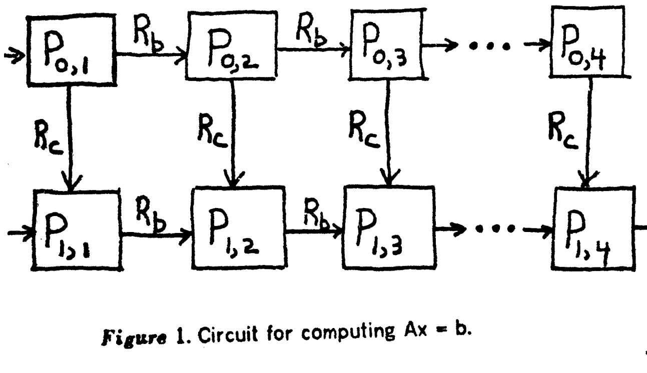


Figure 1. Circuit for computing Ax = b.

Now, what are the operations that we require of the registers in these processors. All of the registers can accept and send data over their register connection lines. In addition each register can transfer its data to another register in the same processor. Finally each register can perform the operation $R_c \leftarrow R_c + R_a * R_b$ (and similarly for R_a and R_b), plus the operation $R_c \leftarrow R_c/R_b$ (or $R_b \leftarrow R_b/R_c$ or $R_a \leftarrow R_a/R_b$).

We assume that all of the operations take the same amount of time, which we call one time unit. We also assume that all of the processors execute some instruction at every time unit, though that instruction may be a "do-nothing" instruction. The algorithms presented here are assumed to reside in the processors so no broadcasting is necessary. Note that the algorithms given here are not systolic in the sense of Kung and Leiserson, [Kung79], but they definitely have been motivated by that paper. The algorithm for computing Ax now follows.

```
zero out all registers
for i ← 1 to n do
  in parallel do            //all statements in the scope of an in parallel//
    read $x_i$ into $R_b(P_{0,k})$, $1 \le k \le n$.   //statement are executed concurrently//
    read column i into $R_b(P_{1,k})$, $1 \le k \le n$.
    $R_c(P_{0,k}) \leftarrow R_b(P_{0,k})$,  $1 \le k \le n$.
  end
    $R_c(P_{1,k}) \leftarrow R_c(P_{0,k})$, $1 \le k \le n$.   // This operation is called Shift-down //
    $R_a \leftarrow R_a + R_b * R_c$ for all processors in row 1.  // multiply-and-add //
repeat
$R_b(P_{1,k}) \leftarrow R_a(P_{1,k})$,  $1 \le k \le n$.
output $b_1,\dots,b_n$ from $R_b(P_{1,k})$ // move-right-row1 n times //
```

Algorithm 1. Computing Ax = b.

In figure 2 one sees snapshots of register R_a for the n processors in row one immediately following the multiply-and-add operation.

$R_a(P_{1,1})$	$R_a(P_{1,2})$	$\cdots$	$R_a(P_{1,n})$
$a_{n1}x_1$	$a_{n-1,1}x_1$	$\cdots$	$a_{1,1}x_1$
$a_{n1}x_1+a_{n2}x_2$	$a_{n-1,1}x_1+a_{n-1,2}x_2$	$\cdots$	$a_{11}x_1+a_{12}x_2$
$\vdots$	$\vdots$		$\vdots$
$a_{n1}x_1+\cdots+a_{nn}x_n$	$a_{n-1,1}x_1+\cdots+a_{n-1,n}x_n$	$\cdots$	$a_{1}x_1+\cdots+a_{1n}x_n$

Figure 2.

If we consider the complexity of this algorithm we first observe that no algorithm can require less than n^2 time units as at least n^2 elements must be read into the chip. Moreover one can show that at least $n^2 + n$ units of time are required before the final output is produced at the output gate. (Consider the last element of A which is input at time n^2 and must at least move to the output gate.) Examination of the body of the *for* loop shows that everything within the in-parallel statement is accomplished in n^2 units of time. The remaining two steps in the *for* loop take unit time each pass for a total of 2n. Outputting the final values takes another n+1 steps plus the time to zero out the registers takes one step, so the total time taken is $n^2 + 3n + 1$. Only an additional $2n + 1$ units of time are required over the theoretical minimum. Therefore we say that the complexity of this algorithm is $2n + 1$.

We now consider the second of the three problems. Given a lower triangular matrix L with elements $a_{i,j}$ for $i \ge j$ and a vector $b = b_1,\dots,b_n$ to determine the values of the vector x such that $Lx = b$. Once again we assume the same configuration of processors having the same capabilities. The algorithm follows.

```
zero out all registers
for i ← 1 to n do
  in parallel do        //total time is n(n+1)/4 //
    read $a_{2i-k,k}$ into $R_b(P_{0,k})$   $1 \le k \le i$.
    $R_c(P_{0,k}) \leftarrow R_b(P_{0,k})$
     Shift-down
    read $b_{2i-1}$ into $R_b(P_{1,1})$
  end
    $R_b \leftarrow R_b - R_a * R_c$  except $P_{1,i}$ where $R_b \leftarrow R_b/R_c$
    $R_a(P_{1,i}) \leftarrow R_b(P_{1,i})$      //save new $x_i$ //
    $R_b(P_{1,k}) \leftarrow R_b(P_{1,k-1})$    //move-right-row1 //
  in parallel do        // total time is n(n+1)/4 //
    read $a_{2i-k+1,k}$ into $R_b(P_{0,k})$   $1 \le k \le i$.
    $R_c(P_{0,k}) \leftarrow R_b(P_{0,k})$
     Shift-down
    read $b_{2i}$ into $R_b(P_{1,1})$
  end
    $R_b \leftarrow R_b - R_a * R_c$
     $R_b(P_{1,k}) \leftarrow R_b(P_{1,k-1})$ //move-right-row1 //
repeat
$R_b(P_{1,k}) \leftarrow R_a(P_{1,k})$ ,  $1 \le k \le n$.
output $x_1,\dots,x_n$  from $R_b(P_{1,k})$
```

Algorithm 2. Computing x: Lx = b.

Figure 3 shows snapshots of the algorithm as the computation progresses. As for the complexity we observe that the matrix A contains n(n+1)/2 elements which is the total time for both read statements which input A. The time to read in the b values is overlapped with the reading of A.

The remaining statements within the main *for* loop each take n units of time for a total of 5n, zeroing out the registers takes unit time as does the transfer from register R_a of row one to register R_b. The actual output takes an additional n units of time and totalling these gives n(n+1)/2 + 5n + 1 + 1 + n = n(n+1)/2 + 6n + 2. Using an argument similar to the one used before, a lower bound is given by n(n+1)/2 + n + 1. Therefore the complexity of this algorithm is no more than 5n + 1.

i	row	
1	0	$(-, a_{11}, a_{11})$
	1	$(-, b_{11}/a_{11}, a_{11})$
	0	$(-, a_{21}, a_{21})$
	1	$(x_1, b_2 - a_{21}x_1, a_{21})$
2	0	$(-, a_{31}, a_{31})\ (-, a_{22}, a_{22})$
	1	$(x_1, b_3 - a_{31}x_1, a_{31})(x_2, \frac{b_2 - a_{21}x_1}{a_{22}}, a_{22})$
	0	$(-, a_{41}, a_{41})(-, a_{32}, a_{32})$
	1	$(x_1, b_4 - a_{41}x_1, a_{41})(x_2, b_3 - a_{31}x_1 - a_{32}x_2, a_{32})$
3	0	$(-, a_{51}, a_{51})(-, a_{42}, a_{42})(-, a_{33}, a_{33})$
	1	$(x_1, b_5 - a_{51}x_1, a_{51})(x_2, b_4 - a_{41}x_1 - a_{42}x_2, a_{42})(-, x_3, a_{33})$
	0	$(-, a_{61}, a_{61})(-, a_{52}, a_{52})(-, a_{43}, a_{43})$
	1	$(x_1, b_6 - a_{61}x_1, a_{61})(x_2, b_5 - a_{51}x_1, a_{52})(x_3, b_4 - a_{41}x_1 - a_{42}x_2, a_{43})$

3-tuple = (R_a, R_b, R_c)

Figure 3. Computation of x given Lx = b, L lower triangular

We now consider the last of the three problems. Given a matrix A we are to determine a factorization A = LU where L is a lower triangular matrix and U is an upper triangular matrix. One way to produce this factorization is to use Gaussian elimination. If we assume that the matrix is either symmetric, positive definite or irreducibly diagonally dominant (as first observed in [Kung 79]) then pivoting is not necessary. A sequential version of Gaussian elimination without pivoting follows.

```
for k ← 1 to n do
    for i ← k+1 to n do
        $m_{i,k} ← a_{i,k}/a_{k,k}$
        for j ← k+1 to n do
            $a_{i,j} ← a_{i,j} - m_{i,k}\, a_{k,j}$
        repeat
    repeat
repeat
```

Algorithm 3. Sequential Gaussian elimination.

When executed on a sequential processor the complexity of this algorithm is $O(n^3)$. The $m_{i,k}$ are the values of the upper triangular matrix U, while the new values of $a_{i,j}$ are the values of the lower triangular matrix L. As there are n^2 elements to be input we cannot expect to do any better than to overlap all of the computation as the input values are being read. The circuit for this computation is given below. Observe that this circuit contains in its first two rows the circuit we were using to compute b = Ax and to compute x where Lx = b.

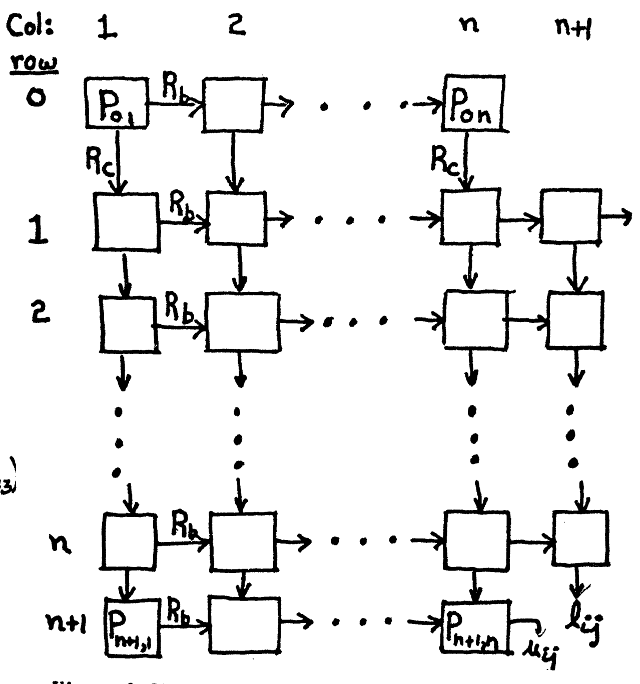


Figure 4: Circuit for determining LU = A, Ax and x: Lx = b.

In figure 4 we assume that the processors are numbered as P(0:n+1, 1:n+1) with the input processors in row zero and row one, and the output processors in row n + 1 and column n + 1. Before we present the algorithm we need to explain some abbreviations. The Shift Down operation is now generalized and is equivalent to $R_c(P_{i,j}) \leftarrow R_c(P_{i-1,j})$ for $1 \le i,j \le n + 1$. The algorithm follows.

```
zero out all registers ; $R_a(P_{1,1}) ← a_{1,1}$
for i ← 1 to 2n-1 do
    if i ≤ n then $R_b(P_{1,1})$ ← first element of row i
                  $R_b(P_{1,1}) ← R_b(P_{1,1})/R_a(P_{1,1})$
    endif
    for j ← 0 to FLOOR(i/2) in parallel do
        $R_b(P_{j,k}) ← R_b(P_{j,j})$ for $j+1 ≤ k ≤ n$
        if $i > n$ then $R_b(P_{n,k})$ goes to output for $i - n \le k \le n$.
    repeat
    $R_c(P_{0,k}) \leftarrow R_b(P_{0,k})$
    Shift down
    Multiply and Add //$R_c \leftarrow R_c - R_a * R_b$ //
    if odd(i) then $R_a(P_{CEIL(i/2),k}) \leftarrow R_c(P_{CEIL(i/2),k})$, $CEIL(i/2) \le k \le n$
    for j ← 2 to CEIL(i/2) in parallel do
        $R_c(P_{j,j}) \leftarrow R_c(P_{j-1,j})$
        $R_b(P_{j,j}) \leftarrow R_c(P_{j,j})/R_a(P_{j,j})$
    repeat
    if odd(i) then $R_c(P_{CEIL(i/2),k}) \leftarrow R_a(P_{CEIL(i/2),k})$
repeat
```

Algorithm 4. Computing L,U: A = LU.

Examining the complexity of this problem we observe that n^2 elements must be input, so that is an obvious lower bound. In the algorithm each row is input separately for a total of n^2 units. At the same time as the input is being read in, register R_b in the processors on the main diagonal of the grid are having their values propogated across their rows. This is done simultaneously with the inputting or the outputting. Thus the total time for the first inner *for* loop is n^2. The operations described by the second inner *for* statement are also done in parallel, for a total cost of 4n-2. All other statements within the main *for* loop take n units to total time. Summing up we get $1 + 1 + 2n + n^2 + 3n + n + 4n-2 + n = n^2 + 11n$. Below is a snapshot of the algorithm as it processes a 4 x 4 matrix.

Conclusions

For all of the three problems we have seen, a single chip design was used to compute the solutions. Each of the processors on the chip requires a small set of simple capabilities and they are locally connected. The computing time analyses have intentionally included I/O as well as processing time as we feel this is an especially critical parameter for VLSI. The algorithms given here show that methods can be devised which overlap almost all of the computation with the input and output.

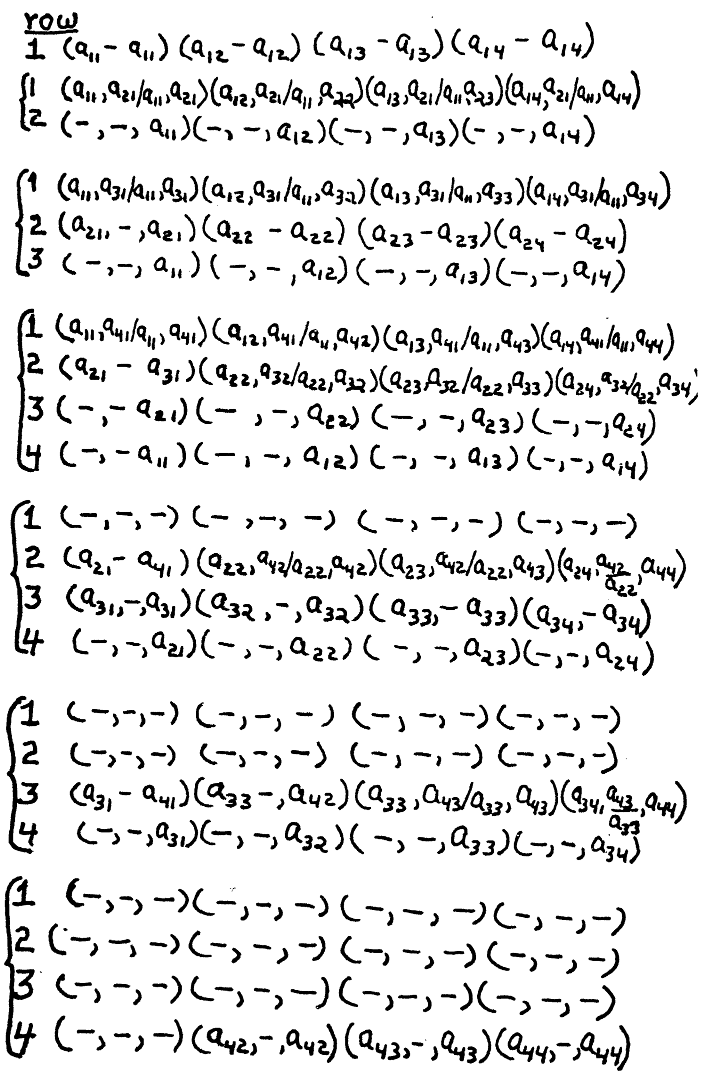


Figure 4. Computation of L, U such that A = LU.

References

[Mead78] C.A. Mead and L.A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (to appear in 1979). In particular Chapter 8, "Highly concurrent systems".

[Kung79] H.T. Kung and Charles E. Leiserson, "Systolic arrays (for VLSI)", Computer science research reviews, Carnegie-Mellon University, 1977-78.

EFFICIENT FUNCTION IMPLEMENTATION FOR BIT-SERIAL PARALLEL PROCESSORS

Anthony P. Reeves and John D. Bruner
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Abstract -- Parallel processors with bit-serial PE's usually implement arithmetic functions by a sequence of word parallel arithmetic operations; however, basic operations must be specified at the bit-serial level. In this paper the possibility of more efficiently implementing a function with a special, tailored sequence of bit-serial operations is considered. A general scheme is described for generating efficient programs to implement arbitrary functions on bit-serial-arithmetic processors. This scheme is based on logic design methodology and involves designing a logic network to realize a desired function. The parallel processor is then used to efficiently simulate a set of these networks. Heuristic design algorithms are used to generate the logic networks; several algorithms are described and compared with some benchmark functions.

This scheme is suitable for implementing any arbitrary function, however the computation cost increases very rapidly with the number of inputs to the function. Examples are given of some 8-input function implementations.

Programs may be generated for most bit-serial parallel processors. Several efficient PE designs are described and analyzed.

Introduction

Bit-serial parallel processors usually specify arithmetic functions at the word parallel level. However, basic operations must be specified at the bit serial level which opens the possibility of developing special, tailored, instruction sequences for implementing specific functions. On conventional computers, operations are at the word level and very efficient multiply and divide operations are usually available; moreover, for fast function implementation a table look-up method can be used. Fast multiplication and division is difficult to implement on bit-serial parallel processors and an indexing mechanism, necessary for a table look-up scheme, is usually not available.

A function may be considered as a mapping rather than an arithmetic expression. For example, in an image processing application a logarithmic transform of an image may be required; typically the image may consist of 8-bit integer brightness values and the desired result may also be a scaled 8-bit integer. Therefore we wish to compute the function $\lfloor 255 \cdot \log(X+1)/\log(256) \rfloor$ where X is the value of each element of the image. The conventional approach to this problem is to use an arithmetic approximation technique to compute the function which must be implemented with sufficient precision to avoid round off errors. An alternative, more direct, approach is to consider the function as a specified mapping from 8 inputs to 8 outputs and achieve this mapping in one stage with a sequence of single bit operations. This direct approach is useful for existing bit-serial parallel processors such as STARAN [1] and also for emerging parallel processors based on LSI technology such as CLIP4 [2], MPP [3] and DAP [4].

A general scheme for implementing arbitrary functions on a bit-serial SIMD processor is described. To utilize this scheme the PE's of the processor must be capable of computing logical functions of 2 operands; however, only a subset of the 16 possible Boolean functions is required.

The general procedure for implementing a function is as follows: First a truth table of all possible outputs for all possible inputs is created. A conventional computer could use a table-look-up algorithm at this stage; however bit-serial SIMD processors usually cannot efficiently implement such algorithms. In a SIMD processor each PE would have to store the complete truth table; moreover each PE would also need a special, individual indexing mechanism to access the table data.

Second, an efficient program is developed from the truth table which will implement the desired function. Several algorithms for automatically generating efficient programs from the truth table are described in this paper; these algorithm are based on logic design methodology. A logic network is designed which will realize the desired truth table, then this design is translated into a program sequence for the parallel processor. Typically, one processing instruction on a bit-serial PE may be considered to simulate a two-input logic gate.

A suitable design algorithm should generate networks with the following characteristics:

(a) The total gate count in the network should be a minimum since each gate implies at least one process instruction in the final program sequence.

(b) The network can consist only of two input gates; however, many different Boolean functions may be available at equal cost.

(c) The number of logic levels in the network is unimportant. In fact, a cascade of N 2-input gates can usually be realized more easily than a balanced binary tree of N gates.

(d) Usually multi-output logic networks will be required.

Currently there is no simple, efficient design algorithm which will generate a minimum logic network with the above characteristics. An exhaustive selection algorithm would require too much computation for practical sized problems. Several different heuristic algorithms have been developed for designing efficient multi-output-logic networks with 2-input gates.

An alternative design strategy is to create networks with Universal logic modules (ULM's). A ULM is a 3-input device having two data inputs and one selection input. In some processor designs a one-bit mask register is associated with each PE which, when set, inhibits a PE from processing data. These processors can efficiently simulate a network of ULM's by using the mask register for the selection input.

The main advantage of the described scheme is its generality in that a program for any arbitrary function may be generated including a random mapping. The two main problems with the technique are: (a) the computation time for the logic design algorithms is proportional to two to the power of the number of inputs, and (b) the logic design algorithms do not generate minimum designs and in some cases a more efficient program can be developed with analytic methods, e.g. a program for bit-serial addition. Several illustrative functions which have been programmed with the logic design scheme are discussed.

Logic Design Algorithms

Two schemes for designing networks with 2-input gates have been investigated. In the first scheme a two-level minimal sum of products design is decomposed into a network of 2-input AND gates followed by 2-input OR gates. In the second method the best Reed-Muller-Unate expression of the logic function is decomposed into 2-input AND gates followed by 2-input Exclusive-OR gates. Both of these schemes have been extended to multi-output designs.

An alternative approach is the method of disjunctive decomposition [5] [6]. However, those techniques work best for disjunctive functions which do not frequently occur in practice. Moreover, it is more difficult to make such schemes share logic in multi-output designs.

Several minimization algorithms have been developed for networks of Universal Logic Modules (ULM's). A recent paper by Voith [7] describes a minimization scheme for single control input ULM's. Some single output functions have been realized with ULM's; this design method, like disjunctive decomposition, is difficult to extend to efficient multi-output designs.

AND-OR Network Designs

For the AND-OR network design, the prime implicants for the required function are generated by a recursive concensus tree algorithm [8]. The following simple algorithm is then used to select an irredundant cover. It uses three lists: a list of selected implicants, SI; a list of remaining implicants, RI; and a list of terms to be covered, TC.

Prime implicant selection algorithm:

1. Set SI to all essential prime implicants; set RI to all non-essential prime implicants; and set TC to all minterms in the desired function which are not covered by essential prime implicants.
2. Select the prime implicant from RI which covers the greatest number of minterms in TC.
3. Add the selected prime implicant to SI and delete from TC all covered minterms.
4. If TC is empty, the selection is complete and SI contains the required irredundant cover. If TC is not empty return to step 2.

The following general gate selection algorithm, which can decompose a set of terms into two input gates, is then used to decompose all the product terms into 2-input AND gates. Some of the 2-input AND gates which are connected to inputs of the function may have inverted inputs.

Gate Selection Algorithm:

1. Every possible 2-input gate between function inputs (including any intermediate gate outputs) which could contribute to any of the prime implicants is examined. The number of implicants to which each of these gates contribute is recorded.

2. The gate which contributes to the most prime implicants is selected. The representations of the prime implicants which are affected by the selected gate, are modified to indicate its presence.

3. If all prime implicants are completely implemented then the selection is complete; otherwise go back to step 1.

For a single output network, the product terms may be connected by a cascade of OR gates. For multiple-output networks, the product terms for all outputs are first generated, then a network of OR gates is generated using the gate selection algorithm described above. In this case the inputs to the OR gates are the outputs of the product terms and the outputs of the OR gates are the function outputs.

AND-Exclusive-OR Network Design

In this section networks consisting of a set of product terms followed by a set of exclusive-OR gates are considered. It has been conjectured that networks of this form may be constructed more economically than the conventional AND-OR form; algorithms have been developed for minimizing a subset of the AND-exclusive-OR networks called the generalized Reed Muller canonic forms [9-11]. An important feature of these canonic forms is that they are unate, i.e. an input may occur either in its true or complemented form but not both for the same function.

A common expression for a Reed Muller expansion is:

$$f(x_1 \cdots x_n) = a_0 \oplus a_1 \dot{x}_1 \oplus a_2 \dot{x}_2 \cdots a_n \dot{x}_n \oplus a_{n+1} \dot{x}_1 \dot{x}_2 \oplus \cdots a_{2^n-1} \oplus \dot{x}_1 \dot{x}_2 \cdots \dot{x}_n$$

Where a_i are 0,1 coefficients and $\dot{x}_i$ is consistently either x_i or $\overline{x}_i$.

There are a maximum possible of 2^n terms in each expression. Also, for a completely defined function there are 2^n different canonic forms, one for each possible arrangement of true and complemented inputs. Conventionally, a minimum unate network is determined by selecting the canonic form containing the minimum number of terms; this results in a minimal unate design. For a two input gate network the product terms are decomposed with the gate select algorithm and a cascade of exclusive-OR gates is used to combine the product terms. Due to the unate property of the product terms a large amount of sharing of two input gates between the product terms is usually possible.

For multi-output designs two algorithms have been investigated. In the first, called multi-polarity, the terms for the best canonic form for each output are selected and the gate selection algorithm is used to implement them. Then the gate selection algorithm is used to generate a network of exclusive - OR gates in a similar way to the AND-OR multi-output design. As a different canonic form may be selected for each of the outputs, the multi-output network is no longer unate. The second algorithm determines which canonic form requires the least number of terms to generate all the outputs. The multi-output network is then generated in a similat way to the multi-polarity algorithm except that the same canonic form is used for all outputs; this results in a unate multi-output design. The unate scheme has the advantage that the product terms are selected from the same set for each output which increases the possibility of two or more outputs sharing similar terms. The problem with the unate algorithm is that some outputs may have a very simple realization with a different canonic form.

One fact which is not used in either algorithm is that, unlike the conventional product of sums canonical form, the number of variables in each product term is not a constant. Therefore in selecting the canonic form to be implemented, the cost of its constituent product terms should be considered.

Universal Logic Module Network Design

An algorithm for designing networks of arbitrary functions with ULM's has been described by Voith [7]. We have developed the following simple recursive algorithm which is currently being evaluated.

ULM Minimization Algorithm:

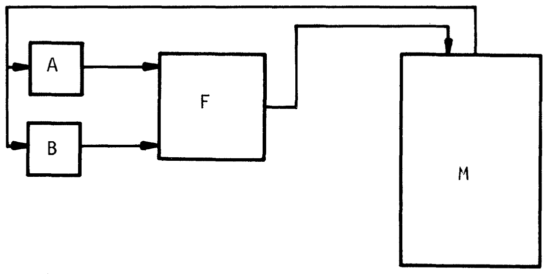
1. Find the input which is most correlated with the output, i.e. select the input which has the most matches with the output in either true or complemented form.
2. Use this input x_i to control a ULM which partitions the function into two sub-functions

   $f_L(X) = x_i \cdot f(x_1 \cdots x_n)$ $\quad f_r(X) = \overline{x}_i \cdot f(x_1 \cdots x_n)$
3. Apply this algorithm recursively to $f_L(X)$ until all inputs are constant 1 or 0.
4. Apply this algorithm recursively to $f_r(X)$ until all inputs are constant 1 or 0.

For the worst case the above algorithm would generate a binary tree of n levels involving 2^n-1 ULM's. However, the lowest level can always be removed by applying the last variable values to the inputs of the next level; therefore a maximum of $2^{n-1}-1$ ULM's could be required. Moreover, by choosing a suitable selection strategy at step 1 of the minimization algorithm the total number of ULM's required is usually significantly less than the worst case. So far this algorithm has only been used to generate single output networks; there is no simple way to extend to efficient multi-output designs.

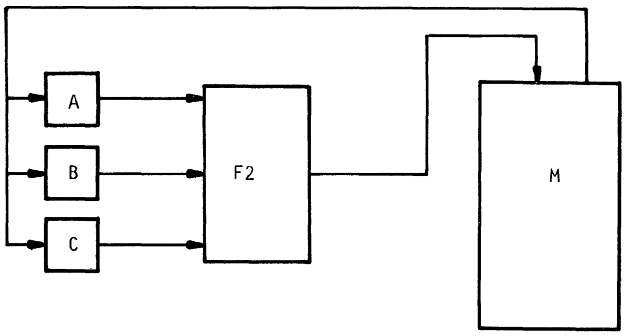
PE Architectures

In this section several possible PE architectures for efficiently simulating logic networks



PE1 basic architecture

Fig. 1

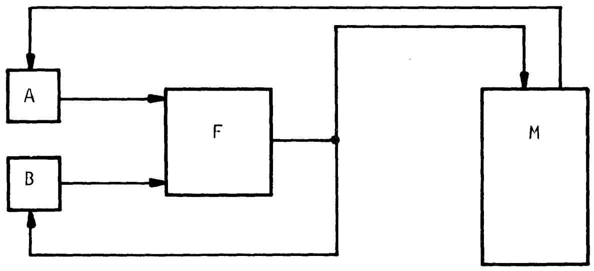


PE2 BASE architecture

Fig. 2

are considered. The basic functional units of a PE are shown in Fig. 1. Operands (gate inputs) are obtained from a 1-bit-wide memory M and each two-input gate is simulated by a Boolean processor F, which can realize any of the 16 possible functions of its inputs. Two single-bit registers, A and B are used to temporarily hold the inputs for F. A single isolated gate requires three memory cycles: two to load A and B, and one to store the result.

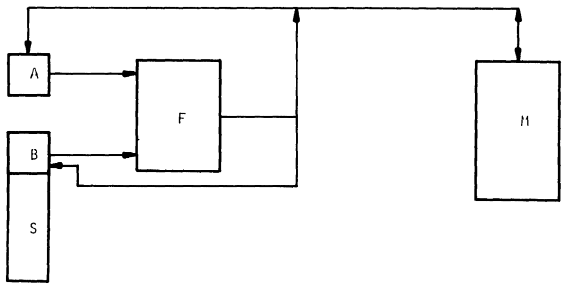
The implementation of a single output network of N gates with a fan-out limit of 1 will be discussed first. The more general case for a multi-output function involving gates with a fan out greater than 1 will then be considered. The architecture PE1 can implement any network in 3N memory cycles. One way to improve the PE performance is to use the BASE architecture [12] shown in Fig. 2. In this case a three-input, general Boolean processor, F2, is used. A pair of connected 2-input gates can be simulated in an operation involving four memory cycles. Therefore, any network can be implemented in 2N memory cycles.



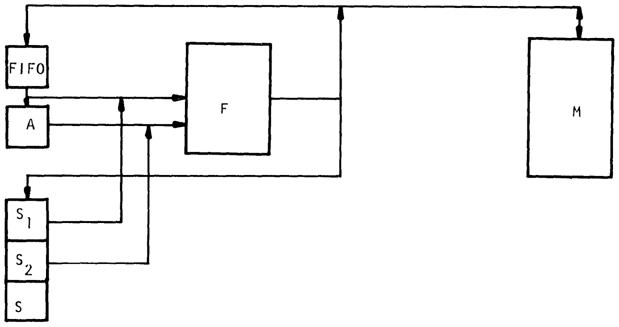
PE3 accumulator architecture
Fig. 3

In the architecture PE3 shown in Fig. 3, one of the holding registers, B, is used as an accumulator. The B register is not directly connected to M; however, it may obtain data from M through A and F without involving any more memory or clock cycles; when B is loaded from A, A is loaded from M. This scheme only involves the same number of control inputs (7 + memory address) that PE1 requires. In fact one control input may be discarded, as the enable inputs for the A and B registers always have the same value. The efficiency of this PE depends upon the topology of the network to be implemented. Two extreme cases exist: a cascade network which is the best case and a balanced tree network which is the worst case. Note, this is the reverse of the conventional logic design scheme where it is important to minimize the number of logic levels. The PE3 architecture requires N+2 memory cycles to implement a cascade network and a worst case of 2N+1 memory cycles to implement a balanced tree network.

Various alternative architectures have been developed for more efficiently implementing balanced tree networks. A simple modification to PE3 is shown in Fig. 4; the register B is now the top element of a stack S. An extra control signal is required so that A may be loaded with the output from F. This architecture requires 1·5(N+1) memory cycles to implement a balanced



PE4 Stack accumulator architecture
Fig. 4



PE5 Optimal balanced tree architecture
Fig. 5

binary-tree network. An optimal architecture is shown in Fig. 5, it involves a FIFO buffer and additional selection logic. The minimum number of memory cycles to implement a balanced tree, with a single memory M and a two-input Boolean processor F is $N+1 + \log_2 (N+1)$.

The main characteristics of the various architectures shown in Figs. 1-5 are summarized in Table 1. The worst case for a multi-output network with K outputs occurs when there are K disjoint networks with N/K gates each. If a gate has a fan out greater than one then an extra memory cycle is required to store the output of this gate into M. Therefore, if R gates in a network have a high fan-out then R additional memory cycles will be required. The worst case number of memory cycles to implement a network having K outputs and R high-fan-out gates is given in Table 1.

Designs with ULM modules usually result in tree networks. In Table 2 the number of memory cycles for implementing a balanced tree of N ULM's is given. The performance of the PE's is considerably improved if an extra register is used to store the selection input (plus a small amount of additional logic). The performance of the PE's with this modification is also given in Table 2.

Results

Several benchmark functions have been used to evaluate the various design schemes; these are described in Table 3. The gate counts for implementing these functions with different algorithms are given in Table 4.

	No. of memory cycles			Control inputs
Architecture	Cascade	Balanced Tree	Worst Case	(excluding M address)
PE1	3N	3N	3N	7
PE2	2N	2N	2N	12
PE3	N+2	2N+1	2N+K+R	6
PE4	N+2	1•5(N+1)	1•5(N+K)+R	7
PE5	N+2	$(N+1)+\log_2(N+1)$	$N+K+R+K\log_2(N/K+1)$	9

Table 1.

Performance of PE Architectures

	No. of Memory Cycles	
Architecture	Original design	With extra register
PE1	8N	4N
PE2	4N	-
PE3	6N	3N+1
PE4	5•5(N+1)	2•5(N+1)
PE5	$3N+1+\log_2(3N+1)$	$2N+1+\log_2(2N+1)$

Table 2.

Efficiency of PE Architectures
for a Balanced Tree of N-ULM's

No.	Function	No. of inputs	No. of outputs
1	X^2	4	8
2	$\sqrt{X}$	8	4
3	$\sqrt{X^2+Y^2}$	4 each	5
4	$\lfloor 255\cdot\log(X+1)/\log(256)\rfloor$	8	8
5	$\lfloor 255\cdot X^{0.4}/255^{0.4}\rfloor$	8	8
6	random function	4	4
7	random function	8	8

Table 3
Test Functions

	AND-OR		AND-Exclusive-OR			ULM (single output)
				multi-ouput		
Function	single output	multi output	single output	multi polarity	Unate	ULMs + 2-input Gates
1	26	22	20	17	26	4 + 12
2	105	105	71	70	60	14 + 33
3	573	485	757	651	435	97 + 158
4	686	530	812	702	396	120 + 185
5	779	610	997	968	415	140 + 208
6	39	35	38	33	30	8 + 14
7			1505	1228	502	342 + 350

Table 4
Gate Count for Different Algorithms

For the AND-OR networks figures are given for a set of single output networks (one for each output) and for a multi-output design. For the AND-Exclusive-OR networks figures are given for both the multi-polarity and unate multi-output algorithms. The multi-output designs are consistently better than the single output designs and for all 8 input functions the unate AND-Exclusive-OR algorithm gives significantly the best results.

The number high fan-out gates affects the cost of implementing a network. However, the number of high fan-out gates for the AND-Exclusive-OR unate implementations of the log function is only 32 or 7.6% of the total gates required.

In the lst column of Table 4 figures for single output ULM designs are given. It was found that more than half the ULM's in each design could be reduced to 2-input gates; therefore figures for each ULM design are given as a number of ULM's plus a number of 2-input gates. The gate counts for ULM designs compare well with the other algorithms; however, the ULM is a 3-input device and requires at least one more memory cycle to implement than a 2-input gate.

Conclusion

A scheme for implementing arbitrary functions on bit-serial parallel processors has been presented. Several relevant, heuristic logic design algorithms and special PE architectures have been described and evaluated. The PE architectures of most current bit serial parallel processors are suitable for implementing this scheme and a small improvement in performance could be achieved by adding a stack mechanism to the PE's.

Results from current design algorithms indicate that an arbitrary 8-input-8-output function could be implemented in less than 1000 memory cycles on many bit-serial PE architectures. This is an efficient way to implement functions with a complex form; however if the function has a simple arithmetic solution, e.g. addition or two's complement, then conventional arithmetic techniques may be more efficient. Further work is needed to improve the heuristic logic design algorithms.

References

[1] Batcher K.E. "STARAN Parallel Processor System Hardware" National Computer Conference, 1974, pp. 405-410.

[2] Duff M.J.B. "CLIP4: A Large Scale Integrated Circuit Array Parallel Processor" 3rd International Joint Conference on Pattern Recognition, pp. 728-832, 1976.

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

[4] Fung L., "A Massively Parallel Processing Computer," High Speed Computer and Algorithm Organization, Academic Press, 1977, pp. 203-204.

[5] Butler J.T. and Breeding, "Some Characteristics of Universal Cell Nets", IEEE Transactions on Computers Vol. C-22 No. 10 October 1973.

[6] Chakrabarti K. and Kolp O. "Fan-in Constrained Tree Networks of Flexible Cells", IEEE Transactions on Computers Vol. C-23 No. 12 December 1974 pp. 1238-1249.

[7] Voith R. 'ULM Implicants for Minimization of Universal Logic Module Circuits' IEEE Transactions on Computers Vol. C-26 No. 5 May 1977.

[8] Dunworth A. "Recursive Tree and Preprocessing Algorithms for Determining the Prime Implicants of Switching Functions. APL Quote Quad Vol. 7 No. 4 1977 pp. 23-26.

[9] Mukhopodhyay A., Schmitz G. "Minimization of Exclusive-OR and Logical Equivalence Switching Circuits", IEEE Transactions on Computers Vol. C-19 No. 2 February 1970.

[10] Swamy S. "On Generalized Reed-Muller Expansions" IEEE Transactions on Computers, (Corresp.) Vol. C21 pp. 1008-1009, September 1972.

[11] Fisher L.T. "Unateness Properties of AND-Exclusive-OR Logic Circuits", IEEE Transactions on Computers, Vol. C-23, No. 2, February 1974, pp. 166-172.

[12] Reeves A.P., "A Systematically Designed Binary Array Processor", submitted to IEEE Transactions on Computers. Also available as a Purdue Internal Report, 1977.

A BIT-SLICE MULTI-MICROPROCESSOR SYSTEM FOR RANGE ARITHMETIC

Walter H. Burkhardt
Institut für Informatik
Universität Stuttgart
Stuttgart, West Germany

Summary

Numerous scientific applications of computers require reliable results. Several approaches offer assistance to the user in this field [1]. Range arithmetic serves as a valuable tool for the definition of upper and lower limits for the results from numerical computations 2 . Several software implementations as ALGOL or FORTRAN routines of range arithmetic have appeared [3, 4] . All of the programs that give complete algorithms suffer from some or all of the following drawbacks:

a. Extremely poor execution performance;
b. Approximations to the true bound values;
c. Requirements in software for direct machine accesses; etc.

Implementation in a programming language of higher level is required for the routines in order to achieve transportability between machines, which in turn prohibits access to the machine facilities for proper handling of the data. Additionally, computers of existing design are poorly suited to range arithmetic.

A bit-slice multi-microprocessor system has been designed as a better, more appropriate, and efficient solution for the implementation of range arithmetic. The special features of this design appear in a parallel computer structure with, in the minimal configuration, two semi-independent bit-slice microcomputers with common input and intermediate registers under the control of an execution processor. The microcomputers have a common microprogrammed control memory for the execution of the algorithms of the range arithmetic as far as possible and separate control memories for the decision logic and for the specific operations, see Fig. 1.

The control processor executes during operation all statements for control and fixed-point arithmetic, but not the floating-point arithmetic [5] and the decision sequence. These portions are executed by the two bit-slice microcomputers in parallel. One computes the result of the required computation for the upper bound, the other for the lower bound. Number and kind of operations for the execution of an arithmetic instruction are the same, but working on different operands. A common control store can so be used for both. Thus, the system's architecture follows closely the required tasks. Execution time for an instruction is close to the one the control processor needs for fixed-point data [5] . This configuration permits compliance with such an optimal mode of execution.

The data words for the floating-point arithmetic conform at present to the single precision data formats of the 360/70 computer family for reasons of comparative investigations.

The system is being implemented and tested currently with the Intel 3000 series microprocessors [6] for the microcomputers for range arithmetic and an 8080 as control processor. A detailed report of this investigation will appear later [7] . Estimates for the performance of the present configuration show a significant decrease of computer time compared to execution of range-arithmetic on serial computers of medium size.

References

1. Bohlender, G.: Floating-point computation of functions with maximum accuracy. IEEE TC, C-26 (1977), 621-622.

2. Bivius, R.L. and N.C. Metropolis: Significance Arithmetic. IEEE TC, C-26 (1977), 639-42.

3. Gibb,A.: Procedures for Range Arithmetic. Comm. ACM Vol. 4 (1961), 319-320.

4. Dewar, J.K.S.: Procedures for interval Arithmetic, The Computer Journal, Vol. 14 p. 447-451.

5. Herrmann, W.: Gleitkomma-Arithmetik-Einheit mit dem System Intel 3000. Studienarbeit Institut für Informatik, Universität Stuttgart, 1977

6. INTEL series 3000 Reference Manual, INTEL Corporation, Santa Clara, CA.

7. Burkhardt, W.H. and R. Hapatzky: Design and Implementation of a bit-slice multimicroprozessor system for range arithmetic. To appear.

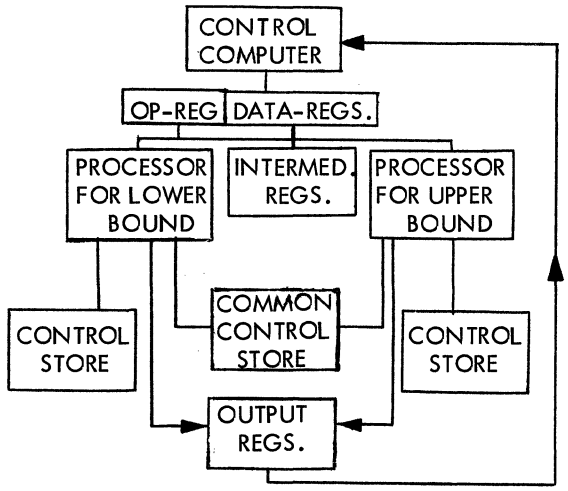


Fig. 1.: Structure of the System

FAST PARALLEL BIHARMONIC SEMIDIRECT SOLVERS

Marián Vajteršic
Institute of Technical Cybernetics
Slovak Academy of Sciences
809 31 Bratislava, Czechoslovakia

Summary

Parallel algorithms FCP and FBP are proposed for solving the first biharmonic boundary value problem on a unit square Q with the boundary $\bar{Q}$. A function u(x,y) is to be found which satisfies

$$\Delta\Delta u(x,y) = f(x,y) \quad \text{in } Q$$

$$\text{and} \quad \begin{aligned} u(x,y) &= g(x,y), \\ u_n(x,y) &= r(x,y) \end{aligned} \quad \text{on } \bar{Q}\,.$$

The functions f, g, r are given and u_n denotes the outward normal derivative of u on $\bar{Q}$.

Superimposing a square grid with mesh size $h = (N+1)^{-1}$ ($N = 2^{k+1} - 1$ for some positive integer k), a semidirect method [1] is applied for solving this problem. For this purpose, the biharmonic equation is treated as a coupled pair of difference equations. Using this method, two block-tridiagonal systems of linear algebraic equations of order N^2 with smoothing need to be solved in one iteration [1] - [3]. This process terminates when the difference of two following iteration values in absolute value is less than a given $\varepsilon > 0$ in all interior grid points.

We shall assume a parallel computer of SIMD type consisting of N^2 identical processors. Any arithmetic operation performed in one time step on several processors is assumed to be one operation step. The usual computation of one iteration [1], [2] can be replaced by the computation of one three-term recurrence formula, as suggested in [3]. Using this formula, parallel algorithms FVP and FWP described in [4], require respectively 14 $\log_2 N$ and 18 $\log_2 N$ parallel steps per iteration.

In both the algorithms presented, a vector of values for one iteration is computed by the formula mentioned, too. In the FCP algorithm, the cyclic odd-even reduction algorithm [5] is applied for evaluating this formula. One iteration can be obtained on N^2 processors in 12 $\log_2 N$ steps only. The stable variant of the cyclic odd-even reduction algorithm was developed by Buneman [5]. The effective application of this method was done in construction the FBP algorithm, requiring 20 $\log_2 N$ parallel steps per iteration on N^2 processors. For comparison, the algorithm presented in [2] requires 24 $\log_2 N$ steps per iteration on N^2 processors. As shown in [3], the number of iterations required for the algorithms mentioned is $O(N^{1/2} \log_2 N)$, when $\varepsilon = O(h^2)$ and the optimal smoothing parameters [1] are used.

The advantage of both the algorithms FCP and FBP lies in replacing N multiplications of real vectors with full matrices of order N by the multiplication with the diagonal ones only. We note that when the number of processors required is reduced by half, both the algorithms can be used in an efficient manner, too.

References

[1] L.W. Ehrlich, "Solving the biharmonic equation as coupled finite difference equation", SIAM J.Num. Analys.,(June, 1971), pp. 278-287

[2] A.H. Sameh, S.C. Chen, and D.J. Kuck, "Parallel Poisson and Biharmonic Slovers", Computing,(1976), pp. 219-230

[3] M. Vajteršic, "A Fast Algorithm for Solving the First Biharmonic Boundary Value Problem", (1978),(submitted for publication)

[4] M. Vajteršic, "A Fast Parallel Solving the Biharmonic Boudary Value Problem on a Rectangle", Proceedings of the 1 - st European Conference on Parallel and Distributed Processing, Toulouse, (February, 1979), pp. 136-141

[5] B.L. Buzbee, G.H. Golub, and C.W. Nielson, "On Direct Methods for Solving Poisson's Equations", SIAM J.Num. Analys., (December, 1970), pp. 627-656

A PARALLEL MICROPROCESSING SYSTEM

Bipin C. Desai
Department of Computer Science
Concordia University
Montreal, Canada
H4B 1R6

Summary

The importance of Integer Programming [1] in solving problems of practical importance has led to a great amount of effort in recent years on methods of their solution. In many of these problems, the integer variables are further restricted to take one of the values, 0 or 1. The further importance of solving the 0-1 problem lies in the fact that the integer problem can be converted to an equivalent problem having only bivalent variables. The solution space of the 0-1 problem involving n variables is finite and consists of 2^n possible points or nodes. A straightforward method of solving the problem is via an exhaustive or explicit examination of each of these possible points in the solution space. This approach may be suitable when the value of n is small, but since the size of the solution space increases rapidly with n, the time required to explicitly enumerate the solution space becomes prohibitive. Thus, the technique of implicit enumeration is used.

A simple, generally applicable implicit enumeration algorithm for the 0-1 problem was presented in [2,3]. In this paper, we present a microprocessor implementation of the algorithm, as a system having a number of processing stages, with each stage having a number of microprocessors (referred to hereinafter as processors). Since the computation involved in the algorithm is divided into a number of stages, it is possible to assign a special processor to each stage to perform their respective tasks. Furthermore, each stage could have a number of processors working independently on different sets of data; the actual number of processors in each stage could be chosen to match the processing load of the stage and thus optimize their utilization. Each stage would have a controller to orchestrate the processors contained therein and control the communication between the stages using appropriate buffers. The overall supervision of the solution process can be performed by a supervisor.

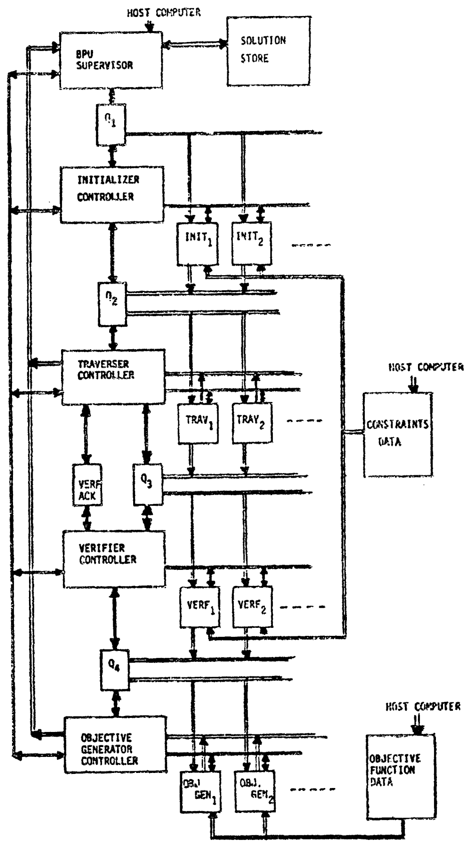
The system, called Bivalent Processing Unit (Figure 1) is an example of a Staged Multiple Instructions Stream, Multiple Data Stream (SMIMD) system [3]. This system could be interfaced to a general purpose digital computer which would communicate the user problem to the processing unit and the BPU would generate the solutions, if any, and transmit them back to the host computer for communication to the user. Simulation results of the operation of the BPU on a set of test problems from the relevant literature confirm the feasibility of the approach and show an improvement in the range of 2 to 3 orders of magnitude over the published results.

References

[1] Dantzig, G.B., "On the Significance of Solving Linear Programming Problems with Some Integer Variables", Econometrica, Vol. 29 (1960), pp. 30-44.

[2] Desai, B.C., "An Implicit Enumeration Algorithm to Solve the 0-1 Problem", to be published.

[3] Desai, B.C., "The BPU, A Staged Parallel Processing System to Solve the Zero-One Problem" Proc. of ICS '78, Taipei, Taiwan, December 1978, pp. 802-817.



THE OVERALL BPU STRUCTURE
Figure 1

DISTRIBUTED ENUMERATION ON NETWORK COMPUTERS

Ossama I. El-Dessouki
Computer Science Dept.
Illinois Institute of Tech.
Chicago,Illinois 60616

Wing H. Huen
Bell Telephone Labs.
Naperville
Illinois 60540

Abstract: Known solutions of a large number of important (and difficult) computational problems, called NP-complete problems, depend on enumeration techniques, which examine all feasible alternatives. This paper considers the design of enumeration schemes in a distributed environment in an attempt to exploit the parallel activities inherent in enumeration algorithms.
An overview for the general enumeration techniques used to solve NP-complete problems, namely Integer Programming, Dynamic Programming, and Branch and Bound is presented together with a discussion of the suitabilty of each as a basis for distributed enumeration algorithms. It is shown that a variation of Branch and Bound is the most suitable enumeration method for distributed processing. An approach that takes into account communication and computation time for analyzing distributed algorithms is then discussed. The approach is then used to analyze the performance of the proposed BB algorithm.

Introduction:

During the last decade, a special class of computational problems, called NP-complete problems [Cook, 1970], that has defied all attempts of analytical (polynomial) solutions has been identified. This class includes many famous and important problems such as the general scheduling problem, the travelling salesman problem, the graph partitioning problem. The only known technique to obtain a solution for any of these problems is to formulate them as an enumeration of all the possible permutations over some set of objects.

Also, in the last few years a new class of problems has emerged in connection with the design of distributed software. Motivated by the evolution of microcomputers, a new type of machines, called network computers([Bell and Newell, 1971] and [Huen et al., 1977]), has been recently designed. They are composed of a number of computers (until now mostly microcomputers) connected together in the form of a network and cooperating to perform computational jobs. One of the most important areas of research in connection with these machines is designing software for them. This software must be designed in a distributed form as a number of cooperating tasks that can run on separate computers of the network. A recent study [El-Dessouki, 1978] shows that many software partitioning problems on network computers are NP-complete problems. Also, enumeration techniques to solve these problems consist of activities that can be carried out in parallel. There is consequently a strong need for designing a parallel enumeration technique for these problems in general.

Partitioning programs automatically to produce clusters distributed to run on a network computer, is an example of distributed software design. This problem is important in that its solution makes network computers self-sufficient, bringing into existence mechanisms that work on network computers to produce software for network computers.

In the distributed environment of network computers, software should be designed with the following charactersitics:

1. Software must be in the form of clusters of small processes cooperating together via messages to perform the computational job.
2. Each cluster must be small enough to fit in the memory of one computer.
3. Processes in the same cluster may communicate using common data structures. But processes in different clusters can communicate only by sending messages.
4. The communication overhead incurred in message sending is significant. Consequently the amount of communication between different clusters should be minimized.
5. In order to utilize the parallel processing capability that the network offers, processing in different computers should be overlapped and equally distributed.

In a network computer environment, information sharing requires message overhead and causes a slow-down and poor utilization of the whole machine. It may be more efficient to duplicate parts of the distributed code in more than one machine to save the time spent in message passing and obtain a better overlap of computations on different computers. This observation is the basis of many techniques proposed in this paper for designing distributed software for network computers.

The problem to be solved in this paper is to determine an appropriate technique for distributed enumeration on a network computer.

The paper first discusses known mthods for designing enumeration schemes on a single computer. Based on a comparison between these techniques, Branch and Bound (BB) is, then, chosen as a basis for designing a new distributed enumeration algorithm. A method for analyzing distributed algorithms, that is characterized by taking into consideration the speed-up factor resulting from multiprocessing, processor utilization, storage requirements, as well as the communication overhead, is then presented. Two types of communication overhead are identified and methods for estimating each of them are presented. On the basis of this analysis scheme, the proposed distributed BB algorithm is evaluated. An example for the use of the technique is also given.

Enumeration Methods:

Many variations of enumeration techniques have been designed for different kinds of problems. On the theoretical (abstract) level, enumeration methods, or backtracking algorithms as they are frequently called, are generally divided into two types:

- Methods for enumerating all the solutions of problems that have more than one feasible solution, and
- Methods for finding the solution of problems that have only one correct solution.

Discrete optimization problems belong to the first kind. They are characterized by the existence of a cost function that is used to evaluate the cost of every feasible solution generated during enumeration. Frequently, this evaluation is done with the purpose of finding the minimum cost solution (the optimal solution).

In this paper, our interest will be focused on discrete optimization problems. However, the algorithms used to solve these discrete optimization problems can be extended without much difficulty to include other types of enumeration problems.

There are three well-known methods that were used to solve many discrete optimization problems using single computers. They are:

Integer Programming
Dynamic Programming (DP)
Branch and Bound (BB)

The following discussion aims at the suitability of each of these methods as a basis for a distributed enumeration technique that works on network computers.

Integer Programming In integer programming a set of decision variables is defined and the problem is formulated as an objective function of these variables to be minimized (or maximized) and a set of constraints to be observed for the generated values of these decision variables. Commercial packages are available for providing solutions for problems formulated in this way provided that the model is linear (Mixed Integer Linear Programming or MILP for short). Many optimization problems (e.g. the partitioning problem mentioned above) is more naturally expressable, however, as a nonlinear integer programming problem. Most of them can be linearized (for example see [El-Dessouki and Huen, 1977] for a linear model of the partitioning problem), but the resulting models are ill-conditioned and their numerical stability cannot be guaranteed. Furthermore, many supplementary methods for pruning the search space in MILP packages are actually BB techniques. Hence, it is quite logical to investigate how to distribute BB techniques first. In addition to BB techniques, methods for distributed matrix operations are needed. It is not clear at this point that these distributed matrix operations can be designed without a considerable amount of communication overhead.

Dynamic Programming: The underlying search space for the enumeration algorithms can be modeled as a finite tree of partial solutions. The objective of these techniques is to locate and identify an optimal solution while explicitly searching a small portion of the whole tree. As Kohler and Steiglitz point out in [Coffman, 1976], Dynamic Programming and Branch and Bound, have recently emerged as the principal general methods for finding solutions to discrete optimization problems. DP essentially searches breadth-first and uses dominance rules to prune search tree nodes. The main advantage of DP is that breadth-first search provides a global look at the search space, allowing quick discovery and elimination of dominated nodes. Breadth-first searching has been the basis of enumeration techniques designed for some multiprocessor systems. For example, Marshall [1977] studied a breadth-first enumeration technique for PEPE (Parallel Element Processor Ensemble). There are significant differences, however, between multiprocessor machines and network computers so that distributed approaches adopted for multiprocessors have serious drawbacks in network computers. There are mainly two reasons behind this fact:

1. In a multiprocessor all the CPUs are assumed to be capable of addressing all the memory in the system without much difficulty. In a network computer the speed of accessing information heavily depends on whether this information is located in the local memory of the computer or not. Thus, information sharing may be achieved relatively easily in a multiprocessor that contains common memory, but it is extremely inefficient in a network computer.
2. Most of the parallelism provided by a multiprocessor, like PEPE, is on the arithmetic and logic unit level (PEPE has 288 ALUs). In a network computer, however, parallelism is provided mainly in the program control unit (PCU) level [Handler, 1977]. Every computer in the network is capable of interpreting and executing its own instruction stream. Thus, in a parallel computer like PEPE or the ILLIAC IV for example, it was very attractive to design algorithms in the form of successive stages that consist of many similar operations that are executed on different data sets simultaneously by different ALUs. Processes in this form of parallel processing are executed by different ALUs in a synchronous mode performing one stage of computation at a time and creating more processes (sons) for the next stage. Consequently,

breadth-first was a very natural choice as a basis for parallel enumeration on multiprocessors. In network computers, however, the amount of information needed to be communicated by every module to the process that prunes the tree in every stage is huge and causes a drastic slow-down in speed. Moreover, the size of the generated partial solutions tree grows exponentially before generating any solution. Actually, solutions can only be generated if enough memory is available to hold the entire solution's tree. This creates a serious problem in a network computer in which the amount of memory available in each computer is usually small and always limited. The problem is that when the search is stopped before reaching the final stage, not a single solution can be obtained and all the search effort is lost.

Branch and Bound: Branch and Bound is a general technique for backtrack enumeration that refers to a number of algorithms. The algorithms that are subsumed in the literature under this term, may differ widely in the ways used to arrange enumeration and the pruning of the search tree. The feature that distinguishes them from other enumeration techniques is the way they attempt to accumulate information about the optimal solution and the use of this information in pruning the search tree. Information regarding the most recently known best solution is accumulated in the form of upper bounds. A set of lower bounds on the expected quality of the solutions generated from active nodes is used to eliminate nodes of the search tree. When compared to DP, BB has many attractive features that make it more suitable as a basis for designing enumeration algorithms for network computers. These features include:

1. BB tree search mechanisms are more flexible. They can be changed from breadth-first to depth-first or even a combination of both by choosing the appropriate branching policy. In fact, as Steiglitz and Kohler point out in [Coffman, 1976], given a sequencing problem and a DP algorithm to solve it, an equivalent BB algorithm can also be found.
2. BB can generate complete solutions very quickly, specially if a depth-first strategy is used. These solutions may not be optimal, but they are of known (bounded) quality. More importantly, the quality of the required solutions can be adjusted according to the user needs.
3. The tree search can be stopped as soon as the computational facilities (time or storage) are exhausted without losing the generated solutions. This property has a significant value in network computers that have limited memory in each node and no virtual storage capability.
4. The most important and attractive feature of BB, however, is that the amount of global information needed to describe the state of the search is less than that in DP and is adjustable. The state of the search is described in each stage by the most recent upper bounds and the lower bound on the quality of the solutions generated from the active node. It becomes natural to think about a set of processes working independently on different parts of the search tree in an asynchronous mode and exchanging information regarding the most recent bounds. This style of loosely coupled parallel computation is much more suited to network computers.

To effectively compare the different BB algorithms, it is necessary to establish a general classification scheme. Following Kohler and Steiglitz [Coffman, 1976], BB algorithms can be classified according to nine parameters (B, S, E, F, D, L, U, BR, RB). The nine parameters are defined as follows:

1. B is the branching rule that defines the scheme of generating the sons of each node in every stage of the algorithm.
2. S is the selection rule that is used to select the next branching node from the set of currently active nodes. The selection rules that will be studied in this paper are:
   a. S = LLB (least lower bound rule) - Select the currently active node with the least lower-bound cost.
   b. S = FIFO (first-in first-out rule) - Select the currently active node that was generated first. This leads to a breadth-first strategy.
   c. S = LIFO (last-in first-out rule) - Select the currently active node that was generated last, provided it is not a completed solution (leaf). This gives a depth-first search strategy.
   d. S = DF/LLB (depth-first/least lower bound rule) - From the set of most recently generated active sons, select the son with the least lower bound cost, provided this son is not a completed solution (leaf).
3. F is the characteristic function used to eliminate nodes known to have no completion in the set of feasible solutions.
4. D is the dominance relation defined on the set of partial solutions and used to eliminate nodes (dominated nodes) from the search tree before extending them.
5. L is the lower bound function that assigns to each partial solution a real number representing a lower-bound cost for all complete solutions that can be generated from it.
6. U is the upper-bound cost which is actually the cost of some complete solution known at the beginning of the algorithm. At each stage of the algorithm, U is updated to incorporate the least cost known solution up to that stage.
7. E is the set of elimination rules that use D, U, and L to eliminate newly generated and currently active nodes. The most famous elimination rules are:
   a. U/DBAS (upper bound tested for dominance of descendents of branching node and members of currently active set) - If the lower bound of a newly generated descendent of an active node exceeds U, eliminate it before it becomes active. If as a result of this

operation, the lower bound of the active node itself becomes greater than U, eliminate that active node from the set of currently active nodes (e.g. when all sons have been investigated and additional and improved information regarding L was known as a result).

b. AS/DB (active node set tested for dominance of descendants of branching node) - Using D, each node in the active set is tested for dominance of each descendant of the branching node. If dominated, the descendant is eliminated before being considered active.

c. DB/AS (descendants of branching node tested for dominance of currently active node set) - Each descendant that passed the preceding test in b and became active is tested for dominance of each currently active node. Each dominated node is removed from the current active set of nodes.

8. BR is a real number between zero and one representing the desired maximum relative deviation of the optimal cost from an acceptable solution.

9. RB is the resource bound vector whose components are upper bounds on the total expendable execution time and the usable storage for active nodes and immediate descendants of the branching nodes.

Conceptually, in each stage of a BB algorithm, a "list" of the objects that can be permuted to generate all the possible alternatives for feasible solutions is constructed. The algorithm selects one "attractive" alternative using the rule S. This operation can be considered as a selection of one node of the search tree. Then the algorithm generates the direct descendants of this node using B. It then computes L of these new nodes and uses D and E to detect and eliminate dominated nodes. It then updates the set of currently active nodes. When a complete solution (a leaf) is generated, U is updated and used together with BR to check whether an acceptable solution has been reached. The algorithm stops if such a solution is reached. In every stage the RB vector is used to determine whether the available resources have been exhausted or not. If so the best recently known solution is given before halting.

From the above description it is clear that all BB methods generate the whole search tree at some point. This is unavoidable, otherwise the optimality of the generated solution cannot be guaranteed. This is another way of stating the fact that the problem in hand is NP-complete. There are two desirable considerations, however, that make one BB algorithm more attractive than another for a network computer. First because of the limited memory in each node, the search tree size should grow gradually (i.e. in a rate comparable to the rate of generating solutions) and the whole tree should be generated at the latest possible time after generating a number of complete solutions. Second, the BB algorithm should have low communication requirements. These two features will be the basis of designing a distributed BB algorithm in the next section.

Distributed Branch and Bound:

One way to distribute the BB algorithm operations described above on a number of computers is to give every computer a "share" from the list of objects used to generate the solutions. Every computer, then, can start to investigate its own share. The most important point in the algorithm proposed here, however, is to let each computer by itself generate the list, identify its own share, and limit its investigation for a solution within this share. The computers, then, can exchange, via messages, global and time dependent information such as the most recent upper bound, in order to enhance the search for the optimal solution. It must be noticed that the process of generating at least a part of the required lists mentioned above is repeated in more than one computer.

This scheme has two advantages. First, in a network computer environment, it is more efficient to let every computer search a complete subtree independent of the others. Even if subtrees are not completely disjoint and some computations are repeated in more than one computer, still the communication time saved makes this method more attractive and efficient than having a single computer for each node in the decision tree. As a quick example, duplicating the computations corresponding to the root of a decision tree in all the computers of the network does not increase the load of individual computers because each of them must wait anyhow for the result of the root calculation, but this scheme saves the communication time required to send the result of this computation to all the required receivers. Second, the problem of letting a single computer take the decision of splitting up the load among a number of processors is avoided. This is particularly important in applications where it is difficult to determine how to divide the search problem before generating the actual permutations. In these cases, some permutations can be ruled out very quickly as not feasible using the characteristic function F mentioned earlier, for example. But this can only happen after generating these permutations. In these cases the process of dividing the search load requires a significant amount of computations. In the proposed scheme this process is shared among the computers of the network.

In order for a computer to be able to identify its "share" in the enumeration, every computer containing an enumeration subprocess is given an identification number, MYNUMR. The enumeration subprocess must also know the total number of cooperating parallel subprocesses in the enumeration job. This latest number is called NCP.

During enumeration, each subprocess is in one of the following phases:

Phase 1: Selection phase.
Phase 2: Full enumeration phase.
Phase 3: Exchange phase.

In each phase, each subprocess operates as follows:

Phase 1: Selection Phase:

In this phase, the number of nodes in the search tree in each stage under consideration is less than the number of cooperating subprocesses NCP. Since every enumeration tree begins with one root, this is always the initial phase.

At every stage each subprocess constructs the list of objects that can be permuted to generate the partial solutions. The subprocess then selects one node from the decision tree by generating one specific permutation of these objects in the list. This same node may be chosen by more than one subprocess since the number of cooperating subprocesses is greater than the number of possible permutations. The distribution of nodes is made using a circular scheme such that if the number of nodes generated in a stage in the selection phase is denoted by SIZE, then subprocess numbered MYNUMR is assigned node number g where:

$$g = [(\text{MYNUMR} - 1) \ \underline{\text{modulo}}\ \text{SIZE}] + 1.$$

The subprocess calculates new values for MYNUMR and NCP as follows:

$$\text{DELTA} = \begin{cases} 1 & \text{if } g \leq [(\text{NCP}-1)\ \underline{\text{modulo}}\ \text{SIZE}]+1 \\ 0 & \text{Otherwise} \end{cases}$$

$$\text{MYNUMR} = \lceil \text{MYNUMR} / \text{SIZE} \rceil$$
$$\text{NCP} = \lfloor \text{NCP} / \text{SIZE} \rfloor + \text{DELTA}$$

Then the process reenters the selection phase at the next stage.

This phase ends when SIZE becomes equal to or greater than NCP. The parameter SIZE, which indicates the number of possible permutations at each stage, can become larger than NCP at any stage of the selection phase. When this happen, every process picks up a number of permutations equal to $\lfloor \text{SIZE/NCP} \rfloor$. The last process arbitrarily gathers the remainder of the load when SIZE is not divisible by NCP. All processes then enter full enumeration.

Phase 2: Full Enumeration:

When the number of alternative permutations of the generated list of objects reaches NCP, each subprocess can choose a permutation, being sure that it is not shared with any other subprocess. Each subprocess can proceed to search the generated subtree in a normal BB fashion since it has a unique subtree of its own.

Each computer in the network should include three tasks that cooperate to perform enumeration and communication with other processes in the other computers. These three tasks are called:

The Main enumeration task, M,
the Receiving task, R, and
the Transmitting task, T.

R receives the messages, updates the local information regarding enumeration (e.g. the most recent value of UB), and may activate special procedures in its node (e.g. a procedure to send information regarding unsearched parts of the local tree as will be explained in the exchange phase). T formats and sends the required messages.

In this section, different BB algorithms are examined using the nine-tuple characterization presented earlier in order to compare them as a basis for full enumeration in our distributed environment. The algorithms are analyzed on the basis of the amount of communication overhead, the effect of limited memory in each node, and the utilization of the parallel processing capability of the network.

Information regarding the two parameters L, the lower bound, and U, the upper bound will be exchanged between various nodes sharing in enumeration. Consequently, these two parameters are assumed to be accessed and/or modified by the transmitting and receiving tasks, T and R, in each node in a way that will be discussed shortly.

The four parameters: the branching rule B, the characteristic function F, the resource bound vector RB, and the bracket BR, are in fact user and/or problem dependent parameters. They can be incorporated in a direct way in each main task M on each node. A simple modification for the definition of these four parameters is that they should all be defined on the local set of tree nodes generated in each processor. Thus, for example B becomes the rule that is used to generate the sons of an active node among those stored in the processor in which task M is executing.

The main problem to be discussed now is the kind of S, D, and E rules that should be used and their effects on the communication overhead, speed, and storage.

Two general types of S, D, and E rules can be proposed for the distributed algorithm:
Type 1 : Global rules
Type 2 : Local rules

In the first approach, the parameters are defined in the same way as in the uniprocessor case e.g. the dominance relation D is defined for all pairs of generated solutions irrespective of the node in which they are generated. A second example is a global elimination rule in which partial solutions are eliminated if they are dominated by any partial or complete solution in any processor in the network. In the second approach, the rules are defined to operate on the partial and/or completed solutions generated within the processor only (local solutions). As an example, a local selection rule would be to select the node with the least lower bound cost among the local set of active nodes in each computer.

To compare the two approaches, it is first observed that a global selection rule has no advantage over a local one. A global selection rule may let one computer to work at a time (the one which has the "selected" node) resulting in very poor utilization of the parallelism in the network. Aother alternative of global selection is to select a set of nodes and send information regarding these nodes to a set of computers to allow them to search in parallel. This may well lead to a situation in which the information regarding the whole enumeration tree is sent many times around the network causing a prohibitive communication overhead. Thus, a local selection rule as the one mentioned earlier is a clear choice.

The advantage of a global dominance relation and an elimination rule based on it is that it may increase the probability of discovering dominated solutions earlier in the enumeration and hence reduce computation time. However, this fact is not necessarily true. For every global dominance relation, D, a local dominance relation, D' can be found such that $D' \subseteq D$. This can be achieved by using the same definition of D on the restricted local set of solutions. Kohler and Steiglitz [1976] proved that using a stronger dominance relation like D instead of D' may increase the computational requirements of a BB algorithm (theorem 6.3 p 256 of [Coffman, 1976]). Moreover, the communication overhead incurred in exchanging the information needed to test every pair of solutions will further slow down significantly the resulting algorithm. As a result the local approach is chosen as a basis to design the distributed BB algorithm in this paper. Thus, the dominance relation and all the elimination rules based on it are defined for the local set of solutions in each computer.

A final point in this comparison is the difference between various selection rules. A significant advantage of a local LIFO (depth-first) and/or DF/LLB (depth-first least lower bound) rules over a local FIFO (breadth-first) and/or LLB rule in a network is that in the first set of rules many complete solutions (one in each computer) are generated quickly. Since the value of the most recent UB found in each computer is exchanged among all the computers via messages, this should result in increasing the chance of improving the UB value at each stage. This in effect can be thought of as using a tighter UB in each computer which, as indicated in the studies of uniprocessor BB algorithms, is a significant factor in speeding up enumeration. Moreover, in depth-first search, the UB contains all the necessary information needed to be retrieved during successive stages of the enumeration. The storage space of complete solutions can be reused in generating new solutions. The full specifications of the best known complete solution is not needed during enumeration and can be stored on a secondary storage file, for example. This fact has a special significance in network computers with limited memory in each node.

Thus, in the distributed BB algorithm proposed in this paper the main process M in each computer searches its own subtree using a depth-first selection rule. When a complete solution is generated the UB in that computer is updated and the Transmit task, T, is activated to "broadcast" the upper bound value to all the other computers. When a message concerning a new value of UB is received in any computer, the Receiving task, R, is activated. It tests the received UB value and uses it to update the local variable that contains the most recently known UB. It should be noticed that the location of that local UB is accessed by both M and R asynchronously. Consequently, this location must be considered a critical resource and appropriate mechanisms for mutual exclusion should be used.

Phase 3: Exchange Phase:

The exchange phase is entered when an enumeration process p completes the search for a solution in its own subtree and terminates its active phase of full enumeration, while some other subprocesses are still searching their subtrees.

Process p declares its new status and initiates a dialogue aiming at redistributing the remaining enumeration load. Essentially, process p switches to search a new as yet unsearched subtree that was originally a part of the load of another process q. In order to achieve this, the following steps must be taken:

1. Process p must identify an appropriate process q to share the enumeration load with.
2. Each process in the system must identify the unsearched part of its subtree.
3. The information necessary to switch from one subtree to another must be specified. Process q sends a message to process p including such information.
4. Process p should carry most of the overhead load associated with the switching operation, since it is the idle process.
5. Process q must identify the subtree given to process p and updates its search space properly by eliminating this subtree from it.
6. A "thrashing" situation in which the processes spend most of the time exchanging subtrees should be avoided.

As was mentioned before , each process at the beginning the selection phase knows its own name, MYNUMR, and the number of computers in the network, NCP. The initial values of these two parameters are called MYNUMR0 and NCP0, respectively. Every process in the system is assumed to be identified by its MYNUMR0. Thus, any message in the system should be associated with two parameters, the values of MYNUMR0 of the sender and the receiver. This identification should act as a channel number for the communication network. Broadcast messages mentioned before, however, are treated specially and may not include this kind of information. (It must be noticed that the values of NCP and MYNUMR are updated in every stage of the selection phase. Consequently, their final values will, in general, be different from NCP0 and MYNUMR0). Hence, a process p can identify its nearest neighbor, p*, as:

p* = [(MYNUMR0 + 1) modulo NCP0]

Note that this scheme makes process 1 in the enumeration the successor of process number NCP0.

When process p enters the exchange phase, it starts asking the other processes in the system one at a time beginning with p* for a share of their enumeration load. If p* does not want to give any part of its load, process p asks the nearest neighbor of p* and so on. If no process wants to give any part of its load, process p informs the group that it is quitting and it halts. Otherwise the first process that shows willingness for exchange is identified as q.

The next step is to design the mechanism by which each process can identify the unsearched part of its own subtree. To do that, it must be noticed that phase 2 incorporates a depth-first search strategy. Moreover, one of its main characteristics is that each enumeration subprocess is searching a complete subtree. In order to maintain these desirable features, process p should switch to the upper-most nodes (i.e. one closest to the root) in the unsearched part of q's subtree. Furthermore, p should pick up the subtree that will be searched last by q. To achieve this while avoiding the occurrence of a thrashing situation the following steps are executed:

1. Each process in the system (like q) maintains a pointer, UNSEARCH, to the last son of the upper-most root of its unsearched subtree. UNSEARCH can be initialized at the point of entering full enumeration. Two cases must be considered at the last stage of the selection phase. If SIZE became greater than NCP and the share of each computer at the moment of entering full enumeration is greater than one tree node (i.e. when |SIZE/NCP| > 1), UNSEARCH should be initialized to point to the last node of each computer share. Otherwise, UNSEARCH should be initialized to point to the last "son" of the only node picked by the computer while entering full enumeration.
2. When a process begins the search in the subtree with root UNSEARCH, the value of UNSEARCH should be updated to point to its last son.
3. When a process receives a message asking for a share of its enumeration load it tests whether the node pointed to by UNSEARCH has brothers that are active but not yet branched from. If there is, the process sends the address of UNSEARCH to the requesting process p (i.e. it becomes q). If there is no brothers for the node UNSEARCH, however, the process initiates a branching from UNSEARCH if more than one son results from the branching it passes the address of the last son to p. Otherwise, the process expresses its refusal for exchange (the remaining load is too little and it is better not to send it around). UNSEARCH is updated to point to the last son remaining in the processor.

The advantage of this scheme is that the free computer, p, takes a share of the load proportional to the remaining load in q. Also, process q always keeps a part of that remaining load in order to avoid another exchange in case q is about to enter its unsearched part of the tree when it received the exchange message from p. In this way unnecessary exchanges of tree nodes are substantially reduced. The next step is to specify what information must be exchanged between p and q before a new active phase can begin in p. Since each process in the network has a copy of the original set of objects that can be permuted to obtain partial solutions at each stage, process p can resume enumeration at any node of the search tree if it is given the description of the partial solution corresponding to that node. This information must be supplied by q. After sending this information to p, process q eliminates that partial solution from its search space and update UNSEARCH as indicated above.

An example for distributing a simple enumeration tree on 3 processors using the above scheme is given below.

Example: The original tree is shown in Figure A. In the first step of the selection phase each process generates the initial root node, identifies its neighbor, and performs the root computations as shown in Figure B.
In the second iteration of the selection phase, the alternatives examined in each process and the computation of g are shown in Figure C.
Based on the value of g, processes P1 and P3 "select" node 2, and process P2 "selects" node 3 as shown in Figure D. The values of MYNUMR and NCP are updated accordingly.
In the next iteration, process P2 realizes that no other process is sharing with it node 3 (NCP became 1 which is less than or equal to the number of entries in the list {3} selected by process P2 at this point). Thus, P2 enters full enumeration phase and identifies node 9 as the uppermost root of the "unsearched" part of its tree. Processes P1 and P3, however, begin a new iteration in the selection phase as shown in Figure E.
At this point, both processes P1 and P3 realize that the number of nodes in {4,5,6} is 3 and P1 "picks-up" node 4 while P3 "picks-up" 5 and 6. Each process stores a value for UNSEARCH and

enters full enumeration phase as shown in Figure F.
Each process continues full enumeration of its subtree until, for the purpose of completing this example, a point is reached when P1 finishes its full enumeration phase and wants to share in the remaining enumeration load. The status of the enumeration is assumed to be as shown in Figure G.
Process P1 recognizes that 2 is the process number of its "neighbor". Hence, P1 initializes an exchange phase with P2. After the exchange is complete, the new status of the enumeration is shown in Figure H.

It is clear that in a real application the trees are much bigger than the one shown in the example and one or more exchange phases may be initiated by various processes in the system.

Algorithm Analysis:

In this section, the performance of the distributed BB algorithm presented above is analysed. In recent research ([Agerwala and Lint, 1978], [Baudet and Stevenson, 1978] and others), it has been pointed out that the previous work on the design of parallel algorithms for SIMD machines largely ignored communication delays. New methods for analyzing algorithms for machines in which communication overhead cannot be ignored are beginning to appear in recent research. The following analysis considers the computational complexity and the communication overhead of the distributed enumeration algorithm.

Before examining the algorithm, the computation model on which the analysis is based must be described. As was mentioned earlier, the system architecture is a network computer consisting of multiple processors each having its own local memory and control. There is no shared memory in the system. Computers are connected by a fixed topology network. The time to send a unit of information between two computers is proportional to the distance between them (i.e. the number of links or data movements required between them). A linear cost criterion [Agerwala and Lint, 1978] is used to define message length in this paper. Under this criterion a message can be considered of length equal to an integral multiple of some basic unit. It is further assumed that one processor can broadcast one unit of information to all the processors of the network. The time to send such a broadcast message is proportional to the number of computers in the network.

In the distributed BB algorithm presented above, processors are assumed to execute in an asynchronous manner. Three modes of interaction among computation and communication can be directly identified:
1. In the selection phase each processor executes its computation using local data only. No interaction or communication of any kind between different computers takes place during this phase.

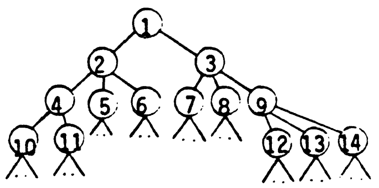


Figure A.

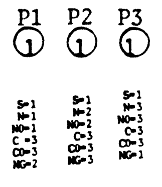


Figure B.

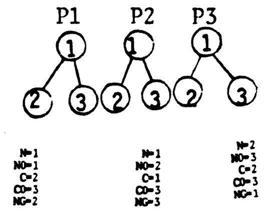


Figure C.

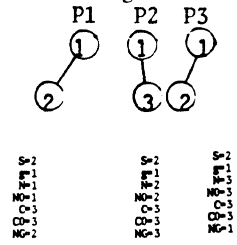


Figure D.

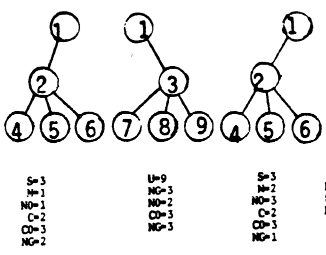


Figure E.

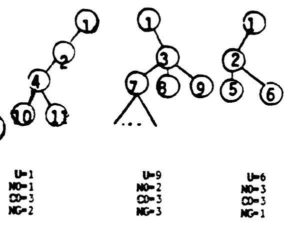


Figure F.

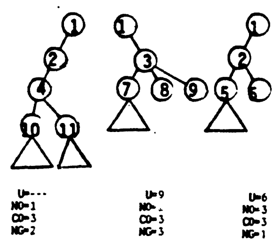


Figure G.

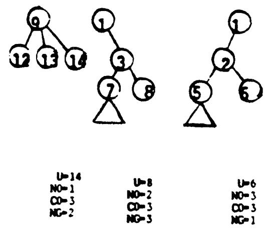


Figure H.

legend: S- SIZE, N- MYNUMR, NO- MYNYMRO, C- NCP, CO- NCPO, NG- NEIGHBOR, U- UNSEARCHED

2. In the full enumeration phase, each processor's main enumeration task M executes its computations in the form of a loop. In each iteration (stage) of the loop selection, branching, and elimination steps are executed. The main task does not wait for messages from any other processor. When a message is received, however, the receiving task R in that processor is activated. As a result, tasks R and M share the same CPU until the message is processed. Also, when a message needs to be sent, task T is activated and it takes its share from the processor time. It is therefore clear that the

communication delay in this case overlaps with computation completly. The communication overhead is directly proportional to the processing time of T and R. For the purpose of this paper, it is convenient to measure this kind of overhead as:

$$NT*YT + NR*YR$$

where NT is the number of times T is activated,
YT is the amount of processing done by T each time it is activated
NR is the number of times R is activated,
YR is the amount of processing done by R each time it is activated.

3. In the exchange phase, the processes involved in the exchange operation can be modeled using a similar technique as suggested in [Agerwala and Lint, 1978] i.e. each process is assumed to be a sequence of non-overlapping cycles of computation followed by communication. The communication in this case is carried out by the network and is modeled using the linear model discussed at the beginning of this section.

With this model defined, the distributed BB can be discussed. Given a network of N computers and an enumeration problem in which the number of possible solutions in stage i is given by SIZE(i), the selection phase ends at stage j at which:

$$\mathrm{SIZE}(j) \geq N$$

and the number of tree nodes generated in the selection phase is:

$$(N+1) \leq \sum_{i=1}^{j} \mathrm{SIZE}(i) \leq 2N$$

In this phase each computer is working to generate one path of a tree with N leaves. Consequently, the height of such a tree is less than or equal to $\lceil \log N \rceil$.
Thus, the speed-up obtained in this phase is given by:

$$S \geq N/\lceil \log N \rceil$$

In the full enumeration phase, the number of solutions generated in each computer, Q can be bounded by:

$$1 \leq Q \leq \mathrm{SIZE(FINAL)}/N$$

where FINAL is the number of stages in the whole enumeration tree. Thus, the number of messages transmitted by each node is Q and the number of messages received is less than (N-1)Q. This is in fact a very pessimistic estimate on the number of messages sent and received in every node. It is assumed that the computer that generates the worst solution is the fastest one and the computer with the next worst solution is the second fastest and so on. Many of these messages in a real case will not be sent since some of the good solutions may be arrived at before the bad ones. The probability of obtaining good solutions first is better in the case of a network of independent computers working on different subtrees simultaneously (as in the above distributed BB algorithm) than the case of a single processor. This probability also increases with the number of processors N. A better estimate for the number of received messages may therefore be N^q, where q is a fraction between o and 1.

Each received message causes UB to be updated. This operation should not take more than an O(1) time units. Also, sending a message with a new UB is another O(1) operation. Consequently, the amount of communication processing is bounded by $Q(N^q+1)$. In order to estimate the amount of processing done by the main enumeration task M in every stage, it is assumed that the number of active tasks to be considered for dominance and elimination processing using E and D is proportional to the number of branching operations done up to that stage. Thus, the processing done by the main enumeration task in every satge can be bounded by:

$$[\lceil \log (\mathrm{SIZE(FINAL)}/N) \rceil]^c$$

where $c \geq 1$ (depending on the sophistication of the elimination rule used, c may vary between 1 and 3 mostly). Consequently, the total amount of processing in all the stages done by the main task M may be estimated as:

$$[\log(\mathrm{SIZE(FINAL)}/N)]^c * 2 * \mathrm{SIZE(FINAL)}/N$$

The factor 2*SIZE(FINAL)/N represents the upper bound on the number of tree nodes visited by one computer i.e. the number of branching operations executed.

For a uniprocessor the amount of processing should be
$[\lceil \log \ \mathrm{SIZE(FINAL)} \rceil]^c * \mathrm{SIZE(FINAL)}$ Thus, the speed-up in the full enumeration phase is given by:
calling SIZE(FINAL) SF we get

$$S \geq N/\left[(1-(\log N/\log SF))^c + N^q/(\log SF)^c\right]$$

For the case c=q=1, this expression reduces to:

$$N/[1 + (N-\log N)/\log(\mathrm{SIZE(FINAL)})]$$

In the exchange phase, every computer begin asking its nearest neighbor for a share in enumeration. If it is considered very rare for a neighbor to refuse an exchange, then the distance of all the messages in this phase can be considered unity. As was mentioned before in the description of the exchange protocol, process q may have to execute a branch operation and an elimination operation for one node. Also, the amount of exchanged information is essentially equal to the size of one partial solution. All these operations take an amount of time independent of the number of computers in the network and the number of nodes in the enumeration tree. Consequently, the overhead associated with the exchange will be bounded by the number of exchange operations that occur. If the amount of processing needed in each enumeration subtree is roughly the same, the number of exchange operations should be small. However, it will greatly depend on the distribution of actual load on the computers, a factor that does not only vary on a statistical basis but also does depend on the nature of the problem.

Conclusions:

A distributed enumeration algorithm has been presented that has the following characteristics:
1. It consists of a number of loosely coupled processes that operate in parallel. Each process enumerates a complete subtree of the enumeration space. A depth-first BB technique is used to enumerate all feasible solutions.
2. The information exchanged between various processes is reduced to one value per complete solution (the most recent upper bound), to minimize the communication overhead. This was shown to be the most attractive feature of depth-first BB as compared to all other enumeration techniques. The communication overhead is reduced in some phases of the algorithm by duplicating some of the computations in more than one process. These duplicatedcomputations do not increase the load of any process sharing in the enumeration.
3. The technique is general enough to be applied in many enumeration problems and in multiprocessors as well as network computers. The reduced communication overhead makes it specially attractive for the network computer case. In a multiprocessor with interleaved memory, this feature can be used to reduce memory conflicts.
A method for analyzing distributed algorithms was discussed and used to analyze the performance of the proposed algorithm. It was shown that the nature of the communication overhead can vary depending on the way asynchronous processes are designed to interact together. One case was studied in which the amount of communication overhead does not depend on the characteristics of the communication network but rather on the complexity of some message handling task in each computer of the network. Finally, it was shown that the speed up gained by the proposed algorithm can reach the number of computers in the network for trees with huge sizes.

References:

1. D. Agerwala and B. Lint, Communication in parallel algorithms for Boolean matrix multiplication, Proc. 8th International Conf. on Parallel Processing, Traverse City, Mich., August 1978.

2. G. Baudet and D. Stevenson, Optimal sorting algorithms for parallel computers, IEEETC, Jan. 1978, pp. 84-87.

3. Bell, G., Newell, A. 1971. Computer Structures: Readings and Examples, pp. 470-488, McGraw-Hill Book Company, New York.

4. Coffman, E.G., Jr. (Ed.) 1976. Computer and Job-Shop Scheduling Theory, John Wiley and Sons, New York.

5. El-Dessouki, O., Huen, W. 1977. Automatic partitioning for a network computer, Technical report 77-6, Dept. of Computer Science, Illinois Institute of Technology, Chicago.

6. El-Dessouki, O. 1978. Program Partitioning and Load Balancing in Network Computers, Ph.D. thesis, Department of Computer Science, Illinois Institute of Technology, Chicago.

7. W. Handler, The impact of classification schemes on computer architecture, Proc. 7th International Conf. on Parallel Processing, Traverse City, Mich., August 1977.

8. Huen, W., Greene, P., Hochsprung, R., El-Dessouki, O. 1977. A network computer for distributed processing, Proc. COMPCON 77 FALL, IEEE, Sept. 1977.

9. Kook, S.A. 1970. The complexity of theorem proving procedures, Proc. 3rd ACM Conf. on Theory of Computing, May 1970.

10. Marshall, D. 1977. A parallel processor approach for searching decision trees, Proc. 7th International Conf. on Parallel Processing, Traverse City, Mich., August 1977.

THE MULTISENSOR DATA CORRELATION AND HANDOVER PROBLEM

Duane David Marshall
System Development Corporation
Huntsville, Alabama 35805

Summary

The situation in which N items (men, radar tracks, jobs) must be assigned or paired with N other items (tasks, incoming objects, computers) so as to minimize some measure of the pairing is called the Assignment Problem (AP) [1,2]. An algorithm to determine the optimal pairing is well-known and has been studied extensively. The algorithm is a modification of a branch-and-bound tree search and takes advantage of the special structure of the problem.

The multisensor data correlation and handover problem occurs when object tracks pass from one radar's field of view to another's. In order to provide the maximum information transfer, the two radars must be able to identify the same objects when observed from two different aspects. This is an assignment problem.

The assignment algorithm is a sequential process, and execution times tend to be on the order of N^3 [3]. For a large number of objects, such as the ballistic missile defense problem, algorithm execution times are excessive, and means to obtain solutions quickly must be determined. Several ways to reduce the AP times have been explored. One approach is to use a non-optimal heuristic [4]. Execution times for the heuristic are on the order of N^2. However, the number of incorrect assignments made makes this approach unacceptable.

Another way to reduce the AP time is to take advantage of the parallelism found in the sequential algorithm. A study was conducted using the Parallel Element Processing Ensemble (PEPE) to determine any advantages found by solving the AP on a parallel associative processor. A testbed of approximately 50 problems was generated. These problems ranged in size from 3 to 12 objects. Included for each size were several randomly generated problems and the sequential algorithm "worst case" example. The worst case runs in N^4 time.

These test problems were then solved by four different means to compare the relative solution times and solution accuracies. The methods used were (1) the sequential algorithm, (2) the sequential heuristic, (3) the parallel algorithm, and (4) the parallel form of the heuristic. Test results for execution times are shown in Figure 1.

As reported in the literature, the execution time for the sequential algorithm is a function of N^3. The parallel form of the optimal algorithm gives an execution time proportional to N^2. The sequential heuristic runs in N^2 time while the parallel heuristic runs in a time proportional to N.

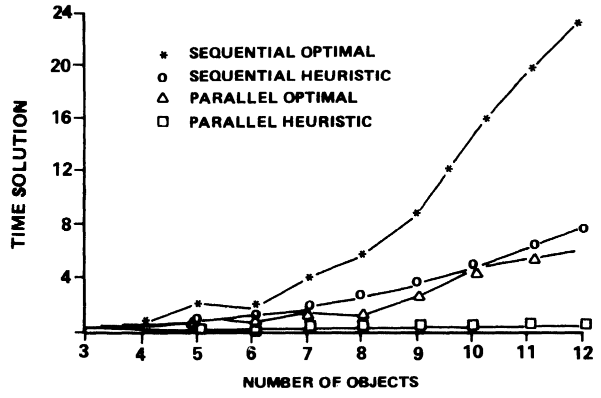


Figure 1. Test Results

The sequential heuristic was found to produce a large number of errors in the pairings. In addition, the times for the heuristic were longer than those for the parallel algorithm even though both ran in N^2 time. Thus, the parallel algorithm provided the optimal solution while execution times were reduced by a factor of N when compared with the sequential implementation. In a critical environment, such an improvement could prove to be decisive.

References

[1] J. Munkres, "Algorithms for the Assignment and Transportation Problems," Journal of the Society for Industrial and Applied Mathematics, March 1957, pp. 32-38.

[2] R. Silver, "An Algorithm for the Assignment Problem," Communications of the Association for Computing Machinery, November 1960, pp. 605-607.

[3] F. Bourgeois and J. C. Lassale, "An Extension of the Munkres Algorithm for the Assignment Problem to Rectangular Matrices," Communications of the Association for Computing Machinery, December 1971, pp. 802-806.

[4] C. G. Pfeiffer, Advanced Data Processing Engagement Logic and Algorithms for Midcourse Acquisition, D^3, C^3, Multisensor Data Correlation and Handover, TRW Report TR-A008.

A FRAMEWORK FOR THE STUDY OF PERMUTATIONS AND APPLICATIONS TO MEMORY PROCESSOR INTERCONNECTION NETWORKS*

D. K. Pradhan[†] and K. L. Kodandapani[§]
School of Engineering
Oakland University
Rochester, Michigan 48063

Abstract--This paper develops a set of systematic techniques to facilitate the study of permutations and permutation networks. First, we present a switching theory framework through which certain well-known permutations that are useful in parallel processing are characterized. A procedure is then developed, using these results, which is useful in characterizing certain shuffle-exchange networks and, in general, any permutation networks. This procedure is based on a technique whereby complex permutations are decomposed into a sequence of elementary permutations. Several new results are thus derived.

Introduction

A goal of this paper is to extend, as well as unify, the understanding of both permutations and permutation networks. It is hoped that these new results and insights may lead to further research.

The study of permutations and permutation networks has been an important topic of research in parallel processing [3-8]. Permutations of the data, as well as of the intermediate results, are required in order to execute the algorithms that are used in parallel processing. Also, the ability to simultaneously access multiple data elements from memory is key to successful parallel processing. This simultaneous access is achieved by the use of multiple memory modules [1], where those data items that may be simultaneously needed are stored in different modules. Several permutation techniques [3-8]. have been proposed for arranging the data items so that conflict-free access to the memory modules is achieved.

In the literature, certain permutations, such as uniform shift, unscrambling of t-ordered vectors, etc., have been identified as important in parallel processing (especially in SIMD-type machines). These findings have influenced the design of some of the permutation networks that have been developed for interconnection between memory modules and processors [3-8].

In this paper, we first introduce a switching theory framework of permutations. Then, using this formulation, we derive some interesting results that characterize some of the permutations that are known to be useful in parallel processing. We then develop a technique that characterizes certain shuffle-exchange networks and illustrate how this technique can be used to study any arbitrary permutation network. This technique is based on the process where a complex permutation (which is realized by a network in one or more passes) is decomposed into a sequence of elementary permutations.

First, we study the class of permutations which is admitted by the shuffle-exchange network proposed by Lawrie [3]. We then derive results which provide new insight into these permutations. Next, we focus our attention on the simplified version of the shuffle-exchange network proposed, subsequently, by Lang-Stone [4]. We assume a more general condition for these simplified networks and allow for all possible combinations of control functions for different passes through the network. Under this generalized condition, we derive results which exhibit certain characteristics of the permutations which are admitted by these networks. Furthermore, a tighter upper bound on the number of permutations admitted by the network is derived.

This paper is organized into two main sections. In the following section, a switching theory formulation of permutations is presented. This is useful in the second section where a procedure for characterizing memory processor interconnection networks is developed.

A Technique for Characterizing Permutations and Its Related Results

A permutation p can be defined as a one to one and on to mapping from a set of integers into itself. The permutation p is usually represented as $\{(i,p(i))|0 \le i \le N-1\}$ where $p(i)$ represents the mapping of i, $0 \le i \le N-1$.

In this paper, as in some of the previous work [3 - 8], we assume $N = 2^n$ for some n. In the following, we introduce F, an alternate representation of p.

Consider the binary numbers 0 to $N-1$. Let B^n represent these binary numbers. The set B^n consists of 2^n distinct binary n-tuples. Let $F:B^n \to B^n$ be a mapping, as defined below.

For each i, $0 \le i \le N-1$, let $\underline{i} = (i_n, i_{n-1},\ldots,i_1)$ denote the binary number i. Let $F(\underline{i}) = \underline{j}$ where $\underline{j} = (j_n,j_{n-1},\ldots,j_k,\ldots,j_1)$ denotes the binary number j, where $j = p(i)$. Thus, F is a one to one and on to mapping of B^n into B^n and is the permutation p expressed in terms of the binary numbers.

Now F, in turn, can be expressed as a collection of n switching functions $f_1,f_2,\ldots,f_k,\ldots,f_n$. Each of these functions f_k, $1 \le k \le n$ are n-variable functions, as defined below.

For $1 \le k \le n$, let

*Research reported supported in part by AFOSR Contract No. F49620-79-C-0119.

[†]Formerly with University of Regina, Regina, Canada.

[§]Currently with the Department of Computer Science, Wichita State University, Wichita, Kansas.

$$Y_k = f_k(y_n, y_{n-1}, \ldots, y_1)$$

and let

$$j_k = f_k(y_n = i_n, y_{n-1} = i_{n-1}, \ldots, y_1 = i_1)$$

where j_k is the kth component of the binary number $\underline{j}$, given by $F(\underline{i}) = \underline{j}$.

Example 1.

Consider the following permutation.

		y_3	y_2	y_1	Y_3	Y_2	Y_1
i	j = p(i)	i_3	i_2	i_1	j_3	j_2	j_1
0	0	0	0	0	0	0	0
1	5	0	0	1	1	0	1
2	6	0	1	0	1	1	0
3	7	0	1	1	1	1	1
4	1	1	0	0	0	0	1
5	2	1	0	1	0	1	0
6	3	1	1	0	0	1	1
7	4	1	1	1	1	0	0

$$Y_3 = \bar{y}_3\bar{y}_2y_1 + \bar{y}_3y_2 + y_3y_2y_1$$

$$Y_2 = \bar{y}_3y_2 + y_3\bar{y}_2y_1 + y_3y_2\bar{y}_1$$

$$Y_1 = \bar{y}_3y_1 + y_3\bar{y}_1$$

Thus, we see that every permutation on 2^n elements can be represented in terms of n switching functions.

In the following, we deduce these switching function expressions for several permutations useful in parallel processing. We reveal some interesting characteristics of these permutations.

Perfect Shuffle Permutation

This permutation has been shown to be useful in algorithms which are suitable for parallel processing, those designed for polynomial evaluation, sorting, etc. [4].

The perfect shuffle is defined as

$$p(i) = \begin{cases} 2i & 0 \le i \le 2^{n-1} - 1 \\ 2i - N + 1 & 2^{n-1} \le i \le 2^n \end{cases}$$

Theorem 1. The perfect shuffle permutation results in the following functions.

$$Y_i = \begin{cases} y_n & i = 1 \\ y_{i-1} & 2 \le i \le n \end{cases}$$

Proof.

Proof follows from the fact [2] that

$$p(\underline{i}) = (i_{n-1}, i_{n-2}, \ldots, i_1, i_n)$$

Q.E.D.

Exchange Permutation

This permutation has been shown to be useful in the design of dynamic memory [4] and in memory processor interconnection networks for array processors [3,4].

This exchange permutation is defined as

$$p(i) = \begin{cases} i+1 & i = \text{even} \\ i-1 & i = \text{odd} \end{cases}$$

The following is an immediate consequence of the definition, and hence there is no need for a proof.

Theorem 2. The exchange permutation results in the following functions.

$$Y_i = \begin{cases} \bar{y}_i & i = 1 \\ y_i & 2 \le i \le n \end{cases}$$

Uniform Shift

Uniform shift is an important permutation for parallel processing [4,6,7]. It has several applications, especially in matrix manipulation. The basic permutation performed in Illiac IV consists of four different types of uniform shift permutations. All other permutations are achieved by using these permutations.

Uniform shift is defined as

$$p(i) = (i + d) \bmod N$$

where d represents the amount of shift.

We need the following additional definitions which will be useful for characterizing this and other permutations to be presented later.

Definition. Let $g(x_n, x_{n-1}, \ldots, x_i, \ldots, x_1)$ be an n-variable function. Then, $dg/dx_i = g(x_n, x_{n-1}, \ldots, x_i=0, \ldots, x_1) \oplus g(x_n, x_{n-1}, \ldots, x_i=1, \ldots, x_1)$ where $\oplus$ is the exclusive-or operator. The function dg/dx_i is said to be the Boolean difference of g with respect to x_i.

Definition. Let $g(i) = g(x_n=i_n, x_{n-1}=i_{n-1}, \ldots, x_1=i_1)$ be the value of the function for the input $(i_n, i_{n-1}, \ldots, i_1)$, representing the binary number i.

Definition. The function $g(x_n, x_{n-1}, \ldots, x_1)$ is said to be a runlength-q function if, for some i, $g(i) = g(i+1) = \ldots = g(i+q-1) = 1$ and $g(i) = 0$ for all other i.

Example 2.

x_3	x_2	x_1	g
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

The above function $g(x_3,x_2,x_1) = \bar{x}_1x_2 + x_1\bar{x}_2x_3$ is a runlength-3 function. For this function, we can compute

$$\frac{dg}{dx_3} = x_1\bar{x}_2$$

It may be noted that, given the ex-or sum of products expression for g, the Boolean difference dg/dx_i can be obtained by simply deleting all terms that do not involve x_i and, also, all appearances of x_i.

Theorem 3. For uniform shift function, Y_n can be expressed as

$$Y_n = y_n \oplus g_n(y_{n-1},y_{n-2}, \cdots, y_1)$$

The other functions $Y_{n-1},Y_{n-2},\ldots,Y_1$ are related to Y_n through the following recursive rule.

If $Y_k = y_k \oplus g_k(y_{k-1},y_{k-2}, \cdots, y_1)$,

then $Y_{k-1} = y_{k-1} \oplus \dfrac{dg_k}{dy_{k-1}}$.

Proof.

Let $(d_n,d_{n-1},\ldots,d_1)$ be the binary-representation of the number d, the amount of shift. The permutation $p(i) = (i+d) \bmod N$ corresponds to addition of the fixed number d to each i. Since the addition is performed modulo $N = 2^n$, any carry from the nth position is discarded.

The variables $Y_n,Y_{n-1},\ldots,Y_1$ represent the sum bits produced by adding d to the number represented by the variables $y_n,y_{n-1},\ldots,y_1$.

Thus, the function Y_n can be expressed as $Y_n = y_n \oplus d_n \oplus C_n(y_{n-1},y_{n-2},\ldots,y_1)$, where C_n is the function representing the carry bit in to the nth position. However, it may be noted that, since d is a fixed integer for all i, C_n is expressed as a function of only the variables $y_{n-1},y_{n-2},\ldots,y_1$.

Let $g_n(y_{n-1},y_{n-2},\ldots,y_1) = d_n \oplus C_n(y_{n-1},y_{n-2},\ldots,y_1)$. Thus, $Y_n = y_n \oplus g_n(y_{n-1},y_{n-2},\ldots,y_1)$.

Similarly, for any k, we can express $Y_k = y_k \oplus g_k(y_{k-1},y_{k-2},\cdots y_1)$ where

$$g_k(y_{k-1},y_{k-2},\ldots,y_1) = d_k \oplus C_k(y_{k-1},y_{k-2},\ldots,y_1) \qquad (1)$$

The function $C_k(y_{k-1},y_{k-2},\ldots,y_1)$ represents the carry into the kth position.

Now, one can express

$$\begin{aligned} C_k&(y_{k-1},y_{k-2},\ldots,y_1) \\ &= d_{k-1}y_{k-1} + d_{k-1}C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) \\ &\quad + y_{k-1}C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) && (2) \\ &= d_{k-1}y_{k-1} \oplus d_{k-1}C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) \\ &\quad \oplus y_{k-1}C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) && (3) \end{aligned}$$

where $C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1)$ is the carry function into the (k-1)st position. Substituting (3) in (1), one has

$$\begin{aligned} g_k&(y_{k-1},y_{k-2},\ldots,y_1) \\ &= d_k \oplus d_{k-1}y_{k-1} \oplus d_{k-1}C_{k-1}(y_{k-2},\ldots,y_1) \\ &\quad \oplus y_{k-1}C_{k-1}(y_{k-2},\ldots,y_1) && (4) \end{aligned}$$

Now,

$$\begin{aligned} g_k&(y_{k-1}=0,y_{k-2},\ldots,y_1) \\ &= d_k + d_{k-1}C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) && (5) \end{aligned}$$

and

$$\begin{aligned} g_k&(y_{k-1}=1,y_{k-2},\ldots,y_1) \\ &= d_k \oplus d_{k-1} \oplus d_{k-1}C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) \\ &\quad \oplus C_{k-1}(y_{k-2},\ldots,y_1) && (6) \end{aligned}$$

From (5) and (6), one has

$$\frac{dg_k}{dy_{k-1}} = d_{k-1} \oplus C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) \qquad (7)$$

But,

$$Y_{k-1} = y_{k-1} \oplus d_{k-1} \oplus C_{k-1}(y_{k-2},y_{k-3},\ldots,y_1) \qquad (8)$$

Substituting (7) in (8), we get

$$Y_{k-1} = y_{k-1} \oplus \frac{dg_k}{dy_{k-1}}$$

Q.E.D.

Corollary 1. The function $g_n(y_{n-1},y_{n-2},\ldots,y_1)$ is a runlength-d function.

Proof.

Consider the following table which illustrates the function Y_n for a uniform shift of d.

y_n	y_{n-1}		...	y_1	Y_n	$Y_n + y_n$	
0	0		...	0	0	0	
0	0		...	1	0	0	
⋮						⋮	
0	0	1	...	1	1	1	
0	1	0	...	0	1	1	d
⋮						⋮	
0	1	1	...	1	1	1	
1	0	0	...	0	1	0	
1	0	0	...	1	1	0	
⋮						⋮	
1	0	1	...	1	0	1	
1	1	0	...	0	0	1	d
⋮						⋮	
1	1	1	...	1	0	1	

It is readily seen that $Y_n \oplus y_n = g_n(y_{n-1}, y_{n-2}, \ldots, y_1)$ is independent of y_n and is a runlength-d function.

Q.E.D.

Example 3.

Consider the following permutation which corresponds to a uniform shift of 3 on $N = 8$.

i	p(i)	y_3	y_2	y_1	Y_3	Y_2	Y_1
0	3	0	0	0	0	1	1
1	4	0	0	1	1	0	0
2	5	0	1	0	1	0	1
3	6	0	1	1	1	1	0
4	7	1	0	0	1	1	1
5	0	1	0	1	0	0	0
6	1	1	1	0	0	0	1
7	2	1	1	1	0	1	0

$$Y_3 = y_3 \oplus y_2 \oplus y_1 \oplus y_1 y_2$$

$$Y_2 = y_2 \oplus y_1 \oplus 1$$

$$Y_1 = y_1 \oplus 1$$

$$g_3(y_3, y_2, y_1) = y_2 \oplus y_1 \oplus y_1 y_2$$

$$g_2(y_3, y_2, y_1) = y_1 \oplus 1$$

$$g_1(y_3, y_2, y_1) = 1$$

It may be seen that

$$g_2(y_3, y_2, y_1) = \frac{dg_2}{dy_1} = \frac{d}{dy_1}\left(\frac{dg}{dy_2}\right)$$

Unscrambling of t-Ordered Vector

This permutation has been found useful in aligning data in memory modules to obtain conflict-free access of related items in an array [3-5].

This permutation is defined as $p(i) = t \cdot i \bmod N$ where t is relatively prime to N.

First, it may be observed that, since $N = 2^n$, t must be an odd number and hence must have a 1 in the least significant position. Let $t_n, t_{n-1}, \ldots, t_2, t_1$ be the binary representation of t. In this, $t_1 = 1$.

Theorem 4. For the permutations that realize the unscrambling of t-ordered vectors, Y_n can be expressed as $Y_n = y_n \oplus g_n(y_{n-1}, y_{n-2}, \ldots, y_1)$. The functions $Y_{n-1}, Y_{n-2}, \ldots, Y_1$ are related to Y_n through the following recursive rule.

If $Y_k = y_k \oplus g_k(y_{k-1}, y_{k-2}, \ldots, y_1)$,

then $Y_{k-1} = y_{k-1} \oplus \dfrac{dg_k^*}{dy_{k-1}}$, where

$$\frac{dg_k^*}{dy_{k-1}} = \begin{cases} \dfrac{dg_k}{dy_{k-1}} & \text{if } (t-1)/2 \text{ is even} \\[2ex] 1 \oplus \dfrac{dg_k}{dy_{k-1}} & \text{if } (t-1)/2 \text{ is odd} \end{cases}$$

Proof.

Consider Table 1 which illustrates the various terms in the multiplication of $(t_n, t_{n-1}, \ldots, t_1)$ with the variables $(y_n, y_{n-1}, \ldots, y_1)$.

Since the multiplication is performed modulo $N = 2^n$, we can discard all the partial product bits to the left of the nth bit position, as shown by the dotted line.

First, it may be observed that, in each column, there may be one or more carry bits in addition to the partial product bits. These carry bits are generated during the summation of the columns to the right of this column, which are then propagated to this position. Let C_k, $2 \leq k \leq n$, represent the ex-or sum or these carry bits in each position. Note that $C_1 = C_2 = 0$.

Since t is a fixed number, C_k is a function of $y_{k-1}, y_{k-2}, \ldots, y_1$ only. Thus,

$$Y_n = t_1 y_n \oplus t_2 y_{n-1} \oplus \cdots \oplus t_n y_1$$
$$\oplus\, C_{n-1}(y_{n-1}, y_{n-2}, \ldots, y_1)$$

Let

$$g_n(y_{n-1}, y_{n-2}, \ldots, y_1) = t_2 y_{n-1} \oplus \cdots \oplus t_n y_1$$
$$\oplus\, C_{n-1}(y_{n-1}, y_{n-2}, \ldots, y_1)$$

Since $t_1 = 1$, one has

$$Y_n = y_n \oplus g_n(y_{n-1}, y_{n-2}, \ldots, y_1)$$

Table 1

MULTIPLICATION TABLE

				y_n	y_{n-1}	... y_k	y_{k-1}	...	y_1
				t_n	t_{n-1}	... t_k	t_{k-1}	...	t_1
			t_1y_n	t_1y_k	t_1y_{k-1}	...	t_1y_3	t_1y_2	t_1y_1
		t_2y_n	t_2y_{n-1}	t_2y_{k-1}	t_2y_{k-2}	...	t_2y_2	t_2y_1	
							t_3y_1		
							C_3		
	$t_{k-1}y_n$	$t_{k-1}y$	$t_{k-1}y_{n-k+2}$	$t_{k-1}y_2$	$t_{k-1}y_1$				
t_ky_n		t_ky_{n-k}	t_ky_{n-k+1}	t_ky_1	C_{k-1}				
				C_k					
		t_ny_2	t_ny_1						
			C_n						
			Y_n ...	Y_k	Y_{k-1}	...	Y_3	Y_2	Y_1

Similarly, Y_k can be expressed as

$$Y_k = y_k \oplus g_k(y_{k-1},y_{k-2},\ldots,y_1)$$

where

$$g_k(y_{k-1},y_{k-2},\ldots,y_1) = t_2y_{k-1} \oplus t_3y_{k-2} \oplus \cdots \oplus t_ky_1 \oplus C_k(y_{k-1},y_{k-2},\ldots,y_1)$$

Now,

$$g_k(y_{k-1}=0,y_{k-2},\ldots,y_1) = t_3y_{k-2} \oplus \cdots \oplus t_ky_1 \oplus C_k(Y_{k-1}=0,y_{k-2},\ldots,y_1)$$

and

$$g_k(y_k=1,y_{k-2},\ldots,y_1) = t_2 \oplus t_3y_{k-2} \oplus \cdots \oplus t_ky_1 \oplus C_k(y_{k-1}=1,y_{k-2},\ldots,y_1)$$

Thus,

$$\frac{dg_k}{dy_{k-1}} = t_2 \oplus C_k(y_{k-1}=0,y_{k-2},\ldots,y_1) \oplus C_k(y_{k-1}=1,y_{k-2},\ldots,y_1)$$

$$= t_2 \oplus \frac{dC_k}{dy_{k-1}} \qquad (9)$$

We can express $C_k(y_{k-1},y_{k-2},\ldots,y_1)$, which is the ex-or sum of all the carry bits into the kth position, as

$$C_k(y_{k-1},y_{k-2},\ldots,y_1) = C_k^1(y_{k-1},y_{k-2},\ldots,y_1) \oplus C_k^2(y_{k-2},y_{k-3},\ldots,y_1)$$

In this, $C_k^1(y_{k-1},y_{k-2},\ldots,y_1)$ represents the function for the carry bit produced in the addition of the (k-1)st column only. Whereas $C_k^2(y_{k-2},y_{k-3},\ldots,y_1)$ represents the function for ex-or sum of the carry bits produced in the addition of jth columns $2 \leq j \leq (k-2)$ and which are propagated into the kth column. It may be noted that y_{k-1} does not appear anywhere in the 1st through (k-2)nd column. The first time y_{k-1} appears is in the (k-1)st column. Therefore, the function C_{k-1}^2 is independent of y_{k-1}. Thus,

$$\frac{dC_k}{dy_{k-1}} = C_k^1(y_{k-1}=0,y_{k-2},\ldots,y_1) \oplus C_k^1(y_{k-1}=1,y_{k-2},\ldots,y_1) \oplus C_k^2(y_{k-2},y_{k-3},\ldots,y_1) \oplus C_k^2(y_{k-2},y_{k-3},\ldots,y_1)$$

$$= \frac{dC_k^1}{dy_{k-1}} \qquad (10)$$

Consider the function $C_k^1(y_{k-1},y_{k-2},\ldots,y_1)$. This, by definition, is the carry bit into kth column due to the addition of the terms y_{k-1}, $t_2y_{k-2}, t_3y_{k-3},\ldots,t_{k-1}y_1,C_{k-1}$ which appear in (k-1)st column. This function is represented in Table 2.

The function $C_k^1 = 1$ represents all the rows which have 2i 1's for i = odd integers.

Table 2

THE TRUTH TABLE FOR C_k^1

	y_{k-1}	t_2y_{k-2}	t_3y_{k-3}		$t_{k-1}y_1$	C_{k-1}	C_k^1	
	0	0	0	...	0	0	0	
	0	0	0		0	1	0	
	0	0	0		1	0	0	
B_0	0	0	0		1	1	1	← i
	⋮							
	0	1	1	...	1	1		
	1	0	0	...	0	0	1	
	1	0	0		0	1	1	
	1	0	0		1	0	1	
B_1	⋮							
	1	0	0		1	1	1	← i
	⋮							
	1	1	1	...	1	1		

This follows from the observation that, if the number of 1's in the (k-1)st column is equal to 2, 6, 10, 14, etc., then we get a carry in to the kth column.

Now, consider the function

$$\frac{dC_k^1}{dy_{k-1}} = C_k^1(y_{k-1}=0, y_{k-2}, \ldots, y_1) \oplus C_k^1(y_{k-1}=1, y_{k-2}, \ldots, y_1)$$

The truth table for this function can be computed in the following way. First, note that dC_k^1/dy_{k-1} is independent of y_{k-1}, and hence the truth table will have half as many rows as the above one for C_k^1. The value of the function dC_k^1/dy_{k-1} for the ith row can be computed as follows: Take the value of the function C_k^1 for the ith row in the upper half B_0, and then compute the ex-or sum of this with the value of the function for the ith row in the lower half B_1 as shown in Table 2. The ith rows in the B_0 and B_1 are identical in the $t_2y_{k-1}, t_3y_{k-1}, \ldots, C_{k-1}$ positions. Further, it may be seen that, if the ith rows have an even number of 1's in these positions, then the two values of C_k^1 are identical. On the other hand, if they have an odd number of 1's, then C_k^1 has complementary values for these two rows. This, therefore, implies that dC_k^1/dy_{k-1} is equal to 1 for the ith row only if it has an odd number of 1's in the $t_2y_{k-1}, t_3y_{k-1}, \ldots, C_{k-1}$ positions and is equal to 0 otherwise. From this, it can now be deduced that

$$\frac{dC_k^1}{dy_{k-1}} = t_2y_{k-1} \oplus t_3y_{k-2} \oplus \cdots \oplus t_{k-1}y_1 \oplus C_{k-1}$$

Substituting this first in (10) and then in (9), one has

$$\frac{dg_k}{dy_{k-1}} = t_2 \oplus t_2y_{k-1} \oplus t_3y_{k-2} \oplus \cdots \oplus t_{k-1}y_1 \oplus C_{k-1} \tag{11}$$

But

$$Y_{k-1} = y_{k-1} \oplus t_2y_{k-1} \oplus t_3y_{k-2} + \cdots \oplus t_{k-1}y_1 \oplus C_{k-1} \tag{12}$$

From (11) and (12)

$$Y_{k-1} = y_{k-1} \oplus t_2 \oplus \frac{dg_k}{dy_{k-1}}$$

<u>Case I</u>: (t-1)/2 = even number

In this case, $t_2 = 0$

$$Y_{k-1} = y_{k-1} \oplus \frac{dg_k}{dy_{k-1}}$$

<u>Case II</u>: (t-1)/2 = odd number

In this case, $t_2 = 1$

$$Y_{k-1} = y_{k-1} \oplus \frac{dg_k}{dy_{k-1}} \oplus 1$$

Q.E.D.

The other permutations [4] such as the interchange of elements 2^{n-r} apart and

$p(i) = (t-i) \bmod N$ have characterizations similar to those presented in Theorems 3 and 4.

In the next section, some of the above results will be used in formulating a procedure for characterization of memory processor interconnection networks.

On Memory-Processor Interconnection Networks

In this section, a procedure for characterizing permutation networks is developed. This procedure is based on the following observations: (1) that any permutation which is realized by a network, after n passes through the network, can be expressed as a composition (sequence) of n permutations; (2) that each of these n permutations, in turn, can be expressed as a composition of certain elementary permutations. (These elementary permutations were described in the last section.) Using the earlier derived results (that characterized these elementary permutations), one can characterize the permutations admitted by the network.

In the following, this technique is first developed for shuffle-exchange networks; then, we illustrate its applicability to any arbitrary network. The shuffle-exchange networks considered here are the ones proposed by Lawrie [3], and its simplification, that proposed by Lang-Stone [4].

First, a brief description of these two types of networks is given below. (From this point on, the shuffle-exchange network presented by Lawrie [3] will be referred to as SE, and the simplified version by Lang-Stone [4] will be referred to as an SSE network.) Figure 1 presents an abstraction of these two networks.

These networks are N-input, N-output networks, and they can perform many useful permutations on N data items, where $N = 2^n$, for some n; the data items are circulated through the network n times (passes) to achieve the desired permutation.

These networks are composed of two subnetworks which are denoted as S and E in Fig. 1. The first subnetwork, S, moves the contents of its input, i, to its output, j, where $j = p(i)$, and p is the perfect shuffle map. The second subnetwork, E, moves the contents of its inputs, i, and $(i + 1)$ to its output, $(i + 1)$ and i, respectively, for certain selected i's, where i is an even number. For all other inputs, i, the contents are moved straight through to the outputs, i. Thus, in effect, E performs exchanges on the contents of certain selected pairs of adjacent inputs.

A set of control bits determines the subset of pairs which are selected for exchange. One bit per pass per data item is all that is needed for controlling the exchange operation. The data items carry with them these control bits, and, thus, the control bits form an integral part of the contents of the inputs or outputs.

The number of control bits that are required for operation in the SSE network is a substantial number less than that needed in the SE network.

In the case of SE networks, each data item carries with it n-control bits. During the kth pass, the kth control bits are used for controlling the exchange operation. The contents of a certain pair of inputs are exchanged (not exchanged) during the kth pass, if kth control bits (which are contained in both of these inputs) are 1 (0).

On the other hand, the SSE network receives only one control bit per data item. These bits constitute the control bits that are used during the first pass. These control bits are used in the same way as in the SE network where they control the exchange operation during the first pass. However, for every successive pass after this first pass, a <u>new</u> control bit is computed for each data item, and these new bits now control the exchange operation during that pass. These new bits are computed from the control bits that have been used during the immediately preceding pass; this is described below:

During any pass other than the first, certain predefined Boolean operations are performed on every pair of control bits (which are contained in the ith and (i+1)th inputs of E for all even i). The bits produced by these Boolean operations then replace the existing control bits (in their respective pairs of inputs), and these new bits are then used by E as control bits to perform the exchange operation for that current pass.

Let $\{(i, p(i)) \mid 0 \leq i \leq N-1\}$ be any permutation that is admitted (realized) by an SE or SSE network in n passes through the network. Let this permutation, p, be represented by n-switching functions, $Y_n, Y_{n-1}, \ldots, Y_1$, of the variables $y_n, y_{n-1}, \ldots, y_1$. In the following, we derive several results that characterize these Y_j functions; these will provide a new perspective on the permutations admitted by SE and SSE networks:

First, we introduce additional notations which will be useful in deriving the results. Let $\{(i, p_k, (i)) \mid 0 \leq i \leq N-1\}$ be any permutation which is admitted by the network in k passes. Let the permutation, p_k, be represented by n-switching functions, $Y_n^k, Y_{n-1}^k, \ldots, Y_1^k$, of the variables $y_n, y_{n-1}, \ldots, y_1$.

Next, let $\{(i, p_{ks}(i)) \mid 0 \leq i \leq N-1\}$ be that intermediate permutation which is realized at the output of the shuffle network, during the kth pass, which results, at the completion of the kth pass, in the permutation, p_k, at the network output. Let this permutation, p_{ks}, be represented by n-switching function, $X_n^k, X_{n-1}^k, \ldots, X_1^k$ of the variables, $y_n, y_{n-1}, \ldots, y_1$.

Finally, let C_i^k represent that control bit used in E for the contents of its ith input during the kth pass.

It follows from the above notations that for all j, $1 \leq j \leq n$:

$$Y_j^k = \begin{cases} y_j & \text{for } k = 0, \text{ and} \\ Y_j & \text{for } k = n \end{cases} \tag{13}$$

Lemma 1.

$$X_j^k = \begin{cases} Y_{j-1}^{k-1} & \text{for } 2 \leq j \leq n, \text{ and} \\ Y_n^{k-1} & \text{for } j = 1 \end{cases}$$

Proof.

Proof is a direct consequence of Theorem 1 in conjunction with the fact that the outputs of the (k-1)th pass are fed back to form the inputs to the kth pass.

Q.E.D.

Theorem 5. The Y_k functions that represent any permutation which is admitted by the SE network can be expressed as

$$Y_k = y_k \oplus f_k(Y_n, Y_{n-1}, \ldots, Y_{k+1}, y_{k-1}, y_{k-2}, \ldots, y_1) \quad \text{for all } k, 1 \leq k \leq m$$

where the function, f_k, is defined by the control bits C_i^k that are used during the kth pass.

Proof.

For the sake of simplicity, this proof will be developed in two parts: First, we will show that

$$Y_n = y_n \oplus f_n(y_{n-1}, y_{n-2}, \ldots, y_1)$$

Then, we prove, in general, that

$$Y_k = y_k \oplus f_k(Y_n, Y_{n-1}, \ldots, Y_{k+1}, y_{k-1}, y_{k-2}, \ldots, y_1)$$

Using both Lemma 1 (for $k = 1$) and the relationship (13), one has

$$X_j^1 = \begin{cases} Y_{j-1} & 2 \leq j \leq n \\ Y_n & j = 1 \end{cases} \tag{14}$$

The following table describes the relationship between $X_n^1, X_{n-1}^1, \ldots, X_1^1$ and $Y_n^1, Y_{n-1}^1, \ldots, Y_1^1$.

Table 3 is derived by using the following observations regarding the mapping of input addresses to output addresses, as performed by E:

(a) The output pair is identical to the input pair when the input pair is not exchanged.

(b) On the other hand, when an input pair <u>is</u> exchanged, the resulting output pair has the following characteristic:

The binary numbers that represent the output pair are identical to the binary numbers that represent the input pair in all the positions except the least significant position. The bit in the least significant position of the output pair is exactly the complement of the bit in the least significant position of the input pair.

(c) The least significant bit of any input pair, i, and (i+1) is 0 and 1, respectively, because i is even.

(d) The input pair, i, and (i+1), is exchanged if $C_i^1 = C_{i+1}^1 = 1$; and the pair is not exchanged if $C_i^1 = C_{i+1}^1 = 0$.

Table 3

Y AND X VARIABLES DURING THE FIRST PASS

	X_n^1	X_{n-1}^1	$\ldots$	X_2^1	X_1^1	Y_n^1	Y_{n-1}^1	$\ldots$	Y_2^1	Y_1^1
	0	0		0	0	0	0		0	C_0^1
	0	0		0	1	0	0		0	$\bar{C}_1^1$
	0	0		1	0	0	0		1	C_2^1
i	- -	- -	- -	- -	0	- -	- -	- -	- -	C_i^1
i+1	- -	- -	- -	- -	1	- -	- -	- -	- -	$\bar{C}_{i+1}^1$
	1	1		1	0	1	1		1	$C_{2^n-2}^1$
	1	1		1	1	1	1		1	$\bar{C}_{2^n-1}^1$

As a direct consequence of the above observations, it is evident that all the Y_j's are identical to all of the X_j's, except for $j=1$. The column, Y_1^1, can be represented as C_i^1 and $\bar{C}_{i+1}^1$, in the ith and (i+1)th rows, respectively for all i.

From this, it can be derived that:

(1) $Y_j^1 = X_j^1, \quad 2 \leq j \leq n,$

(2) Y_1^1 is a function of $X_n^1, X_{n-1}^1, \ldots, X_1^1$.

The function, Y_1^1, can be expressed as shown below:

$$C_0^1 \bar{X}_n^1 \bar{X}_{n-1}^1 \cdots \bar{X}_2^1 \bar{X}_1^1 \oplus \bar{C}_0^1 \bar{X}_n \bar{X}_{n-1} \cdots \bar{X}_2^1 X_1^1 \oplus \cdots$$
$$\oplus \bar{C}_{2^n-2}^1 X_n^1 X_{n-1}^1 \cdots X_2^1 \bar{X}_1^1 \oplus C_{2^n-2} X_n^1 X_{n-i}^1 \cdots X_1^1 \tag{15}$$

since $C_i^1 = C_{i+1}^1$ for all even i.

Consider the following well-known identities in Boolean algebra:

(a) $P \oplus Q = P + Q, \quad \text{if} \quad PQ = 0$ (16)

(b) $\bar{P} = 1 \oplus P$ (17)

(c) $P \oplus \bar{P} = 1$ (18)

Using these identities, one can express (15) as:

$$Y_n \oplus C_0^{1}\bar{y}^{1}_{n-1}\bar{y}_{n-2}\cdots\bar{y}_2\bar{y}_1 \oplus C_2^{1}\bar{y}_{n-1}\bar{y}_{n-2}\cdots\bar{y}_2 y_1 \oplus \cdots$$

$$\oplus\, C^{1}_{2^n-4} y_{n-1}y_{n-2}\cdots y_2\bar{y}_1 \oplus C^{1}_{2^n-2} y_{n-1}\bar{y}_{n-2}\cdots y_2 y_1$$

using (13) and Lemma 1.

Let $f_n(y_{n-1}, y_{n-2}, \ldots, y_1)$

$$= C_0^{1}\bar{y}_{n-1}\bar{y}_{n-2}\cdots\bar{y}_2\bar{y}_1 \oplus C_2^{1}\bar{y}_{n-1}\bar{y}_{n-1}\bar{y}_{n-2}\cdots\bar{y}_2 y_1 \oplus \cdots$$

$$\oplus\, C^{1}_{2^n-4} y_{n-1}y_{n-2}\cdots y_2\bar{y}_1 \oplus C^{1}_{2^n-2} y_{n-1}y_{n-2}\cdots y_2 y_1$$

Thus,

$$Y_1^1 = y_n \oplus f_n(y_{n-1}, y_{n-2}, \ldots, y_1) \qquad (19)$$

Using Lemma 1 and the above observation regarding the least significant Y-bit, one can state that, in general:

(a) $Y_j^k = y_{j-1}^{k-1} \quad 2 \le j \le n,$ and

(b) Y_1^k is a function of the variables, $y_n^{k-1}, y_{n-1}^{k-1}, \ldots, y_1^{k-1}$. This, in turn, implies the following, in general, for any k:

$$Y_1^{n-k+1} = Y_k^n = Y_k \qquad 1 \le k \le n \qquad (20)$$

$$Y_k^j = Y_{n-k+j}^n = Y_{n-k+j} \qquad 1 \le j \le k \qquad (21)$$

$$Y_j^k = Y_{j-k}^0 = Y_{j-k} \qquad (k+1) \le j \le n \qquad (22)$$

Substituting $k=n$ in (20), one has:

$$Y_1^1 = Y_n$$

Thus, Eq. (19) now becomes:

$$Y_n = y_n \oplus f_n(y_{n-1}, y_{n-2}, \ldots, y_1)$$

Now, to prove the theorem, in general, for any Y_k, consider the $(n-k+1)$th pass through the network. For the sake of convenience, let $n-k+1$ be denoted as m.

One can derive the following equation for $Y_1^m = Y_1^{n-k+1}$, by using techniques similar to those used for deriving Y_1^1.

$$Y_1^m = X_1^m \oplus C_0^m\bar{X}_n^m\bar{X}_{n-1}^m\cdots\bar{X}_3^m\bar{X}_2^m \oplus C_2^m\bar{X}_n^m\bar{X}_{n-1}^m\cdots\bar{X}_3^m X_2^m + \cdots$$

$$\oplus\, C^m_{2^n-2} X_n^m X_{n-1}^m\cdots X_3^m X_2^m \qquad (23)$$

Using Lemma 1 in conjunction with Eqs. (20) through (22), one can deduce the following:

$$X_1^m = X_1^{n-k+1} = Y_k \qquad (24)$$

$$X_j^m = X_j^{n-k+1} = Y_{k+j-1} \qquad 2 \le j \le m \qquad (25)$$

$$X_j^m = X_j^{n-k+1} = Y_{j-m} \qquad m+1 \le j \le n \qquad (26)$$

Substituting (24) through (26) in (23), one has

$$Y_k = y_k \oplus C_0^m\bar{y}_{k-1}\bar{y}_{k-2}\cdots\bar{y}_1\bar{Y}_n\bar{Y}_{n-1}\cdots\bar{Y}_{k+2}\bar{Y}_{k+1}$$

$$\oplus\, C_2^m\bar{y}_{k-1}\bar{y}_{k-2}\cdots\bar{y}_1\bar{Y}_n\bar{Y}_{n-1}\cdots\bar{Y}_{k+2}Y_{k+1} + \cdots$$

$$\oplus\, C^m_{2^n-2} y_{k-1}y_{k-2}\cdots y_1 Y_n Y_{n-1}\cdots Y_{k+2}Y_{k+1}$$

$$= y_k + f_k(Y_n, Y_{n-1}, \ldots, Y_{k+1}, y_{k-1}, y_{k-2}, \ldots, y_1)$$

where

$$f_k(Y_n, Y_{n-1}, \ldots, Y_{k+1}, y_{k-1}, y_{k-2}, \ldots, y_1)$$

$$= C_0^m\bar{y}_{k-1}y_{k-2}\cdots\bar{y}_1\bar{Y}_n\bar{Y}_{n-1}\cdots\bar{Y}_{k+2}\bar{Y}_{k+1}$$

$$\oplus\, C_2^m\bar{y}_{k-1}\bar{y}_{k-2}\cdots\bar{y}_1\bar{Y}_n\bar{Y}_{n-1}\cdots\bar{Y}_{k+2}\bar{Y}_{k+1} + \cdots$$

$$\oplus\, C^m_{2^n-2} y_{k-1}y_{k-2}\cdots y_1 Y_n Y_{n-1}\cdots Y_{k+2}Y_{k+1}$$

Q.E.D.

In the following, $\log N$ denotes $\log_2 N$. The following corollaries are a direct consequence of the above theorem.

Corollary 2. There are exactly $\sqrt{2^N \log N}$ distinct permutations which are admitted by the SE network.

In the following, we derive an interesting consequence of Theorem 5 which characterizes the permutations that are admitted by the SE network.

Let $p(i) = j$ and $p(i') = j'$ for some permutation p where $0 \le i, j, i', j' \le N-1$. Let

$$\underline{i} = (i_n, i_{n-1}, \ldots, i_k, \ldots, i_1)$$

$$\underline{j} = (j_n, j_{n-1}, \ldots, j_k, \ldots, j_1)$$

and

$$\underline{i} = (i'_n, i'_{n-1}, \ldots, i'_k, \ldots, j_1)$$

$$\underline{j} = (j'_n, j'_{n-1}, \ldots, j'_k, \ldots, j_1)$$

be the binary representation of i, j, i', and j', respectively.

Definition. A permutation p is said to be symmetric in the kth bit if it satisfies the following: Given any i, i', for which

$i_j = i'_j$ for all $j \neq k$, and i_k is the complement of i'_k ($i'_k = \bar{i}_k$), then the kth bits of the resulting pair, j and j', are also the complement of each other, i.e., $j'_k = \bar{j}_k$.

Definition. A permutation will be said to be symmetric if it is symmetric in all the n-bits.

Corollary 3. Any permutation which is admitted by the SE network is symmetric in the most significant bit.

What is interesting about the above corollary, also, is that it provides an explanation as to why such a permutation as bit reversal is not admissible by the SE network.

Next, we extend our study to the SSE networks and present several results that reveal the structure of the permutations which are admitted by these networks.

In the following, we discuss certain generalized versions of the SSE networks; this enables us to derive results with broader implications.

Let the control function that is used in any pass be a function of certain tag bits which are transmitted with the data items. Thus, one can select any arbitrary control function for any pass. (However, it may be noted that we restrict the control function to be the same for all data items during a particular pass.) Therefore, so as to produce the desired permutation, the use of any combination of control functions for the n-different passes is available.

Four tag bits are required in order to specify 1-out-of-16 possible different two-variable functions. Since there are n-passes, only 4n additional tag bits are required altogether. (For example, given 256 data items, only 32 tag bits are required--a small number when compared to the 256 control bits that are used.) As it will be seen later, the use of these additional bits can produce a much larger number of permutations. This is significant when compared to the number of permutations that are admitted by the network when the control functions are prespecified.

This version of the SSE network, which allows for arbitrary control functions, will be hereafter referred to as the "SSEAC" network.

In the following, we present two lemmas which are then used to derive Theorem 6. This theorem provides a switching function characterization of the permutations which are admitted by the above SSEAC networks. Then, certain consequences of the theorem are examined, and these provide further insight into the permutations admitted by these networks.

Lemma 2.

$$(a_0P_0 \oplus a_1P_1 \oplus \cdots + a_tP_t) * (b_0P_0 \oplus b_1P_1 \oplus \cdots \oplus b_tP_t) = (a_0 * b_0)\, P_0 \oplus (a_1 * b_1)\, P_1 \oplus \cdots \oplus (a_t * b_t)\, P_t$$

where

(1) a_i's and b_i's are constants and are equal to 0 or 1

(2) P_i's are product terms over some variables

(3) $*$ is any Boolean operation

(4) $P_iP_j = 0$ for all i,j and $i \neq j$

(5) $P_0 \oplus P_1 \oplus \cdots \oplus P_t = 1$

Lemma 3. The control bits that are used in the SSEAC network satisfy the following relationship:

$$C_{2i}^{m+1} = C_{2i+1}^{m+1} = C_i^m * C_{i+2^{n-1}}^m \qquad 0 \leq i \leq 2^n - 1$$

where $*$ is that Boolean operation used during the (m+1)th pass to produce the new control bits.

Theorem 6. Any permutation that is admitted by the SSEAC network can be represented by functions defined as given below:

$Y_n = y_n \oplus f_n(y_{n-1}, y_{n-2}, \ldots, y_1)$ and if

$Y_k = y_k \oplus f_k(y_{k-1}, y_{k-2}, \ldots, y_1)$ then

$Y_{k-1} = y_{k-1} \oplus f_{k-1}(y_{k-2}, y_{k-3}, \ldots, y_1)$ where

$$f_{k-1}(y_{k-2}, y_{k-3}, \ldots, y_1) = f_k(y_{k-1}=0, y_{k-2}, \ldots, y_1) * f_k(y_{k-1}=1, y_{k-2}, \ldots, y)$$

and $*$ is that Boolean operation which is used during the (n-k+2)th pass to compute the control bits.

Proof.

The proof is based on Lemmas 2 and 3 as well as some of the observations made in Theorem 5.

Q.E.D.

Now, it may be noted that, if the control function that is used in the network is either an exclusive-or function or an equivalence function (as it is proposed in [2]), then the recursive relationship between Y_k and Y_{k-1} reduces to the following.

If $Y_k = y_k \oplus f_k(y_{k-1}, y_{k-2}, \ldots, y_1)$, then

$$Y_{k-1} = \begin{cases} y_{k-1} \oplus \dfrac{df_k}{dy_k} & \text{if the control function is ex-or} \\ y_{k-1} \oplus \dfrac{df_k}{dy_k} \oplus 1 & \text{if the control function is equivalence} \end{cases}$$

It is also interesting to note that the above relationship is precisely the same relationship derived for those permutations described in Theorems 3 and 4.

Corollary 4. Any permutation which is admitted by the SSEAC network is symmetric.

Proof.

Proof is an immediate consequence of Theorem 6.

Q.E.D.

Corollary 5. The number of permutations that are admitted by the SSEAC network is, at most, equal to

$$\sqrt{2^{N+8\log N-8}}$$

Corollary 6. The fraction of SE permutations which are admitted by the SSEAC network tends to 0, asymptotically, with N.

Some implications of the above results that relate to SSE networks are as follows.

(1) All of the permutations which are admitted by those SSE networks that use ex-or and equivalence as control functions have representations that are similar to those characterizing permutations such as uniform shift, unscrambling of t-ordered vectors, etc.

(2) The use of additional tag bits so as to provide for any desired combination of control functions can produce a much larger number of permutations.

(3) Any permutation which is admitted by the SSE network is symmetric.

(4) For large values of N, the set of permutations which are admitted by the SSE network is a relatively small subset of those permutations which are admitted by the SE network (even all possible combinations of control functions are provided for the SSE network).

We discuss in the following how to characterize any arbitrary permutation network. It is a well-known fact that any permutation can be expressed as a composition of transpositions [9]. Since any transposition can be realized by a combination of uniform shift permutations and exchange permutations, it follows that any arbitrary permutation can be expressed as a composition of uniform shift permutations and exchange permutations. Thus, any permutation that is realized after any number of passes through the network can be expressed as a composition of uniform shift and exchange permutations. Therefore, using the techniques presented in this section, it is possible to characterize any permutation network, given its structure.

(Any of the proofs omitted here can be obtained directly from the authors.)

Conclusion

A general framework for the study of permutations and permutation networks has been introduced in this paper. Through this formulation, we derived certain characterizations for some well-known permutations. These results then led to the development of a procedure to study the characteristics of permutation networks, in general. Also, several results that directly relate to shuffle-exchange networks were derived as well.

References

[1] J.P. Hayes, Computer Architecture and Organization, McGraw-Hill (1978).

[2] H.S. Stone, "Parallel Processing with Perfect Shuffle," IEEE Trans. on Computers, Vol. C-20 (Feb 1971), pp. 153-161.

[3] D.H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers, Vol. 4-21, No. 12 (Dec 1975), pp. 1145-1175.

[4] T. Lang and H. Stone, "A Shuffle-Exchange Network with Simplified Control," IEEE Trans. on Computers, Vol. C-25, No. 1 (Jan 1976).

[5] R.C. Swanson, "Interconnection of Parallel Memories to Unscramble p-Ordered Vectors," IEEE Trans. on Computers, Vol. C-23 (Nov 1974), pp. 1105-1116.

[6] M.C. Pease, "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Computers, Vol. C-26 (May 1977), pp. 458-473.

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

[8] H.J. Siegel, "The University of Various Types of SIMD Machine Interconnection Networks," Proc. 4th Ann. Symp. on Computer Architecture (Mar 23-25, 1977).

[9] A.R. Mitchell and R.W. Mitchell, An Introduction to Abstract Algebra, Brook-Cole Publishing Co. (1970).

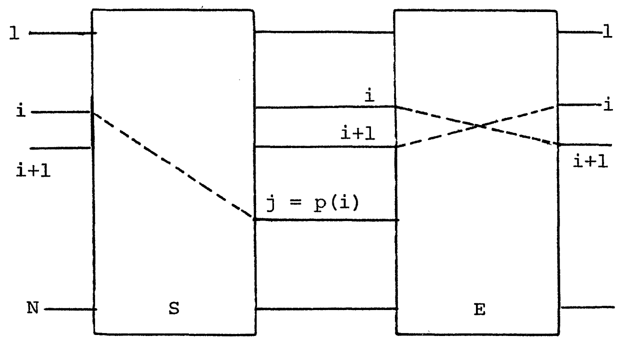


Fig. 1. SHUFFLE-EXCHANGE NETWORK.

On Conflict-free Permutations in Multi-Stage Interconnection Networks

Mohammad A. Abidi and Dharma P. Agrawal
Electrical and Computer Engineering
Wayne State University
Detroit, Michigan 48202

Summary

Various multistage interconnection networks proposed in the literature provide fast and efficient communication between processors and memory modules. These include Data Manipulator, Flip network, Omega network, regular SW Banyan network, indirect Binary n-cube network and Baseline network. In these networks, there exist certain numbers of interconnection permutations which cannot be achieved in one pass. In these permutations, at least one common link is required for establishing communication between two or more distinct pairs of desired inlets-outlets. Such a link is said to be in conflict. The objective of this paper is to identify the families of permutations which can be passed conflict-free for multistage networks mentioned earlier.

Lenfant [1] has classified certain families of these permutations or bijections which are most likely to occur in parallel processing. Three of these families are conflict-free for Omega network [2]. Even though the other multistage networks are topologically equivalent [3], these bijections cannot be performed in a single pass in any other network. In this paper conflict-free families for various multistage interconnection networks are obtained. The number of such permutations for each network is seen to be the same. But they are disjoint to each other. Hence all these networks combined together provide much more coverage of conflict-free permutations. It might be pertinent to mention that if the frequency of occurrence of certain types of conflict-free permutations are predetermined, then this paper does provide a novel technique for selecting the type of network(s) best suited for that application.

In a general $\ell x \ell$ Omega network if we define a mapping for each switching element $(p_\ell p_\ell p_{\ell-2} \cdots p_1)$ for i^{th} stage as under:

$$\gamma_i[(p_\ell p_{\ell-1} \cdots p_2 p_1)_i] = (p_i \cdots p_1 p_{i+1} \cdots p_\ell) \quad \text{for} \quad 0 \leq i \leq \ell \qquad (1)$$

we obtain Omega network which is equivalent to Baseline network. The objective here is to identify families of permutations for Baseline network which corresponds to given families in Omega network. As we are interested only in the equivalent permutations, we consider the switching elements only at the input and output terminals. Thus substituting i=0 and i=ℓ in relation (1) we get:

for i=0

$$\gamma_0[(p_\ell p_{\ell-1} \cdots p_1)_0] = (p_1 p_2 \cdots p_\ell)_0 \qquad (2.a)$$

for i=ℓ

$$\gamma_\ell[(p_\ell p_{\ell-1} \cdots p_1)_\ell] = (p_\ell p_{\ell-1} \cdots p_1)_\ell \qquad (2.b)$$

Though these mappings are defined for switching elements, they are also true for the input and output links of the Omega network. Now equation (2.a) indicates that the input terminal is simply a bit reversal of its original binary representation. Equation (2.b) shows no change in the respective output terminals. Thus, if there is a permutation

$$\begin{matrix} x_0 & x_1 \cdots x_{2^n-1} \\ \Pi(x_0) & \Pi(x_1) \cdots \Pi(x_{2^n-1}) \end{matrix} \qquad (3.a)$$

in the Omega network, the corresponding equivalent permutation in the Baseline network would be

$$\begin{matrix} \tilde{x}_0 & \tilde{x}_1 & \cdots \tilde{x}_{2^n-1} \\ \Pi(x_0) & \Pi(x_1) & \Pi(x_{2^n-1}) \end{matrix} \qquad (3.b)$$

where x_i and $\tilde{x}_i$ are given by

$$x_i = (b_\ell \; b_{\ell-1} \; .. \; b_1)$$

$$\tilde{x}_i = (b_1 \; b_2 \; .. \; b_{\ell-1} \; b_\ell)$$

Using the equivalence of (3.a) and (3.b), conflict-free permutations for Baseline network are developed. Similarly general relations of equivalence between other interconnection networks are obtained which provide the conflict free families for other networks.

Another problem dealt with in this paper is the recognition of conflict-free families for some particular network, so that such bijections can be performed in one pass. If we are given a one-to-one permutation and we want to see whether it can be passed by some particular network, we just check whether it belongs to the conflict-free families of that particular network or not. For this purpose, suitable hardware [4] is also presented in this paper.

References

[1] Lenfant, Jacques, "Parallel Permutations of Data: A Benes Network Control Algorithm for frequently used Permutations", IEEE Trans. Comput., Vol. 27, July 1978, pp. 637-647.

[2] Lawrie, D., "Access and Alignment of data in an array processor", IEEE Trans. Comput. Vol. C-24, Dec.1975, pp. 1145-1155.

[3] Wu, Chuan-lin and Feng, T., "Routing Techniques for a class of Multistage Interconnection Networks", Proceedings of the 1978 International Conference of Parallel Processing.

[4] Agrawal,D.P., "High Speed Arithmetic Arrays," IEEE Transaction on Computers,Vol.28,Mar.1979,pp215-224.

THE REVERSE-EXCHANGE INTERCONNECTION NETWORK

Chuan-lin Wu and Tse-yun Feng
Department of Electrical and Computer Engineering
Wayne State University
Detroit, Michigan 48202

Abstract -- Interconnection networks usually affect the overall cost and performance of parallel processing systems very significantly. Designers should work out various ways to enhance the efficiency of the interconnection networks. However there is limited work (or no work at all) concerning the following areas: how to control the blocking-type multistage interconnection networks to accomplish arbitrary permutation in multiple passes; taking advantage of existing Benes network control algorithms to realize arbitrary permutations on blocking-type multistage interconnection networks; exploiting the relationships between the admissible permutations and their control information; reconfiguring a network to accomplish various functions of different networks. The reverse-exchange multistage interconnection network is used to attack these unsolved problems. We first show a way to reconfigure a multistage interconnection network to accomplish various functions of multistage interconnection networks after comparing the functional relationships among these networks. The relationships between the admissible permutations and their control information of the reverse-exchange network are then exploited. Finally we show that arbitrary permutations can be realized in two passes (or $2(\log_2 N)$ switching steps, where N is the network size). Taking advantage of Benes network control algorithms, we also provide a way to control the two-pass structure.

I. Introduction

In many proposed or existing parallel processing architectures, an interconnection network is used to realize various permutations of data between processors or between processors and memory modules [1-11]. The interconnection network in such architectures significantly, and sometimes even dominantly, affects the overall cost and performance. However, there are still problems on the way of designing a cost-effective interconnection network. As a continuation of previous work [11], this paper first addresses some critical problems and then provides relevant contributions through the use of a newly configured multistage interconnection network.

The possibility of realizing arbitrary permutations on a multistage interconnection network is an important topic to consider. It is noted that all the multistage interconnection networks so far considered [1-6,11] cannot realize arbitrary permutations in a single pass. Pease [4] did point out the necessity of the multiple-pass realization on multistage interconnection networks. However, there is no work available on how to control the multistage interconnection networks to accomplish arbitrary permutations in multiple passes. Using single stage networks, some investigators have presented multiple-pass schemes for realizing arbitrary permutations. These include Lang's $O(\sqrt{N})$ shuffle-exchange steps [7], Orcutt's $O(\sqrt{N}\log_2 N)$ steps of Illac IV [9], Stone's $O((\log_2 N)^2)$ perfect-shuggle steps [10], and Siegel's $O((\log_2 N)^2)$ algorithm [12]. In reviewing these multiple-pass schemes we feel that $O(\sqrt{N})$ or $O((\log_2 N)^2)$ is still an impractical number when N is large. An interconnection network which can realize arbitrary permutations in much less passes with efficient control techniques available should be favorably considered.

Another important topic to consider is to design an efficient network control mechanism for both the single pass and the multiple passes. The permutation function and its control information of an interconnection network are closely related. However, it is most likely fair to say that there is only a limited understanding on their relationships. Lawrie [5] proposed a control technique named destination-tag scheme, which relaxes the requirement of understanding the relationship. This destination-tag scheme can route data from the input side to the output side according to the binary address of the destination. A homogeneous routing technique which removes the direction restriction of the destination-tag scheme has also been presented [11]. These routing techniques have to take care of the conflict resolution and to decide the route switching element by switching element. In the case that the needed permutation function is known, there is no ground to loose time in calculating the route switching element by switching element. Batcher [3] has done some work on specifying the control information of the flip network according to the permutation function. However, his work is restricted on the flip permutation functions. On the contrary, for the Benes binary network [13], there are well defined looping algorithms [14,15] which calculate the control information according to the permutation function. It is interesting to note that with these looping algorithms Benes binary network can realize arbitrary permutations while its hardward facilities are less than twice those of multistage interconnection networks [1-6,11]. So far there is not work which can take advantage of these looping algorithms on the way of realizing arbitrary permutations on a multistage interconnection network in multiple passes. However, Lenfant pointed out [8] that these looping algorithms are both time-consuming and space-consuming. In order to meet the time constraints arising from the use of a Benes binary network as the alignment network of a parallel computer, Lenfant [8] presented a Benes network control algorithm that can calculate the control information according to the name of five frequently used permutation families. Lenfant and Tahé [16] also derived an external control mechanism for the reverse omega network. Lenfant's approach does represent an alternative way in searching for an efficient control mechanism. It is worthy of exploiting useful permutation families of an interconnection network and then developing the control mechanism accordingly.

The possibility of increasing the combinatorial power of a network through reconfiguring that network has not been exploited. The topological equivalence of a class of multistage interconnection networks has been defined previously [11]. However, the topological equivalence is not equal to the functional equivalence. It would be a favored step to define the functional equivalence and explicitly express the functional relationships among the topologically equivalent interconnection networks. These functional relationships can facilitate obtaining the necessary reconfiguration information so that one can use just one topology and accomplish various functions through reconfiguring.

This paper is organized as follows. In Section II we review our mathematical model developed previously [11] and introduce terminologies of the reverse-exchange network which will be used to demonstrate our solutions to these problems described. Section III provides the functional relationships among topologically equivalent multistage interconnection networks. We also address a way of reconfiguring a network to increase the network combinatorial power. The capability and the recusive control algorithm of the reverse-exchange network are exploited in Sections IV and V, respectively. In Section VI we prove that the reverse-exchange network can realize arbitrary permutation in two passes. Both the construction and the routing scheme are provided. The network utilization and limitations are discussed in Section VII.

II. The Reverse-Exchange Network

A mathematical model was previously presented to specify various multistage interconnection networks [11]. For a self-contained purpose, we give a brief review here. A multistage interconnection network can be defined by specifying its configuration and its control structure. We shall discuss the control structure in the next section. By the configuration of an interconnection network we mean the topology and the logical names of the network components: interconnection links and switching elements. The topology is defined in terms of three parameters: the number of communication paths of each switching element; the number of switching element stages; the connectivities of the interconnection links between switching elements. A set of mathematical rules, called topology describing rules, is used to describe the connectivities of the interconnection links. Fig. 1 shows a configuration of a 16×16 baseline network. Note that the logical names of the interconnection links do not appear in Fig. 1 but they can be derived from the logical names of the switching elements. With such a configuration, we can uniquely identify every component and unambiguously assign the interconnection task by specifying the input logical name and the output logical name.

The baseline network can be used as a reference to evaluate the relationship among various multistage interconnection networks. A class of topologically equivalent multistage interconnection networks has been defined [11] by using the fact that any network topology in this class can be described by the same topology describing rules of the baseline network if the logical names are properly assigned to the network components. In the rest of this paper, to configure a network means to assign logical names to network components. Note that a network configured in our way may have different functional capabilities from those of the network with the same topology but configured in the other way. We like to exploit some terminologies of this class of multistage interconnection networks before we use it in solving the described problems. Since the networks in the class, configured in our way, are functionally identical although they have different topologies, we could get the same results no matter which topology is used in demonstration. In this paper, we use the topology of the baseline network.

The permutation function of a network is accomplished by two components - the interconnection links and the switching elements. Assume the binary representation of integer X is $x_\ell x_{\ell-1}\cdots x_0$, where $\ell=n-1$ and $n=\log N$. The link connectivity of level i of the network performs the following permutations:

$$R_i(x_\ell x_{\ell-1}\cdots x_0) = x_\ell\cdots x_{\ell-i+2}x_0x_{\ell-i+1}\cdots x_1, \quad (1)$$

for $0 < i \le \ell$. For $i = 0$ and $\ell + 1$,

$$R_i(x_\ell x_{\ell-1}\cdots x_0) = x_\ell x_{\ell-1}\cdots x_0. \quad (2)$$

By Eqs. (1) and (2) we have

$$R_{\ell+1}\,(R_\ell(R_{\ell-1}\cdots(R_0(x_\ell x_{\ell-1}\cdots x_0))\cdots)) = x_0x_1\cdots x_\ell, \quad (3)$$

Eq. (3) implies that the overall interconnection links of the network perform the bit-reverse permutation. The exchange is performed by a switching element on two inputs named by adjacent numbers. The exchange permutation is defined as

$$E(x_\ell x_{\ell-1}\cdots x_0) = \begin{cases} x_\ell x_{\ell-1}\cdots x_0 & \text{if } c = 0 \\ x_\ell x_{\ell-1}\cdots \bar{x}_0 & \text{if } c = 1 \end{cases} \quad (4)$$

where c is the control bit of the switching element. Since there are N/2 switching elements in a stage, there is a control vector associated with each stage. The notation of $C_i(j)$ and E_i are used to denote, respectively, the control bit for jth switching element in stage i and the exchange permutation of stage i associated with control vector C_i. Assume X is permuted to P(X) by the network. Then

$$P(x_\ell x_{\ell-1}\cdots x_0) = R_{\ell+1}(E_\ell(R_\ell(E_{\ell-1}\cdots (R_1(E_0(R_0(x_\ell x_{\ell-1}\cdots x_0))))\cdots))) = e_0(x_0)\cdot e_1(x_1)\cdots e_{\ell-1}(x_{\ell-1})\cdot e_\ell(x_\ell), \quad (5)$$

where $e_i(x_i)$, $0 \le i \le \ell$, is equal to x_i or $\overline{x}_i$ depending on the exchange performed by the associated switching element in stage i. Fig. 2 shows an example of a permutation realized on the network of size $N = 2^3$ with control vectors as specified. Because of the bit-reverse and the exchange attributes we use the reverse-exchange network to name this uniquely configured class of multistage interconnection networks.

III. Functional Relationships and Reconfiguration

In this section, we compare functional relationships among topologically equivalent multistage interconnection networks [11] and exploit the possibility to reconfigure a network to accomplish functions of various networks. First of all, we would like to make it clear that the topological equivalence between two networks implies the one-to-one and onto relationship between network components of the two networks, and does not necessarily imply the identical functionality between the two networks. For example, Fig. 3(a) shows an omega network reconfigured by using the binary tree coding method [11]. This network is actually a reverse-exchange network and functionally different from the original omega network proposed by Lawrie [5] as shown in Fig. 3(b). Previously, we use the topology of the baseline network [11] to obtain the topological relationship among various networks. Here, again, we shall use the functions of the baseline network to obtain functional relationships among various networks. But before evaluating the functional relationships, we should define some terminologies and mathematical tools.

The functions of a network are also confined by its control structure. Siegal [17] proposed that the control structure of a network sets the states of switching element. Thus the control structure of a network sends the control information to switching elements to realize the interconnection function. We therefore propose here that two multistage interconnection networks are functionally equivalent (or identical) if they have a same set of admissible interconnection functions and they can realize any admissible interconnection function using the same control information (without duplication, but a proper mapping on its location is allowed). Hence, in order to compare the functional capability of various networks, we have to assume those networks have the same control structure which includes the states of each switching element and the way to set them. In this section, the functional relationships among various networks are assessed under this assumption. Under the assumption, let the set of admissible network functions of the baseline network, the reverse baseline network, the omega network, the reverse omega network, the indirect binary n-cube network, the reverse indirect binary n-cube network, the modified data manipulator, the reverse modified data manipulator, the flip network, the reverse flip network, the regular SW banyan network (S=F=2) and the reverse regular SW banyan network (S=F=2) be B_N, B_N^{-1}, Ω_N, Ω_N^{-1}, C_N, C_N^{-1}, D_N, D_N^{-1}, F_N, F_N^{-1}, G_N, G_N^{-1}, respectively. Using the previous notation of the binary representation, we define the bit reversal permutation ρ by

$$\rho(x_\ell x_{\ell-1}\cdots x_0) = x_0 x_1 \cdots x_\ell, \quad (6)$$

and the bit switch permutation δ by

$$\delta(x_\ell x_{\ell-1} x_{\ell-2}\cdots x_3 x_2 x_1 x_0) = x_1 x_{\ell-1} x_{\ell-2}\cdots x_3 x_2 x_\ell x_0, \quad (7)$$

for $\ell \ge 1$.

Using the above definitions and the binary tree coding method we can now evaluate the functional relationships. A simple example will be demonstrated on the networks shown in Fig. 3. The reverse-exchange network shown in Fig. 3(a) is configured by using the binary tree coding method and has B_N as its set of admissible network functions where N=8. The omega network shown in Fig. 3(b) of course has Ω_N as its set of admissible network functions where N=8. Now if we take a bit-reverse permutations, ρ, on the logical names of the input links of the omega network shown in Fig. 3(b), we should have the same logical names for the input links as those shown in Fig. 3(a). Therefore from this example, we can see, for N=8,

$$\Omega_N = B_N \circ \rho, \quad (8)$$

where $B_N \circ \rho$ implies $B_N(\rho)$. Eq. (8) means that the omegan network function can be decomposed into two ordered functions: First, a bit-reverse permutation and then, a baseline network function. The argument can be extended for any N. The functional relationships between the baseline network and other topological equivalent networks can be similarly developed. The result is listed in the $\theta = B_N$ column of Table 1. Note that the bit switch permutation, δ, is used in the D_N, D_N^{-1}, G_N and G_N^{-1} rows. It is interesting to see that the baseline network and the reverse baseline network are functionally identical as indicated in Table 1 by

$$B_N^{-1} = B_N. \quad (9)$$

The above results in which we use the B_N as the reference can be manipulated so that new formulas can be formed in terms of other network functions. For example, we can see

$$B_N = B_N \circ \rho \circ \rho, \quad (10)$$

since $\rho \circ \rho$ is equal to the identity. Then according to Eqs. (8) and (10) we have

$$B_N = \Omega_N \circ \rho \qquad (11)$$

This result is shown in the first entry in the $\theta = \Omega_N$ column of Table 1. The other entries of Table 1 are obtained in the same way. From the $\theta = \Omega_N$ column of Table 1 we can see $\Omega_N = C_N^{-1} = F_N^{-1}$ which means that the omega-network, the reverse indirect binary n-cube network, and the reverse flip network are functionally identical. Similarly, from other columns, we can also see that $C_N = \Omega_N^{-1} = F_N$, $D_N = G_N^{-1}$, and $G_N = D_N^{-1}$.

The functional relationships shown in Table 1 lay the ground for the reconfiguration problem. The reconfiguration problem can be defined as the problem to reassign the logical names of the input links and/or the output links of an interconnection network in a parallel processing system so that the interconnection network can at least realize various functions of the networks listed in Table 1. For a simple illustration, assume the baseline network of size N is installed in a parallel processing system. In order to reconfigure the baseline network to accomplish various functions listed in Table 1, two permutation functions, ρ and σ, are needed to reassign the logical names of terminal links. Assume we want to realize Ω_N on the topology of the baseline network. Then by Eq. (8) the logical name of the input link should be $\rho(x_\ell x_{\ell-1} \ldots x_0)$ where $x_\ell x_{\ell-1} \ldots x_0$ is the original logical name of the input link in the baseline network, and the logical name of the output link should remain the same. On the other hand, if we want to realize G_N, then by the following formula

$$G_N = \delta \circ B_N, \qquad (12)$$

the logical name of the input link should remain the same, and the logical name of the output link should be $\delta(x_\ell x_{\ell-1} \ldots x_0)$. In concluding the way to reconfigure a reference network from Eqs. (8) and (12), we see that in Table 1, the permutation appearing on the right hand side of the reference network function should be used to assign the logical name of the input link and the permutation appearing on the left hand side should be used to assign the logical name of the output link.

A reconfiguration function in the control unit of a parallel processing system is highly recommended. The reconfiguration function should choose a proper network configuration according to the algorithm to be processed. It seems necessary to have a table which establishes the mapping between the algorithm and the network configuration and the mapping on the control information in various configurations. Furthermore, some hardware facilities may be needed for this reconfiguration function. Some consequent questions may arise from the use of the reconfiguration: How many configurations on a network topology are necessary to accomplish arbitrary permutations? What is the relationship between the configuration and the frequently used permutations [8]? Does the reconfiguration have any impact on the multi-dimensioning-access scheme [5,19]? We will discuss these questions in Section VII on the network utilization and limitations.

IV. Admissible Permutations

The admissible permutations, i.e. permutations which can be realized by the reverse-exchange network in one pass, are a subset of $2^n!$ distinct permutations. In this section we will identify these admissible permutations.

Assume that $(A_j, Z_j) = (a_{j,\ell} a_{j,\ell-1} \ldots a_{j,0}, z_{j,\ell} z_{j,\ell-1} \ldots z_{j,0})$ represents the source-destination pair of the permutation request $P(A_j) = Z_j$. Define $A_{p,q} = a_{p,\ell} a_{p,\ell-1} \ldots a_{p,\ell-q+1}$ and $Z_{p,q} = z_{p,\ell} z_{p,\ell-1} \ldots z_{p,\ell-q+1}$.

Theorem 1:

Given a set of distinct permutation requests, $P_N = \{(A_i, Z_i) \mid 0 \leq i < N\}$, P_N can be realized by the reverse-exchange network if and only if $A_j \neq A_k$ and $A_{j,\ell-m} = A_{k,\ell-m}$ implies $Z_{j,m+1} \neq Z_{k,m+1}$ for $j \neq k$, $0 \leq j,k < N$ and $0 \leq m \leq \ell$.

Proof: Recall from previous work [11] that for the permutation request $P(A_j) = Z_j$, stage m will switch source $a_{j,\ell} a_{j,\ell-1} \ldots a_{j,0}$ to link $z_{j,\ell} z_{j,\ell-1} \ldots z_{j,\ell-m+1} a_{j,\ell} a_{j,\ell-1} \ldots a_{j,m+1} z_{j,\ell-m}$ in the level m+1. A conflict occurs if some other source is also switched to this link. That is, for some pairs of permutation requests, say (A_j, Z_j) and (A_k, Z_k), and for some m, we have $a_{j,\ell} a_{j,\ell-1} \ldots a_{j,m+1} = a_{k,\ell} a_{k,\ell-1} \ldots a_{k,m+1}$ and $z_{j,\ell} z_{j,\ell-1} \ldots z_{j,\ell-m} = z_{k,\ell} z_{k,\ell-1} \ldots z_{k,\ell-m}$. The conflict condition can be represented as $A_{j,\ell-m} = A_{k,\ell-m}$ and $Z_{j,m+1} = Z_{k,m+1}$ for permutation requests $P(A_j) = Z_j$ and $P(A_k) = Z_k$. Therefore if $A_j \neq A_k$ and $A_{j,\ell-m} = A_{k,\ell-m}$ implies $Z_{j,m+1} \neq Z_{k,m+1}$ for $j \neq k$, $0 \leq j,k < N$ and $0 \leq m \leq \ell$, both (A_j, Z_j) and (A_k, Z_k) are realizable permutations. On the other hand, since there is only one interconnection path existing for a permutation, the reverse way of the above statement is also true. Q. E. D.

Note that if we take Eq. (11) into account, Theorem 1 can also be inferred from the similar arguments of Lawrie for Ω_N [5]. Using Theorem 1 we will identify some of the permutations which can be realized by the reverse-exchange network in one pass.

Theorem 2:

Define X_i^r to be the number whose binary representation is the reverse binary representation of X_i and define $P_N = \{(X_i, X_i^r) | 0 \leq i < N)\}$ to be the bit-reverse permutation. Then P_N is realizable by the reverse-exchange network.

Proof: Assume $X_{j,\ell-i} = X_{k,\ell-i}$ for $0 \leq i \leq \ell-1$ where $j \neq k$. Since $X_j \neq X_k$ for $j \neq k$ we then obtain $X^r_{j,i+1} \neq X^r_{k,i+1}$ from the assumption. The proof immediately follows Theorem 1. Q.E.D.

Theorem 3:

If $P_N = \{(A_i, Z_i) | 0 \leq i < N\}$ is realizable by the reverse-exchange network and a is an odd integer, then $P_N' = \{(A_i, aZ_i) | 0 \leq i < N\}$ is also realizable by the network.

Proof: Define $Y_i = aZ_i$. We will prove that $A_{j,\ell-m} = A_{k,\ell-m}$ for $j \neq k$ and $0 \leq m < \ell$ implies $Y_{j,m+1} \neq Y_{k,m+1}$.

Since $A_{j,\ell-m} = A_{k,\ell-m}$ for $j \neq k$ and $0 \leq m \leq \ell$ implies $Z_{j,m+1} \neq Z_{k,m+1}$, hence, $a_{j,\ell}a_{j,\ell-1}\ldots a_{j,m+1} = a_{k,\ell}a_{k,\ell-1}\ldots a_{k,m+1}$ implies $z_{j,m}z_{j,m-1}\ldots z_{j,0} \neq z_{k,m}z_{k,m-1}\ldots z_{k,0}$. Assume further that $z_{j,t} \neq z_{k,t}$ for some t, $0 \leq t \leq m$ and $z_{j,s} = z_{k,s}$ for $t < s \leq m$. Let $b_\ell b_{\ell-1}\ldots b_0$ be the binary representation of a where $b_0=1$ since a is odd. The products of $Y_j=a_jz_j$ and $Y_k= a_kz_k$ will result in $Y_{j,t} \neq Y_{k,t}$. This result concludes that $A_{j,\ell-m} = A_{k,\ell-m}$ for $j \neq k$ and $0 \leq m < \ell$ implies $Y_{j,m+1} \neq Y_{k,m+1}$. Q.E.D.

Theorem 4:

If $P_N = \{(A_i, Z_i) | 0 \leq i < N\}$ is realizable by the reverse-exchange network and b is an integer, then $P_N = \{(A_i, Z_i+b) | 0 \leq i < N\}$ is also realizable by the network.

Proof: Let $Y_i = Z_i+b$. It is obvious that $Z_{j,m+1} \neq Z_{k,m+1}$ for $0 \leq m < \ell$ implies that $Y_{j,m+1} \neq Y_{k,m+1}$. Hence $A_{j,\ell-m} = A_{k,\ell-m}$ for $j \neq k$ and $0 \leq m < \ell$ implies $Z_{j,m+1} \neq Z_{k,m+1}$ and also $Y_{j,m+1} \neq Y_{k,m+1}$. Q.E.D.

Corollary 1: The permutation defined by $P_N= \{(X_i, aX_i^r+b) | 0 \leq i < N$, a is an odd integer and b is an integer$\}$, is realizable by the reverse-exchange network.

Corollary 2: The permutation defined by $P_N= \{(X_i, T-X_i^r) | 0 \leq i < N$ and T is an integer$\}$ is realizable by the reverse-exchange network.

Corollaries 1 and 2 are consequences of Theorems 2, 3 and 4. Note that the omega network has the same properties shown by Theorems 3 and 4 [5].

Theorem 5:

If $P_N = \{(A_i, Z_i) | 0 \leq i < N\}$ is realizable by the reverse-exchange network and k is an integer, then $P_N' = \{(A_i, Z_i \oplus k | 0 \leq i < N$ and $\oplus$ is the bit-by-bit EXCLUSIVE OR$\}$ is also realizable by the network.

Proof: Define $Y_i = Z_i \oplus k$. It can be seen that $Y_{j,m} \neq Y_{k,m}$ if $Z_{j,m} \neq Z_{k,m}$ for $0 \leq m \leq \ell$. Since P_N is realizable by the reverse-exchange network, $A_{j,\ell-m} = A_{k,\ell-m}$ for $j \neq k$ and $0 \leq m < \ell$ also implies $Y_{j,m+1} \neq Y_{k,m+1}$. Q.E.D.

Corollary 3: The permutation defined by $P_N= \{(X_i, X_i^r \oplus k) | 0 \leq i < N$ and k is an integer$\}$ is realizable by the reverse-exchange network.

Corollary 3 is a consequence of Theorems 2 and 5.

Theorem 6:

Define the following binary representations:

$$X_i = (x_{i,\ell}x_{i,\ell-1}\ldots x_{i,j}x_{i,j-1}\ldots x_{i,0}),$$

$$U = (x_{i,j-1}\ldots x_{i,0}),$$

$$Y_i = (Y_{i,0}\ldots Y_{i,j-1}x_{i,j}\ldots x_{i,\ell-1}x_{i,\ell}),$$

and $V = (Y_{i,j-1}\ldots Y_{i,0})$.

Assume $V = U + k \bmod 2^j$, where k is an integer. Then the permutation defined by $P_N = \{(X_i, Y_i) | 0 \leq i < N\}$ is realizable by the reverse-exchange network.

Proof: By the definition, if $(x_{p,m}x_{p,m-1}\ldots x_{p,0}) \neq (x_{q,m}x_{q,m-1}\ldots x_{q,0})$, then $(y_{p,m}y_{p,m-1}\ldots y_{p,0}) \neq (y_{q,m}y_{q,m-1}\ldots y_{q,0})$ for either $m \leq j$ or $m > j$. Hence $X_{p,\ell-m} = X_{q,\ell-m}$ for $p \neq q$ and $0 \leq m < \ell$ implies $Y_{p,m+1} \neq Y_{q,m+1}$. Q.E.D.

Theorem 7:

If $P_N = \{(A_i, Z_i) | 0 \leq i < N\}$ is realizable by the reverse-exchange network, then $P_N' = \{(Z_i, A_i) | 0 \leq i < N\}$ is also realizable by the network.

Proof: Since P_N is realizable by the reverse-exchange network, $a_{j,\ell}\ldots a_{j,m+1} = a_{k,\ell}\ldots a_{k,m+1}$ for $j \neq k$ and $0 \leq m < \ell$ implies $z_{j,\ell}\ldots z_{j,\ell-m} \neq z_{k,\ell}\ldots z_{k,\ell-m}$. By contradiction, assume that in the case of $z_{j,\ell}\cdots z_{j,q+1} = z_{k,\ell}\cdots z_{k,q+1}$ for $j \neq k$ and $0 \leq q < \ell$ we can have $a_{j,\ell}\cdots a_{j,\ell-q} = a_{k,\ell}\cdots a_{k,\ell-q}$. Then there exists $m = \ell-q-1$ such that $a_{j,\ell}\cdots a_{j,m+1} = a_{k,\ell}\cdots a_{k,m+1}$ and $z_{j,\ell}\cdots z_{j,\ell-m} = z_{k,\ell}\cdots z_{k,\ell-m}$. This contradicts the statement shown in the beginning of this proof. Hence $z_{j,\ell}\cdots z_{j,q+1} = z_{k,\ell}\cdots z_{k,q+1}$ for $j \neq k$ and $0 \leq q < \ell$ implies $a_{j,\ell}\cdots a_{j,\ell-q} \neq a_{k,\ell}\cdots a_{k,\ell-q}$. Q.E.D.

Corollary 4: The permutation defined by $P_N = \{(aX_i + b, X_i^r) \mid 0 \leq i < N$, a is an odd integer and b is an integer$\}$ is realizable by the reverse-exchange network.

Proof: According to Theorems 2 and 7, we can see that $P_N' = \{(X_i^r, X_i) \mid 0 \leq i < N\}$ is realizable by the network. Then by Theorems 3 and 4 it is obvious that $P_N'' = \{(X_i^r, aX_i + b) \mid 0 \leq i < N$, a is an odd integer and b is an integer$\}$ is realizable. From Theorem 7 again, we see that $P_N = \{(aX_i + b, X_i^r) \mid 0 \leq i < N$, a is an odd integer and b is an integer$\}$ is realizable. Q.E.D.

We have presented some theorems which identify the one-pass realizable permutations of the reverse-exchange network. In the next section we will use these theorems to classify some specially interesting realizable permutations and exploit the relationships between the permutation and its related control information.

V. Recursive Routing Mechanism for Admissible Permutations

The homogeneous routing procedure described previously in [11] already provides a simple routing mechanism. It employs the n-bit destination tag to determine the valid state of the in-path switching elements. However, because of the prescribed reasons, it is preferable to determine the control pattern by the name of the admissible permutation. We will first identify some classes of usefull admissible permutations and then derive recursive algorithms which determine control patterns according to the permutation names.

It is easy to verify, using Theorems 1-7, that the following categories of important permutations are admissible on the reverse-exchange network.

1. $F_k^{(n)}$ $(0 \leq k < 2^n) = P(X^r \oplus k) = X$ and $P(X \oplus k) = X^r$.

2. $C_{j,k}^{(n)}$ $(0 \leq j,k < 2^n$, j odd$) = P(jX^r + k) = X$ and $P(jX + k) = X^r$.

3. $S_{q,k}^{(n)}$ $(0 < q \leq n, 0 < k < 2^q) =$ $P(X) = (X + k - \varepsilon \cdot 2^q)^r$ and $P(X + 2^q - k - \varepsilon \cdot 2^q) = X^r$.

4. $\tilde{F}_k$ $(0 \leq k < 2^n) = P(X) = X^r \oplus k$ and $P(X^r) = X \oplus k$.

5. $\tilde{C}_{j,k}$ $(0 \leq j,k < 2^n$, j odd$) =$ $P(X) = jX^r + k$ and $P(X^r) = jX + k$.

6. $\tilde{S}_{q,k}$ $(0 < q \leq n, 0 < k < 2^q) =$ $P((X + k - \varepsilon \cdot 2^q)^r) = X$ and $P(X^r) = X + 2^q - k - \varepsilon \cdot 2^q$.

Note that Category 4 (5 and 6) is a consequence of applying Theorem 7 to Category 1 (2 and 3). Also $S_{q,k}^{(n)}$ of Category 3, a direct result of Theorem 6, is a cyclic shift of amplitude k within each segament of size 2^q. The ε in $S_{q,k}^{(n)}$ can be defined as follows. Let $X = x_\ell x_{\ell-1}\cdots x_q x_{q-1}\cdots x_0$ and $k = 0\ldots0k_{q-1}\cdots k_0$. Then ε is equal to the carry out bit of summing $x_{q-1}x_{q-2}\cdots x_0$ and $k_{q-1}k_{q-2}\cdots k_0$.

Denote by $K^{(n)}(P) = (v_{ij})$, $0 \leq i \leq 2^{n-1}$ and $0 \leq j \leq n-1$, the control patterns of $2^{n-1} \times n$ matrix associated with a permutation P. Denote also by $v^{(n-j)}(b)$ the 2^{n-j} bit vector whose components are all equal to b. A binary tree whose root is a vector v and whose upper subtree and lower subtree are K_u and K_v, respectively, is denoted by $[v; K_u, K_v]$. The cascaded matrix whose left part and right part are L and R, respectively, is denoted by $[L;R]$.

Let k be a positive integer and denote, respectively, by k' and k_0, its quotient and its remainder in the division by 2, i.e., $k = 2k' + k_0$.

Theorem 8:

$$K^{(n)}(F_k^{(n)}) = [v^{(n-1)}(k_0); K^{(n-1)}(F_{k'}^{(n)})]$$
$$\text{where } K^{(1)}(F_{k'}^{(n)}) = [v^{(n-1)}(k')].$$

Proof: Let $X = x_\ell x_{\ell-1}\cdots x_0$ and $k = k_\ell k_{\ell-1}\cdots k_0$ where $\ell = n-1$. Then $X^r \oplus k = (x_0 \oplus k_\ell)(x_1 \oplus k_{\ell-1})\cdots(x_\ell \oplus k_0)$. According to the properites of the reverse-exchange network described by Eq. (5), we have $x_i = e_{\ell-i}(x_i \oplus k_{\ell-i})$, $0 \leq i \leq \ell$, for $P(X^r \oplus k) = X$. Therefore we can see that, for $0 \leq i \leq \ell$,

$$x_i = \begin{cases} e_{\ell-i}\,(x_i) \text{ if } k_{\ell-i} = 0, \\ e_{\ell-i}\,(\bar{x}_i) \text{ if } k_{\ell-i} = 1. \end{cases} \qquad (13)$$

Eq. (13), in turn, implies that the control bit of switching elements in stage $\ell-i$ should equal to $k_{\ell-i}$. Thus we obtain

$$K^n(F_k^{(n)}) = [V^{(n-1)}(k_0); V^{(n-1)}(k_1); \ldots; V^{(n-1)}(k_\ell)]. \qquad (14)$$

Eq. (14) is exactly what the theorem implies. The same arguments can be made for $P(X \oplus k) = X^r$. Q.E.D.

Example: Assume

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 5 & 3 & 7 & 0 & 4 & 2 & 6 \end{pmatrix}.$$

P can be described by

$$F_4^{(3)}: P(X^r \oplus 4) = X.$$

According to Theorem 8, we have

$$K^{(3)}(F_4^{(3)}) = [v^{(2)}(0);v^{(2)}(0);v^{(2)}(1)].$$

Hence

$$K^{(3)}(P) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

The setting of the network is depicted in Fig. 4.

<u>Theorem 9:</u>

Let $j = 2j' + 1$.
$K^{(n)}(C_{j,k}^{(n)}) = [v^{(n-1)}(k_0);K^{(n-1)}(C_{j,k'}^{(n-1)}),\ K^{(n-1)}(C_{j,j'+k'+k_0}^{(n-1)})]$, where $K^{(1)}(C_{j,k}^{(1)}) = [k]$.

<u>Proof:</u> Consider $P(jX^r + k) = X$. We first define some notations:

$X = x_\ell x_{\ell-1}\cdots x_0$,
$X' = x_\ell x_{\ell-1}\cdots x_1$,
$j = j_\ell j_{\ell-1}\cdots j_0$,
$j' = j_\ell j_{\ell-1}\cdots j_1$,
$k = k_\ell k_{\ell-1}\cdots k_0$,
$k' = k_\ell k_{\ell-1}\cdots k_1$,
$Y = jX^r + k = y_\ell y_{\ell-1}\cdots y_0$, and
$Y' = y_\ell y_{\ell-1}\cdots y_1$.

Note that $X = 2X' + x_0$, $Y = 2Y' + y_0$, $j = 2j' + 1$ and $k = 2k' + k_0$. Furthermore, $y_0 = x_\ell \oplus k_0$ and $Y'=j(X')^r + j'x_\ell + k' + x_\ell\cdot k_0$.

According to Eq. (5), we have

$$e_0(y_0)e_1(y_1)\cdots e_\ell(y_\ell) = x_\ell x_{\ell-1}\cdots x_0. \tag{15}$$

Hence by Eq. (15) the control bit of switching elements in stage 0 is determined by $e_0(y_0) = x_\ell$. Since $y_0 = x_\ell \oplus k_0$, the control vector of stage 0 is then equal to $V^{(n-1)}(k_0)$. The next n-1 control vectors are dependent on $Y'=y_\ell y_{\ell-1}\cdots y_1$. These control vectors can then be partitioned into two halves: upper half where $x_\ell=0$, and lower half where $x_\ell=1$. For the upper half, $Y'=j(X')^r + k'$ since $x_\ell=0$. The permutation for this upper half then becomes $P(j(X')^r + k') = X'$. For the lower half, $Y' = j(X')^r + j' + k' + k_0$ since $x_\ell=1$. The permutation for this lower half then becomes $P(j(X')^r + j' + k' + k_0) = X'$. The above discussion can recursively be applied to these two permutations of the upper half and the lower half as shown in the theorem. Similar arguments can be made for $P(jX + k) = X^r$. Q.E.D.

<u>Example:</u> Assume

P =

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 10 & 4 & 15 & 3 & 9 & 6 & 12 & 0 & 11 & 5 & 14 & 2 & 8 & 7 & 13 & 1 \end{pmatrix}.$$

P can be described by $C_{5,7}^{(4)}$: $P(5X^r + 7) = X$.
According to Theorem 9, we have

$$K^{(4)}(C_{5,7}^{(4)}) = \left[\begin{bmatrix}1\\1\\1\\1\\1\\1\\1\\1\end{bmatrix} \begin{matrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\\ \begin{bmatrix}0\\0\\0\\0\end{bmatrix}\end{matrix} \begin{matrix}\begin{bmatrix}1\\1\end{bmatrix}\\ \begin{bmatrix}C\\0\end{bmatrix}\\ \begin{bmatrix}1\\1\end{bmatrix}\\ \begin{bmatrix}1\\1\end{bmatrix}\end{matrix} \begin{matrix}[0]\\ [1]\\ [0]\\ [0]\\ [1]\\ [0]\\ [0]\\ [1]\end{matrix}\right]$$

The setting of the network is depicted in Fig. 5.

<u>Theorem 10:</u>

For $q \le n$ and $0 < k < 2^q$,
$K^{(n)}(S_{q,k}^{(n)}) = [V^{(n-1)}(k_0);\ K^{(n-1)}(S_{q,k'+k_0}^{(n-1)}),\ K^{(n-1)}(S_{q,k'}^{(n-1)})]$, and $K^{(n-q)}(S_{q,k}^{(n-1)}) = K^{(n-q)}(C_{1,0}^{(n-q)})$. If q=n then $K^{(1)}(S_{q,k}^{(1)})=[k]$.

<u>Proof:</u> Consider $P(X) = (X + k -\varepsilon\cdot 2^q)^r$ here. According to the value of ε defined previously, we can see that the permutation functions like $P(X) = (X + k)^r$ for stage i where $i < q$ and $P(X)= X^r$ for stage j where $j \ge q$. Let ξ_i be the carry out of the sum of x_m and k_m for $0 \le m < i$ where $\xi_0=0$. For stage i, $i < q$, we have

$$e_i(x_i) = x_i \oplus k_i \oplus \xi_i. \tag{16}$$

From Eq. (16) we have $e_0(x_0) = x_0 \oplus k_0$ which implies $V^{(n-1)}(k_0)$ for stage 0. Similarly to the proof of Theorem 9, we partition the next (n-1) control vectors into two halves: upper half where $x_0 \oplus k_0 = 0$ and lower half where $x_0 \oplus k_0=1$. In the upper half, if $k_0=1$, then $x_0=1$ and if $k_0=0$ then $x_0=0$. Therefore $\xi_1=k_0$ for the upper half. Thus the control vectors of the upper half can be described by $K^{(n-1)}(S_{q,k'+k_0}^{(n-1)})$. Since $\xi=0$ for the lower half, its control vector can be described by $K^{(n-1)}(S_{q,k'}^{(n-1)})$. Concluding above arguments for stage i where $i < q$. We have $K^{(n)}(S_{q,k}^{(n)}) = [V^{(n-1)}(k_0);\ K^{(n-1)}(S_{q,k'+k_0}^{(n-1)}),\ K^{(n-1)}(S_{q,k'}^{(n-1)})]$. On the other hand, for stage j where $j \ge q$, we can obtain the control vectors from Theorem 9 since $P(X) = X^r$ is a case considered in Theorem 9 where j=1 and k=0. Thus $K^{(n-q)}(S_{q,k}^{(n-q)}) = K^{(n-q)}(C_{1,0}^{(n-q)})$. Similar arguments can be made for $P(X + 2^q - k - \varepsilon\cdot 2^q) = X^r$.
Q.E.D.

<u>Example:</u> Assume P =

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 12 & 0 & 8 & 4 & 14 & 2 & 10 & 6 & 13 & 1 & 9 & 5 & 15 & 3 & 11 & 7 \end{pmatrix}.$$

P can be described by $S_{2,3}^{(4)}$ which is an cyclic shift of amplitude 3 within each segament of size 2^2 as shown in Fig. 6. The application of the algorithm shown in Theorem 10 results in

$$K^{(4)}(S^{(4)}_{2,3}) = \left[\begin{matrix} \begin{bmatrix}1\\1\\1\\1\\1\\1\\1\\1\end{bmatrix} & \begin{matrix}\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\\ \begin{bmatrix}1\\1\\1\\1\end{bmatrix}\end{matrix} & \begin{matrix}\begin{bmatrix}0\\0\end{bmatrix}\\ \begin{bmatrix}0\\0\end{bmatrix}\\ \begin{bmatrix}0\\0\end{bmatrix}\\ \begin{bmatrix}0\\0\end{bmatrix}\end{matrix} & \begin{matrix}[0]\\ [0]\\ [0]\\ [0]\\ [0]\\ [0]\\ [0]\\ [0]\end{matrix} \end{matrix}\right]$$

The setting of the network is depicted in Fig. 7.

Corollary 5: The control bit pattern of Category 4 (5 and 6) is the same as that Category 1 (2 and 3) only reversed and with properly permuting the bit location within the stage.

Proof: Since the baseline network and the reverse baseline network are topologically and functionally equivalent, the reverse and the permutation of the control bit location are obvious from one to another. The rule for the location permutation of the control bit is defined in [11] and is recited here:

$$r[(P_\ell P_{\ell-1}\cdots P_1)_i] = (P_i\cdots P_1 P_\ell\cdots P_{i+1})_i \text{ for } 0 \le i \le \ell. \tag{17}$$

In Eq. (17), $(P_\ell P_{\ell-1}\cdots P_1)_i$ is the position of a control bit in stage i after the stage order is reversed, and $(P_i\cdots P_1 P_\ell\cdots P_{i+1})_i$ is the correct bit position after permuting. Q.E.D.

VI. Realization of Arbitrary Permutations

In this section we will first show that all permutations can be realized by the reverse-exchange network in two passes. Next, we consider the routing scheme for this two-pass structure.

A. Two-Pass Permutation

The fact that the Benes binary network can realize all permutations between its inputs and outputs follows the result of Slepian-Dupid theorem [13]. Using the above fact, we will prove that the reverse-exchange network can realize all permutations in two passes.

Theorem 11:

The reverse-exchange network can realize all permutations in two passes.

Proof: The theorem will be proven by showing that the functions of the Benes binary network can be simulated by the baseline network in two passes. An example structure for a two-pass implementation using a baseline network is shown in Fig. 8. The input data are fed in on Side 1. The output data of the first pass are stored in the shift register files on Side 2. In the second pass, the data in the register files are fed back to the input lines on Side 1 and the final results are again stored in the register files.

The two-pass structure is equivalent to the implementation of cascading two baseline networks. According to the previous result we can obtain a reverse baseline network via properly permuting the switching elements and its related links of the baseline network. For switching elements in the reverse baseline network, the mapping, γ_i, from physical names, $(p_\ell p_{\ell-1}\cdots p_1)_i$ to logical names $(b_\ell b_{\ell-1}\cdots b_1)_i$ is shown in Eq. (17). The mapping, γ_i^{-1}, from logical names to physical names is shown in the following:

$$\gamma_i^{-1}[(b_\ell b_{\ell-1}\cdots b_1)_i] = (b_{\ell-i}\cdots b_1\, b_\ell\cdots b_{\ell-i+1})_i, \tag{18}$$

for $0 \le i \le \ell$. If we rearrange the switching elements and its related links of the baseline network for the second pass in the equivalent structure in ascending order of the physical names, which are obtained by applying γ_i^{-1} of Eq. (18) on the logical names, we can obtain a structure which is formed by a baseline network and a reverse baseline network connected front to end. An example is shown in Fig. 9. The labellings shown in Fig. 9 are the logical names. Now, set the switching elements in the first stage of the reverse baseline network in the equivalent structure on the state of the direct connection. The setting is also shown in Fig. 9 for the example. It can be seen that the structure shown in Fig. 9 is equivalent to a Benes binary network. Q.E.D.

B. Routing Scheme

There exist several algorithms which compute the control information for the Benes binary network. A simple algorithm is previously shown [18]. However, three systematic algorithms have been proposed by Opferman and Tsoo-Wu [14], Anderson [15], and Lenfant [8]. We shall use these three algorithms to derive the control information for our two-pass structure.

The control pattern computed by these three algorithms should properly be permuted before it can be applied to the reverse-exchange network. As shown in Theorem 11, the leftmost n stages of the Benes binary network are one-to-one correspondent to the reverse-exchange network of the first pass from left to right, and the rightmost n-1 stage of the Benes binary network are one-to-one correspondent to the reverse-exchange network of the second pass from right to left. The switching elements in the leftmost stage of the network in the second pass are refined to be in the valid state of the direct connection. Assume that signals 0 and 1 represent the valid states for the direct and the crossed connection. We can represent the control pattern of the Benes network and the reverse-exchange network by the following matrices:

1. Benes binary network

$$B = \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,2n-2} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,2n-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ b_{\frac{N}{2},0} & b_{\frac{N}{2},1} & \cdots & b_{\frac{N}{2},2n-2} \end{bmatrix},$$

2. Reverse-exchange network of the first pass

$$R_1 = \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ b_{\frac{N}{2},0} & b_{\frac{N}{2},1} & \cdots & b_{\frac{N}{2},n-1} \end{bmatrix},$$

3. Reverse-exchange network of the second pass

$$R_2 = \begin{bmatrix} 0 & a_{0,1} & \cdots & a_{0,n-1} \\ 0 & a_{1,1} & \cdots & a_{1,n-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & a_{\frac{N}{2},1} & \cdots & a_{\frac{N}{2},n-1} \end{bmatrix}.$$

Hence, given matrix B, we can immediately obtain matrix R_1 and derive matrix R_2 by performing the following permutation according to Eq. (18)

$$a_{j,i} = b_{k,n+i-1}, \qquad (19)$$

where $j = r_i^{-1}(k)$ and $1 \le i \le n-1$.

VII. Utilization and Limitations

Like the other blocking-type multistage interconnection networks, the reverse-exchange network also has limitations. It cannot do the identity permutation, +1 mod N, etc. in one pass. But with its help we find the way to reconfigure the network to achieve all interconnection functions (including identity permutation and +1 mod N, etc.) of other networks in one pass. It also facilitates the two-pass structure which can realize arbitrary permutations. In this section we shall look for impacts of using the reverse-exchange network, along with the capabilities of the reconfiguration and the two-pass structure, in paralell processing systems.

The bit-reversal permutation is vitally important to the computation of the fast Fourier transformation. The flip network and the shuffle exchange network cannot realize the bit-reverse permutation in one pass. The permutation class of $C_{i,k}^{(n)}$ and $\tilde{C}_{i,k}^{(n)}$ which are realizable by the reverse-exchange network in one pass clearly indicates that the scrambled data can be aligned in bit-reverse order and the bit-reverse data can be restored in the original scrambled order.

In the memory to processor interconnection organization, the multi-dimensioning access memory [5,19] can be accessed by words, by bit slices, by byte-slices, etc. A scramble/unscramble network function is required to scramble the data when it is stored into memory and unscramble the data when it is read from memory. In the course of reconfiguration, the terminal link names (i.e. the memory module names and the processor names) are transformed by proper permutations such as ρ, δ and others. However if we modify the scrambling and unscrambling accordingly, we can achieve the same purpose of the multi-dimensioning access in various configurations. Thus the reverse-exchange network is not only good for FFT problems but also good for other problems in one pass through the use of reconfiguration. In the processor to processor interconnection organization, the output link is fed back to the input link, and the reconfiguration done on terminal links in one side automatically affect the other side. Therefore the reconfiguration scheme seems not promising in the processor to processor interconnection organization if the implementation cannot remove this restriction. However, the two-pass structure is good for both interconnection organizations.

An array computer can be composed of large numbers of processors for the fast realization of large problems. However, in some circumstances, the computation should be divided into subgroups and each groups, either identical or heterogeneous, can be performed in a small subarray of processors and achieve the efficiency through parallelism. Hence, it is convenient in these cases to be able to partition the computer into various subarrays. In the two-pass structure, the partition can be done for various sizes of subarrays. The permutation which may not be exactly of size N or of 2's power size is allowed. However, in one-pass, the partition can only be done for an equal size of 2's power. We have shown the partition definitions, $S_{q,k}$ and $\tilde{S}_{q,k}$, for the reverse-exchange network. The partition of some other networks can be found in [17].

There are still more work to be done for implementing the reconfiguration and the two-pass structure. It is interesting to know how many configurations are needed to realize arbitrary permutations in one pass. It seems also important to develop computation algorithms in terms of the configurations of a network. To realize a permutation required in an algorithm we should have the knowledge on how to name the terminal links and how to control the switching elements. Some algorithms may need two or more permutations. A mechanism to work out compromise configuring ways should be developed. If we cannot work out a general way, at least we should try to specify the relationships between frequently used permutations and configurations of a network. On the other hand, the routing scheme used in the two-pass structure needs the control algorithm background on Benes binary network. The looping algorithms need memory storage for computing the control pattern and the computing time needed is in the order of $N/2 \log_2(N/2)$. Lenfant's algorithm [8] overcomes those deficits. However it is restricted on some frequently used permutation families. It is worthwhile to extend his control mechanism to more generalized cases.

VIII. Conclusion

The reverse-exchange network is a valuable interconnection network not only because it can be well adapted to some important algorithmic processes such as FFT problems but also because it serves as an excellent reference from which a lot of fruitful results can be obtained. Previously, it was used to identify isomorphic structures among a class of multistage interconnection networks. In this paper, we first derive the functional relationships among the class of multistage interconnection networks as shown in Table 1, using the functional property of the reverse-exchange network. According to the functional relationships, we propose a reconfiguration scheme which can enhance the network capability. Then we specify the admissible permutations through a set of theorems and derive a set of recursive control algorithms to realize some useful permutations. Using the reverse-exchange property we also prove that the recursive control algorithms actually work. Finally we prove that arbitrary permutations can be realized on a two-pass structure of the reverse-exchange network. The routing scheme of the two-pass structure is also derived.

The feasibility of reconfiguration presents the great reality that a network can accomplish various interconnection functions of other networks in one pass. From the study of functional relationships we observe that we can accomplish other networks on a reference network through permuting names of terminal links of the reference network. The number of permutation patterns needed for such a renaming purpose is surprisingly small. For example, the baseline network needs only two patterns, ρ and δ, to accomplish various interconnection functions (see Table 1). The reconfiguration scheme is especially good for the memory to processor interconnection organization.

The two-pass structure offers many flexibilities for parallel processing. The structure can realize arbitrary permutations. With this structure the parallel processing system can be partitioned into subsystems of arbitrary sizes. This kind of system can be used to execute any algorithmic process. The two-pass structure is not only good for the memory to processor interconnection organization but also good for the processor to processor interconnection organization.

Finally, we like to point out that we do not count down the merits of other types of multistage interconnection networks. On the contrary, these glorious works on various types of networks mentioned in Table 1 are useful and more works are encouraged since any one of these networks can be configured on the reference network topology. Furthermore, any network can serve as the reference network for the reconfiguration and the two-pass structure since they are topologically equivalent. It is our intention to use the reconfiguration and the two-pass structure to remove the limitations of any one of these networks.

Acknowledgment

The authors wish to acknowledge the constructive comments from "referee #2" in reorganizing the contents of this paper.

References

[1] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comput., Vol. C-23, No. 3, March 1974, pp. 309-318.

[2] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," IEEE Trans. Comput., Vol. C-26, No. 2 Feb. 1977, pp. 153-161.

[3] K.E. Batcher, "The flip network in STARAN," Proc. of the 1976 International Conference on Parallel Processing, pp. 65-71.

[4] M. C. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comput., Vol. C-26, No. 5, May 1977, pp. 548-573.

[5] D. H. Lawrie, "Access and alignment of data in an array processor," IEEE Trans. Comput., Vol. C-24, Dec. 1975, pp. 1145-1155.

[6] L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," Proc. First Annual Symposium on Computer Architecture, Dec. 1973, pp. 21-28.

[7] T. Lang, "Interconnections between processors and memory modules using the shuffle-exchange network," IEEE Trans. Comput., Vol. C-25, No. 5, May 1976, pp. 496-503.

[8] J. Lenfant, "Parallel permutations of data: A Benes network control algorithm for frequently used permutations," IEEE Trans. Comput., Vol. C-27, No. 7, July 1978, pp. 637-647.

[9] S. E. Orcutt, "Implementation of permutation functions in Illiac IV-type computer," IEEE Trans. Comput., Vol. C-25, No. 9, Sept, 1976, pp. 929-936.

[10] H. S. Stone, "Paraller processing and the perfect shuffle," IEEE Trans, Comput., Vol. C-20, No. 2, Feb. 1971, pp. 153-161.

[11] C. Wu and T. Feng, "Routing techniques for a class of multistage interconnection networks," Proc. of the 1978 International Conference on Parallel Processing, pp. 197-205.

[12] H. J. Siegel, "The universality of various types of SIMD machine interconnection networks," Fourth Annual Symp. Computer Architecture, Mar. 1977, pp. 70-79.

[13] V. Benes, Mathematical Theory of Connecting Networks, New York: Academic Press, 1965.

[14] D. Opferman and N. Tsaŏ-Wu, "On a class of rearrangeable switching networks," Bell Syst. Tech. J., Vol. 50, May-June 1971, pp. 1579-1618.

[15] S. Andersen, "The looping algorithm extended to base 2^t rearrangeable switching networks," IEEE Trans, Commun., Vol. COM-25, No. 10, Oct. 1977, pp. 1057-1063.

[16] J. Lenfant and S. Tahé, "Permuting data with the omega network," RADC Final Report, Nov. 1978.

[17] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," Proc. Fifth Annual Symposium Computer Architecture, April 1978, pp. 223-229.

[18] T. Feng, C. Wu and D. Agrawal, "A micro-processor-controlled asynchronous circuit switching network," Proc. Sixth Annual Symp. on Computer Architecture," April 1979, pp. 202-215.

[19] K. E. Batcher, "The multi-dimensioning-access memory in STARAN," Proc. of the 1975 Sagamore Computer Conference, p. 167; also in IEEE Trans, Comput., Vol. C-26, No. 2, Feb. 1977, pp. 174-177.

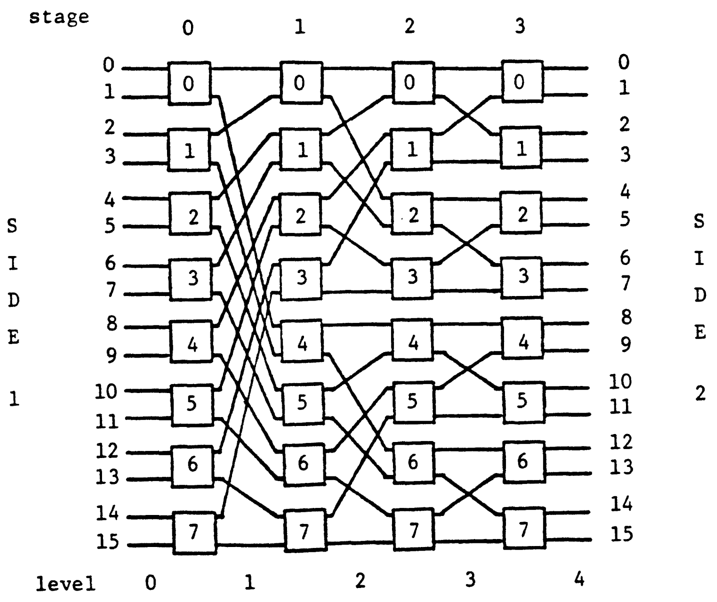


Fig. 1 A configuration of a 16 × 16 baseline network.

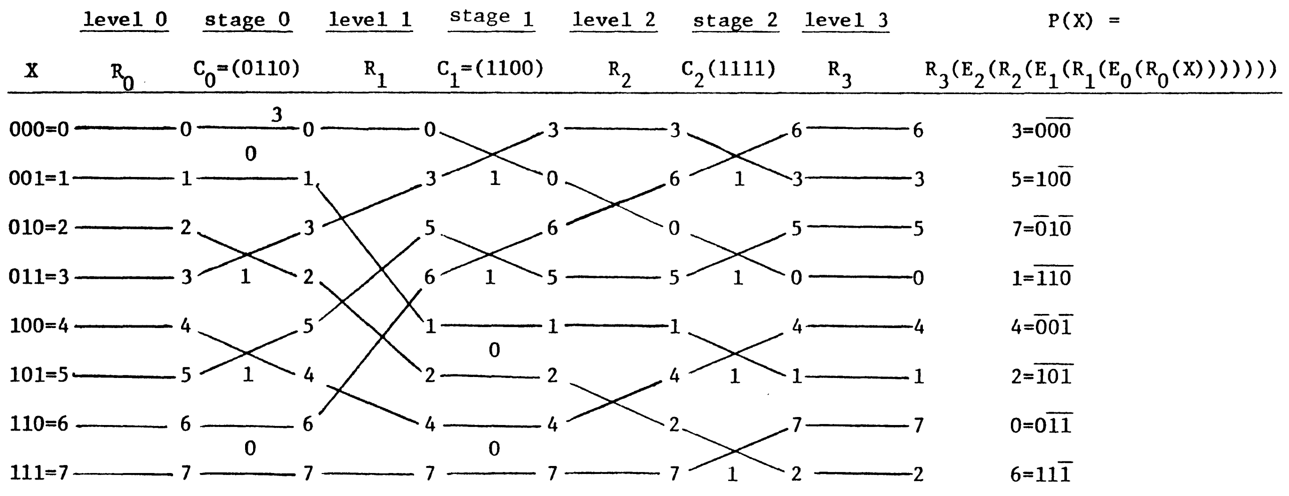


Fig. 2 A permutation realized by the reverse-exchange network of size 2^3.

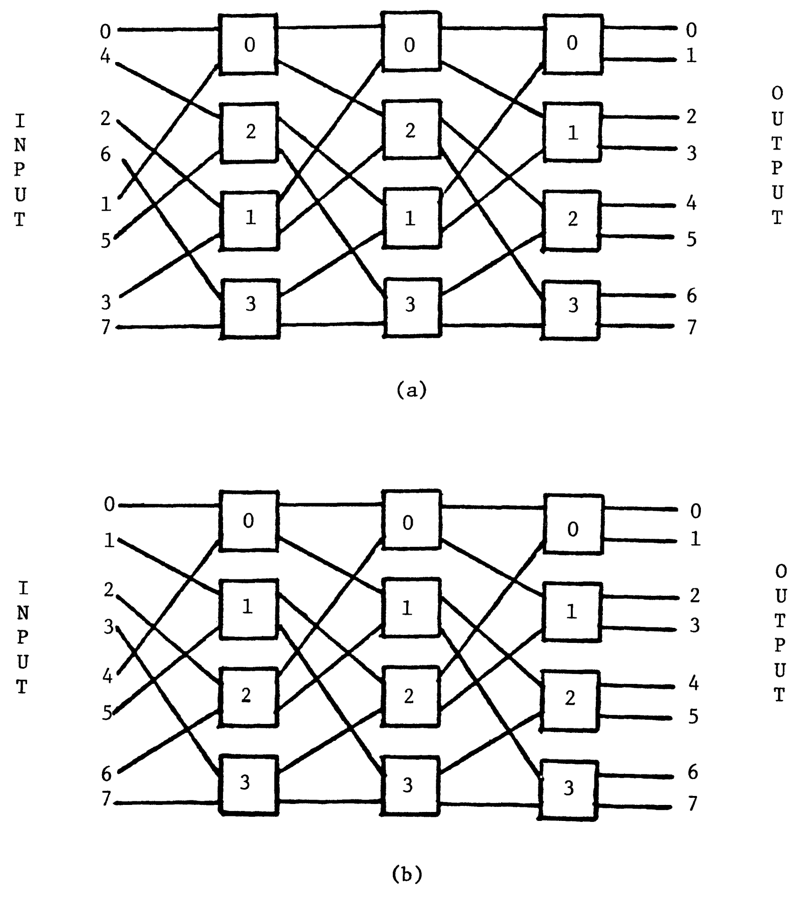


Fig. 3 (a) A reverse-exchange network.
(b) An omega network.

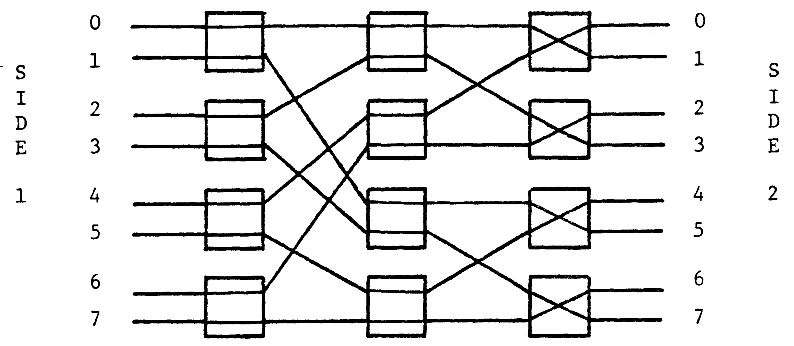


Fig. 4 Setting for $F_4^{(3)}$.

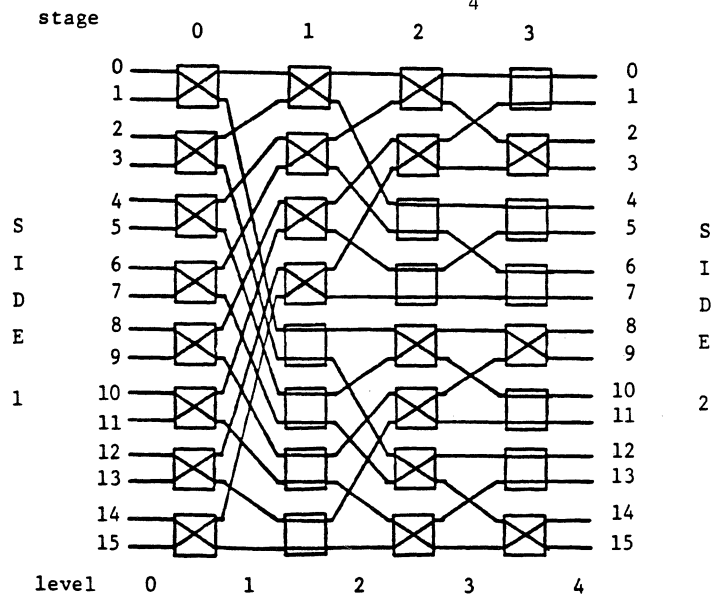
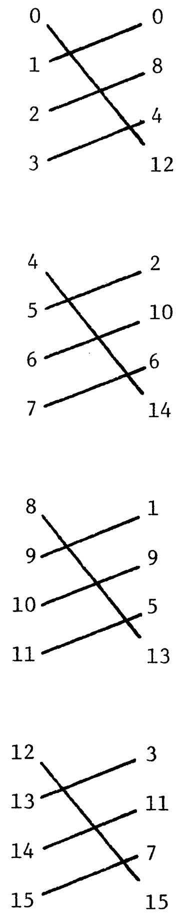


Fig. 5 Setting for $C_{5,7}^{(4)}$.



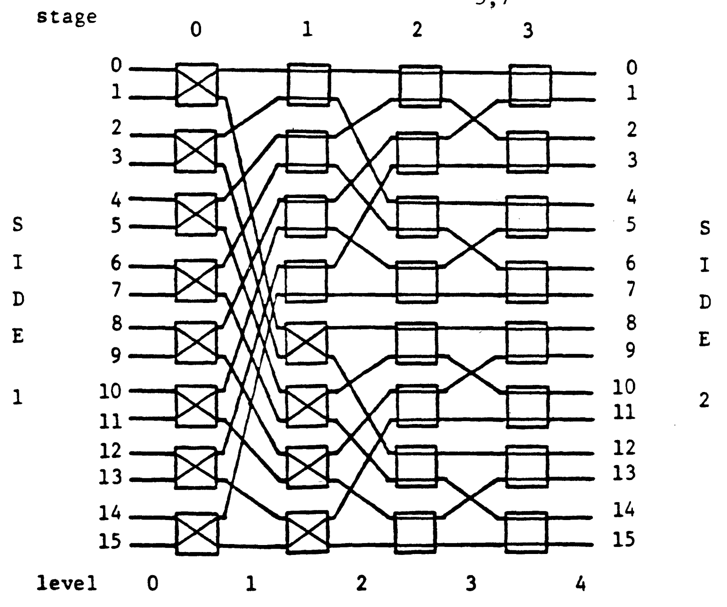


Fig. 6 $S_{2,3}^{(4)}$.

Fig. 7 Setting for $S_{2,3}^{(4)}$.

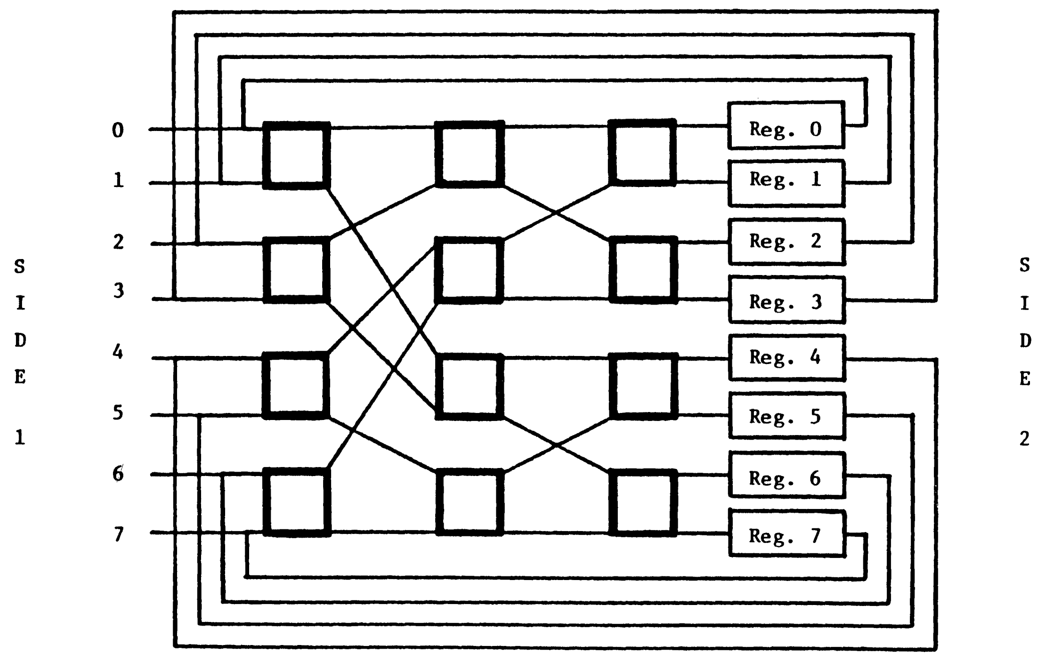


Fig. 8 Two-pass structure of a reverse-exchange network.

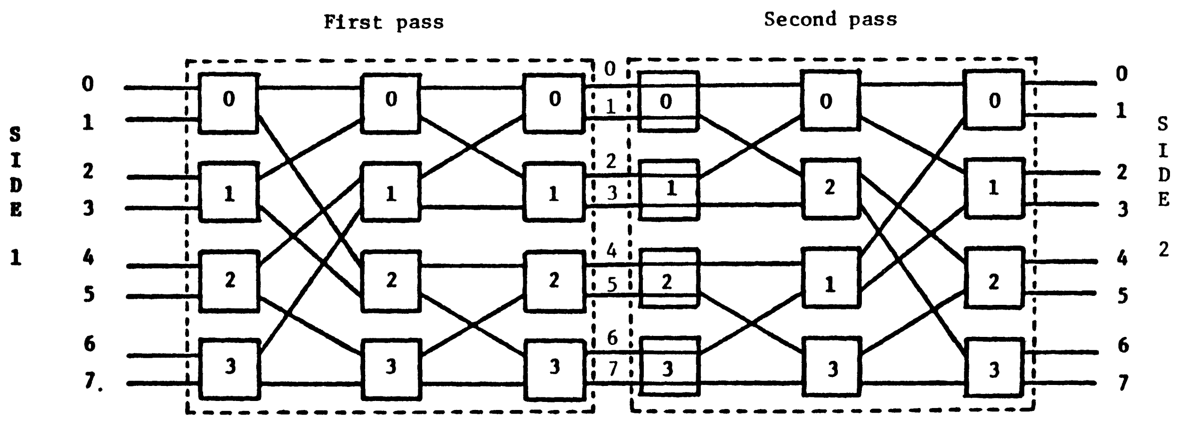


Fig. 9 Equivalent structure and switching element confinement of a two-pass structure.

Table 1 Functional Relationships Among Various Multistage Interconnection Networks ($P=f(\theta)$).

Network Function (P)	$f(\theta)$				
	$\theta = B_N$	$\theta = \Omega_N$	$\theta = C_N$	$\theta = D_N$	$\theta = G_N$
Baseline Network (B_N)	B_N	$\Omega_N \circ \rho$	$\rho \circ C_N$	$D_N \circ \delta$	$\delta \circ G_N$
Reverse Baseline Network (B_N^{-1})	B_N	$\Omega_N \circ \rho$	$\rho \circ C_N$	$D_N \circ \delta$	$\delta \circ G_N$
Omega Network (Ω_N)	$B_N \circ \rho$	Ω_N	$\rho \circ C_N \circ \rho$	$D_N \circ \delta \circ \rho$	$\delta \circ G_N \circ \rho$
Reverse Omega Network (Ω_N^{-1})	$\rho \circ B_N$	$\rho \circ \Omega_N \circ \rho$	C_N	$\rho \circ D_N \circ \delta$	$\rho \circ \delta \circ G_N$
Indirect Binary n-Cube Network (C_N)	$\rho \circ B_N$	$\rho \circ \Omega_N \circ \rho$	C_N	$\rho \circ D_N \circ \delta$	$\rho \circ \delta \circ G_N$
Reverse Indirect Binary (C_N^{-1}) n-Cube Network	$B_N \circ \rho$	Ω_N	$\rho \circ C_N \circ \rho$	D_N	$\delta \circ G_N \circ \rho$
Modified Data Manipulator(D_N)	$B_N \circ \delta$	$\Omega_N \circ \rho \circ \delta$	$\delta \circ C_N \circ \delta$	$\delta \circ D_N \circ \delta$	$\delta \circ G_N \circ \delta$
Reverse Modified Data Manipulator(D_N^{-1})	$\delta \circ B_N$	$\delta \circ \Omega_N \circ \rho$	$\delta \circ \rho \circ C_N$	$\delta \circ D_N \circ \delta$	G_N
Flip Network (F_N)	$\rho \circ B_N$	$\rho \circ \Omega_N \circ \rho$	C_N	$\rho \circ D_N \circ \delta$	$\rho \circ \delta \circ G_N$
Reverse Flip Network (F_N^{-1})	$B_N \circ \rho$	Ω_N	$\rho \circ C_N \circ \rho$	$D_N \circ \delta \circ \rho$	$\delta \circ G_N \circ \rho$
Regular SW Banyan (G_N) Network (S=F=2)	$\delta \circ B_N$	$\delta \circ \Omega_N \circ \rho$	$\delta \circ \rho \circ C_N$	$\delta \circ D_N \circ \delta$	G_N
Reverse Regular SW Banyan (G_N^{-1}) Network (S=F=2)	$B_N \circ \delta$	$B_N \circ \rho \circ \delta$	$\rho \circ C_N \circ \delta$	D_N	$\delta \circ G_N \circ \delta$

PARTITIONING PERMUTATION NETWORKS: THE UNDERLYING THEORY

Howard Jay Siegel
Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

Abstract--The age of the microcomputer has made feasible large-scale multiprocessor systems. In order to use this parallel processing power in the form of a flexible multiple-SIMD (MSIMD) system, the interconnection network must be partitionable and dynamically reconfigurable. The theory underlying the partitioning of MSIMD system permutation networks into independent subnetworks is explored. Conditions for determining if a network can be partitioned into independent subnetworks and the ways in which it can be partitioned are presented. The use of the theory is demonstrated by applying it to a variety of SIMD networks.

I. Introduction

An SIMD (single instruction stream - multiple data stream) machine [8] typically consists of a set of N processors, N memories, an interconnection network, and a control unit (e.g., the Illiac IV [1,5]). The control unit broadcasts instructions to the processing elements (PEs), where each PE is a processor/memory pair. All active ("turned on") PEs execute the same instruction at the same time. Each PE executes instructions using data taken from a memory with which only it is associated. The interconnection network allows interprocessor communication. When the interconnection network connects at most one input to each output it is also called a permutation network.

An MSIMD (multiple-SIMD) system is a parallel processing system which can be structured as one or more independent SIMD machines. The original design of Illiac IV was as an MSIMD system [1]. As the microprocessor revolution makes processors less expensive, multimicroprocessor systems which can operate in MSIMD mode are being proposed [6,13-15,19,22,24-26].

The partitionability of an interconnection network is the ability to divide the network into independent subnetworks of different sizes [23,27]. Each subnetwork of size N' must have all of the interconnection capabilities of a complete network of that same type built to be of size N'. A partitionable network can be characterized by any limitations on the way in which it can be subdivided. MSIMD systems use partitionable interconnection networks to dynamically reconfigure the system into independent SIMD machines of varying sizes.

This work was sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-78-3581. The United States Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

The theory underlying the partitioning of MSIMD permutation networks into independent subnetworks is explored in Section VI. The interconnection networks to be studied are the Cube, PM2I, and Shuffle-Exchange. In Sections III, IV, and V, these networks are defined and analyzed in terms of permutations. This analysis is required as preparation for the evaluation of partitionability in Section VI. The next section discussed how to view interconnections as permutations.

II. Interconnection Networks and Permutations

Formally, an interconnection network is defined to be a set of interconnection functions [17]. Each interconnection function is a bijection (a one-to-one and onto mapping) on the sets of input/output addresses, the integers from 0 to N-1. When interconnection function f is applied, the data at input i is moved to output f(i), for all i simultaneously, $0 \leq i < N$. Since an interconnection function is a bijection from the set of integers 0,1,2,...,N-1, onto that same set, it is also a permutation. In later sections it is assumed that N is a power of two.

A cyclic notation can be used to represent the bijection f as a permutation. The permutation is represented as the product of cycles, where the cycle

$$(j_0 \; j_1 \; j_2 \cdots j_{x-1} \; j_x)$$

means $f(j_0) = j_1$, $f(j_1) = j_2$, ..., $f(j_{x-1}) = j_x$, and $f(j_x) = j_0$. The length of this cycle is x+1. The physical interpretation of this cycle is that input j_0 is connected to output j_1, input j_1 is connected to output j_2, ..., input j_{x-1} is connected to output j_x, and input j_x is connected to output j_0.

The product of cycles is the composition of the bijections the cycles represent. If p and q are cycles, then the product pq represents the effect of first applying p and then applying q. For example, if

$$p = (0 \; 1) \text{ and } q = (1 \; 2),$$

then

$$pq = (0 \; 1) (1 \; 2) = (0 \; 2 \; 1),$$

since p maps 0 to 1 and q maps 1 to 2, pq maps 0 to 2, etc. The composition of cycles is not commutative, e.g.,

$qp = (\ 1\ 2\ )\ (\ 0\ 1\ ) = (\ 0\ 1\ 2\ ) \neq pq.$

The product of two or more permutations is defined similarly. For example, if

$g = (\ 0\ 1\ )\ (\ 2\ 3\ )$ and $h = (\ 0\ 1\ 2\ 3\ ),$

then

$gh = (\ 0\ 2\ )\ (\ 1\ )\ (\ 3\ ).$

That is, since g maps 0 to 1 and h maps 1 to 2, gh maps 0 to 2; since g maps 1 to 0 and h maps 0 to 1, gh maps 1 to 1; etc. In general, the composition of permutations is not commutative. For the example above,

$hg = (\ 1\ 3\ )\ (\ 0\ )\ (\ 2\ ) \neq gh\ .$

Every permutation can be uniquely represented as the product of disjoint cycles [10]. The cycle structure of an interconnection function is its unique disjoint cycles representation [18]. Cycles of length one (that is, $f(j_i) = j_i$) are typically not included. For example, the cycle structure of the function (permutation)

$f(x) = x + 2$ modulo 8, $0 \leq x < 8,$

is

$(\ 0\ 2\ 4\ 6\ )\ (\ 1\ 3\ 5\ 7\ ).$

The cycle structure of gh defined above is

$gh = (\ 0\ 2\ )\ (\ 1\ )\ (\ 3\ ) = (\ 0\ 2\ )\ .$

Sections III, IV, and V use the terminology reviewed in this section to define the Cube, PM2I, and Shuffle-Exchange interconnection networks. The definitions and permutation properties discussed in these sections will then be used to analyze partitionability in Section VI.

III. The Cube Interconnection Network

The Cube interconnection network consists of $n = \log_2 N$ interconnection functions

$$\text{cube}_i(s_{n-1}\cdots s_1 s_0) = s_{n-1}\cdots s_{i+1}\overline{s}_i s_{i-1}\cdots s_0$$

where $S = s_{n-1}\cdots s_1 s_0$, $\overline{s}_i$ is the complement of s_i, $0 \leq S < N$, and $0 \leq i < n$. For example, for $N \leq 8$, $\text{cube}_2(0) = 4$. The Cube network can be implemented as a recirculating (single stage) network or as a multistage network.

Figure 1 shows a general model for a recirculating network. Conceptually, a recirculating network may be viewed as N input selectors and N output selectors. The way in which the input selectors are connected to the output selectors determines the allowable connections. Since the network consists of only a single stage of connections, multiple passes through the network may be required to perform a data transfer, that is, the data may have to recirculate through the network several times.

For the Cube network, input selector $s_{n-1}\cdots s_1 s_0$ is connected to output selectors

$$s_{n-1}\cdots s_{i+1}\overline{s}_i s_{i-1}\cdots s_0, \quad 0 \leq i < n.$$

Output selector $t_{n-1}\cdots t_1 t_0$ gets its inputs from input selectors

$$t_{n-1}\cdots t_{i+1}\overline{t}_i t_{i-1}\cdots t_0, \quad 0 \leq i < n.$$

To execute the cube_i interconnection function, input selector j selects the $\text{cube}_i(j)$ output line and output selector j selects the $\text{cube}_i(j)$ input line, for all j, $0 \leq j < N$. Each recirculating Cube interconnection function for N = 8 is shown in Figure 2.

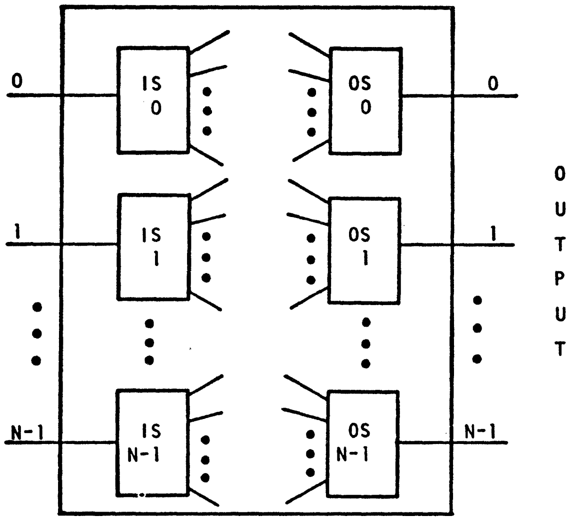


Figure 1: Model of a recirculating network. "IS" is input selector, "OS" is output selector.

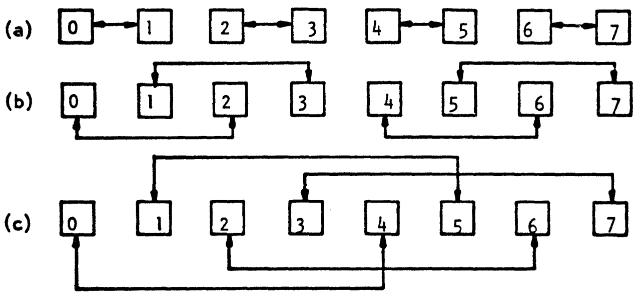


Figure 2: Cube recirculating network for N = 8. (a) Cube_0. (b) Cube_1. (c) Cube_2.

Various properties of the recirculating Cube SIMD network are presented in [17,18,20,21,28,29]. The CHoPP MIMD (multiple instruction stream - multiple data stream) machine [8] uses a type of recirculating Cube network [23,31].

Figure 3 is a model of a multistage Cube network. The boxes in this figure are called interchange boxes. In general, a multistage Cube network consists of n stages of N/2 interchange boxes. For each interchange box, the upper and lower outputs are labeled with the same numbers as the upper and lower inputs, respectively. Each interchange box can connect its lower input to its lower output and its upper input to its upper output (the straight state) or connect its lower input to its upper output and its upper input to its lower output (the exchange state). It is assumed that each box can be controlled individually (independent box control [21,27]). Stage i of this multistage network can implement the cube_i function, that is, connect an input line to the output line that differs from it only in the ith bit position.

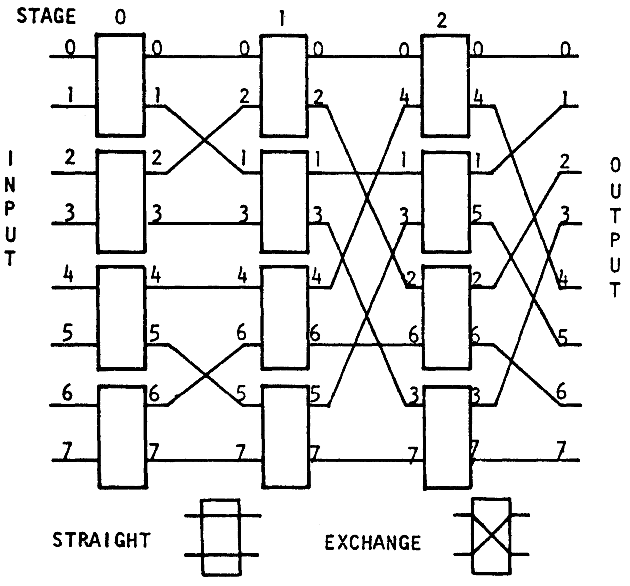


Figure 3: Model of a multistage Cube network for N = 8.

The STARAN SIMD network and the indirect binary n-cube SIMD network are multistage Cube networks, and their capabilities are discussed in [2-4,16]. The SW-banyan (S=F=2) proposed for the varistructured array processor is also based on the multistage Cube topology [9,14]. Other information about multistage Cube networks can be found in [23,27,29,32].

In terms of permutations, the cube_i interconnection function can be expressed uniquely as a product of N/2 disjoint cycles of size two by

$$\prod_{\substack{j=0 \\ i^{th} \text{ bit of } j=0}}^{N-1} (\; j \;\; \text{cube}_i(j) \;).$$

For example, for N = 8 cube_2 is

(0 4) (1 5) (2 6) (3 7) .

Consider a recirculating Cube network (Figure 2). As stated previously, all active PEs execute the same interconnection function (instruction) at the same time. In order for a data transfer to be representable as a permutation, if one PE in a cycle is inactive, the other PE in that cycle must be inactive also. For example, consider cube_2 for N = 8. If PEs 0 and 4 are inactive, the (0 4) cycle is removed, and the cube_2 permutation becomes

(1 5) (2 6) (3 7) .

If only PE 0 was inactive in the above example, then (1) PE 0 would "keep" its own data (0 0) and PE 4 would send it data (4 0), a two-to-one, not one-to-one, transfer; and (2) PE 4 would not receive any data, so the transfer would not be onto. Thus, in general, for each cycle in a permutation, either all PEs in the cycle must be active or all PEs in the cycle must be inactive in order for the resulting data to be representable as a permutation [18].

In general, when function cube_i is executed, the way in which the data is permuted is

$$\prod_{j=0}^{N-1} (\; j \;\; \text{cube}_i(j) \;),$$

where the ith bit of j = 0 and PE j (and PE $\text{cube}_i(j)$) is active.

Consider a multistage Cube network (Figure 3). Stage i of the network corresponds to the cube_i permutation if all the interchange boxes in stage i are set to exchange. For example, if all interchange boxes in the network are set to exchange, the way in which the data is permuted is

$$(\text{cube}_0)(\text{cube}_1)(\text{cube}_2)\ldots(\text{cube}_{n-1})$$

$$= \prod_{i=0}^{n-1} (\prod_{\substack{j=0 \\ i^{th} \text{ bit of } j=0}}^{N-1} (\; j \;\; \text{cube}_i(j) \;).$$

For example, for N = 4, the permutation is

(0 1) (2 3) (0 2) (1 3) .

In general, the way in which the data is permuted is

$$\prod_{i=0}^{n-1} (\prod_{j=0}^{N-1} (\; j \;\; \text{cube}_i(j) \;),$$

where the ith bit of j=0 and the stage i interchange box whose inputs are labeled j and $\text{cube}_i(j)$ are set to exchange. For example, if in Figure 3 only the top row of boxes are set to exchange (and the rest set to straight), the permutation is

(0 1) (0 2) (0 4) = (0 1 2 4) .

This section discussed how to describe the actions of the recirculating Cube network and a multistage Cube with independent box control in terms of permutations. These descriptions will be used in Section VI to analyze how these networks can be partitioned into independent subnetworks.

IV. The PM2I Interconnection Network

The Plus-Minus 2^i (PM2I) interconnection network consists of 2n interconnection functions

$$\text{PM2}_{+i}(j) = j + 2^i \text{ modulo } N,$$

$$\text{PM2}_{-i}(j) = j - 2^i \text{ modulo } N,$$

where $0 \leq j < N$, $0 \leq i < n$, $n = \log_2 N$, and j-x modulo N = j+(N-x) modulo N. For example, for $N \geq 8$, $\text{PM2}_{+2}(0) = 4$. Note that $\text{PM2}_{+n-1} = \text{PM2}_{-n-1}$. The PM2I network can be implemented as a recirculating (single stage) network or as a multistage network.

Consider the model of recirculating networks shown in Figure 1. For the PM2I network, input selector j is connected to output selectors

$$j+2^i \text{ and } j-2^i, \quad 0 \leq i < n.$$

Output selector j gets its inputs from input selectors

$$j-2^i \text{ and } j+2^i, \quad 0 \leq i < n.$$

To execute the $PM2_{+i}$ interconnection function, input selector j selects the $PM2_{+i}(j)$ output line and output selector j selects the $PM2_{-i}(j)$ input line, for all j, $0 \leq j < N$. To execute the $PM2_{-i}$ interconnection function, input selector j selects the $PM2_{-i}(j)$ output line an output selector j selects the $PM2_{+i}(j)$ input line, for all j, $0 \leq j < N$. Each recirculating PM2I interconnection function for N = 8 is shown in Figure 4.

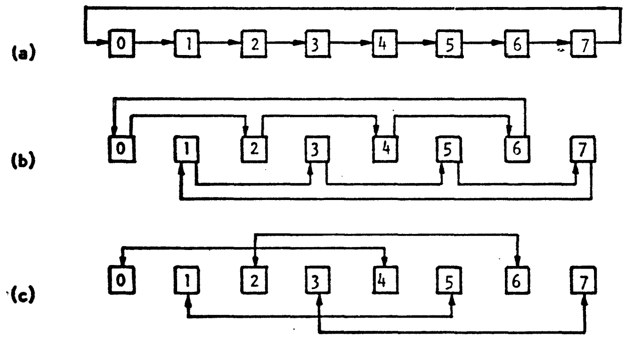


Figure 4: PM2I recirculating network for N = 8. (a) $PM2_{+0}$. (b) $PM2_{+1}$. (c) $PM2_{+2}$. For the $PM2_{-i}$ connections, reverse the directions of the arrows.

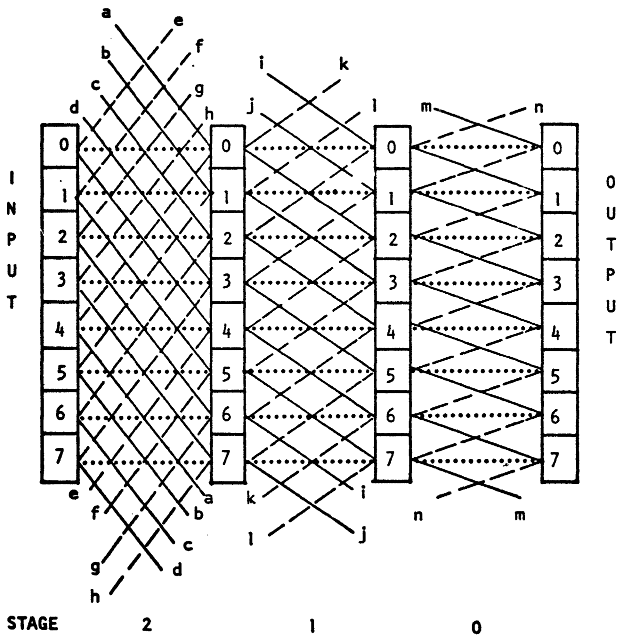


Figure 5: Augmented Data Manipulator multistage PM2I network for N = 8. The lower case letters represent "end-around" connections.

Various properties of the recirculating PM2I network are presented in [17,18,20,21,28,29].

Figure 5 shows a multistage PM2I network topology called the <u>data manipulator</u> network [7]. In general, the data manipulator consists of n stages of N cells. For $0 \leq j < N$ and $0 \leq i < n$, there are three sets of interconnections at stage i: one sends the data from input cell j to output cell $j+2^i$ modulo N ($PM2_{+i}$), one sends the data from input cell j to output cell $j-2^i$ modulo N ($PM2_{-i}$), and one sends the data from input cell j to output cell j (straight).

The control scheme originally proposed for the data manipulator is not flexible enough for partitioning because sets of cells would receive the same control signals. For this reason, the more flexible (and more costly) <u>augmented data manipulator</u> (ADM) network has been proposed [27]. In the ADM network, each cell receives its own control signals. Specifically, for $0 \leq i < n$, each cell at stage i can get any of the signals D ("down" = $PM2_{+i}$, the solid line in Figure 5), U ("up" = $PM2_{-i}$, the dashed line), or H ("horizontal" = straight, the dotted line).

More information about the data manipulator and augmented data manipulator (ADM) multistage PM2I networks can be found in [7,21,26-29].

In terms of permutations, the $PM2_{+i}$ interconnection function can be expressed uniquely as a product of the following 2^i disjoint cycles of size 2^{n-i} by

$$\prod_{j=0}^{2^i-1} (\ j\ \ j+2^i\ \ j+2*2^i\ \ j+3*2^i\ \ldots\ j+N-2^i\ ).$$

For example, for N = 8, $PM2_{+1}$ is

$$(\ 0\ 2\ 4\ 6\ )\ (\ 1\ 3\ 5\ 7\ ).$$

The $PM2_{-i}$ interconnection function can be expressed uniquely as a product of the following 2^i disjoint cycles of size 2^{n-i} by

$$\prod_{j=0}^{2^i-1} (\ j+N-2^i\ \ldots\ j+3*2^i\ \ j+2*2^i\ \ j+2^i\ \ j\ ).$$

For example, for N = 8, $PM2_{-1}$ is

$$(\ 6\ 4\ 2\ 0\ )\ (\ 7\ 5\ 3\ 1\ ).$$

Consider a recirculating PM2I network (Figure 4). As with the Cube network, if one PE in a cycle is inactive, the other PEs in that cycle must also be inactive if the data transfer is to be representable as a permutation. For example, consider $PM2_{+1}$ for N = 8. If PEs 0, 2, 4, and 6 are inactive, the (0 2 4 6) cycle is removed and the permutation becomes (1 3 5 7).

In general, when the interconnection function $PM2_{+i}$ is executed, the way in which the data is permuted is

$$\prod_{j=0}^{2^i-1} (\ j\ \ j+2^i\ \ j+2*2^i\ \ j+3*2^i\ \ldots\ j+N-2^i\ ),$$

where for each j PEs $j+k*2^i$, $0 \leq k < (2^{n-i}-1)$, are all active. When the interconnection function $PM2_{-i}$ is executed, the way in the data is permuted is

$$\prod_{j=0}^{2^i-1} (\ j+N-2^i\ \ldots\ j+3*2^i\ \ j+2*2^i\ \ j+2^i\ \ j\ ),$$

where for each j PEs $j+k*2^i$, $0 \leq k < (2^{n-i}-1)$, are all active.

Consider the ADM network (Figure 5). In order for the entire data transfer (from the input of the network to the output of the network) to be representable as a permutation, no data can be destroyed at any stage. This implies that, for $0 \leq i < n$, the transfer of data from the input cells of stage i to the output cells of stage i must be representable as a permutation, that is, each input cell must be connected to exactly one output cell. In general, at stage i, the flexible control scheme allows some cells to execute $PM2_{+i}$, while others execute $PM2_{-i}$, while still others execute "straight." With the recirculating structure this was not allowed, i.e., either all active PEs executed $PM2_{+i}$ or all active PEs executed $PM2_{-i}$ (inactive PEs being equivalent to the "straight" state for the multistage network). The following lemma examines how this increased flexibility affects the set of permutations performable by the ADM network. This lemma will be used to analyze the partitionability of the ADM network in Section VI.

Lemma: If all data transfers are representable as permutations, then in the ith stage of the ADM network, $0 \leq i < n$, the transfer of data from input cell j can be represented only as any one of the following five cycles:

1. (j),
2. $(j \;\; j+2^i \;\; j+2*2^i \;\; j+3*2^i \; \ldots \; j+N-2^i)$
3. $(j+N-2^i \; \ldots \; j+3*2^i \;\; j+2*2^i \;\; j+2^i \;\; j)$,
4. $(j \;\; j+2^i)$, or
5. $(j \;\; j-2^i)$,

where all arithmetic is modulo N.

Proof: There are three cases for the ith stage: input cell j connects to output cell j, $j+2^i$, or $j-2^i$.

Case 1: Input cell j connects to output cell j. This is form 1.

Case 2: Input cell j connects to output cell $j+2^i$. Since the data transfer must be representable as a permutation, input cell $j+2^i$ can not be connected to output cell $j+2^i$, so it must be connected to either output cell (a) $j+2^i-2^i = j$ or (b) $j+2^i+2^i$. Subcase (a) is, obviously, form 4. Subcase (b) is form 2 because for k = 2,3,4,..., (in that order) input cell $j+k*2^i$ can not be connected to either output cell $j+k*2^i$ or $j+(k-1)*2^i$, since they are already connected to input cells. Therefore, for this subcase, $j+k*2^i$ must be connected to output cell $j+(k+1)*2^i$, $0 \leq k < 2^{n-i}$.

Case 3: Input cell j connects to output cell $j-2^i$. Using arguments similar to those in Case 2, it can be shown that this case must generate a cycle of either form 3 or 5 (recall that $j-k*2^i = j+N-k*2^i = j+(2^{n-i}-k)*2^i$ modulo N). For i = n-1, forms 2 and 3 are the same and forms 4 and 5 are the same, since $j+2^{n-1} = j-2^{n-1}$ modulo N. □

Thus, a permutation is performable at stage i if and only if it can be represented as the product of disjoint cycles of the forms 1 to 5 given above. For example, for N = 8, at stage 1, the permutation

$(0 \; 2 \; 4 \; 6) (1 \; 3) (5 \; 7)$

is performable. If $perm_i$ represents the permutation performed at stage i of the ADM network, then the permutation performed by the entire network is

$$\prod_{i=n-1}^{0} perm_i .$$

Note that i goes from n-1 to 0 because data travels from stage n-1 to stage n-2 to stage n-3, etc.

This section discussed how to describe the actions of the recirculating PM2I network and multistage ADM network in terms of permutations. These descriptions will be used in Section VI to analyze how these networks can be partitioned into independent subnetworks.

V. The Shuffle-Exchange Network

The Shuffle-Exchange interconnection network consists of two interconnection functions, the shuffle and the exchange.

$$\text{shuffle}(s_{n-1}\ldots s_1 s_0) = s_{n-2}\ldots s_1 s_0 s_{n-1}$$

where $S = s_{n-1}\ldots s_1 s_0$, $0 \leq S < N$, and $n = \log_2 N$. This will be referred to as a shuffle function based on N elements. For example, for $N \geq 4$, shuffle(1) = 2.

$$\text{exchange}(S) = \text{cube}_0(S)$$

where $0 \leq S < N$. For example, for $N \geq 2$, exchange(1) = 0. The Shuffle-Exchange can be implemented as a recirculating (single step) or as a multistage network.

Consider the model of recirculating networks shown in Figure 1. For the Shuffle-Exchange network, input selector $s_{n-1}\ldots s_2 s_1 s_0$ is connected to output selectors

$$s_{n-2}\ldots s_2 s_1 s_0 s_{n-1} \text{ and } s_{n-1}\ldots s_2 s_1 \overline{s}_0 .$$

Output selector $t_{n-1}\ldots t_2 t_1 t_0$ gets its inputs from input selectors

$$t_0 t_{n-1}\ldots t_2 t_1 \text{ and } t_{n-1}\ldots t_2 t_1 \overline{t}_0 .$$

To execute the shuffle interconnection function, input selector $s_{n-1}\ldots s_1 s_0$ selects the $s_{n-2}\ldots s_1 s_0 s_{n-1}$ output line and output selector $t_{n-1}\ldots t_1 t_0$ selects the $t_0 t_{n-1}\ldots t_2 t_1$ input line. To execute the exchange interconnection function, input selector $s_{n-1}\ldots s_1 s_0$ selects the $s_{n-1}\ldots s_1 \overline{s}_0$ output line and output selector

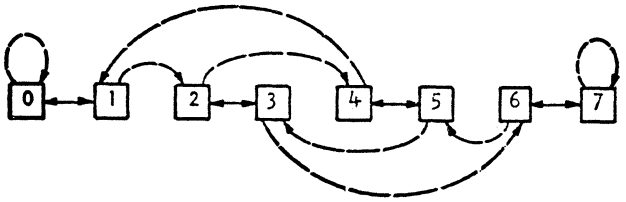


Figure 6: Shuffle-exchange recirculating network for N = 8. Solid line is exchange, dashed line is shuffle.

$t_{n-1} \cdots t_1 t_0$ selects the $t_{n-1} \cdots t_1 \overline{t}_0$ input line. The shuffle and exchange interconnection functions for N = 8 are shown in Figure 6.

The use of a recirculating Shuffle-Exchange network for parallel processing was first proposed in [30]. Various properties of this network are discussed in [17,18,20,21,28,30].

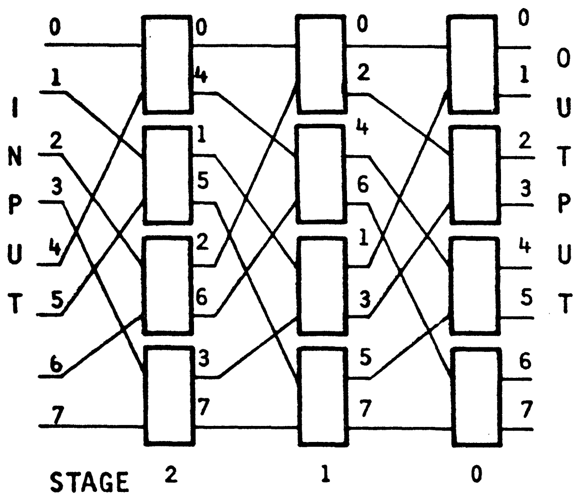


Figure 7: Model of a multistage Shuffle-Exchange network for N = 8.

Figure 7 is a model of a multistage Shuffle-Exchange network. Like the multistage Cube network model, the multistage Shuffle-Exchange network consists of n stages. Each stage of a Shuffle-Exchange network consists of the shuffle interconnection (connecting the line at position x to position shuffle(x), $0 \leq x < N$) followed by a column of N/2 interchange boxes. Recall that the upper and lower outputs of the interchange boxes are labeled with the same numbers as the upper and lower inputs, respectively. It is assumed that each interchange box is controlled independently and may be in either the straight or exchange state.

Various properties of multistage Shuffle-Exchange networks are described in [11,12,21,27]. (The interchange boxes of the "omega" multistage Shuffle-Exchange network [12] can be in "broadcast" states in addition to the straight and exchange states, but here only the latter states are considered.)

In terms of a permutation, the exchange interconnection function can be expressed uniquely as a product of N/2 disjoint cycles of size two by

$$\prod_{\substack{j=0 \\ j \text{ even}}}^{N-2} (\ j \ \ j+1\ ) .$$

For example, for N = 8 the exchange is

$$(\ 0\ 1\ )\ (\ 2\ 3\ )\ (\ 4\ 5\ )\ (\ 6\ 7\ ) .$$

Let "shuffle^i" mean apply the shuffle function i times. Then, in terms of a permutation, the shuffle interconnection function can be expressed uniquely as a product of disjoint cycles by

$$\prod_{\substack{j=0 \\ j \text{ not in a} \\ \text{previous cycle}}}^{N-1} (\ j \ \ \text{shuffle}(j) \ \ \text{shuffle}^2(j) \ \ldots\ )$$

For example, for N = 8 the shuffle is

$$(\ 0\ )\ (\ 1\ 2\ 4\ )\ (\ 3\ 6\ 5\ )\ (\ 7\ )$$
$$= (\ 1\ 2\ 4\ )\ (\ 3\ 6\ 5\ ).$$

In general, for a shuffle based on N elements, the sizes of the cycles in the product of disjoint cycles representation of the shuffle will vary. However, the largest a cycle can be is n, since $\text{shuffle}^n(x) = x$, $0 \leq x < N$.

Consider a recirculating Shuffle-Exchange network (Figure 6). Recall that if one PE in a cycle is inactive, the other PEs in that cycle must also be inactive if the data transfer is to be representable as a permutation. For example, consider the shuffle function for N = 8. If PEs 1, 2, and 4 are inactive, the (1 2 4) cycle is removed and the permutation becomes (3 6 5). In general, when the shuffle interconnection function is executed, the way in which data is permuted is

$$\prod_{j=0}^{N-1} (\ j \ \ \text{shuffle}(j) \ \ \text{shuffle}^2(j) \ \ldots\ ),$$

where, for each cycle, j has not appeared in a previous cycle and PEs $\text{shuffle}^i(j)$, $0 \leq i < n$, are all active.

The permutation analysis for the exchange interconnection function of a recirculating Shuffle-Exchange network is the same as the analysis for cube_0 presented in Section III.

To describe the permutations performable by multistage Shuffle-Exchange networks, their relationship to the Generalized Cube network is examined. The Generalized Cube network [27] is identical to the multistage Cube network shown in Figure 3, except the data travels in the opposite direction, that is, through stage n-1, then stage n-2, then stage n-3, etc. Thus, the permutations performable by the Generalized Cube network can be expressed as

$$\prod_{i=n-1}^{0} \left(\prod_{j=0}^{N-1} (\ j \ \ \text{cube}_i(j)\ ) \right),$$

where the ith bit of j=0 and the stage i interchange box whose inputs are labeled j and $\text{cube}_i(j)$ are set to exchange. Notice that the outer product goes from i = n-1 to n-2 to n-3, etc., due to the order in which the stages of the network are traversed.

In [27] it is shown that structure and connection capabilities of multistage Shuffle-Exchange networks are the same as those of the Generalized Cube network. At stage i of both networks the labels of inputs to interchange boxes differ only in their i^{th} bit position. This shows the relationship between the Shuffle-Exchange and Cube multistage networks. In particular, the interchange boxes in stage i of multistage Shuffle-Exchange networks implement the cycles of the $cube_i$ interconnection function. Therefore, the expression above for the permutations performable by the Generalized Cube network also describes the permutations performable by multistage Shuffle-Exchange networks.

This section discussed how to describe the actions of recirculating and multistage Shuffle-Exchange networks in terms of permutations. The description of the recirculating Shuffle-Exchange network will be used in Section VI to show that it can not be partitioned. The description of multistage Shuffle-Exchange networks will be used to show how to partition these networks into independent subnetworks.

VI. Partitioning

A. Definitions

To analyze formally the partitioning of interconnection networks, the following definitions are introduced:

(a) $P = \{0, 1, 2, \ldots N-1\}$, the set of physical PE addresses;

(b) $L_i = \{L_{i0}, L_{i1}, L_{i2}, \ldots L_{i(w_i-1)}\}$, the set of logical PE addresses in the i^{th} partition;

(c) w_i is the size of L_i (that is, $|L_i| = w_i$), where $0 < w_i \leq N$ and w_i is a power of two;

(d) v is the number of partitions, where $0 < v \leq N$;

(e) $L = L_0 \cup L_1 \cup L_2 \ldots \cup L_{(v-1)} = \bigcup_{j=0}^{v-1} L_j$, the set of logical PE addresses for all partitions, where

$$|L| = \sum_{i=0}^{v-1} w_i = N;$$

(f) m is a bijection from P to L, such that if $m(p_k) = L_{ij}$ then $m^{-1}(L_{ij}) = p_k$, where $p_k \in P$ and $L_{ij} \in L$.

The physical interconnection network of a system is defined in terms of P.

B. Partitioning the Cube Network

The $cube_q$ interconnection function causes the PE whose logical address is x to send its data to the PE whose logical address is y if and only if $cube_q(m^{-1}(x)) = m^{-1}(y)$. In order to partition the Cube network into independent subnetworks, the mapping m must have certain properties. For $0 \leq i < v$, the i^{th} partition, L_i, must be such that $\log_2 w_i$ Cube interconnection functions are available for its independent use. Furthermore, L_{ij}, $0 \leq j < w_i$, must be connected to each of the $\log_2 w_i$ PEs in L_i whose logical addresses differ from j in only one bit position.

Theorem 1: In terms of the cycle structure of the Cube interconnection functions, the network will be partitioned into independent subnetworks if and only if m is such that $\forall i$, $0 \leq i < v$, for each of $\log_2 w_i$ distinct Cube functions exactly $w_i/2$ of the cycles contain only elements of P which are mapped to elements of L_i by m. In addition, for $0 \leq r < \log_2 w_i$, if $cube_{q_r}(m^{-1}(L_{ij})) = m^{-1}(L_{ik})$, then j and k can differ in only the r^{th} bit position, $\forall i,k$, $0 \leq j,k < w_i$.

Proof: Since the cycles in the cycle structure are disjoint, if exactly $w_i/2$ of the size two cycles contain only elements of P which are mapped by m to elements of L_i, all of the elements of L_i are included, and only elements of L_i are included. Thus, because the cycles are disjoint, no element which maps to an element of L_i is in a cycle other than one of these $w_i/2$. Therefore, the collection of the $w_i/2$ cycles in each of $\log_2 w_i$ Cube functions constitute a complete and independent Cube network for L_i. The constraint that, for $0 \leq r < \log_2 w_i$ and $cube_{q_r}(m^{-1}(L_{ij})) = m^{-1}(L_{ik})$, j and k can differ only in the r^{th} bit position, $\forall i,j$, $0 \leq j,k < w_i$, establishes a correspondence between the physical Cube connections and the logical connections for a partition, maintaining the properties of the Cube network. This constraint requires the $cube_{q_r}$ interconnection function to connect PEs whose logical addresses differ only in the r^{th} bit position. Without this constraint, m would not preserve the properties of the Cube network (e.g., $m(0) = L_{i0}$, $m(1) = L_{i3}$, $m(2) = L_{i1}$, $m(3) = L_{i2}$ would incorrectly be allowed for a partition of size four). □

For example, if N = 8, v = 3, $w_0 = 4$, $w_1 = 2$, and $w_2 = 2$, then one possible choice for m is $m(0) = L_{00}$, $m(1) = L_{10}$, $m(2) = L_{01}$, $m(3) = L_{20}$, $m(4) = L_{02}$, $m(5) = L_{11}$, $m(6) = L_{03}$, $m(7) = L_{21}$. The mapping m meets the requirements by picking the following sets of cycles: (a) for L_0: (0 2) (4 6) from $cube_1$, (0 4) (2 6) from $cube_2$; and (b) for L_1: (1 5) from $cube_2$; and (c) for

L_2: (3 7) from cube_2. In general, the physical addresses of all the PEs in a partition L_i must agree in the $n-\log_2 w_i$ bit positions not corresponding to the $\log_2 w_i$ Cube functions the partition will use for communications [27].

For interprocessor communications within the partition, only a subset of the cycles need to be used. For example, to connect L_{0j} to $L_{0(j+1 \text{ modulo } 4)}$, $0 \leq j \leq 3$, use:

$$(0\ 2) (4\ 6) (0\ 4) = (0\ 2\ 4\ 6)$$

which, by applying m, is (L_{00} L_{01} L_{02} L_{03}).

Consider the model of a recirculating Cube network (Figure 1). In order to have multiple independent SIMD machines, multiple controllers must be available. If each PE sets its own input and output selectors, based on the transfer instructions it receives from its controller, each partition can perform Cube cycles independent of the other partitions. Subsets of the cycles available to a partition are chosen by disabling the appropriate PEs during the data transfer.

In a multistage Cube network (Figure 2), the cycle (x y), where $x,y \in P$, is implemented by the interchange box uniquely determined by the inputs labeled x and y. The assumption that each interchange box is controlled individually is needed so that the different partitions can operate independently and concurrently.

C. Partitioning the PM2I Network

The partitionability of the recirculating PM2I network is first examined. Following that, the partitionability of the ADM is explored.

Theorem 2: In terms of the cycle structure of the PM2I functions, recirculating PM2I networks will be partitioned into independent subnetworks if and only if m is such that for $\forall i$, $0 \leq i < v$, for each of $2*\log_2 w_i$ PM2I functions there exist cycles containing all of the elements of P which are mapped to elements of L_i by m, and nothing else. In addition, for $0 \leq r < \log_2 w_i$, if $PM2_{+x_r}(m^{-1}(L_{ij})) = m^{-1}(L_{ik})$, then $k = j+2^{x_r}$, and if $PM2_{-x_r}(m^{-1}(L_{ij})) = m^{-1}(L_{ik})$, then $k = j-2^{x_r}$, $\forall j,k$, $0 \leq j,k < w_i$.

Proof: Let $w_i = N/2^a$, $j \in P$, and $m(j) \in L_i$. The PM2I function $PM2_{+x}$ and $PM2_{-x}$, $0 \leq x < a$, can not be used by j because their cycles are of length $N/2^{a-1}$ or longer. If L_i uses a cycle longer than w_i it will not be independent of L_j, $0 \leq i,j < v$, $i \neq j$. By the definition of the PM2I network, if a partition is of size w_i it must have $2*\log_2 w_i$ PM2I functions to use. Since $a = n-\log_2 w_i$ and $PM2_{+-x}$, $0 \leq x < a$ can not be used, this leaves the $2*\log_2 w_i$ functions $PM2_{+-x}$, $a \leq x < n$. It must now be shown that for each of these functions, there are cycles which contain all $j \in P$ such that $m(j) \in L_i$ and no $b \in P$ such that $m(b) \notin L_i$. $PM2_{+-a}$ must be available for use by L_i. Therefore, if $m(j) \in L_i$ all of the w_i elements in the cycle containing j must be in L_i. Thus, L_i is defined to be $j+k*2^a$, $0 \leq k < 2^{n-a}$, that is all those elements of P whose low-order "a" bits equal j [27]. For $a < x < n$, any cycle containing one element whose low-order "a" bits are j will contain only elements whose low-order "a" bits are j. Thus, the choice for L_i of $j+k*2^a$, $0 \leq k < 2^{n-1}$, is the only way to provide $2*\log_2 w_i$ PM2I functions that can be used independently by number of L_i. The constraint stated at the end of the Theorem statement ensures that m is such that the mathematical properties of the PM2I permutations are preserved. For example, without the constraint, for $N = w_0 = 4$, $m(0) = L_{00}$, $m(1) = L_{03}$, $m(2) = L_{01}$, $m(3) = L_{02}$ would incorrectly be allowed. □

Theorem 3: The ADM network can be partitioned based on the criteria described in Theorem 2.

Proof: The Lemma in Section IV showed the five forms of cycles needed to partition the ADM into independent subnetworks with the properties of the complete network. For stage i, the elements of the cycles are a subject of $j+k*2^i$, $0 \leq k < 2^{n-i}$, and, include all of $j+k*2^i$, $0 \leq k < 2^{n-i}$. The rest follows from Theorem 2.□

For example, if $N = 8$, $v = 3$, $w_0 = 4$, $w_1 = 2$, and $w_2 = 2$, then one possible correct choice for m is $m(0) = L_{10}$, $m(1) = L_{00}$, $m(2) = L_{20}$, $m(3) = L_{01}$, $m(4) = L_{11}$, $m(5) = L_{02}$, $m(6) = L_{21}$, $m(7) = L_{03}$. The mapping m meets the requirements by picking the following set of cycles: (a) for L_0: (1 3 5 7) from $PM2_{+1}$, (7 5 3 1) from $PM2_{-1}$, (1 5) (3 7) from $PM2_{+2}$; (b) for L_1: (0 4) from $PM2_{+2}$; and (c) for L_2: (2 6) from $PM2_{+2}$. (Recall $PM2_{+2} = PM2_{-2}$.) For interprocessor communications within the partition, only a subset of the cycles need to be used. For example, to connect the processor pairs L_{0j} and L_{0k}, where j and k differ in the high-order bit position, use

$$(1\ 3\ 5\ 7) (1\ 5) = (1\ 3) (5\ 7)$$

which, by applying m, is

$$(L_{00}\ L_{02}) (L_{01}\ L_{03}) .$$

Consider the model of a recirculating PM2I network (Figure 1). As in the case of partitioning the recirculating Cube network, there must be multiple control units. Subsets of the cycles available to a partition are chosen by disabling PEs during the data transfer.

In the ADM, the cycles corresponding to a given stage can be selected by sending the appropriate control signals (H, D, or U). Cycles of the form "(j)" are established by the "H" connection.

D. Partitioning the Shuffle-Exchange Network

If implemented as a recirculating network the Shuffle-Exchange network can not be used to partition the set of PEs into independent groups whose sizes, w_i, $0 \leq i < v$, are powers of two.

Theorem 4: The Shuffle-Exchange recirculating network can not be partitioned into independent subnetworks.
Proof: To have a complete recirculating Shuffle-Exchange network for a partition of size w_i, it must first be possible to partition the set P into subsets of size w_i, $0 \leq i < v$, such that all PEs whose physical addresses map to logical addresses in L_i have a shuffle interconnection based on w_i elements. In general, this is not possible. The assumption that it is possible will be made, and as a result, a contradiction will be reached. Let $G = 00...01 \in P$ and $m(G) \in L_i$ for some i, $0 \leq i < v$, where $0 < w_i \leq N$ and w_i is a power of two. Based on the definition of the shuffle interconnection function, the size of the largest cycle of a shuffle function based on w_i elements is $\log_2 w_i$, where $0 \leq \log_{w_i} < n$. But G will be in a cycle of size n. In particular, G will be in a cycle containing the PEs whose physical addresses are 00...010, 00...0100, ..., 10...00. Thus, if $m(G) \in L_i$, then $L_i = P$ and $w_i = N$. □

To evaluate the partitionability of multistage Shuffle-Exchange networks, the permutation expression derived in Section V is used. The results of Theorem 1 are then applied.

Theorem 5: Multistage Shuffle-Exchange networks can be partitioned based on the criteria described in Theorem 1.
Proof: The partitioning of the Cube network in Theorem 1 is based upon selecting cycles of the permutation representation of the Cube interconnection functions. It is independent of the order of the cycles that comprise the permutation. Thus, Theorem 1 applies to the Generalized Cube network. Since multistage Shuffle-Exchange networks are equivalent to the Generalized Cube network, Theorem 1 holds for them also. The choice of interchange boxes to implement cycles is as described for the Cube. □

VII. Conclusions

A formal approach to studying the partitionability of permutation networks was presented. Three types of networks were analyzed. Based on conditions on the mapping m, it was shown which networks can be partitioned and how to choose partitions where they are possible.

Acknowledgements

Discussions with L. J. Siegel, S. D. Smith, and P. T. Mueller, Jr., have been very helpful.

References

[1] G. Barnes, et al., "The Illiac IV computer," IEEE Trans. Comp., Vol. C-17, No. 8, Aug. 1968, pp. 746-757.

[2] K. E. Batcher, "The flip network is STARAN," 1976 Int'l. Conf. Parallel Processing, Aug. 1976, pp. 65-71.

[3] K. E. Batcher, "The multi-dimensional access memory in STARAN," IEEE Trans. Comp., Vol. C-26, No. 2, Feb. 1977, pp. 174-177.

[4] L. H. Bauer, "Implementation of data manipulating functions on the STARAN associative array processor," 1974 Sagamore Comp. Conf. Parallel Processing, Aug. 1974, pp. 209-227.

[5] W. J. Bouknight, et al., "The Illiac IV system," Proc. IEEE, Vol. 60, Apr. 1972, pp. 369-388.

[6] F. Briggs, K. S. Fu, K. Hwang, and J. Patel, "PM4 - a reconfigurable multimicroprocessor system for pattern recognition and image processing," Nat'l. Comp. Conf., June 1979, pp. 255-265.

[7] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comp., Vol. C-23, No. 3, Mar. 1974, pp. 309-318.

[8] M. J. Flynn, "Very high-speed computing systems," Proc. IEEE, Vol. 54, Dec. 1966, pp. 1901-1909.

[9] L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," 1st Annual Symp. Comp. Arch., Dec. 1973, pp. 21-28.

[10] I. N. Hernstein, Topics in Algebra, Xerox College Publishing, 1964.

[11] T. Lang and H. S. Stone, "A shuffle-exchange network with simplified control," IEEE Trans. Comp., Vol. C-25, No. 1, Jan. 1976, pp. 55-65.

[12] D. H. Lawrie, "Access and alignment of data in array processor," IEEE Trans. Comp., Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.

[13] G. J. Lipovski, "On a varistructured array of microprocessors," IEEE Trans. Comp., Vol. C-26, No. 2, Feb. 1977, pp. 125-138.

[14] G. J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor," 1977 Int'l. Conf. Parallel Processing, Aug. 1977, pp. 165-174.

[15] G. J. Nutt, "Microprocessor implementation of a parallel processor," 4th Annual Symp. Comp. Arch., Mar. 1977, pp. 147-152.

[16] M. C. Pease, "The indirect binary n-cube multiprocessor array," IEEE Trans. Comp., Vol. C-26, No. 5, May 1977, pp. 458-473.

[17] H. J. Siegel, "Single instruction stream - multiple data stream machine interconnection network design," 1976 Int'l. Conf. Parallel Processing, Aug. 1976, pp. 273-282.

[18] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," IEEE Trans. Comp., Vol. C-26, No. 2, Feb. 1977, pp. 153-161.

[19] H. J. Siegel, "Preliminary design of a versatile parallel image processing system," 3rd Biennial Conf. on Computing in Indiana, Apr. 1978, pp. 11-25.

[20] H. J. Siegel, "Partitionable SIMD computer system interconnection network universality," 16th Annual Allerton Conf. on Communication, Control, and Computing, Oct. 1978, pp. 586-595.

[21] H. J. Siegel, "Interconnection networks for SIMD machines," Computer, Vol. 12, No. 6, June 1979, pp. 57-65.

[22] H. J. Siegel, F. Kemmerer, and M. Washburn, "Parallel memory system for a partitionable SIMD/MIMD machine," 1979 Int'l. Conf. Parallel Processing, Aug. 1979.

[23] H. J. Siegel, R. J. McMillen, and P. T. Mueller, Jr., "A survey of interconnection methods for reconfigurable parallel processing systems," Nat'l. Comp. Conf., June 1979, pp. 529-542.

[24] H. J. Siegel and P. T. Mueller, Jr., "The organization and language design of microprocessors for an SIMD/MIMD system," 2nd Rocky Mt. Symp. on Microcomputers, Aug. 1978, pp. 311-340.

[25] H. J. Siegel, P. T. Mueller, Jr., H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor systems," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 9-17.

[26] H. J. Siegel, L. J. Siegel, R. J. McMillen, P. T. Mueller, Jr., and S. D. Smith, "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," 1979 IEEE Comp. Soc. Conf. Pattern Recog. and Image Processing, Aug. 1979.

[27] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," 5th Annual Symp. Comp. Arch., Apr. 1978, pp. 223-229.

[28] S. D. Smith and H. J. Siegel, "Recirculating, pipelined, and multistage SIMD interconnection networks," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 206-214.

[29] S. D. Smith and H. J. Siegel, "An emulator network for SIMD machine interconnection networks," 6th Int'l. Symp. Comp. Arch., Apr. 1979, pp. 232-241.

[30] H. S. Stone, "Parallel processing with the perfect shuffle," IEEE Trans. Comp., Vol. C-20, No. 2, Feb. 1971, pp. 153-161.

[31] H. Sullivan, T. R. Bashkow, and K. Klappholz, "A large scale homogeneous, fully distributed parallel machine," 4th Annual Symp. Comp. Arch., Mar. 1977, pp. 105-124.

[32] C. Wu and T. Feng, "Routing techniques for a class of multistage interconnection networks," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 197-205.

LSI IMPLEMENTATION OPTIONS FOR THE SHUFFLE-EXCHANGE NETWORK IN A MICROPROGRAMMED SIMD ARRAY

by

Smil Ruhman
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

A promising approach to the interconnection of SIMD processor arrays combines the shuffle-exchange network [1] with processor address masking [2] of signal reception. But a straightforward implementation would require P bidirectional busses of width W, where P is the number of processors and W is the word length. For 256 processors of 32-bit word length this amounts to 8192 bidirectional lines whose terminations alone would dissipate nearly 2 kilowatts at TTL signal levels. Furthermore, if the network is to interconnect an array of (bit-slice) microprocessors, its hardware requirements may become a significant portion of the total array hardware. These arguments urge consideration of series-parallel approaches to network implementation and their optimization. Smith and Siegel [3] treat this question in considerable depth for three network types including the shuffle-exchange network. However, their treatment uses standard logic, whereas a realistic evaluation of both hardware and speed must be based on LSI implementation. This paper presents and compares a number of implementation options, both recirculating and pipeline, each based on a single repetitive LSI chip. They include network microprogramming hardware and consider the processor interface and network timing.

The recirculating network uses a 2 processor-line × 8 bit chip which contains 153 gates and requires 38 pins. This chip is a universal building block independent of the number of processors or the word-length (full utilization obtained for a multiple of bytes). Operation is speeded up by local recirculation through an internal register. Series-parallel handling of the word (down to a single byte) with proportional reduction in hardware and increase in time is possible without any change in the chip or any external hardware addition. Exchange control is supplied by a 256 word rewritable memory fast enough to keep pace with the recirculation rate.

The pipeline network uses a 4 processor-line chip which contains 197 gates and requires 54 pins. It is independent of word length but does vary with the number of processors, hence is not truly universal. A bypass path is provided to make the pipeline fully equivalent to the recirculating network. Three features are introduced that improve speed or economy, and sometimes both. Thus, two shuffle-exchanges (with optional bypass) are implemented in a single AND-NOR complex to save both hardware and time in the pipeline. Further, the four processor lines per chip are grouped so as to share all data and control lines, thus enhancing the packing factor (gates per pin). The possibility of grouping in this manner is shown to be an inherent property of the shuffle-exchange logic, independent of array size. Thirdly, an AND-NOR complex may be latched up simply by inverting its output and feeding it back through an additional AND-leg, yielding a particularly fast and economical circuit. Pipelining may be implemented to any degree desired but soon reaches a point of diminishing returns. Parallel-serial handling of the word with proportional decrease in time and increase in hardware is possible without any change in the pipeline chip or any external hardware addition.

Expressions are given for the hardware and time requirements as a function of array size and word length, and the performance achievable with Schottky-TTL technology is tabulated for the array size mentioned earlier. The implementation approaches described here may be extended to other interconnection networks except for the grouping property which is specific to the shuffle exchange pipeline and the Omega network.

References

[1] H.S. Stone, "Parallel processing with the perfect shuffle", IEEE Trans. Comput., Vol. C-20, (Feb. 1971), pp. 151-161.

[2] 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.

[3] S.D. Smith, and H.J. Siegel, "Recirculating, pipelined, and multistage SIMD interconnection networks", 1978 International Conference on Parallel Processing (August, 1978).

SCHEDULING PARALLEL PROCESSES WITHOUT A COMMON SCHEDULER[†]

George Holober and Lawrence Snyder
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Abstract: An algorithm which solves the critical section problem for distributed processes is presented. We extend the solution of Lamport [LL76] by continuing to allow processes to access their respective critical sections in any arbitrary user-specified order, but with greatly reduced storage requirements for each process. In addition, we supply a facility for testing the presence of deadlock among processes waiting to enter their critical code. We show our scheme to be tolerant of several malfunctioning processors, and derive an equation relating the probability of total system failure to the probability of many individual failures occurring simultaneously among the processors.

INTRODUCTION

The "critical section" problem, which involves developing a synchronization scheme for a set of processes that enforces solo occupancy of common code, is further complicated when we generalize the circumstances under which the scheme will work or restrict the allowable solutions in some manner. For example, we will assume that the processes execute asynchronously (i.e. nothing is known about one process' rate of execution relative to that of another process nor to the same process' rate of execution at a different time) and that each process must have the same solution as every other process. Another reasonable objective is to avoid possible deadlock resulting from two or more processes waiting for each other.

A number of solutions to the critical section problem have been developed and studied since Dijkstra's initial paper [EWD65]. The results reported in that paper, along with the subsequent refinements outlined by Knuth [DEK], deBruijn [deB], and Eisenberg and McGuire [EM], assumed that concurrent processes would be implemented on multiprogrammed systems. These systems allow different processes to read from or write into any memory location.

Only recently have researchers begun to look at multiprocessor or distributed systems. In such a system, a process may read or write in its local memory and may read from another processor's memory, but may not write into another processor's address space. This restriction prevents the use of global variables, but does yield one important advantage over multiprogrammed computers: if one process fails, the entire systems does not <u>necessarily</u> crash, though system performance will likely be degraded.

One of the first examinations of distributed systems was done by Dijkstra [EWD74]. This paper studied the possibility of processors independently recognizing that they had failed and correcting themselves to some prescribed state. At about the same time Lamport [LL74] presented a solution to Dijkstra's original problem with critical sections that obeyed the constraints of distributed computers. Rivest and Pratt [RP] improved upon this scheme by bounding the values of the variables necessary for inter-process coordination and by preventing a process that continually fails and restarts from deadlocking the system. Further improvements (in terms of smaller ranges of values for variables, greater fairness when sequencing processes for entry into their critical regions, and reduced waiting times for processes before entering their respective regions) were developed by Peterson and Fischer [PM]. Finally, Katseff [HPK] incorporated the best aspects of each of these solutions, including the servicing of processes in the order in which they arrive (FIFO), into one algorithm.

Taking a somewhat different approach, Lamport [LL76] recognized the fact that it is not always desirable to allow processes to enter their critical regions in the same order in which they attempted to access these regions. It is frequently the case that a process may not

---

†This work was funded in part by Office of Naval Research Grant N00014-75-C-0752.

conflict with another process in the sense that they may enter critical regions simultaneously, though both these processes may conflict with a third process. Furthermore, given a set of processes that are currently prevented from entering their critical regions, we may wish to impose some priority on these processes so that when conflicting processes eventually do leave their critical regions, the process having the highest priority, rather than the process that has been waiting the longest, will be the first to access its own region.

In this paper we present a modification of Lamport's system that corrects some drawbacks of both his and Katseff's solutions. In particular:
1) We maintain the basic capabilities of Lamport's design but add a facility to detect the formation of anamalous situations in which a set of processes will deadlock because each process believes another process has priority over it.
2) One variable that is used in Lamport's solution may grow unboundedly large (though in practice this may have little effect). We show how to limit to a finite range the possible values of all variables used for synchronization purposes.
3) Lamport's and Katseff's code requires that each process contain an array, the length of which is equal to the total number of processes. With the recent advances in computer-on-a-chip hardware designs, it is quite likely that future machine architectures will involve huge numbers of communicating processors (capable of running a proportionately large number of processes), each processor possessing a fairly limited amount of memory. Such a hardware scheme is clearly incompatible with Lamport's and Katseff's routines. In our program, each process will need to keep track of only a constant number of other processes.

SYSTEM OVERVIEW

The architecture of the system we will use for our studies is conceptually simple: we have a set of processors, each processor capable of executing at most one process from a set of N processes, and each processor communicating with a subset of the other processors. By "communicate" we mean that one processor may read from another's memory or possibly transmit an interrupt signal (this latter condition is not essential); however, conforming to the definition of a truly distributed system, it may not store into any memory but its own.

We further assume that a processor may fail, though it does so in a somewhat orderly fashion. A read request issued to a process immediately after this process has malfunctioned may return arbitrary values. Eventually only some default value will be returned by read requests to a failing processor, hence it is impossible to accurately examine the memory contents of such a processor. Each processor has the ability to detect its own deviation from normal operating protocol and shut itself down without transmitting spurious interrupts and without writing incorrect information on a disk to which it is linked. The process that had been running on a processor until that processor malfunctioned may be restarted at some predefined point.

As noted in the previous section, the early solutions to the critical section problem require disjoint processes to store into common memory locations. Many of the synchronization schemes that have been proposed to date, such as PV [EWD68], monitors [CARH74], and path expressions when implemented in terms of semaphores [CH, ANH], seem to rely upon a dedicated scheduling routine. Unfortunately, such schemes are incompatible with the desired autonomy of processors. For if the processor in which global data is stored or a dedicated scheduler should fail, the entire system fails. Lamport has explored many aspects of a synchronization scheme that avoids this drawback, though he only touches briefly upon the issue of scheduling. We examine this last issue in greater detail.

The synchronization primitive used by Lamport is an extension of the conditional critical region first proposed by Hoare [CARH71] and later described by Brinch-Hansen [PBH72, PBH73a, PBH73b]. This new primitive takes the form

region <mode> when <condition>
do <critical-section> od

The metavariable <mode> is an expression (typically a constant or a single variable) which evaluates to an element of some arbitrary finite set M (subject to Restriction #1 below); <condition> is a Boolean expression; <critical-section> is an arbitrary length of code (subject to Restriction #3) which comprises the critical region.

It may not be the case that all critical regions will conflict with all other critical regions in the sense that we may desire two processes to be executing their critical regions simultaneously, though either or both of these processes may in turn prevent a third process from entering its region. To formalize this notion, we define a symmetric, time-independent function conflict: M x M --> {true, false}. We then say that two processes conflict if and only if they are both attempting to execute region statements with respective <mode> values of model and mode2, and conflict (model, mode2) = true.

The semantics of the region statement can be stated quite simply: the code in the <critical-section> may not begin execution if a conflicting process has already entered the <critical-section> of a region statement or if <condition> evaluates to false. To prevent certain anomalous situations from arising, we must enforce the following restrictions on our synchronization primitive:

Restriction #1: The value of <mode> must remain constant during the entire execution of the region statement to which it is associated.

Restriction #2: To prevent races between instructions which alter and examine a when <condition>, arguments of the <condition> of one process' region statement which are stored in the memory of another process may only be modified by this second process within a region statement which conflicts with the first region statement.

Note that if the <condition> of a region statement does not depend upon the contents of another process' address space, then this <condition> must always evaluate to true, for if this were not so, then the process would enter the region statement, halt execution until the <condition> became true, thereby preventing assignments to the very variables that can satisfy the <condition> and causing the process to deadlock with itself.

Restriction #3: A region statement may not be one of the instructions in the <critical-section> of another region statement.

One problem frequently associated with conditional critical regions is the difficulty they pose in expressing some synchronization problems. These problems usually have a "scheduling" flavor to them: given a set of conflicting processes that are all competing to enter their respective critical regions, which will take precedence? To remedy this flaw, we define a new function must precede: {1, 2, ..., N} x {1, 2, ..., N} --> {true, false} which may depend upon any information that is available to the system. Therefore given a particular i and j in the set {1, ..., N}, must precede (i, j) need not remain constant over a period of time. (Lamport actually calls this function "should precede"; we will save this term to denote a different function.)

This very general definition of must precede is actually too permissive. The following argument illustrates this point. Suppose that in addition to i and j, the names of the two processes, the function must precede depends on K other sources of information, e.g. which processes are in their critical regions, which processes are awaiting permission to enter their regions and how long they've been in this state, which processes have failed, the values stored in the memories of various processes, etc. It is very unlikely that a process can examine all K+2 arguments and instantly determine the value of must precede (i, j). Rather, the process would probably scan one or two arguments at a time and combine this information with previously computed results to obtain a partial answer. This procedure would repeat this until all arguments have been examined and must precede (i, j) has been fully determined. Consider the case in which a process is scanning the xth argument of must precede (x is in the interval [2, ..., K+2]) when another process alters the value of the yth argument (y is in the interval [1, ..., x-1]). The first process will never rescan the yth argument, so the value it finally obtains for must precede (i, j) will be incorrect. To overcome this difficulty, Lamport assumes that must precede is strongly constant, meaning that its value will not change when we are in the midst of computing it. This convention simplifies matters greatly (and in fact probably does not pose a severe restriction), so we will adopt it as well.

The interpretation of the must precede function is self-evident, but it is important to point out that it has meaning only on those processes that are simultaneously waiting to enter their critical regions and that conflict with one another. Putting together the mechanisms we have described so far, it becomes clear that a process i can enter the <critical-section> of a region statement only if the following three conditions are satisfied:

Condition #1: All processes that conflict with process i are executing code outside of their critical regions.

Condition #2: The when <condition> evaluates to true.

Condition #3: For all processes j that are presently executing region statements but have not yet entered their <critical-section>'s, and that conflict with process i

$$\text{must precede}\ (i, j) = \begin{cases} \text{true if } j \text{ has been waiting longer than } i \\ \text{false if } i \text{ has been waiting longer than } j \end{cases}$$

In other words, of all the processes that do not conflict with another process that is in a <critical-section> (#1), that have true when <condition>'s (#2), and that have no predecessors (in the sense that there is no conflicting process j for which must precede (j, i) holds true), time of arrival is the final arbiter (#3). We impose one more condition on our system that guarantees that no process can be locked out of a <critical-section> once it has begun executing a region statement:

Condition #4: Assuming no further processes encounter region statements, a process satisfying Conditions 1 - 3 will enter its <critical-section> after a finite delay.

This condition will follow if we assume that all processes make progress executing their instructions (though our previous assumption of asynchronous operation may make this progress very slow) and if a permanent deadlock situation does not exist among the processes that are waiting to enter their critical regions.

THE ALGORITHM

In the last section we briefly mentioned the possibility of two or more processes causing a deadlock while waiting to enter critical regions. To see how this might happen, consider the most trivial case for the moment. Suppose that process i has just encountered the statement

region model when true do <anything> od

where conflict (model, model) = true and must precede (i, i) = true. Using our rules for selecting processes to enter their <critical-section>'s, process i must wait for itself to leave its <critical-section> before it can enter it, a clear impossibility. A deadlock is present, and Condition #4 is violated (unless must precede (i, i) changes to false at some future point). Although this may seem like a contrived example, and therefore not a very convincing justification for our attempts to determine the existence of deadlocks, these deadlocks can arise in far more subtle ways. The following theorem characterizes the situations in which a deadlock will be present.

Cycle Theorem: A deadlock will exist among the processes that are awaiting entrance to their critical regions if and only if there exists a subset {P(0), P(1), ..., P(L)} of these processes which form a "cycle" in the sense that for all i in the set {0, 1, ..., L}

(1) P(i) is in a region statement with <mode> value M(i), and
(2) the functions must precede (P(i), P(i+1 mod L)) and conflict (M(i), M(i+1 mod L)) evaluate to true.

Proof: The "if" part follows immediately from our definitions. The "only if" part stems from the following fact: if we trace backwards over the must precede and conflict relations on a finite set of processes, we must eventually either return to a process which has already been visited (thereby showing the presence of a cycle), or else we will arrive at a process i for which there are no processes j such that must precede (j, i) = true and processes i and j are in conficting region statements. In this latter case there is no cycle, but process i can enter its <critical-section> and there is no deadlock.

We must establish several ground rules for manipulating faulty processes so that we will have a common convention with which to work. In addition to assuming that a failing process does not behave "maliciously," e.g. it sends off spurious interrupts to the remaining operational processes, we further assume that we have some reliable mechanism for determining whether a particular process has failed. A process can be thought of as emitting a "carrier signal"; when the signal dies, the process has failed.

Processes which fail while on the queue remain there until some external device repairs them so that they can eventually enter their critical sections. We adopt this convention on the basis of its being the most general scheme for dealing with the failure of enqueued processes. "Most general," in the sense used here, means the ability of this scheme to simulate any other scheme. This generality arises from the flexibility of scheduling provided by the must precede. For example, we could easily alter the value of must precede to effectively ignore the presence of a failed process on the queue. Of course, we are assuming that in such a situation the values of the arguments to must precede can be determined despite the loss of accessibility to data that has been stored by malfunctioning processes.

Processes which fail while executing their critical sections can block many other processes with which they conflict, thereby causing serious degradation in system performance. We will assume such processes are to be removed from their critical sections by the external mechanism before being repaired and returned to normal operation. Note that once in its critical section, a process is beyond the effects of the must precede function. Thus we do not have the run-time flexibility we had when dealing with the failure of enqueued processes, and we appear to be quite rigidly bound by whatever scheme we choose for servicing processes which fail in the midst of their critical code.

It would be unreasonable to assume that a process can be made to stop, perform some desired operation, and resume unless it is under our control. Thus we cannot expect the cooperation of processes which are executing their critical sections or non-critical sections. The only times a process does come under our control so that it can be made to perform synchronization tasks is when it is waiting on the queue and leaving its critical section.

Because concurrent computations are inherently difficult to understand (and rigorous mathematical proofs of their correctness are even more difficult to comprehend), we will break down the development of the algorithm into three steps. In the first version, we deal with a sequential program that will temporarily serve as our scheduler and that is easy to comprehend. In the next version, we transform the sequential program into a parallel program. At this point we are halfway to our target program: control of instruction sequencing has been removed from the central scheduler and is now managed by the individual processes, but shared memory is still utilized. In the final version, we convert this parallel program into fully distributed code by passing out the common storage locations among the component processes. (For notational convenience, we say i ==> j if conflict(i,j) = true and must precede(i,j) = true.)

There are several advantages to treating the development of a distributed program as code synthesis beginning with a simple statement of the solution rather than as a programming task followed by a verification phase. Not only are proofs of programs (especially parallel programs) difficult to devise, they are almost as difficult to understand due to their ad hoc nature. Even if the rules of verification could be formalized, mechanical verifiers invariably suffer from extremely poor efficiency, as the task they are meant to perform is almost surely intractible. Synthesizing code by means of simple transformations need not require a major effort, just as the compilation of high level sequential languages into machine level code can be accomplished efficiently and in a straightfoward manner (presumably because this is a well understood task). Furthermore, programming techniques demanding verification suffer because it is difficult to build each new program upon old programs. Instead, many papers dealing with parallel processes seem to begin afresh, defining low level features, expanding upon them, and finally verifying what has been developed. On the other hand, synthesis begins with a small collection of requisite parameters, and modifies these to mesh with the low level features of the system in a top-down fashion.

Version 1

In this initial version, we are dealing with a very simple sequential program. The scheduler exists as a separate routine (which we will presently assume is immune to failure), and governs the operation of all other processes. A macroscopic view of the operation of the scheduler is given by the flowchart in Figure 1.

There is one very important issue that we have avoided so far: how do we deal with two or more processes that simultaneously begin execution of region statements? Or in terms of our system, how do we treat processes that signal their intention to interact with the scheduler when the scheduler is already busy servicing some other process? Before proceeding with our description of the algorithm, we must put this issue to rest by establishing a method for determining the relative ordering of such processes.

Optimally, we would like the scheduling routine to service processes in the same chronological order these processes signal the scheduler. One solution to this problem, performed at the implementation level of the system, would be to let each process dispatch an interrupt when it wants the attention of the scheduler. The scheduler, in turn, serves as our interrupt handler, and it disables all other interrupts until the process requesting attention is fully serviced. In this solution, we have pushed the problem back onto the hardware mechanism.

Another possible solution might be to let each process maintain a timer while it is awaiting the attention of the scheduler. The timer could be a mechanical clock, or we could let the program idle in a loop. On each iteration of the loop, a variable TIMER would be incremented by one. When the scheduler becomes available, it picks the process whose timer indicates the longest wait. This solution suffers several drawbacks. Depending upon the response time of the scheduler, the value stored in the timer could grow unboundedly large. Even worse, we are dealing with an asynchronous system, so the timer may not reflect a true measure of the waiting time (though if we assume a finite bound on the speed of one process relative to another, we are guaranteed that all processes will eventually command the attention of the scheduler).

In both of these solutions, we have relied upon an external agent to assume the burden of the problem. Is it possible to avoid the use of an external device entirely? We maintain the answer is no. In any realistic system, there will be a lower bound on the length of time that can be measured. If two events occur within this time span, we are faced with the problem of taking these seemingly simultaneous events and determining which of them actually came first. What choice do we have, but to rely upon an external arbiter to resolve this dilemma? Hopefully, such an arbiter would either be capable of measuring time on a more refined basis, or would have some other information, unknown to us, for ordering events.

In our system, the lower bound for measuring time is the maximum response time of the scheduler. What we have done, in effect, is to treat time as a resource, and to insist that mutual exclusion be maintained on this resource at those points in time when a process is interacting with the scheduler. We note in passing that many systems that have been described in the literature finesse the issue of simultaneity by assuming the availability of indivisible or atomic operations.

Version 2

Continuing with the synthesis of our final program, we now "snip" the control mechanism to eliminate the explicit scheduler. The scheduler, which is still failure-free, can instead be thought of as existing only in an conceptual form, transmitting instructions to the individual processes. By this we mean that the scheduler issues an instruction which all the processes compete for and execute. The execution of such an instruction is finished when all the processes have completed their portions of the code, or have failed. The result is a parallel program which utilizes shared memory.

In reality, each process will have a copy of the scheduler. These individual copies will operate in asynchronous parallel manner by using a "mutual handshake" concept. When one component of the scheduler finishes some instruction, it polls the other components to determine if they have finished their respective instructions, and waits until they have done so before proceeding with the next instruction. Setting a flag at the beginning and end of each instuction would be a simple mechanism for determining whether or not each process had finished its scheduling instuction. Figure 2 illustrates a sample instruction for enqueuing a process that begins executing a region statement.

We also begin to decompose the queue at this point. Instead of having one process, the common scheduler, store the configuration of the enqueued processes, we now let each component process remember its location within the queue. The processes on the queue will be strung together in a linear sequence by a set of multiple pointers. Each process contains s-element arrays BEFORE and AFTER. The value of BEFORE[i] is the identifier of the process which arrived on the queue i arrivals before the process in which this array is stored. AFTER has the complementary meaning. We will sometimes subscript a variable with index i to emphasize that this variable is local to process i. Figure 3 provides a global view of the structure of these arrays.

The purpose of the multiple links between processes is twofold. First, should a process fail, we can still determine which processes follow it or precede it on the queue simply by following an alternative link around the malfunctioning process. And second, the redundancy of these pointers can be useful for detecting the failure of processes. Many previous solutions to the critical section problem assume that when a process fails, it turns on some sort of signal that beacons its failure to the remaining functional processes, so that the operation of these processes will not be affected. Clearly this is not an entirely realistic assumption. We note that if $\text{BEFORE}_i[j] = k$ and $\text{AFTER}_k[j]$ does not equal i, then it is quite likely that either or both of processes i and k have failed. Further tests involving comparisons with links from other processes could aid in pinpointing the exact identity of the malfunctioning process.

Version 3

In the third and final version of our routines, we are ready to eliminate the scheduler completely and to distribute both the memory and control mechanism to the individual processes. Each process has a copy of the scheduler and can be thought of as issuing instructions to itself. The processes then operate in conjuction to determine which instructions should be executed and when.

Possibly the first feature of version 2 that strikes the reader is that memory management has been almost entirely divided among the constituent processes. This division of memory management has been one of our prime objectives from the beginning, for in order to conform to the definition of a distributed system and reap the fault-tolerant capabilities such systems have to offer, we must insure that individual processes perform write operations only on their own local memories. An examination of the instructions of version 2 reveals that all of the instuctions cause process i to alter only the contents of its own memory.

We are not quite finished, however, due to the memory requirements that would result from a naive implementation of the instructions. A restriction we have placed on our system, along with the need for a distributed control mechanism, is that each process use a limited amount of memory. In other words, each process should have an address space whose size is independent of n, the number of processes. Nearly all of the instructions obey this property, the sole exception being the deadlock-test operation.

The Cycle Theorem tells us that testing for deadlock is equivalent to testing for the presence of cycles in the ==> relation. Phrasing this another way, a deadlock will exist if and only if some process p obeys the relationship $p \overset{*}{==>} p$, where $\overset{*}{==>}$ is the (non-reflexive) transitive closure of ==>. A deterministic algorithm for computing the transitive closure on n objects will undoubtedly proceed by following the ==> relation from one object to the next and backtracking where necessary. To prevent some sequence a ==> b ==> c ==> ... ==> z of processes from being examined repeatedly, it appears necessary to keep a record of the processes along such chains that have already been scanned and need not be re-examined. The number of markers needed to maintain this record yields O(n) space complexity in the worst case. Linear space complexity is unfortunate from our point of view, for even though an amount of memory proportional to n will be needed to test for deadlock, no single process can directly utilize that much space. Thus each of the n processes must devote a constant amount of memory toward executing the deadlock-test instruction.

To see if process i has caused a deadlock, process i turns a flag CYCLE to ON. Each process k other than i checks to see if there is a process j such that CYCLE_j = ON and j ==> k. If so, process k sets CYCLE_k to ON, establishing one more link in the potential deadlock cycle. Eventually, either no more processes can set their values of CYCLE to ON (in which case there can be no deadlock) and the test ends, or some process k such that k ==> i sets CYCLE_k to ON, and process i notes the completed cycle and announces the presence of a deadlock. This deadlock check algorithm is outlined by the flowchart in Figure 4.

An analysis of the requirements for the deadlock-test instruction shows that it can fail in either of two circumstances: more than the designated number of consecutive processes fail simultaneously (in which case the remaining operational processes will not be able to assume responsibility for all of their malfunctioning counterparts), or all the processes on the queue fail simultaneously. However, neither of these conditions is too important. We have ruled out the first case (or at least know the probability of its happening). In the second case, there are no operational processes on the queue, so that none could possibly enter their critical sections, and the existence of a deadlock is therefore inconsequential.

Note that we have been very liberal in allowing the user to risk potential deadlock situations. As a result, our deadlock detection routine incurs a great deal of run-time expense in the form of process cross-talk. One possible alternative to the scheme presented here is somewhat more conservative in nature. Instead of permitting the possibility of deadlock at compile-time and checking for its presence at run-time, we disallow definitions of *must precede* that would allow a deadlock to develop when certain combinations of processes are enqueued. This compile-time check is simple: we assume all processes are on the queue, and use our deadlock tester to see if a cycle is present. If no cycle exists under these circumstances, no cycle can ever exist, and the system is guaranteed to be deadlock-free. Otherwise, the user is informed that deadlock may develop in the future. Thus we need to test for deadlock only when *must precede* changes, and not whenever a new process enters the queue.

FAILURE ANALYSIS

One drawback of our system is that under extreme circumstances the entire system may fail. Such a situation would arise if groups of operational enqueued processes were separated by so many failed processes that the former could not use the information contained in BEFORE and AFTER to derive the relative ordering of the groups. If each of these arrays has s elements, at least 2s consective processes on the queue must be down at the same time for the system to collapse. The probability of such a failure occurring is given by the formula

$$\sum_{i=0}^{n-s-1}\left(1+\left[\sum_{j=1}^{\lfloor i/s+1\rfloor}\binom{i-js}{j}\,[(p-1)p^{s}]^{j}\right]\right)$$

$$\cdot(1-p)\cdot p^{s}\cdot(1-p^{n-i-s-1})$$

where p is the probability that an individual process will be nonoperational at any particular moment. By making s as large as we desire, this probability becomes arbitrarily small.

CONCLUSIONS

We have demonstrated a solution to the critical section problem for distributed systems that satisfies the stated design requirements:

1) It permits arbitrary processes to conflict/not confict depending upon the particular critical regions they are attempting to enter.
2) It allows granting access to critical regions based upon an arbitrary scheduling function. The order of requests for entering the critical regions is maintained and can be used for scheduling purposes.
3) All variables involved in the synchronization process assume values from a finite range.
4) All processes need to store only a small amount of data to maintain the synchronization scheme. By "small" we mean an amount that is independent of of the number of processors in the system.
5) The failure and subsequent restart of any individual process or even a reasonably small subset of processes will not cause a widespread system malfunction.
6) The creation of a cycle of *must-precede*-related processes and the resulting deadlock can be detected, though we do not specify what course of action should be taken from that point on.

Most importantly, we have demonstrated a technique for transforming an easy-to-understand sequential program into a distributed program. Each step of the transformation is reasonably straightforward. We have attempted to find natural lines along which to decompose our program. With a greater effort, we might hope to formalize the transformation process, possibly to the point where it could be mechanized.

Our results point to several other areas that should be examined. For example, we have described one notion of deadlock, when in fact there exists another rather obvious form of deadlock with which we have not dealt. If a process is waiting on the queue for its *when* condition to turn true, but no other conflicting process has yet arrived which can alter this *when* condition, then this process, along with all enqueued conflicting processes which it *must-precede*, will sit idle. Determining whether a process will alter any variables and thereby change *when* conditions is recursively undecidable, so it may not be feasible to build a mechanism to accurately detect or correct this type of deadlock. Is this an important consideration among real parallel routines? If so, will heuristic deadlock testers suffice to make this a negligible problem?

Furthermore, we have been able to develop a reasonably simple algorithm by passing the details of scheduling, in the form of *conflict* and *must-precede* relations, to the user. While this gives the user a great deal of flexibility, this flexibility must be accompanied by a certain measure of responsibility. Is all this flexibility necessary? Or must the user pay for it in terms of the extreme care taken to program scheduling relations? And are there techniques he might employ developing these relations that would allow the synchronization protocols to execute with greater efficiency?

Another area for further study centers around the implementation of the queue. Our multiply-linked list is a "stretched-out" data structure, in the sense that it does not require a large set of malfunctioning processes to form a cut set and thereby cause the system to fail. Are there alternative data structures which require a larger cut set to separate and therefore present a lower probability of system failure? And exactly what would be the tradeoff between the improved reliability of these structures and the increased complexity and reduced efficiency of the code for the critical section problem?

Acknowledgements: Prof. Mark R. Brown provided some useful suggestions related to the failure analysis. Prof. Alan J. Perlis made several very insightful comments upon the nature and limitations of parallel systems.

REFERENCES

[PBH72] Per Brinch-Hansen, "A Comparison of Two Synchronizing Concepts," Acta Informatica, Vol. 3, Fasc. 1, 1972, pp. 190 - 199.

[PBH73a] Per Brinch-Hansen, "Concurrent Programming Concepts," ACM Computing Surveys, Vol. 6, No. 4, Dec. 1973, pp. 223 - 245.

[PBH73b] Per Brinch-Hansen, Operating System Principles, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[deB] N.G. deBruijn, "Additional Comments on a Problem in Concurrent Programming Control," Comm. ACM, Vol. 10, No. 3, March 1967, pp. 137 - 138.

[CH] R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," Operating Systems, Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, 1974, pp. 89 - 102.

[EWD65] E.W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," Comm. ACM, Vol. 8, No. 9, Sept. 1965, p. 569.

[EWD74] E.W. Dijkstra, "Self-stabilizing Systems in Spite of Distributed Control," Comm. ACM, Vol. 17, No. 11, Nov. 1974, pp. 643 - 644.

[EM] M.A. Eisenberg and M.R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Problem," Comm. ACM, Vol. 15, No. 11, Nov. 1972, p. 999.

[ANH] A.N. Habermann, "Path Expressions," Technical Report, Dept. of Computer Science, Carnegie-Mellon University, June 1975.

[CARH71] C.A.R. Hoare, "Towards a Theory of Parallel Programming," International Seminar on Operating System Techniques, Belfast, Northern Ireland, Aug. - Sept. 1971. Also in Operating Systems Techniques, Ed. by C.A.R. Hoare and R.H. Perrott, Academic Press, 1972, pp. 61 - 71.

[CARH74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, No. 10, Oct. 1974, pp. 549 - 557.

[HPK] H.P. Katseff, "A Solution to the Critical Section Problem with a Totally Wait-free FIFO Doorway," Internal Memorandum, Computer Science Division, University of California, Berkley. Extended abstract in "A New Solution to the Critical Section Problem," Proc. of the Tenth Annual ACM Symp. on Theory of Computing, May 1978, pp. 86 - 88.

[DEK] D.E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control," Comm. ACM, Vol. 9, No. 5, May 1966, pp. 321 - 322.

[LL74] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," Comm. ACM, Vol. 17, No. 8, August 1974, pp. 453 - 455.

[LL76] L. Lamport, "The Synchronization of Independent Processes," Acta Informatica, Vol. 7, Fasc. 1, 1976, pp. 15 - 34.

[PF] G.L. Peterson and M.J. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," Proc. of the Ninth Annual ACM Symp. on Theory of Computing, May 1977, pp. 91 - 97.

[RP] R.L. Rivest and V.R. Pratt, "The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report," Proc. of the 17th Annual Symp. on Foundations of Computer Science, Oct. 1976, pp. 1 - 8.

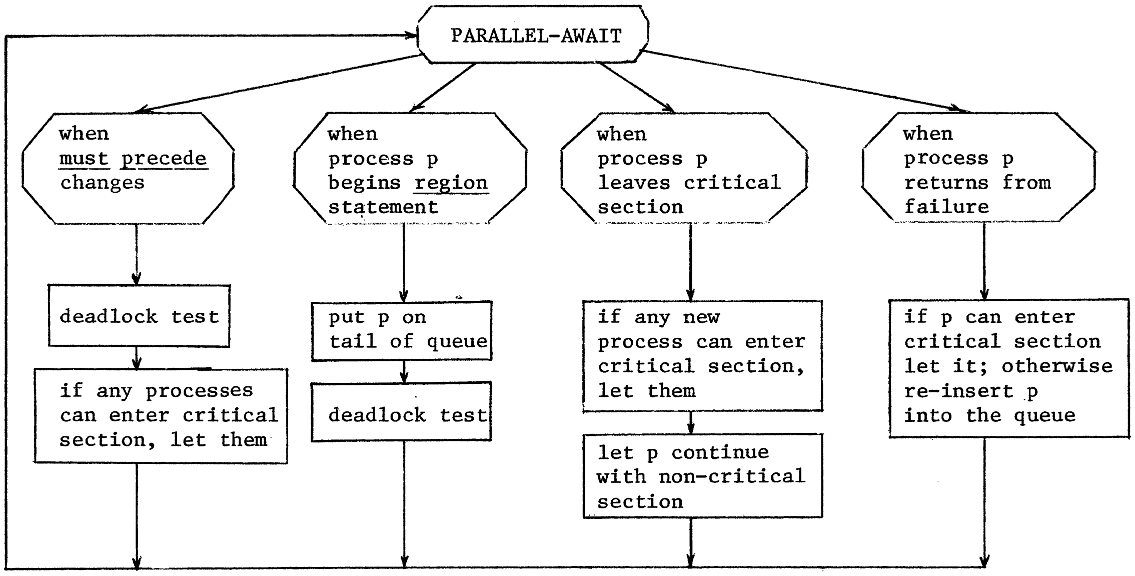

PARALLEL-AWAIT
when must precede changes
deadlock test
if any processes can enter critical section, let them
when process p begins region statement
put p on tail of queue
deadlock test
when process p leaves critical section
if any new process can enter critical section, let them
let p continue with non-critical section
when process p returns from failure
if p can enter critical section let it; otherwise re-insert p into the queue

Figure 1: Version 1 Common Scheduler

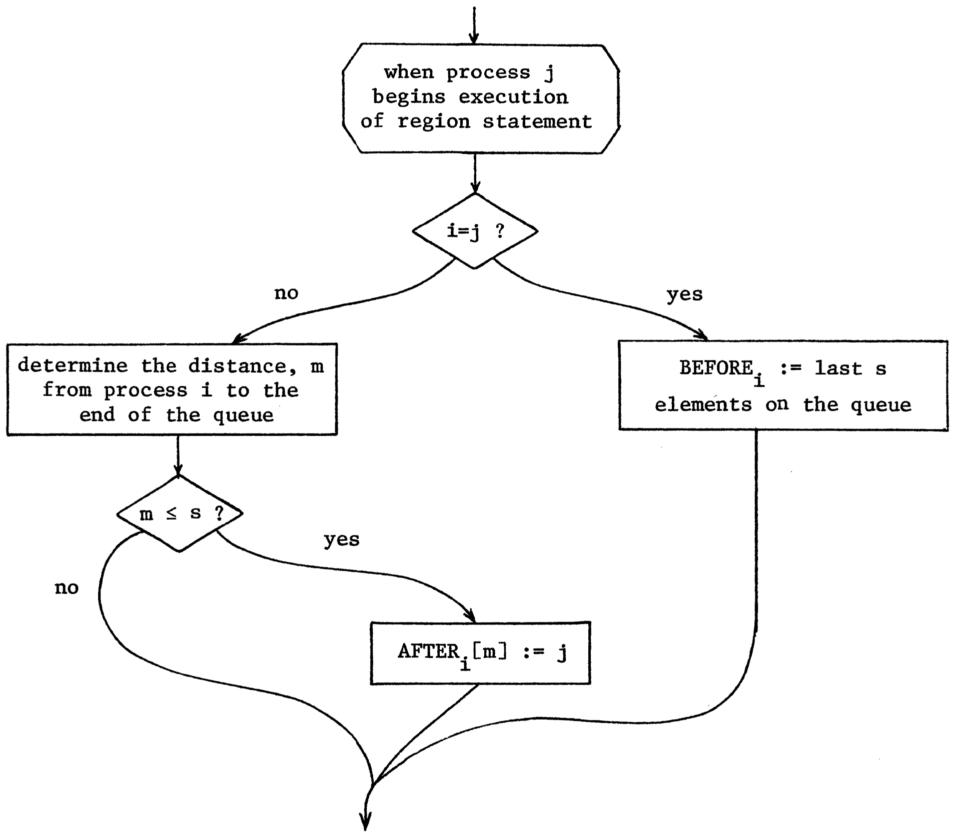

when process j begins execution of region statement
i=j ?
no
yes
determine the distance, m from process i to the end of the queue
BEFORE$_i$:= last s elements on the queue
m ≤ s ?
no
yes
AFTER$_i$[m] := j

Figure 2: Version 2 Instruction for Process i : Enqueue Process j.

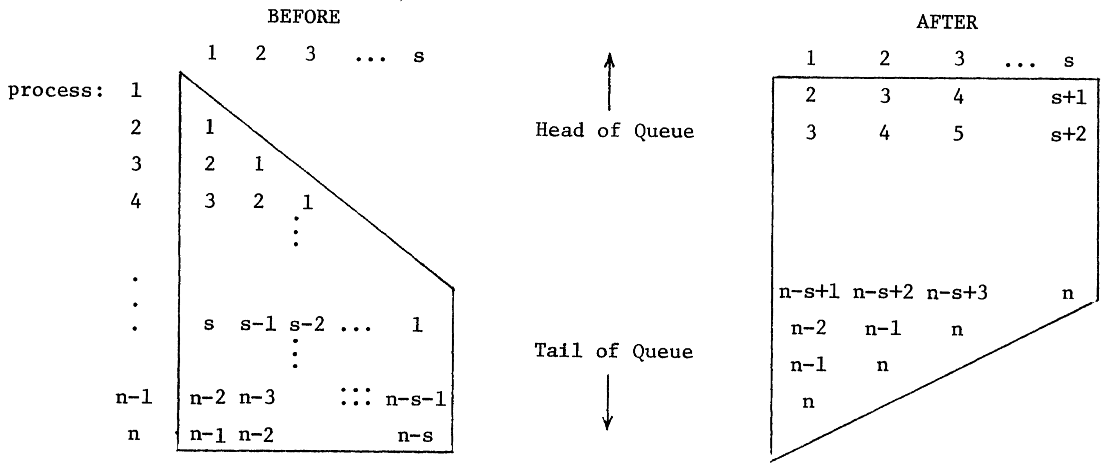


Figure 3: Structure of the BEFORE and AFTER arrays.

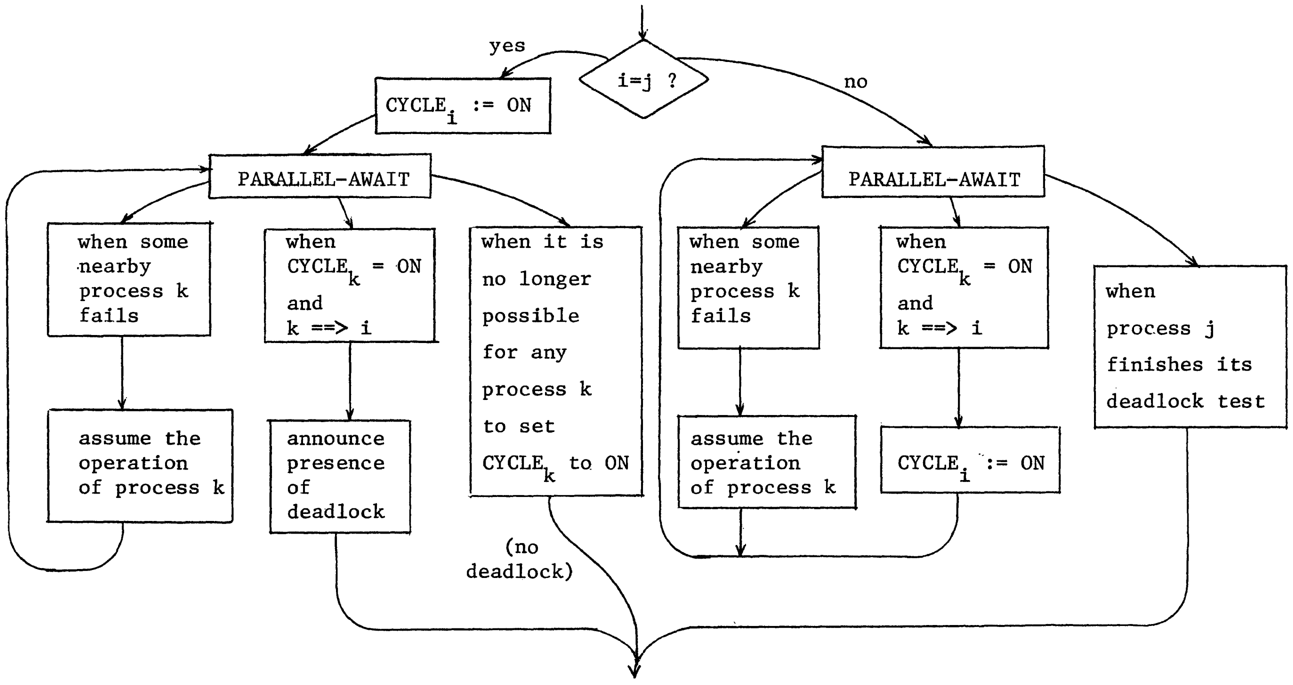

i=j ?
yes
no
CYCLE$_i$:= ON
PARALLEL-AWAIT
when some nearby process k fails
when CYCLE$_k$ = ON and k ==> i
when it is no longer possible for any process k to set CYCLE$_k$ to ON
assume the operation of process k
announce presence of deadlock
(no deadlock)
PARALLEL-AWAIT
when some nearby process k fails
when CYCLE$_k$ = ON and k ==> i
when process j finishes its deadlock test
assume the operation of process k
CYCLE$_i$:= ON

Figure 4: Version 3 instruction for process i to determine if process j is caught in a deadlock.

MODELLING OF CONFLICTS, PRIORITY HIERARCHIES AND REENTRANCY IN CONCURRENT SYNCHRONIZATION STRUCTURES

K.B. Irani
Department of Electrical and Computer Engineering
The University of Michigan
Ann Arbor, Michigan 48109

C.R. Zervos
Catedra Calculatoare
Facultatea de Automatica
Polytechnic Institute of Bucharest
Spl. Independentei no. 313
Bucharest, Romania.

Abstract -- A new variant of the Generalized Petri Nets called the C-Colored Petri Model is presented. The elegance of this model for representing certain concurrent systems is exhibited by examples. Theoretical results concerning the representation power of this model are also presented.

1. Introduction

This paper presents a new variant of the Generalized Petri Nets ([6]) called the C-Colored Petri Model (C-CPM). The C-CPM is a model of systems exhibiting concurrency particularly well suited for the representation of:

1. process synchronization structures involving dynamic priority hierarchies among processes;
2. conflict resolution among competing processes;
3. reentrant coordination structures.

Section 2 formally defines the C-CPM. Sections 3 and 4 present a few modelling examples using the C-CPM. The selected modelling examples emphasize the naturalness and compactness of the representation offered by the C-CPM for coordination situations of the types mentioned above. Section 4 also presents some theoretical results concerning the representation power of the C-CPM. In particular, the modelling capabilities of the C-CPM are compared with the modelling capabilities of other variants of the Generalized Petri Nets such as the Priority Petri Model ([5], [11]), the Extended Petri Model ([1], [11]), the Coordination Petri Model ([8], [11]) and the Petri Net Model with switches, disjunctive logic and token absorbers ([2], [11]). Finally, Section 5 draws a few conclusions regarding the usefulness of the C-CPM.

2. Formal Definition of the C-CPM

Before defining the C-CPM we first give a few basic notations and definitions.

Definition 2.1. A Generalized Petri Net (GPN) is a system N=(T,P,I,O) where:

1. T is a finite set whose elements are called transitions; let $T = \{t_1,\dots,t_n\}$.
2. P is a finite set whose elements are called places; let $P = \{P_{n+1},\dots,P_{n+m}\}$ where $T \cap P = \emptyset$.
3. $I:P\times T \to Z^\circ$ is a function called the input incidence function. $Z^\circ = \{0,1,2,3,\dots\}$ denotes the set of nonnegative integers.
4. $O:T\times P \to Z^\circ$ is a function called the output incidence function.

A GPN is represented graphically by a bipartite directed graph, where:

1. transitions are represented by bars,
2. places are represented by circles, and
3. bars and circles are connected by directed arcs. The set of arcs is denoted by A. For every pair $(p_j,t_k) \in P\times T$ there are exactly $I(p_j,t_k)$ arcs directed from p_j to t_k. If $I(p_j,t_k) > 0$ then p_j is called an input place of the transition t_k and the arcs directed from p_j to t_k are denoted by a^i_{jk}, $i=1,\dots, I(p_j,t_k)$. Each such arc a^i_{jk} is called an input arc of the transition t_k (from the input place p_j). Similarly, for every pair $(t_k,p_j) \in T\times P$ there are exactly $O(t_k,p_j)$ arcs directed from t_k to p_j. If $O(t_k,p_j) > 0$ then p_j is called an output place of the transition t_k and the arcs directed from t_k to p_j are denoted by a^i_{kj}, $i=1,\dots,O(t_k,p_j)$. Each such arc a^i_{kj} is called an output arc of the transition t_k (to the output place p_j).

In what follows, by a "set of colors", we shall mean a lattice $C=(X,\leq)$, where X is a finite set. An element of the set X is called a color.

Definition 2.2. Let N=(T,P,I,O) be a GPN and let $C=(X,\leq)$ be a set of colors associated with N. A color marking of N is a function $CM:P \to \langle X\rangle^*$, where $\langle X\rangle^*$ denotes the bag closure of the set X, i.e., the set of all bags over the set X.

For a detailed discussion concerning the concept of a bag the reader is referred to [3,4,10]. Pertinent notations and definitions employed in this paper are compatible with [11].

For each $p \in P$, $CM(p)$ is the color bag of the place p (in the color marking CM). For each element of the bag $CM(p)$ there exists a token of that color in the place p for the color marking CM. A token is represented graphically by drawing a circle inside the corresponding place. The circle corresponding to each token is filled with the respective color (see Figure 2.1).

Definition 2.3. A C-Colored Petri Net (C-CPN) is a system $CN=(N,C,F)$ where:

1. $N = (T,P,I,O)$ is a GPN.
2. $C = (X,\leq)$ is a set of colors associated with N.
3. $F : A \rightarrow X$ is a function.

For each arc $a^i_{jk} \in A$, $F(a^i_{jk})$, also denoted by c^i_{jk}, is a color of the set X. If a^i_{jk} is an input arc of some transition t_k then c^i_{jk} is called the color threshold of the input arc a^i_{jk}. On the other hand, if a^i_{jk} is an output arc of some transition t_j then c^i_{jk} is called the output color of the output arc a^i_{jk}. Let us now explain the roles of the color thresholds and of the output colors. Suppose CN is a C-CPN in some color marking CM^r. Let t_k be a transition of N and let p_j be an input place of t_k. Let us consider an input arc a^i_{jk}, $1 \leq i \leq I(p_j,t_k)$.

Definition 2.4. A color c of a token present in the place p_j in the color marking CM^r is a candidate enabling color of the transition t_k on the input arc a^i_{jk} in the color marking CM^r if $c \geq c^i_{jk}$, and there exists no other token of color c' in p_j in the color marking CM^r such that $c > c' \geq c^i_{jk}$.

We note that there may exist tokens of colors c and c' in the color bag $CM^r(p_j)$ such that c and c' are incomparable under the ordering relation $\leq$ and each of the two colors is a candidate enabling color of t_k on the input arc a^i_{jk} in the color marking CM^r. An enabling color of the transition t_k on the input arc a^i_{jk} in the color marking CM^r is a color arbitrarily selected from the set of candidate enabling colors for that transition t_k on the input arc a^i_{jk} in the color marking CM^r. The input arc a^i_{jk} is said to claim the enabling token of color c in the color marking CM^r.

On the other hand, if there exists no color $c \geq c^i_{jk}$ in the color bag $CM^r(p_j)$ then there exists simply no candidate enabling color, and hence no enabling color, of t_k on the input arc a^i_{jk} in the color marking CM^r. Thus, the color c^i_{jk} actually sets a color threshold for the selection of the enabling color on the corresponding input arc a^i_{jk}.

The bag of enabling colors of t_k from the input place p_j in the color marking CM^r is:

$$\theta^r_{jk} = \langle c \mid c \text{ is the enabling color of } t_k \text{ on the input arc } a^i_{jk} \text{ in the color marking } CM^r,\ 1 \leq i \leq I(p_j,t_k)\rangle .$$

The bag of enabling colors of the transition t_k in the color marking CM^r is then:

$$\theta^r_k = \bigcup_{\substack{p_j \in P \\ I(p_j,t_k) > 0}} \theta^r_{jk}$$

We can now define the notion of an enabled transition.

Definition 2.5. A transition t_k is enabled in the color marking CM^r is following conditions hold:

1. There is an enabling color in the color marking CM^r for each input arc of t_k.
2. $\theta^r_{jk} \subseteq CM^r(p_j)$ for each input place p_j of t_k.

In other words, the transition t_k is enabled in the color marking CM^r if each input arc of t_k can claim in CM^r a distinct enabling token.

A transition enabled in some color marking may be selected to fire in that color marking. The rule under which this selection is made will be given later in this section. We shall first describe the effect produced by the firing of a transition.

Definition 2.6. A transition t_k, enabled in some color marking CM^r, fires in the color marking CM^r by performing the following operations:

1. For each input arc a^i_{jk}, $i=1,\ldots,I(p_j,t_k)$, for all input places p_j of t_k, a token of color c is removed from the corresponding input place

p_j, where c is the enabling color of t_k on the arc a^i_{jk} in the color marking CM^r.

2. For each output arc a^i_{ks}, $i=1,\ldots,O(t_k,p_s)$, for all output places p_s of t_k, a token of color $F(a^i_{ks})$, also denoted by C^i_{ks}, is placed in the corresponding output place p_s.

Let us now present the rule under which an enabled transition is firable.

Let us define for each $p_j \in P$ the set:

$$T(p_j) = \{t_k | t_k \in T \text{ and } I(p_j,t_k) > 0\}$$

We shall denote by E^r the set of transitions enabled in the color marking CM^r.

Definition 2.7. There exists a conflict in the color marking CM^r at the place p_j for the color c if and only if:

$$\sum_{t_k \in E^r \cap T(p_j)} \#(c,\theta^r_{jk}) > \#(c,CM^r(p_j))$$

Here $\#(c,\theta^r_{jk})$ and $\#(c,CM^r(p_j))$ denote the number of occurrences of the color c in the bags θ^r_{jk} and $CM^r(p_j)$, respectively. In other words, a conflict for the color c has occurred in the color marking CM^r at the place p_j if the number of tokens of color c claimed from p_j by the transitions enabled in CM^r is larger than the number of tokens of this color present in p_j in the color marking CM^r. Note that there may exist conflicts at the same place for several distinct colors.

Two transitions t_k and t_s are said to conflict in the color marking CM^r (denoted by $t_k o t_s$) if following conditions are true:

1. $t_k \in E^r$ and $t_s \in E^r$.
2. There exists at least one place p_j such that $I(p_j,t_k) > 0$ and $I(p_j,t_s) > 0$ and there is a conflict at p_j in the color marking CM^r for some color c, where $c \in \mathcal{D}(\theta^r_{jk}) \cap \mathcal{D}(\theta^r_{js})$. $\mathcal{D}(\theta^r_{jk})$ and $\mathcal{D}(\theta^r_{js})$ denote the domains of the bags θ^r_{jk} and θ^r_{js}, respectively. In addition we impose that $t_k o t_k$ for each $t_k \in E^r$.

The conflict relation defined above can be extended as follows. Let CONF be the relation on E^r such that for any $t_k \in E^r$ and $t_m \in E^r$ $(t_k,t_m) \in$ CONF if there exists a sequence of transitions.

$$t_, = t_{q1},\ldots,t_{qn} = t_m \text{ and } t_{qj} \text{ o } t_{qj+1}$$

for $j=1,\ldots,n-1$.

We have the following obvious proposition:

Proposition 2.1. CONF is an equivalence relation on E^r. An equivalence class of the relation CONF is called a conflict cluster. The quotient set of E^r modulo CONF, denoted by CF^r, is called the set of conflict clusters of the color marking CM^r.

Let $CF^r = CT^r_1,\ldots,CT^r_\phi$ where CT^r_i, $i=1,\ldots,\phi$ denote individual conflict clusters present in CM^r.

Definition 2.8. Given a conflict cluster CT^r_i, a conflict subcluster of CT^r_i is a maximal subset SCT^r_{ik} of CT^r_i such that for any two transitions $t_m \in SCT^r_{ik}$ and $t_s \in SCT^r_{ik}$, $t_m o\ t_s$.

In order to simplify notations, let us denote by SCF^r_i the set of all conflict subclusters of a conflict cluster CT^r_i. Note that SCF^r_i is covering of CT^r_i but not necessarily a partition.

Definition 2.9. A transition t_k is said to cover another transition t_m in the color marking CM^r (denoted by t_k/t_m) if the following condition hold:

1. $t_k \in E^r$ and $t_m \in E^r$.
2. $\text{g.l.b.}(\mathcal{D}(\theta^r_k)) > \text{g.l.b.}(\mathcal{D}(\theta^r_m))$.

If t_k/t_m in CM^r we also say that t_k has higher priority than t_m in the color marking CM^r.

Definition 2.10. The set of majorants of a conflict subcluster SCT^r_{ik} is a maximal subset $maj\{SCT^r_{ik}\}$ of SCT^r_{ik} such that if $t_m \in maj\{SCT^r_{ik}\}$ then there exists no other transition $t_s \in SCT^r_{ik}$ and t_s/t_m. Note that a conflict subcluster may have several distinct majorants.

Based on the above concepts, we define the conditions under which an enabled transition is firable.

Definition 2.11. A transition t_k is firable in some color marking CM^r if and only if t_k is a majorant of all the conflict subclusters of CM^r in which it is contained.

Firing Rule: Any transition which is firable in a color marking CM^r may be selected to be fired in CM^r.

The question can be raised whether the above rule can always enable one to find a firable transition from a nonempty set of enabled transitions. The answer to this question is positive as shown.

Proposition 2.2. There exists a firable transition in any non-empty conflict cluster.

Proof: Let CT_i^r be a non-empty conflict cluster and let

$$COL_i^r = \{g.l.b.(\mathcal{D}(\theta_s^r)) \mid t_s \in CT_i^r\}.$$

Since $COL_i^r \subseteq X$, $COL_i^r \neq \emptyset$, there must exist a maximal element of the set COL_i^r, let it be c. Suppose $t_k \in CT_i^r$ and $g.l.b.(\mathcal{D}(\theta_k^r) = c$. Hence t_k is a majorant of all conflict subclusters in which it is contained (recall that CF^r is a partition of E^r) and therefore firable.

Note that $g.l.b.(\mathcal{D}(\theta_k^r))=c$, where c is a maximal element of the set COL_i^r, is a sufficient condition for t_k to be firable in the color marking CM^r but not a necessary condition.

Let us now give a few definitions which we shall make use of in later sections. Suppose t is a transition firable in some color marking CM^r. Let us assume that CM^{r+1} is a color marking obtained after the firing of t in CM^r is denoted by $CM^r[t > CM^{r+1}$.

Definition 2.12. Given an arbitrary color marking CM^r, a firing sequence from CM^r is a string $\gamma_k = t_{k1} \dots t_{ks}$ of transition names ($\gamma_k \in T^*$) such that there exist the color markings CM^{r+i}, $1 \leq i \leq s$, and CM^{r+i-1} $[t_{ki} > CM^{r+i}$.

If CM^{r+s} is a color marking obtained after the execution of the firing sequence γ_k from CM^r then we shall say that γ_k can lead from CM^r to the color marking CM^{r+s} and we shall denote this fact by CM^r $[\gamma_k > CM^{r+s}$.

Definition 2.13. A C-Colored Petri Model (C-CPM) is a system CP = (CN,CM°) where:

1. CN = (N,C,F) is a C-Colored Petri Net.
2. CM° is the initial color marking of CN.

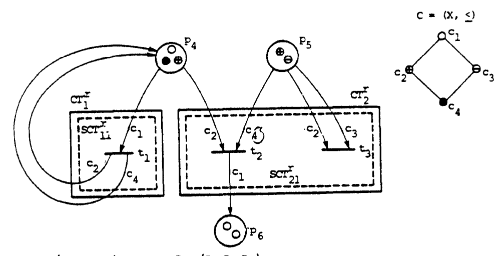
Definition 2.14. Given a C-CPM CP=(CN,CM°), the set of firing sequences of CP is:

$$S(CM^\circ) = \{\gamma \mid \gamma \in T^* \text{ and } CM^\circ[\gamma > CM^r, \text{ for some color marking } CM^r\}.$$

Given a final color marking CM^f, the set of terminal firing sequences of CP is:

$$T(CM^\circ, CM^f) = \{\gamma \mid \gamma \in T^* \text{ and } CM^\circ \ [\gamma \mid CM^f\}.$$

FIgure 2.1 illustrates an example of a C-CPM and its firable transitions.



$T = \{t_1, t_2, t_3\}$; $P = \{P_4, P_5, P_6\}$

I	t_1	t_2	t_3
P_4	1	1	0
P_5	0	1	2
P_6	0	0	0

⊕	P_4	P_5	P_6
t_1	2	0	0
t_2	0	0	1
t_3	0	0	0

P	CM^r
P_4	$<c_1,c_2,c_4>$
P_5	$<c_2,c_3>$
P_6	$<c_1^2>$

Enabling colors selection possibilities	Transition	Bag of enabling colors	$g.l.b.(\mathcal{D}(\theta_k^r))$	Firable transition
1	t_1	$<c_1>$	c_1	t_1, t_2
	t_2	$<c_2,c_2>$	c_2	
	t_3	$<c_2,c_3>$	c_4	
2	t_1	$<c_1>$	c_1	t_1, t_2, t_3
	t_2	$<c_2,c_3>$	c_4	
	t_3	$<c_2,c_3>$	c_4	

Figure 2.1
Sample C-CPM

3. Modelling Examples

In this section we present two examples which illustrate the use of the C-CPM for modelling concurrent systems. We note in passing that it has been shown ([7]) that Generalized Petri Nets cannot correctly model a special case of our first example.

3.1. A Producer-Consumer Synchronization Problem with Dynamic Priority Hierarchy.

Consider the following process synchronization problem (Figure 3.1). The coordination system contains two buffers, B_1 and B_2. Two producer processes, P_{i1} and P_{i2}, and a consumer process, C_i, are connected to each buffer B_i, i=1, 2. Either one of the producer processes can be activated at any time. At that time it produces and deposits an item in the associated buffer. A consumer process can become activated only if the associated buffer is nonempty. Following rules govern the access of each consumer process to its respective buffer. Consumer process C_i may consume from B_i items deposited there by P_{i1} only if B_i does not contain any item produced by P_{i2}. Moreover, it is assumed that C_1 and C_2 can consume items from the corresponding buffers only through a channel with capacity 1, i.e., through a channel which can accomodate only one consumer process at a time. Following priority rule determines which consumer process can seize the channel in case both C_1 and C_2 are simultaneously active:

- if C_1 attempts to consume an item produced by P_{12} while C_2 attempts to consume an item produced by P_{21} then either consumer process can gain control over the channel;

- in all other cases C_1 has priority over C_2.

We note that the coordination system described above incorporates interrelated but distinct priority hierarchies:

1. The consumer processes must select items from the respective buffers according to a fixed priority.
2. Control over the channel is obtained in accordance with a varying priority hierarchy among the consumer processes.

These coordination aspects can be modelled by the C-CPM in a natural way mainly due to the following specific features of this model:

- the rule used for the selection of enabling colors;
- the dependence of the priority of a transition on the current color marking and on the particular selection of the enabling colors.

Figure 3.2 exhibits the C-CPM representation of this producer-consumer synchronization system. There, transitions t_1, t_2, t_3 and t_4 model the producer processes P_{11}, P_{12}, P_{21} and P_{22}, respectively while the transitions t_5 and t_6 model the consumer processes C_1 and C_2, respectively. The places p_7 and p_8 represent the buffers B_1 and B_2, respectively while p_9 can be viewed as the implementation of the shared channel. It can easily be verified that the firing sequences of the C-CPM of Figure 3.2 correctly model the coordination sequences which can be executed by the producer-consumer system under consideration. Using the C-CPM one can correctly model even more sophisticated producer-consumer synchronization problems of this type, involving several producer-consumer processes and more complicated priority hierarchies.

3.2. A Process Coordination Problem with Conflict

Let us examine the following process coordination problem, depicted schematically in Figure 3.3. The system contains four buffers denoted by B_1, B_2, B_3 and B. Three processes, denoted by P_{i1}, P_{i2}, and Proc(i), are connected to each buffer B_i, i=1,2,3. The buffer B can be accessed only by the three processes, Proc(i) and is initially assumed to contain two undistinguishable items. Any one of the processes P_{ij} may be activated at any time. At that time it produces and deposits an item in the associated buffer B_i. The process Proc(i) becomes active only if the associated buffer B_i and the shared buffer B are nonempty. If active, Proc(i) attempts to consume an item from both B_i and B with the restriction

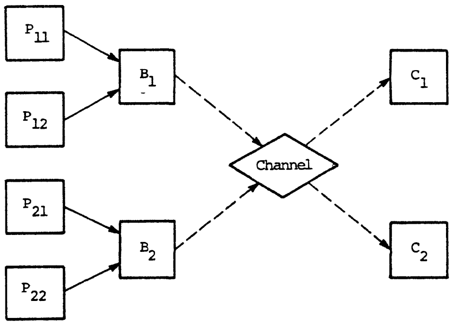


Figure 3.1
Producer-Consumer Synchronization System with Dynamic Priority Hierarchy

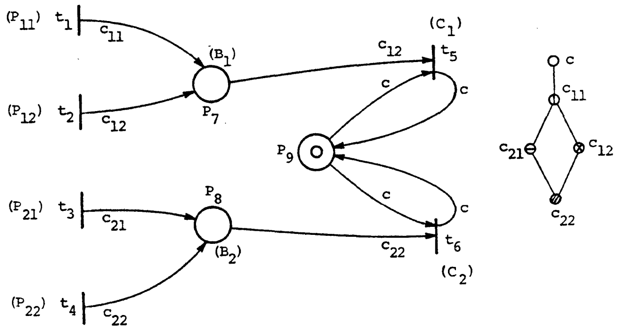


Figure 3.2
C-CPM of the Producer-Consumer Synchronization System of Figure 3.1

that it may consume from B_i an item produced by P_{i1} only if B_i does not contain any item produced by P_{i2}. Moreover, the following priority rule is imposed:

1. A conflict situation arises if the number of items currently contained in the buffer B is strictly less than the number of active processes Proc(i). In this case Proc(1) has highest priority to proceed, Proc(3) has lowest priority and Proc(2) has lower priority than Proc(1) but higher than Proc(3).
2. Otherwise, either active process Proc(i) may proceed.

We note that in both the situations only one process Proc(i) may proceed at a time. After the chosen process, say Proc(k), has consumed the respective items from B_k and B it will temporarily deactivate itself. For example, we can assume that Proc(k) initiates at this stage an abstract operation associated with it which requires an arbitrarily long (possibly null) period of time to execute. Only after the abstract operation is terminated will Proc(k) restore the item consumed from B and regain its active status, as soon as both B_k and B are nonempty.

Meanwhile, however, the pending active processes Proc(i), $i \neq k$, are polled again in order to determine which process proceeds next. From this point of view the processes Proc(i) operate asynchronously and concurrently.

The characteristic aspect of this process coordination problem is the conflict situation which arises in case the number of available resources in a shared pool of homologous resources is less than the number of processes competing for the respective resources. The priority rule designed for solving the conflict situation can be represented in the C-CPM in a natural way mainly due to the conflict structure (conflict cluster, conflict subclusters, etc.), employed for the selection of a transition to be fired. Figure 3.4 exhibits the C-CPM representation of the coordination system of Figure 3.3. Adjacent to the places and transitions of Figure 3.4, we have indicated the entities of the coordination system they model. Let us examine more closely the models of the processes Proc(i). For example, transition t_7 is considered to model the initiation of the abstract operation associated with Proc(1) while t_8 models the termination of that operation. The presence of a token in place p_{17} signifies that the abstract operation is in execution. Conversely, the occurrence of a token in place p_{18} signals that the respective operation is terminated. The processes Proc(2) and Proc(3) are modelled similarly.

This type of coordination problems can certainly be extended further. In particular, we note that we have used a totally ordered priority hierarchy for solving the conflict situation at the buffer B only in order to make the problem simple. Actually, more sophisticated priority hierarchies can be used. For example, we can make the priority of the processes Proc(i) depend on the particular selection of items from the associated buffers B_i. In that case, the conflict situation could be solved according to a dynamic priority hierarchy similar to that described in Section 3.1.

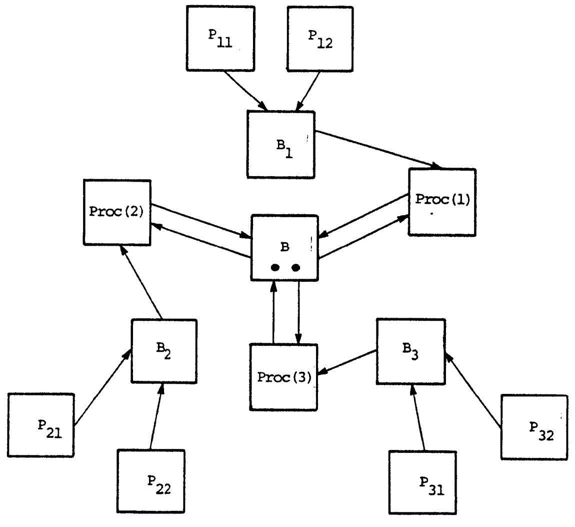


Figure 3.3
Process Coordination System with Conflict

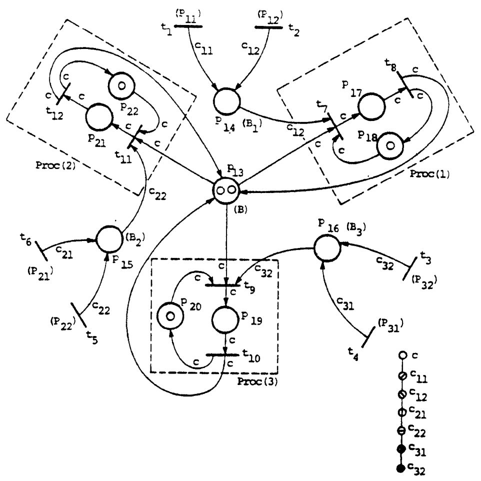


Figure 3.4
C-CPM Representation of the Coordination System of Figure 3.3

4. The Representation Capabilities of the C-CPM

In the previous section we have presented two modelling examples in order to stress the usefulness of the C-CPM with respect to the representation of certain process synchronization structures. In this section we give some results of an in-depth analysis concerning the comparative representation capabilities of the C-CPM and other variants of the Generalized Petri Nets. The family of formal languages generated by a class of Petri Nets is assumed here to be an indication of the representation capability of that class of Petri Nets. For the analysis itself, the reader is referred to [11]. Let us now define the families of languages in question.

Definition 4.1. A Labelled C-Colored Petri Model is a system $CP_\Sigma=(CP,\Sigma,L)$ where:

1. CP = (CN,CM°) is a C-CPM.
2. Σ is a finite alphabet.
3. $L : T \rightarrow \Sigma$ is a function called the labelling function of CP_Σ.

The definition of the labelling function can easily be extended to firing sequences. Let $t\gamma$ be a firing sequence of CP. Then $L(t\gamma)$ is recursively defined as follows:

$$L(t\gamma) = \begin{cases} L(t) & \text{if } \gamma = \lambda \text{ (λ denotes the empty string)} \\ L(t)L(\gamma) & \text{if } \gamma \neq \lambda \end{cases}$$

where $L(t)$ $L(\gamma)$ represents the concatenation of the strings $L(t)$ and $L(\gamma)$. By convention, $L(\gamma)=\lambda$ if and only if $\gamma=\lambda$, the empty firing sequence.

Definition 4.2. $\mathcal{S}$ is the family of languages such that $S \in \mathcal{S}$ if and only if there exists a C-CPM CP = (CN,CM°) where CN = (N,C,F), N = (T,P,I,O) and S = S(CM°). Obviously, $S \subseteq T^*$.

S_o is the family of languages such that $S \in S_o$ if and only if there exists a C-CPM CP = (CN,CM°) and a final color marking CM^f of CN, $CM^f \neq CM°$, such that $S = T(CM°,CM^f)$. Obviously, $S \subseteq T^*$.

Notice that the labelling function L as specified in Definition 4.1 is a nonerasing renaming homomorphism from T^* into Σ^*. For any family of languages A, we shall define the image of A under nonerasing renaming homomorphism to be the set:

$\mathcal{T}(A)$ = {L(A) | A $\in A$ and L is a nonerasing renaming homomorphism on A}.

The families of languages of interest to our study are then defined as follows:

Definition 4.3. The family of computation sequence sets of the C-CPM is the family of languages Λ, where $\Lambda = \mathcal{T}(S)$. Similarly, the family of terminal computation sequence sets of the C-CPM is the family of languages Λ_o, where $\Lambda_o = \mathcal{T}(S_o)$.

Languages families defined similarly were studied in [5] and [9] with respect to the Generalized Petri Nets, the Inhibitor Nets (called Extended Petri Model in [11]) and the Priority Nets (called Priority Petri Model in [11]). Thus, a common basis of comparison between these models and the C-CPM is provided.

In what follows, we shall give a few results regarding the language families Λ_o and Λ defined above. Due to the complexity of the proofs involved and to the lack of space, the results will be given without proof. Detailed proofs can be found in [11].

As already mentioned in the introductory section, we have compared the modelling power of the C-CPM with the modelling power of the Extended Petri Model (EPM), the Priority Petri Model (PPM), the Coordination Petri Model (COPM) and the Petri Net Model with switches, disjunctive logics and token absorbers (PM). Defining Λ_o(EPM), Λ_o(PPM, Λ_o(COPM), Λ_o(PM) and Λ(EPM), Λ(PPM), Λ(COPM), Λ(PM) analoguous to Λ_o and Λ, respectively, we have shown that:

$$\Lambda_o = \Lambda_o(\text{EPM}) = \Lambda_o(\text{PPM}) = \Lambda_o(\text{COPM}) = \Lambda_o(\text{PM})$$

$$\Lambda_o = \{W - \{\lambda\} | W \in \mathcal{C}\} = \{W - \{\lambda\} | W \in \mathcal{CE}\}.$$

Here $\mathcal{C}$ and $\mathcal{CE}$ denote the families of languages accepted in quasi-real-time by counter acceptors and E-counter acceptors (for exact definitions of the families $\mathcal{C}$ and $\mathcal{CE}$ see [11])

Also, $\Lambda = \Lambda(\text{EPM}) = \Lambda(\text{PPM}) = \Lambda(\text{COPM})$ and $\Lambda \subseteq \Lambda(\text{PM})$. It is not known whether $\Lambda(\text{PM}) \subseteq \Lambda$ as well and thus, whether $\Lambda = \Lambda(\text{PM})$. However, $\{W - \{\lambda\} | W\ \Lambda(\text{PM})\} \subset \Lambda_o$.

For any family of languages A, the image of A under renaming homomorphism (not necessarily nonerasing) is the set:

$H_\lambda(A)$ = {h(A) | A $\in$ and h is a renaming homomorphism on A}. Based on the equality results given above, all the results obtained in [5] with respect to the language families Λ(EPM), Λ(PPM) Λ_o(EPM) and Λ_o(PPM) can immediately be extended to the language families Λ and Λ_o respectively. In particular, from theorem 9.3 of [5] it follows that $H_\lambda(\Lambda_o)$ is the family of recursively enumerable languages.

Between the language families Λ and Λ_o we have shown that there exists the following relationship:

$$\{W - \{\lambda\} | W \in \Lambda\} \subset \Lambda_o.$$

The constructs used in order to prove the above results have allowed the following extensions in the definition of the C-CPM. We have shown that the language families Λ_o and Λ are not affected if following color thresholds and output colors are used:

- if a^i_{jk} is an input arc then $F(a^i_{jk}) = f^i_{jk}$ where $f^i_{jk}: \mathcal{P}(X) \rightarrow X$ is a function ($\mathcal{P}(X)$ denotes the power set of X) and in any color marking CM^r, the color threshold used in the selection of the enabling color for the input arc a^i_{jk} is $f^i_{jk}(\mathcal{D}(CM^r(p_j)))$;
- if a^i_{ks} is an output arc then $F(a^i_{ks}) = f^i_{ks}$ where $f^i_{ks}: \mathcal{P}(X) \rightarrow X$ is a function and in any color marking CM^r, the color of the output token placed by a^i_{ks} in the output place p_s in case t_k fires in CM^r, is $f^i_{ks}(\mathcal{D}(\theta^r_k))$.

In fact, the language families Λ_o and Λ are not affected even if the colors of the output tokens of a transition are established according to the following more general rule:

- if a^i_{ks} is an output arc then $F(a^i_{ks}) = f^i_{ks}$ where $f^i_{ks}: \mathcal{P}(X) \times \mathcal{P}(X) \rightarrow X$ is a function and in any color marking CM^r the color of the token produced by t_k on the output arc a^i_{ks} in case t_k fires in CM^r is $f^i_{ks}(\mathcal{D}(\theta^r_k),$ $\mathcal{D}(CM^r(p_s)))$ where

$$CM^{r-}(p_s) = \begin{cases} CM^r(p_s) & \text{if } I(p_s,t_k) = 0 \\ CM^r(p_s) - \theta^r_{sk} & \text{if } I(p_s,t_k) > 0. \end{cases}$$

These extensions of the definition of the C-CPM are useful when one is interested in modelling the flow of control in reentrant subroutines, for example. The efficient and clear modelling of reentrancy becomes especially important when modelling real-life collections of programs such as compilers or reentrant portions of operating systems.

Figure 4.2 exhibits our model of the flow of control in the reentrant subroutines of Figure 4.1. Distinct control flow streams in operating in a reentrant subroutine are represented by distinct colors. We note that upon firing, transition t_5 places in p_9 a token of color c_2 or c_3, depending on its enabling color. On the other hand, due to the particular structure of the lattice of colors, the transitions t_2 and t_4 can select only the colors c_2 and c_3, respectively, as enabling colors from p_9. Thus, control is always returned from the reentrant subroutine to the proper calling routine.

We note that the purpose of the output color function f^1_{59} is to "paint" the output tokens produced by the transition t_5 with the same color as that of the tokens absorbed by the respective transition from its input place. In this way the continuity of the control flow streams operating concurrently in the reentrant subroutine is ensured. The usage of this type of mappings as output color functions enhances the naturalness of the representation of reentrancy. This method applies to nested reentrant subroutines as well. Consider for example, the synchronization situation depicted schematically in Figure 4.3. There, both subroutines A and B may be invoked concurrently by different calling programs. Figure 4.4 displays the C-CPM representation of the flow of control in this synchronization system. Note that due to the structure of the lattice of colors, the return of control from subroutine B is correctly modelled, that is the control flow stream originating from PROGRAM 3 can never be routed to subroutine A upon termination of the execution of subroutine B and vice versa.

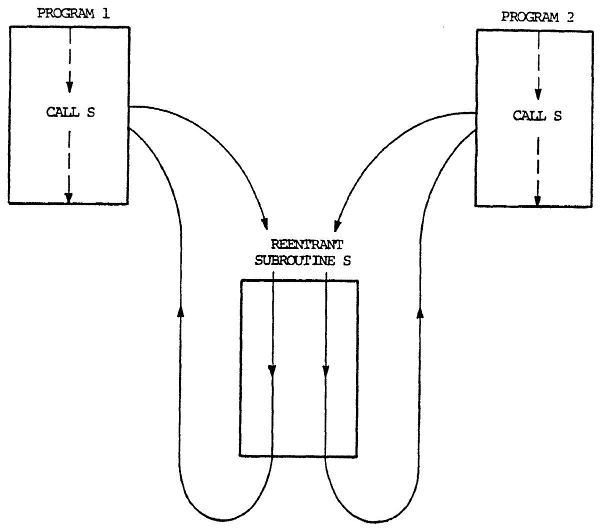
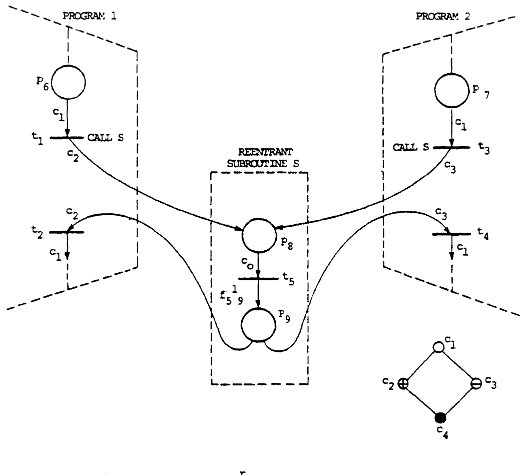


Figure 4.1
Sample Reentrant Subroutine



in any color marking CM^r

$$f_5{}^1{}_9(\mathcal{D}(\theta^r_5) = \begin{cases} c_2 & \text{if } (\theta^r_5) = \{c_2\} \\ c_3 & \text{if } (\theta^r_5) = \{c_3\} \end{cases}$$

Figure 4.2
C-CPM Representation of a Sample Reentrant Subroutine

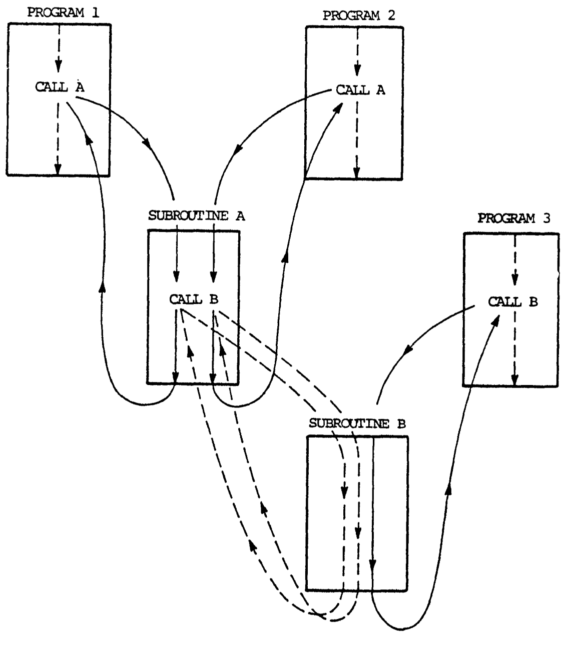
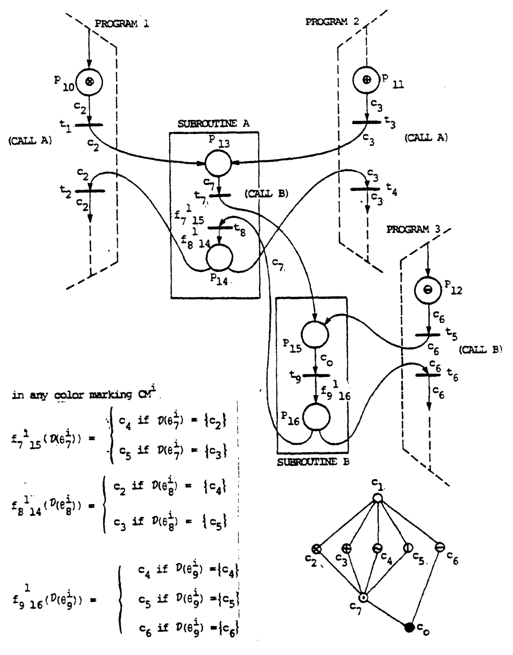


Figure 4.3
Nested Reentrant Subroutines



in any color marking CM^i

$$f_{7\ 15}^{\ 1}(\mathcal{D}(\theta_7^i)) = \begin{cases} c_4 \text{ if } \mathcal{D}(\theta_7^i) = \{c_2\} \\ c_5 \text{ if } \mathcal{D}(\theta_7^i) = \{c_3\} \end{cases}$$

$$f_{8\ 14}^{\ 1}(\mathcal{D}(\theta_8^i)) = \begin{cases} c_2 \text{ if } \mathcal{D}(\theta_8^i) = \{c_4\} \\ c_3 \text{ if } \mathcal{D}(\theta_8^i) = \{c_5\} \end{cases}$$

$$f_{9\ 16}^{\ 1}(\mathcal{D}(\theta_9^i)) = \begin{cases} c_4 \text{ if } \mathcal{D}(\theta_9^i) = \{c_4\} \\ c_5 \text{ if } \mathcal{D}(\theta_9^i) = \{c_5\} \\ c_6 \text{ if } \mathcal{D}(\theta_9^i) = \{c_6\} \end{cases}$$

Figure 4.4
Modelling of Nested Reentrant Subroutines

5. Conclusions

In Section 4 we have seen that the PPM, EPM, COPM, PM and the C-CPM are completely equivalent from the point of view of the Λ_0-language family generated. This fact is rather surprising if one considers that these models were introduced both independently and for distinct purposes. Each class of Petri Nets listed above was defined in order to offer natural representation of a certain class of synchronization systems of interest to the respective authors. Therefore, the equivalence results given in Section 4 do not diminish the usefulness of any of these classes of Petri Nets.

The modelling examples presented in this paper point out the kind of process synchronization structures for which the C-CPM offers a simple and clear representation which preserves the natural process structure. It is worthwhile mentioning that in [11] it was argued in detail that the EPM, PPM, COPM and PM fail to offer a natural and easy to understand representation for the process synchronization structures considered here.

Bibliography

[1] T.K.M. Agerwala, "Towards a Theory for the Analysis and Synthesis of System Exhibiting Concurrency", Ph.D. Thesis, The Johns Hopkins University, Baltimore, Maryland, 1975.

[2] J.L. Baer, "Modelling for Parallel Computation: A Case Study", Sagamore Computer Conference on Parallel Processing, 1973.

[3] V.G. Cerf, "Multiprocessors, Semaphores, and a Graph Model of Computation", Ph.D. Thesis, Computer Science Department, UCLA, 1972 (UCLA-10P14-110).

[4] K.P. Gostelow, "Flow of Control, Resource Allocation, and the Proper Termination of Programs", Ph.D. Thesis, Computer Science Department, UCLA, 1971, (UCLA-10P14-106).

[5] M. Hack, "Petri Net Languages", Computations Structures Group Memo 124, Project MAC, MIT, Cambridge, Massachusetts, June 1975.

[6] M. Hack, "Decidabilty Questions for Petri Nets", Technical Report 161, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, June 1976.

[7] S.R. Kosaraju, "Limitations of Dijkstra's Semaphore Primitives and Petri Nets", The Johns Hopkins University, Baltimore, Maryland, May 1973.

[8] S.S. Patil, "Coordination for Asynchronous Events", Ph.D. Thesis, Project MAC, MAC-TR-72, MIT, Cambridge, Massachusetts, 1970.

[9] J.L. Peterson, "Modelling of Parallel Systems", Ph.D. Thesis, Department of Electrical Engineering, Stanford University, 1973.

[10] J.L. Peterson, "Computation Sequence Sets", Journal of ACM, 13, 1-24 [1976].

[11] C.R. Zervos, "Colored Petri Nets: Their Properties and Applications", Ph.D. Thesis, Program in Computer, Information and Control Engineering, the University of Michigan (SEL-TR-107), January 1977.

A GENERALIZED INSTRUMENTATION PROCEDURE FOR CONCURRENT PASCAL SYSTEMS

O. G. Johnson
University of Houston
and
IBM
Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract — A technique is presented for *instrumenting Concurrent Pascal Systems* at the component level. The key technical problem is that of implanting in each component a *unique index* which can be matched to the name of the component by the instrumentation package. The implantation technique requires two monitors for parent-parent synchronization and parent-offspring synchronization. An additional monitor provides percent utilization statistics by component and a report process periodically produces summary figures.

The entire procedure can be automated. The source Concurrent Pascal System can be read by an instrumentation program and the Instrumented Source Program produced as output. No modifications are required to the Concurrent Pascal Compiler, hence the same compiler can be used to compile and run either the instrumented or uninstrumented version of the system.

Introduction

Program instrumentation has been studied in a number of contexts in the several years since its introduction into the literature [1], [3], [4], [5], [6], [7], [8], [9]. Here we consider the problem of instrumenting programs which are written in Concurrent Pascal [2] for the purpose of determining component utilization. The placement of instrumentation probes in this environment is somewhat specialized since system components may be multiple realizations of system types. Hence the probes, which appear as syntax within the system types, must produce component specific statistics.

It is necessary to imbed in each component a unique index which can be utilized to identify the component during the collection and reporting of statistics. This is accomplished by a flip-flop monitor which synchronizes parent-offspring calls to a "census" table. Upon spawning an offspring, a parent enters its name in the table. The offspring then retrieves the table index of the name. The monitor is written to insure that these events occur in proper order. Section two describes this process in detail.

There is also the problem of simultaneous spawning of offspring by several parent components. It may be necessary to queue a number of parents to wait for access to the census table. It is expedient to handle this problem by a second monitor which acts as a virtual facility which must be seized by each parent in turn. Section three describes the details of parent-parent synchronization.

Section four contains a description of the syntax of each probe along with a set of rules for placing the probes. It follows from the rules that the placement of probes can be automated. Hence it is possible to write an instrumenting system which will automatically instrument an arbitrary Concurrent Pascal system.

Section five presents a sample, producer-consumer, system. The uninstrumented system is given first and is followed by the instrumented version.

The details of the program for automatic instrumentation of Concurrent Pascal systems are not covered here. They are intended to be the subject of a future report.

Parent-Offspring Synchronization

The IMPLANT monitor is given below. It is realized by a system component named IMPLANTER. The parent component alternately performs IMPLANTER.NEWNAME('k_i') and *init* k_i *end*. The entry procedure NEWNAME enters 'k_i' in the census table and updates the table index. In the parent component, the statement *init* $k_i,\dots,k_n$ *end* is replaced by alternate calls to NEWNAME and individual *init* statements as given by probe SI in section four.

The first executable statement of each system type (which must of necessity be realized as one or more offspring) upon instrumentation, is changed to the second executable statement. The new first statement is

COMPONENTNO:=IMPLANTER.INDEXNO

In this way, the table index is implanted in the component. This index value can then be used as a parameter of any calls to attribute gathering components. For instance, if we wish to monitor component utilization, we can have a CLOCK monitor, realized by CLOCKER. we can then execute

CLOCKER.START (COMPONENTNO)

or

CLOCKER.STOP (COMPONENTNO)

The CLOCK monitor syntax is not given here since the coding for this monitor is obvious. There is also the need for a separate process type TABULATE (realized by TABULATOR) which will periodically output the statistics gathered.

The following constants and types are global to the Instrumentation System Types. They can be added to the beginning of the program being instrumented.

*const*NOCOMP = maximum number of components allowed,
IDLENGTH = maximum number of characters in an identifier;
type STRING = *packed array* [1...IDLENGTH] *of char*;
(The key words *packed array* can be replaced by *array* if necessary)

The IMPLANT monitor program is the following:

```
type IMPLANT = monitor ;
var INDEX : [1..NUMCOMP] ; FATHER : boolean ;
     COMPTAB : array [1..NUMCOMP] of STRING ;
     FQUE, CQUE : queue ;
procedure entry NEWNAME (NAME : STRING) ;
```

```
begin
        if not FATHER then delay (FQUE) ;
        FATHER : = false ;
        COMPTAB[INDEX] : = NAME ;
        continue (CQUE)
end ;
function entry INDEXNO : [1..NUMCOMP] ;
begin
        if FATHER then delay (CQUE) ;
        FATHER : = true ;
        INDEXNO : = INDEX ;
        INDEX : = INDEX + 1 ;
        continue (FQUE)
end ;
begin
        INDEX := - 1 ;
        FATHER : = true
end ;
```

Parent-Parent Synchronization

The SPAWN monitor, realized by SPAWNER, provides for an array of queues for parent components which are waiting for access to the census table. In Concurrent Pascal, the queue data type is allowed to delay and thus hold all systems information for one component only. Hence the programmer must provide for an array of queues if more than one component can be delayed at a time.

Before any parent can execute a statement of the form *init* k_i,...,k_n *end* it must first execute

SPAWNER.SEIZE

After the *init* statement, it must execute

SPAWNER.RELEASE

These two statements are part of probe SI mentioned before and described in the next section.

The queue array is programmed as a queue data structure in the usual way:

```
type SPAWN = monitor ;
var LINE : array [0..NUMCOMP] of queue BUSY : boolean ;
HEAD , TAIL : integer :
procedure entry SIEZE ;
begin
        if BUSY then
        begin
                TAIL : = TAIL + 1 ;
                delay (LINE [TAIL mod NUMCOMP])
        end
                else BUSY : = true
end ;
procedure entry RELEASE ;
begin
        if HEAD < TAIL then
        begin
        HEAD := HEAD + 1 ;
        continue (LINE [HEAD mod NUMCOMP])
end
        else BUSY : = false
end ;
begin
        HEAD : = -1 ; TAIL : =-1 ; BUSY = false
end ;
```

Instrumentation Probes

All together there are eight different types of probes. The following list gives each probe a name (e.g. A,C,G, etc.), describes what the probe does and gives the syntax of the probe. In the syntactical description, square brackets indicate syntax which may be required by the contex and braces indicate a choice of syntax. The variation in syntactical detail is quite minor and probe placement can be automated.

A	: Definition of access rights to instrumentation monitors
C	: Definition of component number variable (COMPONENTNO)
G	: Start clock
I	: Implant component number value
ICD	: Instrumentation Components definition
ICI	: Instrumentation Components initialization
S	: Stop Clock
SI	: "Sequentialization" of *init* statement

A : $\left\{\begin{matrix}(\\,\end{matrix}\right\}$ [SPAWNER:SPAWN;]IMPLANTER:
IMPLANT;CLOCKER:CLOCK $\left\{\begin{matrix})\\b\end{matrix}\right\}$

C : [*var*] COMPONENTNO:[1..NOCOMP];
G : CLOCKER.START(COMPONENTNO)[*end*][;]
I : COMPONENTNO:=IMPLANTER.INDEXNO;
ICD : SPAWNER:SPAWN;IMPLANTER:IMPLANT;
CLOCKER:CLOCK;TABULATOR:TABULATE;
ICI : *init* SPAWNER,IMPLANTER,CLOCKER,
TABULATOR *end* ;
S : [*begin*] CLOCKER.STOP (COMPONENTNO) [;]
SI : replace *init* k_1,...,k_n *end* by
[begin] SPAWNER.SIEZE;IMPLANTER.NEWNAME('k_1');
init k_1 *end*;...;IMPLANTER.NEWNAME('k_n'); *init* k_n *end* ;
SPAWNER.RELEASE [*end*][;]

The rules for probe placement can be summarized as follows:

A :	Placed in all system type definition statements. If the type has no access rights, the parenthesis are used else the comma and blank are used to append access to the instrumentation monitors. If the system type contains an *init* statement, access to the SPAWNER must be included.
B :	Placed in the variable definition block of each system type. If the block is void, *var* must be included.
G :	Placed as the new first executable statement of each *entry* and (with I preciding it) at the beginning of each initialization routine. Also, placed after each *delay*. If delay is part of a structured statement, *end* and/or a semicolon may be required.
I :	Placed as the first executable statement of each initialization routine.
ICD :	Placed in the variable definition section of the initial process.
ICI :	Placed as the first executable statement of the initial process.
S :	Placed before each *delay*. If *delay* is part of a structured statement, *begin* may be required and a semicolon is required. Also placed before each *continue*. If continue is part of a structured statement *begin* may be required and a semicolon is required. Also placed before each final *end* of each *entry* and initialization routine which is not immediately preceded by *cycle* ... *end* ; or *continue* (...);
SI :	Replaces all *init* statements.

Example

In the following Producer-Consumer system, the Producer is producing (pseudo) random integers in the range 0 to 1740. The Consumer is keeping a running tally of the largest and smallest numbers produced. The bin between the Producer and the Consumer holds ten numbers and is treated as a stack. The unistrumented system is given first and the placement of probes is indicated in the second program listing.

Uninstrumented Producer — Consumer Example

```
type PRODUCE = process (B:BIN) ;
var V : interger ;
procedure MAKE (var V:integer) ;
begin
      V : = (V*1979) mod 1741
end ;
begin
      V : = 761 ;
      cycle MAKE (V) ; B.SEND (V) end ;
end ;
type CONSUME = process (B:BIN) ;
var V, HI, LO : integer ;
procedure USE (V:integer) ;
begin
      if V > HI then HI : = V ;
      if V < LO then LO : = V ;
end ;
begin
      HI : = - 1 ; LO : = 1741 ; cycle B.RECEIVE (V) ;
      USE (V) end ;
end ;
type BIN = monitor ;
var EMPTY, FULL : boolean ; I : 0..10 ; PQ, CQ : queue ;
STACK : array [1..10] of integer ;
procedure entry SEND (V:integer) ;
begin
   if FULL then delay (PQ) ;
   I : = I + 1 ; EMPTY : = false ;
   STACK [I] : = V ;
   if I = 10 then FULL : = true ;
   continue (CQ)
end ;
procedure entry RECEIVE (var V:integer) ;
begin
   if EMPTY then delay (CQ) ;
   V : = STACK [I] ;
   if I = 0 then EMPTY : = true ;
   continue (PQ)
end ;
begin
   I : = 0 ; FULL : = false ; EMPTY : = true
end ;
var PRODUCER : PRODUCE ; CONSUMER : CONSUME
; BUF : BIN ;
begin
   init BUF, PRODUCER (BUF), CONSUMER (BUF) end
   ;
end.
```

Probe Placement in Producer Consumer Example

Probes indicated by **boldface type**

```
type PRODUCE = process (B:BIN A) ;
var V : integer ;  C
procedure MAKE (V:integer) ;
begin
    V : = (V*1979) mod 1741
end ;
begin  I G
    V ; = 761 ;
    cycle MAKE (V) ;  S B.SEND (V) G end ;
end;
type CONSUME = process (B:BIN A ) ;
var V, HI, LO : integer ;  C
procedure USE (V:integer) ;
begin
    if V > HI then HI : = V ;
    if V < LO then LO : = V ;
end ;
begin  I G
    HI : = - 1 ; LO ; = 1741 ; cycle S B.RECEIVE (V) G ;
    USE (V) end ;
end ;
type BIN = monitor  A ;
var EMPTY, FULL : boolean ; I : 0..10 ; PQ, CQ : queue ;
STACK : array [1..10] of integer ;  C
procedure entry SEND (V:integer) ;
begin  G
    if FULL then S delay (PQ) G ;
     I : = I + 1 ; EMPTY : =  false ;
    STACK [I] : = V ;
    if I = 10 then FULL : = true ;
S   continue (CQ)
end ;
procedure RECEIVE (var V:integer) ;
begin  G
    if EMPTY then S delay (CQ) G ;
    V : = STACK [I] ;
    I : = I - 1 ; FULL : = false ;
    if I = 0 then EMPTY : = true ;
S   continue (PQ)
end ;
begin  I G
    I : = 0 ; FULL : = false ; EMPTY : = true        S
end ;
var PRODUCER : PRODUCE ; CONSUMER : CONSUME
```

```
; BUF : BIN ;  ICD
begin  ICI

    ┌──────────────────────────────────────────────┐
    │ init BUF, PRODUCER (BUF), CONSUMER (BUF)     │
    │ end ;                                        │
    └──────────────────────────────────────────────┘
                                                  SI
end.
```

References

1. B. Bussell, and R. A. Koster, "Instrumenting Computer Systems and Their Programs", Proc. 1970 AFIPS Fall Joint Computer Conference, Vol. 37, 1970.

2. B. Hansen, The Architecture of Concurrent Programs, Prentice Hall, Englewood Cliffs, N.J. 1977.

3. J. C. Hyang, "Instrumenting Programs for Data-Flow Analysis", Tech. Report No. UH-CS-77-4, Department of Computer Science, University of Houston May 1977.

4. J. C. Huang, "Program Instrumentation and Software Testing", Computer, Vol. 11, No. 4 (April 1979), pp 25-32.

5. M. R. Paige and J. P. Benson, "The Use of Software Probes in Testing Fortran Programs", Computer, Vol. 7, No. 7, July 1974, pp 40-47.

6. C. V. Ramamoorthy, K. H. Kim, and W. T. Chen, "Optimal Placement of Software Monitors Aiding Systematic Testing", IEEE Trans. on Software Engineering, Vol. SE-1, No. 4, December 1975, pp 403-411.

7. E. C. Russell, and G. Estrin, "Measurement Based Automatic Analysis of FORTRAN Programs", Proc. 1969 AFIPS Joint Spring Computer Conference, Vol. 34, 1969.

8. J. H. Saltzer and J. W. Gintell, "The Instrumentation of Multics", CACM, Vol. 13, No. 8, Aug. 1970, pp 495-500.

9. L. G. Stucki, "Automatic Generation of Self-Metric Software", Proc. 1973 IEEE Symposium on Computer Software Reliability, New York, N.Y., April 1973.

A DEMON LANGUAGE COMPILER ON A NETWORK FOR PARALLEL CONTROL

Farid El-Wailly, Peter Greene, John Putnam, and Martha Evens
Computer Science Department
Illinois Institute of Technology
Chicago, Illinois, 60616

Summary

We have designed and implemented a demon language for our experiments in heuristic control of loosely coupled complex systems [1]. The compiler itself is written in Fortran on a PRIME 400, but it generates code for TECHNEC [2], a ring network of LSI-11s running in parallel. A critical problem is the control of galactic variables, variables whose values must be available simultaneously on several nodes.

A program for controlling a complex system may be relieved of the need to regulate everything at once, through the presence of independent procedures that monitor system behavior and take corrective action when they detect some particular pattern of events. In the artificial intelligence field such monitors are called demons. In programs that are to be distributed over a network computer, demons are independent enough to run in parallel, and they may be programmed separately.

A demon has five features: a trigger event, an evocation condition, a procedure body, interfaces for output to the outside world, and scheduling policies. When the trigger event occurs, the evocation condition is evaluated. If the condition evaluates "true", the procedure body is executed. The procedure may use the output interfaces to control actuators to influence the physical environment or may cause other demons to be evoked. The scheduling policies control such things as whether the demon is active ("paying attention" or not), and how soon the procedure must be executed after the event and condition cause it to be evoked.

Trigger events can be the completion of a time interval, the arrival of a message, or the modification of a monitored variable. So that demons need not keep scanning variables, a list is kept for each variable of all demons monitoring it. The system is designed to accomodate multiple main programs, each with its set of "owned" demons and procedures. The demons owned by a main program may inhabit different nodes than their owner.

The compiler is organized as a single pass with some forward references left to be resolved by the RT-11 assembler. It is written in the PRIME's recursive extended Fortran, which makes it simple to use recursive descent parsing. The compiler generates indirect threaded code [3] partitioned into separate modules for the LSI-11s.

Design of the runtime environment is strongly affected by the fact that the LSI-11s have no common memory so all communicaton of values of variables between them must take place via messages. The demon language has several kinds of variables. Variables declared within a procedure or a demon are local to the subprogram in which they are declared and do not need special handling. Variables declared within a main program are global to all demons and procedures owned by that main program. These variables may need to be transmitted to other nodes if they are parameters in procedure calls or if some demon in another node needs their values. The variables which cause the greatest problems are those we have named galactic variables, those with values which must be simultaneously available on several nodes.

Galactic variables provide communication between different main programs, and provide demons on remote nodes with fast access to certain variables. Global variable storage is attached to each main program and is accessible by all demons and procedures owned by that main program. Galactic variables, on the other hand, are kept in a common storage area which is replicated on each node. There is only one instance of a galactic variable per node and access is done indirectly through pointers. This storage is common to all main programs, demons, and procedures existing on that node. When a program unit changes the value of a galactic variable that value is instantly available to all program units on the node. A broadcast message is also sent around the ring to update the value of that galactic variable in the galactic storage on all the other nodes.

Changes to galactic and global storage are made through calls to the demon supervisor, a collection of demon support routines resident on each node, which buffer variable access on that node. They alert appropriate demons when a key variable is changed and to initiate the propagation of galactic variable change messages around the ring. They also allow the buffering of variable access between demons and a main program on a different node. It is inevitable in this kind of loosely coupled system that there will be times when copies of the same galactic variable on different nodes will have different values. The system is designed to be robust enough to survive in this situation.

References

[1] P.H. Greene. Organization of Multivariable Control Tasks. Computer Science Dept., Illinois Institute of Technology, TR 78-12, Chicago, Illinois, (1978).

[2] T.W. Christopher, O.I. El-Dessouki, M.W. Evens, P.H. Greene, A. Hazra, W-H. Huen, A. Rastogi, R. Robinson, W. Wojciechowski. "Uniprogramming a Network Computer", Proc. Eighth Annual Int. Conf. on Parallel Processing (August, 1978), pp. 132-138.

[3] R.B.K. Dewar. "Indirect Threaded Code", CACM (June, 1975), pp. 330-331.

AN ALGORITHM FOR THE CONCURRENT UPDATE OF MULTIPLE-COPY DATABASES

Mohamed G. Gouda and Robert G. Arnold
Honeywell Inc.
Bloomington, MN 55420

Summary

The problem of concurrent updates for multiple-copy distributed databases has received much attention in recent years. A good survey of the problem and of some algorithms to solve it can be found in [1]. Although there are no quantitative studies to estimate the performance of different algorithms, most algorithms seem to require a large number of communications between the different sites in the distributed database system. Thus the system performance may be reduced greatly.

To minimize the communication overhead, a recent algorithm [2] takes advantage of the locality of reference which is inherent in some database systems. The algorithm assumes that in 95% of the cases an update request generated at some site will only access data items local to that site. The algorithm allows deadlocks to occur; and proposes a deadlock detection scheme in which a global picture of the distributed system is constructed and analyzed by one machine called the "SNOOP". This seems to contradict the distribution philosophy of a distributed system.

In this paper, we present an alternative algorithm to solve the problem. Like [2], our algorithm assumes locality of reference; but it does not cause deadlocks; thus deadlock detection and resolution (whether centralized or distributed) are not needed.

For convenience, assume that each site has a complete copy of the database. But instead of locking and releasing all copies of a data item, we designate one specific copy of the data item to be locked and released. The site which has the designated copy of a data item is called the primary site [1] of the data item. Different data items may have different primary sites.

Each site s has an update process p which receives update requests from the site users and executes them in cooperation with other update processes at other sites. On receiving an update request from a user at site s, p examines whether the request is local or global. If the primary site of all the data items in the request is s, then the request is local, otherwise it is global. Because of locality of reference, most update requests are local.

Processing Local Updates:
To process a local update request u, the update process p adds u to the wait queue at site s. u waits in the wait queue until all its required data items are released. u is then executed on the database copy at s; then it is broadcasted to all other sites in the system where it is executed on other database copies.

Processing Global Updates:
To process a global update request u, the update process p sends u to a number of sites, namely, the primary sites of the data items required by u. At each one of these sites, u locks the required data items in the site; then moves to the next site. After locking all the required data items, u returns to site s where it is executed on the database copy at s. Then u is broadcasted from s to all other sites to be executed on the other database copies, and to release the data items locked by u.

If global requests are allowed to travel between sites in different orders, then deadlocks can occur. To avoid deadlocks, define an arbitrary total order $<$ on the different sites in the system such that if an update request u has to visit sites s and s' and if $s<s'$ then u must visit s before s'. This is a sufficient condition for freedom of deadlocks.

In the paper, the algorithm is extended to avoid starvation and to handle partially redundant systems. Currently, we are working on the fault-tolerance and performance analysis of the algorithm.

References

[1] Bernstein, P., Goodman, N., Approaches to Concurrency Control in Distributed Data Base Systems. Proc. of NCC '79, pp. 813-820.

[2] Stonebraker, M., Concurrency and Consistency of Multiple Copies of Data in Distributed INGRES. IEEE Trans. on Software Eng., Vol. SE-5, No. 3, May '79, pp. 188-194.

COMPLEXITY MEASURES OF COMPUTER STRUCTURES

W. Händler and V. Sigmund
Universität Erlangen-Nürnberg, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik III), Martensstrasse 3,
D-8520 Erlangen, Federal Republic of Germany

Summary

The central theme of this paper is the measuring of the complexity of computer structures. We show that some commonly used computer classification schemes are based on the notion of complexity measure. A special feature of such measures called structure preservation gives the computer architect a convenient tool for the considerations of machine flexibility and complexity, and for the comparison of computers and applications with regard to their structure.

The great number of new computer designs based on the technological advances of LSI makes it increasingly difficult to recognize which concept is suitable for a given application. Performance criteria alone - such as the number of MOPS, where the number of ALU's or processors is simply multiplied by the number of instructions executed per second by each ALU or processor - are not sufficient for this purpose. Only a part of the executed instructions will actually be useful for the application, while the remainder is required by the overhead (cf. functional and procedural/movement instructions in [1], or execution and transformation in [2]). The ratio of the overhead instructions to the useful instructions (non functional ratio [1]) can become great if the structure of the application does not fit the computer structure.

The only structural feature of computers taken into account in the 1960's was the machine word width. In order to make the parallelism of various computers more explicit Flynn [3] introduced a classification scheme that remains still useful. But one regards a computer that executes multiple instruction streams simultaneously to be "more parallel" than one executing a single instruction stream, and similarly in the case of the multiple data streams. This is a quantitative aspect going beyond the mere classification. In accordance with other similar situations in mathematics and computer sciences, we could speak of a (rather simple) parallelism measure of computers. However, the amount of the quantified information is too small here, even if we additionally distinguish between the serial and parallel ALU operation [4]. Feng [5] considers another parallelism measure of computers, taking the word width and multiplying it by the number of such words that can be processed throughout the computer simultaneously. The measure makes no difference between the various machine levels at which the parallelism is achieved.

The scheme introduced in [4],[6] possesses this last property. It uses the following characteristics to compare computers:

k	the number of processors connected in parallel
k'	" " " phases in macro-pipelining
d	the number of ALU's connected in parallel
d'	" " " phases in instruction pipelining
w	the ALU wordwidth
w'	" number of phases in arithmetic pipelining.

The suggestive notation $(k \times k', d \times d', w \times w')$ for the corresponding 6-tupels shows at a glance the amount of parallelism and pipelining of the computer at the main three machine levels, e.g. $(1\times1, 64\times1, 64\times1)$ for an Illiac IV-quadrant, or $(1\times1, 1\times10, 60\times1)$ for the main processor of the CDC 6600. The componentwise ordering relation induces a partial ordering over these 6-tupels that allows us to compare some computers, or, on the other hand, that shows which computers are incomparable with regard to their structure. We speak generally of a complexity measure of computer structures, since not only parallelism but also pipelining and the hierarchy of operation are quantitatively characterized here.

Certain complexity measures possess the special property that their target partially ordered set can be equipped with operations that correspond to operations on the source set, i.e. computers or applications in our case. Thus the expression $(10,1,12) \times (1,1\times10,60)$ - for the sake of simplicity, we omit the component "$\times1$" - characterizes the complete CDC 6600 with attached peripheral processors. The operator "$\vee$" ("alternative") can be used in order to show the different possible modes or structures of the computer. We obtain the following expression for the pilot EGPA configuration (cf. a companion paper [7]): $[(1,1,32)\vee(1,32,1)] \times$ $\times[(4,1,32)\vee(1\times4,1,32)\vee(1,4,32)\vee(1,128,1)]$.

Structure preserving measures have already proved to be useful for the comparison of computers [4],[6]. They could be even much more useful in the search for optimal embeddings of application structures into fixed computer structure [7]. The establishing of formal tools for this purpose is one of the most ambitious aims of our present work.

References

[1] Flynn,M.J.,"Trends and problems in computer organisations", Information Processing 74, North-Holland, Amsterdam (1974), pp. 3-10
[2] Sigmund,V.,"Parallel compiled interpretation", Proc. of the 1977 Intern. Conf. on Parallel Processing, IEEE, New York 1977, pp. 16-25
[3] Flynn,M.J.,"Very high speed computing systems", Proc. of the IEEE 54 (1966), pp. 1901-1909
[4] Händler,W.,"On classification schemes for computers in the post-von-Neumann-era", GI-4.Jahrestagung, Berlin (1974),Springer,Berlin,439-452
[5] Feng,T.,"Some characteristics of associative/parallel processing", Proc. of the 1972 Sagamore Comp. Conf., Syracuse Univ. (1972), pp. 5-16
[6] Händler,W.,"Impact of classification schemes on computer architecture", cf. 2 , pp. 7-15
[7] Händler,W., Schreiber,H., Sigmund,V.,"Computation structures reflected in general purpose and special purpose multi-microprocessor systems, this conference.

PARALLEL MEMORY SYSTEM FOR A PARTITIONABLE SIMD/MIMD MACHINE

Howard Jay Siegel
Frederick Kemmerer
Mark Washburn
Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

Abstract -- PASM is a large-scale partitionable SIMD/MIMD multimicroprocessor system being designed for image processing tasks. To improve machine throughput, a memory management system employing parallel secondary storage devices and double-buffered primary memories has been devised. The memory system is an intelligent one, using communicating microprocessors which are dedicated to handling data requests and file management. The memory system bus structure is organized to exploit much parallelism in transferring data from the secondary memories to the primary memories of virtual SIMD and MIMD machines.

I. Introduction

As a result of the microprocessor revolution, it is now feasible to build a dynamically reconfigurable large-scale multimicroprocessor system capable of performing image processing tasks more rapidly than previously possible. There are several ways to harness the parallel processing power of a multimicroprocessor system: SIMD, MSIMD, MIMD, and PSM.

An SIMD (single instruction stream - multiple data stream) machine [5] typically consists of a set of N processors, N memories, an interconnection network, and a control unit (e.g. Illiac IV [2]). The control unit broadcasts instructions to the processors and all active ("turned on") processors execute the same instruction at the same time. Each processor executes instructions using data taken from a memory to which only it is connected. The interconnection network allows interprocessor communication. An MSIMD (multiple-SIMD) system is a parallel processing system which can be structured as two or more independent SIMD machines (e.g. MAP [16]). An MIMD (multiple instruction stream - multiple data stream) machine [5] typically consists of N processors and N memories, where each processor may follow an independent instruction stream (e.g. C.mmp [38]). As with SIMD architectures, there is a multiple data stream and an interconnection network. A PSM (partitionable SIMD/MIMD) system [22] is a parallel processing system which can be structured as two or more independent SIMD and/or MIMD machines (e.g. PASM [28]).

PASM, a particular PSM-type system for image processing and pattern recognition, is currently being designed at Purdue University [22]. Due to the low cost of microprocessors, computer system designers have been considering various multimicroprocessor architectures [e.g. 3, 9, 12, 13, 17, 34, 36]. The system described here was the first in the literature to combine the following two features:
1) it may be partitioned to operate as many independent SIMD and/or MIMD machines of varying sizes; and
2) a variety of problems in image processing and pattern recognition are being used to guide the design choices.

In the next section, a brief overview of PASM is presented. The sections following describe various aspects of the PASM memory system. The use of parallel secondary storage devices, double-buffered primary memories, and dedicated microprocessors for memory management are discussed.

II. PASM Overview

PASM, a partitionable SIMD/MIMD system [22, 28, 29], is a dynamically reconfigurable multimicroprocessor machine for image processing. It is a parallel processing system which can be structured as one or more independent SIMD and/or MIMD machines of varying sizes. A block diagram of PASM is shown in Figure 1.

The heart of the system is the Parallel Computation Unit, which contains N processors, N

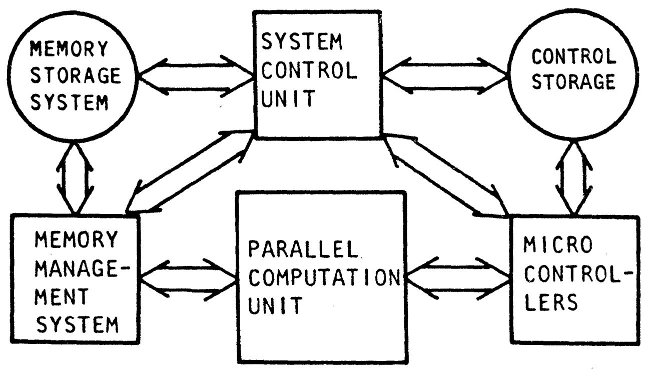


Figure 1: Block diagram overview of PASM.

This research was sponsored in part by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR 78-3581. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

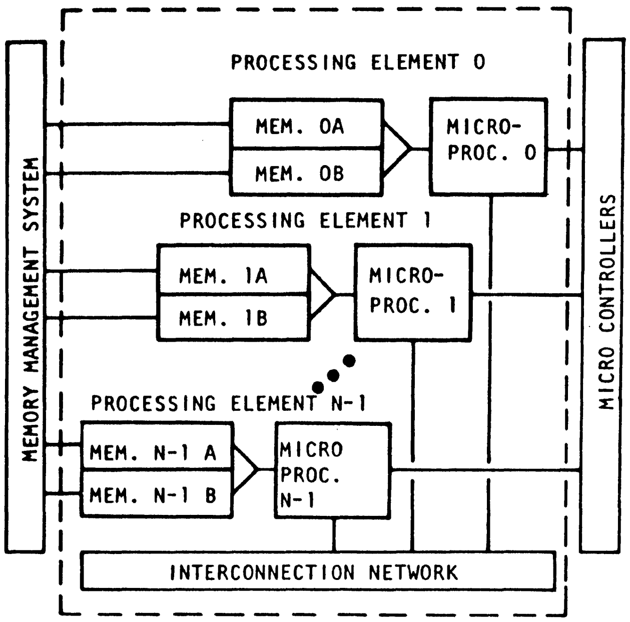


Figure 2: The Parallel Computation Unit.

memory modules, and the interconnection network. The Parallel Computation Unit processors are microprocessors that perform the actual SIMD and MIMD computations. The Parallel Computation Unit memory modules are used by the Parallel Computation Unit processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The interconnection network provides a means of communication among the Parallel Computation Unit processors and memory modules.

The Micro Controllers are a set of microprocessors which act as the control units for the Parallel Computation Unit processors in SIMD mode and orchestrate the activities of the Parallel Computation Unit processors in MIMD mode. Control Storage contains the programs for the Micro Controllers. The Memory Management System controls the loading and unloading of the Parallel Computation Unit memory modules from the Memory Storage System. 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 is organized as shown in Figure 2. A pair of memory units is used for each Parallel Computation Unit memory module so that data can be moved between one memory unit and secondary storage while the Parallel Computation Unit processor operates on data in the other memory unit. The Parallel Computation Unit processors, which are physically numbered (addressed) from 0 to N-1, where $N=2^n$, communicate through the interconnection network. The interconnection network being considered is a variation of the data manipulator [4], a multistage implementation of the "PM2I" network [20, 21, 24, 32], called the Augmented Data Manipulator (ADM) [30]. Other possibilities are cube and shuffle-exchange type networks [10, 11, 17]. Any of these interconnection networks can

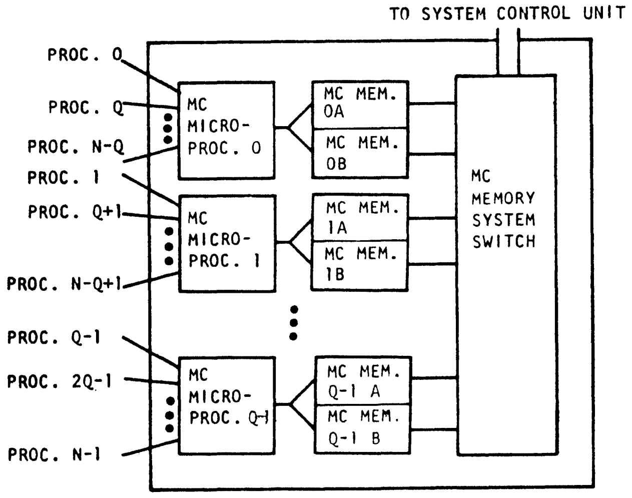


Figure 3: Organization of the Micro Controllers (MCs).

be partitioned into independent sub-networks of varying sizes, which are powers of two, if the physical addresses of the 2^p processors and memory modules in a partition have the same n-p low-order bits [30, 31].

The method used to provide multiple control units is shown in Figure 3. There are $Q=2^q$ Micro Controllers, physically addressed (numbered) from 0 to Q-1. Micro Controller i controls the N/Q Parallel Computation Unit processors whose low-order q physical address bits equal i. Each Micro Controller has a memory module which contains a pair of memory units, allowing memory loading and unloading and computations to be overlapped. A virtual SIMD machine of size RN/Q, where $R=2^r$ and $0 \leq r \leq q$, is obtained by loading R Micro Controller memory modules with the same instructions simultaneously. Similarly, a virtual MIMD machine of size RN/Q is obtained by combining the efforts of the Parallel Computation Unit processors of R Micro Controllers. For either SIMD or MIMD mode, the physical addresses of these R Micro Controllers must have the same low-order q-r bits since the physical addresses of all Parallel Computation Unit processors in a partition must agree in their low-order bit position in order for the interconnection network to function properly.

Given a virtual machine of size RN/Q, the Parallel Computation Unit processors and memory modules for this partition have logical addresses (numbers) 0 to (RN/Q) - 1, $R = 2^r$, $0 \leq r \leq q$. Assuming that the Micro Controllers have been assigned as described above, then the logical number of a Parallel Computation Unit processor or memory module is the high-order r+n-q bits of the physical number. Recall that all of the physical addresses of the processors in a partition must have the same q-r low-order bits. For example, for N = 1024, Q = 16, and R = 4, one allowable choice of Parallel Computation Unit processors to form a partition of size RN/Q is those whose physical addresses are 3, 7, 11, 15,...1023. The high-order r+n-q = 8 bits of these 10-bit physical addresses are 0, 1, 2, 3,...255, respectively. The value of the low-order q-r = 2 bits of all the physical processor addresses are equal to three.

Similarly, the Micro Controllers assigned to the partition are logically numbered (addressed) from 0 to R-1. For R > 1, the logical number of a Micro Controller is the high-order r bits of its physical number. Recall all of the physical addresses of the Micro Controllers in a partition must agree in the low-order q-r bits. For R = 1, there is only one Micro Controller and it is considered logical number 0. For example, if N = 1024, Q = 16, and R = 4, one allowable choice of four Micro Controllers is those whose physical addresses are 3, 7, 11, and 15. The high-order r = 2 bits of these four bit physical addresses are 0, 1, 2, and 3, respectively. The value of the low-order q-r = 2 bits of all the physical Micro Controller addresses are equal to three.

This brief overview of PASM is provided as background for the following sections. More details about PASM and partitionable interconnection networks can be found in [22-31].

The Memory Management System in PASM will have its own intelligence and will use the parallel secondary storage devices of the Memory Storage System. As guidelines for design purposes, it is assumed that N and Q are at least 1024 and 16, respectively. (Systems with 2^{14} to 2^{16} microprocessors have been proposed [17, 36].) Giving the Memory Management System its own intelligence will help prevent the System Control Unit from being overburdened. The parallel secondary storage devices will allow fast loading and unloading of the N double-buffered Parallel Computation Unit memory modules and will provide storage for system image and picture data and MIMD programs. The Memory Management System and Memory Storage System are described further in the following sections.

III. PASM Memory Storage System

Secondary storage for PASM's Parallel Computation Unit memory modules is provided by the Memory Storage System. The Memory Storage System will consist of N/Q independent Memory Storage units, where N is the number of Parallel Computation Unit memory modules and Q is the number of Micro Controllers in PASM. The Memory Storage units will be numbered from 0 to (N/Q)-1. Each Memory Storage unit is connected to Q Parallel Computation Unit memory units. For $0 \leq i < N/Q$, Memory Storage unit i is connected to those Parallel Computation Unit memory modules whose physical addresses are of the form:

$$(Q * i) + k, \quad 0 \leq k < Q.$$

Recall that, for $0 \leq k < Q$, Micro Controller k is connected to those Parallel Computation Unit processors whose physical addresses are of the form:

$$(Q * i) + k, \quad 0 \leq i < N/Q.$$

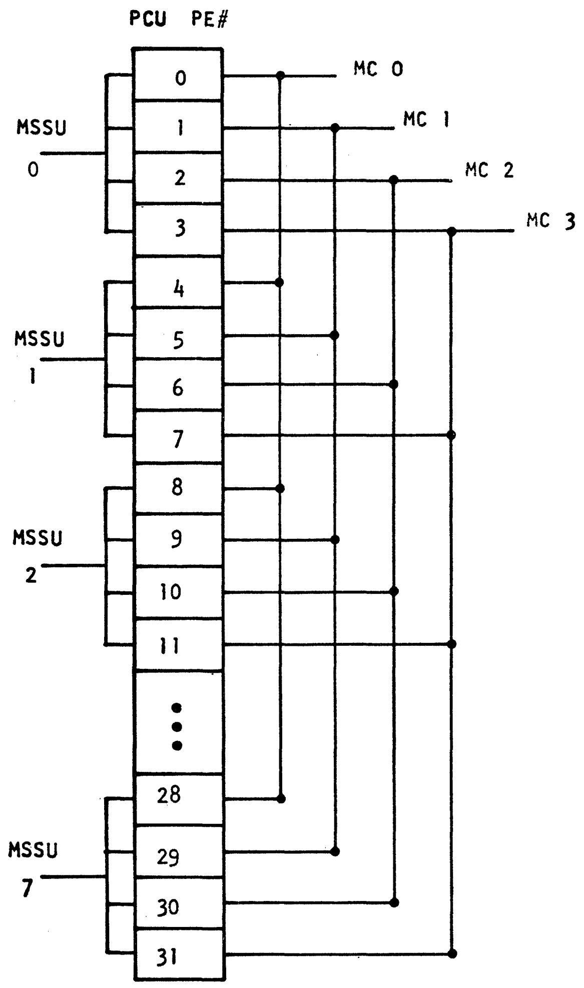


Figure 4: Organization of the Memory Storage System for N = 32 and Q = 4. MSSU is Memory Storage System unit. MC is Micro Controller. PCU PE is Parallel Computation Unit Processing Element (processor - memory module pair).

Thus, Memory Storage unit i is connected to the ith Parallel Computation Unit processor/memory module pair of each Micro Controller. This is shown for N = 32 and Q = 4 in Figure 4.

The two main advantages of this approach for a partition of size N/Q are that (1) all of the Parallel Computation Unit memory modules can be loaded in parallel and (2) the data is directly available no matter which partition (Micro Controller group) is chosen. This is done by storing the data for a task which is to be loaded into the ith Parallel Computation Unit memory module of the virtual machine of size N/Q in Memory Storage unit i, $0 \leq i < N/Q$. Memory Storage unit i is connected to the ith Parallel Computation Unit memory module in each Micro Controller group (i.e., Parallel Computation Unit memory modules Q * i, (Q * i) + 1, (Q * i) + 2,...). Thus, no matter which Micro Controller group of N/Q Parallel Computation Unit processors is chosen, the data from the ith Memory Storage unit can be loaded into the ith Parallel Computation Unit memory module of the virtual machine, for all i, $0 \leq i < N/Q$, simultaneously.

For example, in Figure 4, if the partition of size N/Q = 8 chosen consists of the Parallel Computation Unit processors connected to Micro Controller 2, then Memory Storage unit 0 would load Parallel Computation Unit memory module 2, 1 would load 6, 2 would load 10, etc. If instead Micro Controller 3's Parallel Computation Unit processors were chosen, Memory Storage unit 0 would load Parallel Computation Unit memory module 3, 1 would load 7, 2 would load 11, etc.

Thus, for virtual machines of size N/Q, this secondary storage scheme allows all N/Q Parallel Computation Unit memory modules to be loaded in one parallel block transfer. This same approach can be taken if only $(N/Q)/2^d$ district Memory Storage System units are available, where $0 \leq d \leq n-q$. In this case, however, 2^d parallel block loads would be required instead of just one.

Consider the situation where a virtual machine of size RN/Q is desired, $1 \leq R \leq Q$, and there are N/Q Memory Storage System units. In general, a task needing RN/Q Parallel Computation Unit processors, logically numbered 0 to RN/Q-1, would require R parallel block loads if the data for the Parallel Computation Unit memory module whose high-order n-q logical address bits equal i is loaded into Memory Storage unit i. This is true no matter which group of R Micro Controllers (which agree in their low-order q-r address bits) is chosen.

For example, consider Figure 4, where N = 32 and Q = 4. Assume that a virtual machine of size 16 is desired. The data for the Parallel Computation Unit memory modules whose logical addresses are 0 and 1 is loaded into Memory Storage unit 0, for memory modules 2 and 3 into unit 1, for memory modules 4 and 5 into unit 2, etc. Assume the partition of size 16 is chosen to consist of the Parallel Computation Unit processors connected to Micro Controllers 0 and 2 (i.e., all even physically numbered processors). Then the Memory Storage System units first load Parallel Computation Unit memory modules physically addressed 0, 4, 8, 12, 16, 20, 24, and 28 (simultaneously), and then load memory modules 2, 6, 10, 14, 18, 22, 26, and 30 (simultaneously). As explained in section II, given this assignment of Micro Controllers, the Parallel Computational Unit memory module whose physical address is 2 * i has logical address i, $0 \leq i \leq 16$. Assume the Parallel Computation Unit processors and memory modules associated with Micro Controllers 1 and 3 are chosen. First memory modules physically addressed 1, 5, 9, 13, 17, 21, 25, and 29 are loaded simultaneously, and then modules 3, 7, 11, 15, 19, 23, 27, and 31 are loaded simultaneously. In this case, the Parallel Computation Unit memory module whose physical address is (2 * i) + 1 has logical address i, $0 \leq i < 16$. No matter which pair of Micro Controllers is chosen, only two parallel block loads are needed.

Thus, for a virtual machine of size RN/Q, this secondary storage scheme allows all RN/Q Parallel Computation Unit memory modules to be loaded in R parallel block transfers, $1 \leq R \leq Q$. As stated above for the special case where R = 1, the same approach can be taken for R > 1 if only $(N/Q)/2^d$ distinct Memory Storage System units are available. In this situation, however, $R * 2^d$ parallel block loads would be required instead of just R.

The actual devices that will be used as Memory Storage System units will depend upon the speed requirements of the rest of PASM, cost constraints, and the state of the art of storage technology at implementation time. Possibilities to be investigated include disks, bubble memories, and CCD's.

IV. Local Variable Storage in PASM

The PASM Memory Management System makes use of the double-buffered arrangement of the Parallel Computation Unit memory modules to enhance system throughput. The scheduler, using information from the System Control Unit such as number of Parallel Computation Unit processors needed and maximum allowable run time, will sequence tasks waiting to execute [29]. Typically, all of the data for a task will be loaded into the appropriate Parallel Computation Unit memory units before execution begins. Then, while a Parallel Computation Unit processor is using one of its memory units, the Memory Management System can be loading the other unit for the next task. When the task currently executing completes, the Parallel Computation Unit processor can switch to its other memory unit for doing the next task.

Based on image processing and pattern recognition tasks which have been examined, the following conclusion has been reached. Due to the use of double-buffering, the potentially large Parallel Computation Unit memory modules, and the special purpose design of PASM, the time sharing of the Parallel Computation Unit processors and the use of conventional paging is not desirable.

There may be some cases where all of the data will not fit into the Parallel Computation Unit memory space allocated. Assume a memory frame is the amount of space used in a Parallel Computation Unit memory unit for the storage of data from secondary storage for a particular task. There are tasks where many memory frames are to be processed by the same program (e.g., maximum likelihood classification of satellite data [35]). The double-buffered Parallel Computation Unit memory modules can be used so that as soon as the data in one memory unit is processed, the Parallel Computation Unit processor can switch to the other unit and continue executing the same program. When the Parallel Computation Unit processor is ready to switch memory units, it signals the Memory Management System that it has finished using the data in the memory unit to which it is currently connected. Hardware to provide this signaling capability can be provided in different ways, such as using interrupt lines from the Parallel Computation Unit processors or by using logic to check the address lines between the Parallel Computation Unit processor and its memory modules for a special address code. After the appropriate tests to ensure that the new memory frame is available [29], the processor switches memory units. The Memory Management System can then load the "finished" memory unit with the next memory frame or next task. Such a scheme, however, requires some mechanism which can move variable length portions of programs or data sets (i.e., local data) stored in one unit of a memory module to the other unit when the associated processor switches to access the next memory frame.

Three hardware methods are considered for implementing local variable storage. Each would be used only when multiple memory frames are to be processed. The first method consists of a separate local memory allocated to each Parallel Computation Unit processor for the purpose of storing local variables. This local memory would be in addition to the processor's memory module. Such a local memory would not be affected by the changing connections of memory units associated with it. The second method would consist of splitting the local variable storage, and using a variable length portion of each memory unit as local variable storage. This scheme would require w/2 words of storage in each memory unit to implement w words of local variable storage. This specially allocated space in the memory units would be protected by hardware when the associated Parallel Computation Unit processor changes memory units. The third method stores local variables in the memory units in much the same way as method two, but in this case w words are required in each Parallel Computation Unit memory module for w words of local variable storage. This scheme preserves local variable storage by maintaining a current copy of the local variables in both memory units associated with a given Parallel Computation Unit processor.

Of the three methods described above, method one is the least flexible since it requires a fixed amount of memory to be dedicated to local variable storage at all times. This method may tend to utilize inefficiently the special local variable memories it requires since these memories will have to be large enough to handle tasks which may require amounts of local variable storage many times greater than that of a typical job. For example, a task may require that a portion of a reference image be stored within the local variable storage space. Such a task might be executed infrequently but would require a relatively large amount of local variable storage space. Other tasks run by PASM might be executed far more frequently but would require far less local variable storage space than the reference image example above. The result is that while the tasks requiring a small portion of the available local variable storage space are being run, the bulk of the available local variable storage space is not utilized. Furthermore, if a task requires more local variable storage than expected (i.e., more local variable storage than the fixed size dedicated memory has space for), a problem arises which will require additional hardware and/or software overhead to solve.

The second method described above makes the most efficient use of the memory space available for local variable storage in that for w words of local variable storage required, only w words of actual memory space are used. Since w would be variable, only the amount of local variable storage space required by a given task would need to be allocated to the task. This method, however, has several inherent disadvantages. First, when a Parallel Computation Unit processor executing a given task begins processing the last memory frame associated with the task, the Memory Management System will normally load the inactive memory unit with data for the next task to be run. If the local variable storage system is in use however, the next task cannot be loaded into the memory unit since the w/2 words of local variable storage in the inactive memory unit must be preserved until the current task is complete. A second disadvantage of this method is that the Parallel Computation Unit processor addresses which access local variable storage stored in the inactive memory unit must be translated to properly address the local variable storage in this memory unit. Such address translation is likely to require additional hardware and may cause additional delay in address decoding.

The third method described above would make less than optimal use of the space allocated for local variable storage in the Parallel Computation Unit memory module (2w words of the memory module are needed for w words of local variable storage), but it does not require the address translation of method two and provides much more flexibility than method one. It also eliminates the problems encountered in method two when a new task is loaded into a memory unit which contains local variable storage associated with a previous task. This method maintains a copy of local variables in both memory units associated with a given Parallel Computation Unit processor so that switching memory units does not alter the local variable storage associated with the processor. The implementation of variable size local variable storage for this method is simpler and more straightforward than that of method two above since the total address space for a single processor is fixed at the size of a single memory unit. In method two, the total address space would be the fixed size of a memory unit plus w/2. More image processing and pattern recognition algorithms suitable for implementation on PASM need to be studied to determine if the efficiency gained by optimal utilization of memory space in method two will be significant enough to offset the problems associated with this method. Currently, method three appears to be the most promising.

One possible hardware arrangement to implement method three is described below. The arrangement makes use of two characteristics of the PASM memory access requirements:
1) secondary memory will not be able to load a given memory unit at the maximum rate it can accept data, and
2) Parallel Computation Unit processors will not often be able (or desire) to write to memory on successive memory cycles.
Because of these two characteristics, Parallel Computation Unit processor stores to local vari-

able storage locations in an active memory unit can be trapped by a bus interface register and stored in the inactive memory unit by stealing a cycle on the secondary memory bus. In essence, this technique makes use of the conventional store-through concept as described in [7, 14].

An exception to the second characteristic mentioned above is multiple precision data. If 16 bit words are assumed, then for higher precision it may be desirable to use two or four words as a group. However, a simple buffering scheme can handle this possibility.

The amount of memory allocated as local storage is determined by the contents of a k-bit base register. This register may be altered by the Memory Management System. If 2^p locations are available in each memory unit for Parallel Computation Unit processor use, local storage can be allocated in blocks of 2^{p-k} words. B blocks, $1 \leq B \leq 2^k$, can be allocated for local storage by storing B in a base register. This has the effect of allocating all memory locations from 0 to $B2^k-1$ as local storage. When a processor writes to a local variable location a k bit block comparator causes the memory address and data being written to be trapped by a bus interface register. A cycle request flip-flop is set to indicate to the logic which controls the buses associated with the Parallel Computation Unit memory module that a cycle is needed on the secondary memory bus. When the cycle is granted, the flip-flop is reset and the data in the bus interface register is gated into the inactive memory unit. In this way, the space allocated for local variable storage remains updated in both memory units at all times. It is assumed that the bus interface register will have maximum priority for secondary memory bus usage since this would prevent the processors in the Parallel Computation Unit from having to wait to write to a location designated for use as local storage.

The method described above is applicable to any system which allows its processing tasks to utilize several separate memories and which requires that identical copies of variable amounts of certain data be maintained in all memories so used.

V. Altering Loading Sequences

To further increase the flexibility of PASM, a task may alter the sequence of data processed by it during execution. As an example, consider a task which is attempting to identify certain features within a series of images. The task might examine a visible spectrum copy of an image and, based on features identified within the image, choose to examine an infrared spectrum copy of the same image. Rather than burden the System Control Unit to perform data loading sequence alterations, the task is allowed to communicate directly with the Memory Management System.

In the case of an SIMD task, the associated Micro Controller(s) determines if changes are required in the data loading sequence for the task. If so, a Micro Controller specifies the nature of the changes and communicates them to the Memory Management System without involving the System Control Unit. Each Micro Controller in the PASM system has the capability to generate loading sequence changes. For tasks which require R Micro Controllers ($1 \leq R \leq Q$), logically numbered 0 to R-1, control instructions exist so that logical Micro Controller 0 will handle loading sequence changes. Micro Controller 0 uses logical Parallel Computation Unit processor number 0 of the virtual machine to establish a control information list in logical Parallel Computation Unit memory module 0. (There are Q Parallel Computation Unit processors which can possibly be logically numbered 0 in a virtual machine. They are those Parallel Computation Unit processors which are physically numbered 0, 1, 2,...,Q-1.) This list specifies in a concise fashion the loading sequence alterations required and includes information such as the IDs of the data files to be loaded, the Parallel Computation Unit memory modules which are to receive the data, and the locations within the Parallel Computation Unit memory modules where the data is to be loaded. The Micro Controller initiates the transfer of this list to the Memory Management System by using logical Parallel Computation Unit processor 0 to write a pointer to the list into the highest addressable memory location of its memory module. Through the use of a simple address comparison, the write into this memory location generates an interrupt to the Memory Management System. The Memory Management System recognizes the interrupt as a request for a loading sequence change and determines which Micro Controller is making the request. The Memory Management System uses the list of control information (via the pointer provided) to determine the loading sequence changes required.

An alternative method of interrupt generation is to use an interrupt line from each of the Q possible logical Parallel Computation Unit processor 0's to the Memory Management System. The method selected for interrupt generation will depend upon the interrupt capabilities of the microprocessor used in the Parallel Computation Unit. While loading sequence control information could be passed directly from the Micro Controllers to the Memory Management System, the length of the connections required may make implementation more difficult and costly.

One hardware scheme which can transfer the control information list from a Parallel Computation Unit memory module to the Memory Management System is shown in Figure 5. The hardware system shown is based on having the Memory Management System coordinate the recognition of Micro Controller interrupts and the associated transfers of control information lists from the Parallel Computation Unit memory modules. The interrupt recognition portion is handled by the Parallel Computation Unit processor Interrupt Conrol Logic while the transfer of control information lists is handled by the Parallel Computation Unit memory module Access Control Logic.

Consider the following example in a virtual machine whose processor logically numbered 0 is physically numbered i. Suppose processor i establishes a control information list in one of

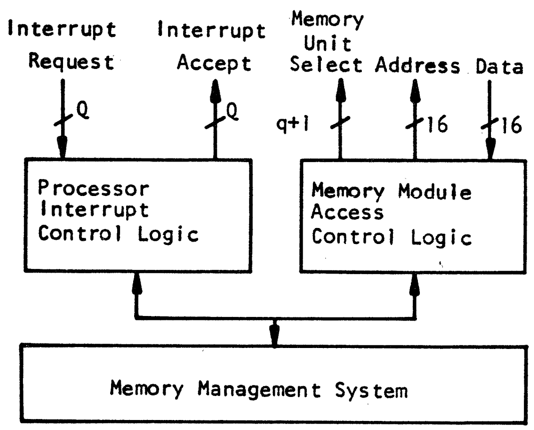


Figure 5: Hardware scheme for dynamically altering the loading sequence of the memory modules.

its memory units and writes a pointer to the list into its corresponding interrupt generation location. The memory write to the interrupt generation location is signaled to the Interrupt Control by a pulse on the Interrupt Request Line corresponding to processor i. This pulse causes the Interrupt Control to signal the Memory Management System that processor i has generated an interrupt to the Memory Management System. The Memory Management System then uses the Access Control to read the interrupt generation location in the Parallel Computation Unit memory module to obtain the pointer to the control information list. The control information list is then read from the Parallel Computation Unit memory module by the Memory Management System. Finally, the Memory Management System signals the Interrupt Control to generate a pulse on the Interrupt Accepted Line to processor i.

The same hardware arrangement described for SIMD tasks is used for MIMD tasks. With each group of N/Q MIMD processors, there is associated a memory supervisor which is logical processor 0 within the group. The memory supervisor possesses the hardware for Memory Management System interrupt generation and loading sequence alterations using the same arrangement described for SIMD mode. All processors associated with a give memory supervisor make requests for loading sequence changes through the memory supervisor, without involving the Micro Controllers or System Control Unit. This reduces System Control Unit contention problems, as mentioned above, and helps prevent the Micro Controller(s) orchestrating the virtual MIMD machine from becoming overburdened.

The scheme described here is well suited to parallel computer systems which execute multiple parallel tasks since it can easily keep track of and arbitrate multiple requests for data loading sequence alterations. This technique also makes efficient use of the multiple memory arrangement of PASM by using the hardware structure of the memory system to provide for communication of loading sequence alteration information from the memory system to the controller which loads data into the memory system.

VI. PASM Memory Management System

Tasks for which the Memory Management System is responsible include file system maintenance, scheduling of Parallel Computation Unit memory module loading and unloading, and Memory Storage System bus control. A set of microprocessors are dedicated to performing the Memory Management System tasks in a distributed fashion, i.e. one processor will handle Memory Storage System bus control, one will handle the scheduling tasks, etc. This distributed processing approach is chosen in order to provide the Memory Management System with a large amount of processing power at low cost. In addition, dedicating specific microprocessors to certain tasks simplifies both the hardware and software required to perform each task.

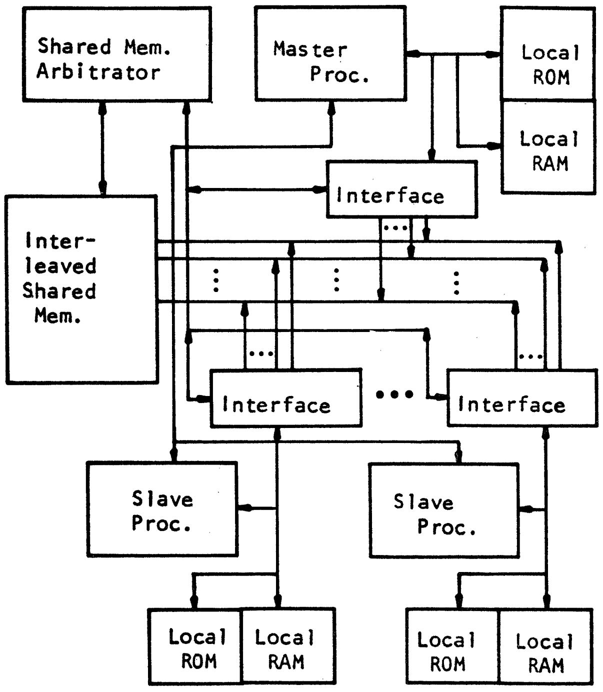


Figure 6: Distributed Memory Management System.

The basic architecture of the Memory Management System is shown in Figure 6. The Memory Management System consists of a master processor which coordinates the concurrent tasks executed by the slave processors, a shared memory for storage of data required by more than one processor, a local ROM and RAM for each processor for storage of code and local data respectively, and an interface to the shared memory for each processor.

A shared memory approach is used to allow the processors to communicate with each other and to share data. This approach is planned due to the need to share relatively large quantities of data such as file tables and task queues. As an example, consider a queue of Memory Storage System to Parallel Computation Unit memory module data transfer operations pending. This queue would need to be accessible to both the processor in charge of the Memory Storage System bus system and the processor in charge of scheduling such transfers.

To reduce contention for the shared memory, each processor uses a local ROM and RAM for storage of code and local data. In addition, the shared memory may be interleaved [14] to further reduce contention. The degree of interleaving desirable may be determined by simulation studies or queuing theory analysis [6] of the Memory Management System.

The processors within the Memory Management System may be implemented using commercially available fixed instruction set microprocessors. The new generation of 16-bit processors [15, 18, 19, 33] are particularly attractive since many provide special hardware for operations such as locked increment and test, memory protection and management, and problem/supervisor state switching. Features such as these would considerably simplify the hardware and software design of the Memory Management System. An alternative to the 16-bit processors are the less expensive 8-bit microprocessors currently available [8, 37]. The choice of a processor type will be governed by the amount of processing required to perform the tasks associated with the Memory Management System and the cost trade-offs involved.

The division of tasks chosen is based on the main functions which the Memory Management System must perform. The functions to be performed include:
1) communication with the System Control Unit and generating slave tasks based on Parallel Computation Unit memory module load/unload requests from the System Control Unit,
2) interrupt handling and generating slave tasks for data loading sequence changes requested by the Parallel Computation Unit processors physically numbered 0 to Q-1 (see previous section),
3) scheduling of Memory Storage System data transfers,
4) control of input/output operations involving peripheral devices and the Memory Storage System,
5) control and maintenance of the Memory Management System file directory information and the creation and deletion of data files, and
6) control of the Memory Storage System bus system.

Most Memory Management System operations will be initiated by the System Control Unit since it will be responsible for coordinating the operation of the PASM system. For this reason, the master processor is chosen to communicate with the System Control Unit and to perform the task spawning operations associated with System Control Unit requests.

Parallel Computation Unit processor interrupt handling is assigned to one slave processor. This slave sends requests for Parallel Computation Unit memory module data loading sequence changes to the master processor.

Scheduling of all Memory Management System operations involving data transfers using the Memory Storage System bus system is assigned to another slave processor. One slave processor is devoted solely to performing scheduling operations since the scheduling of data transfers will be complex and time consuming if near optimal operation of this system is to be realized.

Another slave is devoted to handling input/output between the Memory Storage System and peripheral devices such as magnetic tape units and color video displays. This slave would handle any communications with the peripheral devices and schedule access to the Memory Storage units.

The control and maintenance of the Memory Management System file system is assigned to one or more slave processors. To understand why multiple slave processors may be required, consider the configuration of the Memory Storage System. It will consist of N/Q secondary storage devices which operate in a parallel fashion. The secondary storage devices will be required to locate and transfer data files based on file IDs presented to the Memory Management System. For the suggested values of N=1024 and Q=16, a total of 64 secondary storage devices may be involved in transferring data files at any given time. It is apparent that the file location operations associated with this many devices will exceed the processing capabilities of one slave processor. The exact number of slave processors to be devoted to file directory maintenance will be determined by simulation and/or queuing theory analyses of the Memory Storage System system. Another possibility is to assign a microprocessor to each Memory Storage System unit for file directory maintenance (e.g. intelligent disks), and have a single slave coordinate this activity.

A slave processor is devoted to performing the operations associated with the configuration and control of the Memory Storage System bus system. This would involve setting the control signals needed to connect each Memory Storage System unit to the appropriate Parallel Computation Unit memory module.

The hardware structure of the Memory Management System is such that additional slave processors may be added to perform tasks that are not considered to be part of the Memory Management System processing load at this time. In an actual prototype Memory Management System, interfaces for additional slave processors would be provided to facilitate system expansion and the incorporation of new features into the Memory Management System.

VII. Conclusions

An overview of PASM, a partitionable SIMD/MIMD system for image processing and pattern recognition being designed at Purdue University, was given. To improve the throughput of this large-scale dynamically reconfigurable multimicroprocessor system, a highly parallel memory system was described. The memory system uses double-buffered primary memories, parallel secondary memories, and a set of dedicated microprocessors. The organization of this memory system was presented and its advantages were discussed.

Acknowledgements

The authors thank L. J. Siegel, R. J. McMillen, and P. T. Mueller, Jr., for their comments and suggestions.

References

[1] G. Barnes, et al., "The Illiac IV computer," IEEE Trans. Comp., Vol. C-17, No. 8, Aug. 1968, pp. 746-757.

[2] W. J. Bouknight, et al., "The Illiac IV system," Proc. IEEE, Vol. 60, Apr. 1972, pp. 369-388.

[3] F. Briggs, K. S. Fu, K. Hwang, and J. Patel, "PM4 - a reconfigurable multimicroprocessor system for pattern recognition and image processing," Nat'l. Comp. Conf., June 1979, pp. 255-265.

[4] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comp., Vol. C-23, No. 3, Mar. 1974, pp. 309-318.

[5] M. J. Flynn, "Very high-speed computing systems," Proc. IEEE, Vol. 54, Dec. 1966, pp. 1901-1909.

[6] F. T. Fung and H. C. Torng, "Analysis of memory conflicts in a multiple microprocessor system," IEEE Trans. Comp., Vol. C-28, No. 1, Jan. 1979, pp. 28-37.

[7] J. Hayes, Computer Architecture and Organization, McGraw-Hill, New York, 1978.

[8] Intel, Component Data Catalog, Intel Corp., 1978.

[9] J. Keng and K. S. Fu, "A special computer architecture for image processing," 1978 IEEE Comput. Soc. Conf. Pattern Recognition and Image Processing, June 1978, pp. 287-290.

[10] T. Lang and H. S. Stone, "A shuffle-exchange network with simplified control," IEEE Trans. Comp., Vol. C-25, No. 1, Jan. 1976, pp. 55-65.

[11] D. H. Lawrie, "Access and alignment of data in array processor," IEEE Trans. Comp., Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.

[12] G. J. Lipovski, "On a varistructured array of microprocessors," IEEE Trans. Comp., Vol. C-26, No. 2, Feb. 1977, pp. 125-138.

[13] G. J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor," 1977 Int'l. Conf. Parallel Processing, Aug. 1977, pp. 165-174.

[14] R. E. Matick, "Memory and storage," in Introduction to Computer Architecture, H. S. Stone, editor, Science Research Associates, Chicago, Illinois, 1978.

[15] S. P. Morse, W. B. Pohlman, and B. W. Ravenel, "The Intel 8086 microprocessor: a 16-bit evolution of the 8080," Computer, Vol. 11, No. 6, June 1978, pp. 18-27.

[16] G. J. Nutt, "Microprocessor implementation of a parallel processor," 4th Annual Symp. Comp. Arch., Mar. 1977, pp. 147-152.

[17] M. C. Pease, "The indirect binary n-cube multiprocessor array," IEEE Trans. Comp., Vol. C-26, No. 5, May 1977, pp. 458-473.

[18] B. L. Peuto, "Architecture of a new microprocessor," Computer, Vol. 12, No. 2, Feb. 1979, pp. 10-21.

[19] M. Shima, "Two versions of 16-bit chip span microprocessor, microcomputer needs," Electronics, Vol. 51, No. 26, Dec. 1978, pp. 81-88.

[20] H. J. Siegel, "Single instruction stream - multiple data stream machine interconnection network design," 1976 Int'l. Conf. Parallel Processing, Aug. 1976, pp. 273-282.

[21] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," IEEE Trans. Comp., Vol. C-26, No. 2, Feb. 1977, pp. 153-161.

[22] H. J. Siegel, "Preliminary design of a versatile parallel image processing system," 3rd Biennial Conf. on Computing in Indiana, Apr. 1978, pp. 11-25.

[23] H. J. Siegel, "Partitionable SIMD computer system interconnection network universality," 16th Annual Allerton Conf. on Communication, Control, and Computing, Oct. 1978, pp. 586-595.

[24] H. J. Siegel, "Interconnection networks for SIMD machines," Computer, Vol. 12, No. 6, June 1979, pp. 57-65.

[25] H. J. Siegel, "Partitioning permutation networks: the underlying theory," 1979 Int'l. Conf. Parallel Processing, Aug. 1979.

[26] H. J. Siegel, R. J. McMillen, and P. T. Mueller, Jr., "A survey of interconnection methods for reconfigurable parallel processing systems," Nat'l. Comp. Conf., June 1979, pp. 529-542.

[27] H. J. Siegel and P. T. Mueller, Jr., "The organization and language design of microprocessors for an SIMD/MIMD system," 2nd Rocky Mt. Symp. on Microcomputers, Aug. 1978, pp. 311-340.

[28] H. J. Siegel, P. T. Mueller, Jr., H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor systems," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 9-17.

[29] H. J. Siegel, L. J. Siegel, R. J. McMillen, P. T. Mueller Jr., and S. D. Smith, "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," 1979 IEEE Comp. Soc. Conf. Pattern Recognition and Image Processing, Aug. 1979.

[30] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," 5th Annual Symp. Comp. Arch., Apr. 1978, pp. 223-229.

[31] S. D. Smith and H. J. Siegel, "Recirculating, pipelined, and multistage SIMD interconnection networks," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 206-214.

[32] S. D. Smith and H. J. Siegel, "An emulator network for SIMD machine interconnection networks," 6th Int'l. Symp. Comp. Arch., Apr. 1979.

[33] E. Stritter and T. Gunter, "A microprocessor architecture for a changing world: the Motorola 68000," Computer, Vol. 12, No. 2, Feb. 1979, pp. 43-52.

[34] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: a modular, multimicroprocessor," Nat'l. Computer Conf., June 1977, pp. 645-655.

[35] P. H. Swain, H. J. Siegel, and B. W. Smith, "A method for classifying multispectral remote sensing data using context," Symp. on Machine Processing of Remote Sensing Data, June 1979, pp. 343-353.

[36] Sullivan, H., Bashkow, T. R., and Klappholz, K., "A large scale homogeneous, fully distributed parallel machine," 4th Symp. Annual Comp. Arch., Mar. 1977, pp. 105-124.

[37] J. F. Wakerly, "Intel MCS-48 microcomputer family: a critique," Computer, Vol. 12, No. 2, Feb. 1979, pp. 22-31.

[38] W. A. Wulf and C. G. Bell, "C.mmp - a multi-miniprocessor," Proc. FJCC, Dec. 1972, pp. 765-777.

ADAPTABLE PIPELINE SYSTEM WITH DYNAMIC ARCHITECTURE

Svetlana P. Kartashev
University of Nebraska-Lincoln,

and

Steven I. Kartashev
Dynamic Computer Architecture, Inc.

ABSTRACT - The paper describes an adaptable pipeline system with dynamic architecture. The system performs the following pipeline adaptations toward executed programs: 1) adaptation on the number of pipeline stages when each instruction activates the number of stages in the pipeline which matches the number of operations it realizes: 2) adaptation on operation sequences when pipeline may execute any sequence of operations requiring no reconfiguration and thus no time overhead caused by this reconfiguration; 3) adaptation on operation time in each stage when each stage of the pipeline may execute an operation during minimally required time, because it may change the time of operation. The paper shows that such a pipeline may be organized from DC groups and thus amenable to LSI implementation.

INTRODUCTION

Pipelining is an attractive choice for a faster computation. However since pipeline computations are dedicated, systems in which pipelines are used are confronted with problems of time overheads caused by disparity between the pipeline(s) and executing algorithms. To broaden their cost effective applications, pipeline systems employ various reconfiguration techniques. General ideas of such reconfigurations is to reconfigure the system via software in order to reduce dissimilarity between the system and the program.

Consider how existing systems use reconfiguration. All pipeline systems are divided into two categories: unifunctional and multifunctional. A unifunctional pipeline executes a single dedicated sequence of operations [1-3]. A multifunctinal pipeline system may execute several sequences of operations either in parallel or sequentially [3-8]. For multifunctional pipeline systems, each allowable sequence of operations is executed in the pipeline configured into a matching sequence of operational units. This means that to perform transition from one sequence(s) of operation to another, the system has to reconfigure. Since during this reconfiguration no processing is performed, the time of reconfiguration is a pure time overhead which has to be minimized. To organize pipelined computations, the tasks requiring the same configuration are grouped together. On completing their execution in one configuration the system reconfigures and starts execution of a new block of tasks in the next configuration.

It then follows that one of the major drawbacks of existing reconfigurable pipeline systems are their inability of instantaneous transition from one sequence of operations to another. Furthermore, since many programs usually have different sequences of operations following each other, pipeline type of computations may be used cost-effectively only in a limited sense. Thus a limited applicability is the most severe drawback of existing pipeline systems. Their other drawbacks are associated with the following causes:

(1) Frequent disparity between the number of consecutive operations in the instruction and the number of pipeline stages in the pipeline which processes this instruction. This creates additional delays (dummy time intervals) associated either with instruction propagation through unneeded stages or with conflict resolution when the instruction bypasses the unneeded stages and encounters operands prepared for some of its predecessors [9].

(2) In existing pipelines the time for non-iterative processor dependent operations (addition, subtraction, $>$, etc.) is permanent and does not depend on operand sizes. However, selection of a permanent operation time in each stage requires that it be selected as the time of the longest operation (addition handling maximal word sizes). It then follows that all faster operations (processor dependent operations handling smaller word sizes or Boolean operations, etc.) are slowed down because they are executed during the time of the longest processor dependent operation.

(3) A significant additional waiting time occurs in existing pipelines when the computational result obtained for one instruction in one stage is required as an operand for another instruction in another stage. For instance, if the instruction executes $A \cdot B+(C^2-D^2)$ the temporary result $A \cdot B$ can not be sent to the memory of some pipeline stage which might need it in future computations. Instead, all temporary results needed by a pipeline stage has to be grouped together and then sent to its memory. Or, in order not to waste time on grouping and transferring blocks of information, the needed temporary results are repeatedly computed.

MOTIVATION

As follows from the shortcomings of existing reconfigurable pipeline systems, a further performance improvement can be accomplished if a pipeline system is provided with the following architectural solutions:

(a) the system minimizes the time overhead caused by reconfiguration from one pipeline to another,

(b) it equips each pipeline with variable number of stages. This means that each instruction must activate the number of stages in a pipeline which matches the number of consecutive operations it implements, i.e., an instruction implementing w operations has to pass through w stages and complete its execution.

(c) the system provides each pipeline stage with the variable time interval for operation. Then each stage will be capable of generating a minimal operation time specified either by the sizes of operands for processor-dependent operations or small permanent duration for processor independent operations. Consequently the pipeline equipped with variable time interval in each stage will be capable of working at a variable rate and fan out results much faster if it is filled with short operations.

(d) the system provides for fast exchanges of temporary results between pipeline stages. This will allow one-line feeding of temporary results obtained in one stage to other pipeline stages that will need them in the future.

The architectural solutions which provide pipeline systems with these desirable characteristics can be obtained if a pipeline system is assembled from Dynamic Computer groups (DC-group) discussed in [10, 11]. Such system was called a dynamic pipeline architecture. General concepts of a dynamic pipeline assembled from DC-groups were introduced in [12].

Given paper is dedicated to their further development. It introduces new research results on the following subjects:

(1) organization of a pipeline stage capable to generate minimal time intervals for all its operations;

(2) organization of effective information exchanges between pipeline stages;

(3) addressing procedures which allow on line sending of temporary results to the destination pipeline stages.

DYNAMIC PIPELINE: GENERAL CONCEPTS

Hardware resource for a dynamic pipeline may be organized as follows [12]. It includes a single computer supervisor, C_0, and several k·h-bit computers, C_1, C_2, . . ., C_F, forming consecutive pipeline stages (Fig. 1). Computer C_0 stores instructions in memory M_0 and fetches them to processor P_0. The C_0 computer's size matches that of one instruction. Each pipeline stage C_i has memory M_i for storing initial data and addresses, processor P_i, and general register set m_i, which stores temporary results required by the P_i processor. These are either computed by P_i or by other processors.

Each connecting element MSE_i separates two pipeline stages C_i and C_{i+1} and may assume two modes of transfer: right and no transfer. For the right transfer, MSE_i transfers the pipelined instruction to the next right stage C_{i+1} with delay of one interval. For the

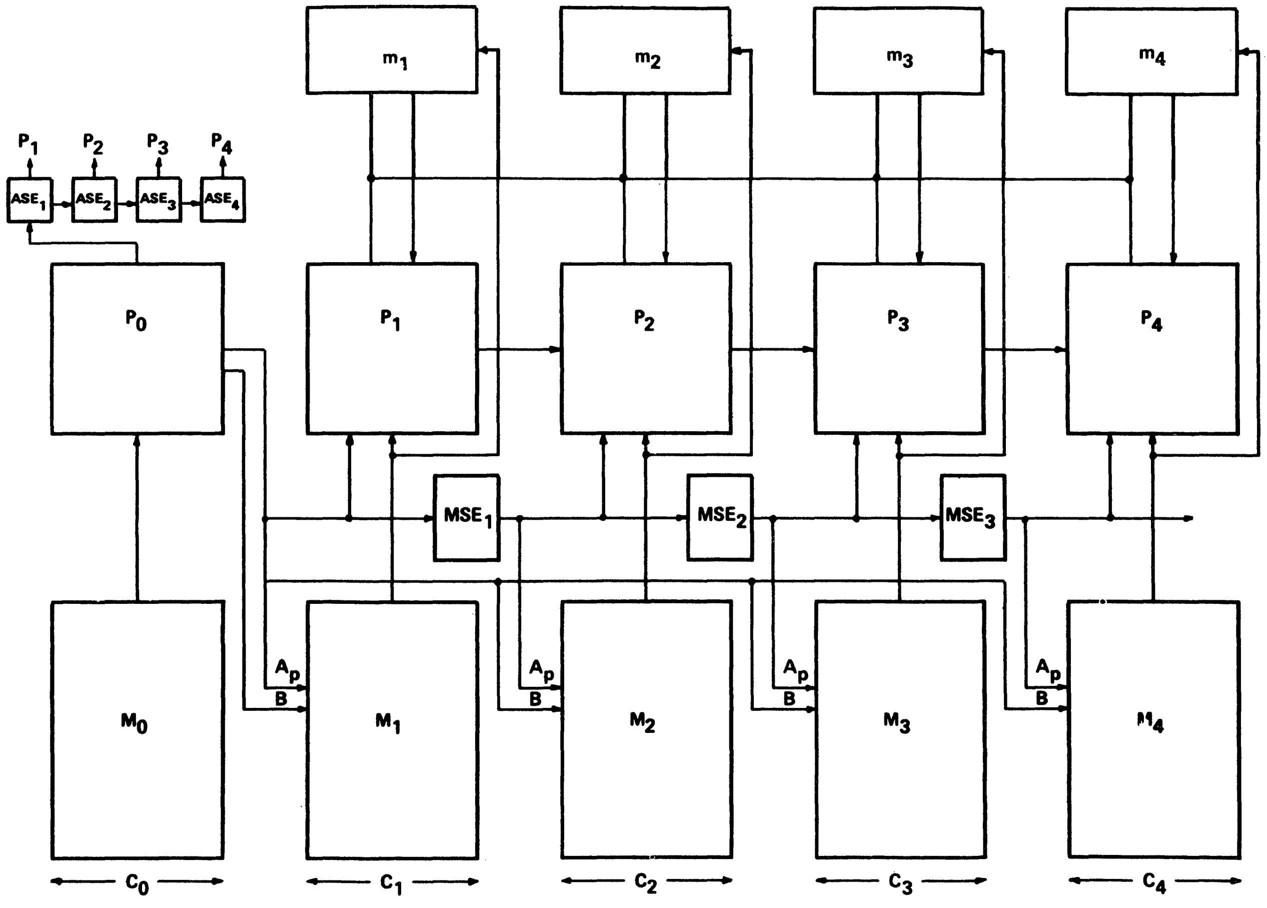


FIGURE 1
Hardware diagram of a dynamic pipeline

no transfer, the instruction received by MSE_i is not transferred to the next C_{i+1}. The MSE element is implemented as a universal module UM equipped with the modular control organization [10, 11].

Address elements $ASE_1, \ldots, ASE_F$, are connected into a shifting sequence, and transfer address portion of the program instruction (AI instruction). Each ASE is an 8-bit microprocessor, UM, equipped with two levels of memory ME-1 and ME-2. These memories store constants which may modify codes and addresses stored in AI instructions. Each time a pipeline stage C_i begins execution of the operation assigned to it by the program instruction, the address element ASE_i, with the same position i, receives the AI instruction shifted from ASE_{i-1}. It then follows that ASE_i is synchronized by stage C_i and stores AI instruction during the time C_i executes operation. If the result of this operation is a future temporary result for another stage C_j, it is written to its general register set m_j. To do this the ASE_i sends the address for the m_j set to stage C_i, which thereafter broadcasts the computational result to the destination. Thus the number of MSE elements is F-1 and the number of ASE elements is F, where F is the number of pipeline stages.

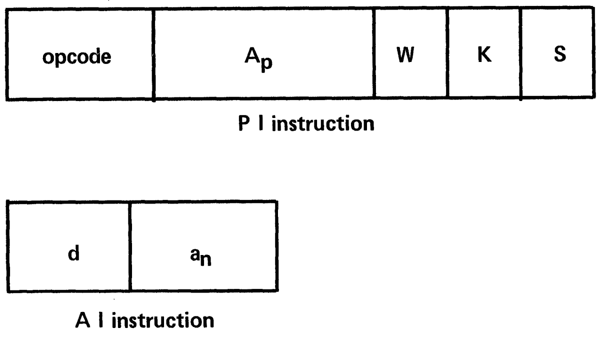


FIGURE 2

Formats of PI and AI instruction

Each instruction fetched from M_0 to P_0 includes two portions: the pipeline portion, PI, the address portion, AI (Fig. 2). By passing through the bus made of MSE connecting elements, the pipeline portion, PI, propagates through consecutive pipeline stages with a delay of one interval, causing execution of an operation assigned to each stage. Concurrently, the address portion, AI, of the instruction propagates through the bus made of ASE elements and specifies a pipeline stage, C_i, which should output the result, and a general register set, m_j, which should receive this result.

FORMATS OF THE PI AND AI INSTRUCTIONS

In order to adapt to the operation, it is necessary that each PI store its own opcode D. However, when the same D propagates through the pipeline stages, it will activate the same operation in each stage. In order that each stage, C_i, execute an individual operation, C_i should store a position code, d_i which is the binary value of its position within the pipeline. For instance for the C_1 stage, d_1 = 001, for C_2 stage d_2 = 010, etc. These two codes, D and d_i achieve the selective activation of an operation assigned to stage C_i by instruction PI.

Adaptation to the length of the pipeline is performed with another code, w, which shows the number of consecutive pipeline stages which execute instruction PI. Clearly w matches the number of consecutive operations which are realized in PI. By propagating through each connecting element, MSE_i, containing position code d_i, w is compared with d_i. If $d_i < w$, MSE_i propagates this PI instruction to the next stage with delay of one time interval. If $d_i \geqslant w$, MSI_i blocks further instruction execution. Therefore, by writing a code w into the PI instruction field, one may select the number of pipeline stages required by a single instruction.

This technique allows one to perform a simple adaptation of the number of pipeline stages that requires no bypassing of unneeded stages and no conflict resolution associated with such bypassing. The entire problem is solved by the code w with size w = log F bits, where F is the maximal length of the dynamic pipeline.

Adaptation of each stage to operation time is performed with another code, k, stored in the PI instruction. Since each LSI module of processor P_i is equipped with the same modular control organization [10, 11, 13] this allows one to organize variable time intervals for any processor dependent operation. Therefore three codes, D, w, and k, effect a pipeline's adaptation to the operation, to the length of the pipeline, and to the time of operation executed by each pipeline stage.

In addition to adaptation codes, each PI instruction stores the relative address, A_p, of M_i memories where each memory M_i stores a word required by the C_i stage or the address of m_i register set. Using this address, a second operand is fetched to P_i from m_i. A special tag bit, in the cell accessed by A_p address recognizes whether the second operand is stored in M_i or m_i. In order not to have any limitations on the height of memory M_i and not to increase the bit size of PI, it is assumed that A_p is a relative 8-bit address. The effective address E (24 bits) is formed from a concatenation of the base address B (16 bits) and address A_p (8 bits). Base address B is fed continuously to all M_i memories during the execution of one task. It is changed when data words have to be fetched from a new page. Such organization provides that each M_i memory may contain 2^{24} words. Finally, the PI instruction stores the S code which shows the registers of the P_i processor to be connected with the adder inputs and the destination of the computational result if other than the next stage.

The AI instruction stores position code d of a pipeline stage which should receive a temporary result produced by some other stage and the address a_n of m register set where the result should be written. In each ASE element, d and a_n values may be modified by addition with constants stored in ME memory assigned to ASE. This allows a programmer to send every temporary result computed by PI instruction in each pipeline stage it propagates to any m register set. Thus every pipeline stage may be provided with all temporary results it may need in the future.

ORGANIZATION OF A PIPELINE STAGE

Consider now organization of a single pipeline stage. Each stage is assembled from one DC-group having n computer elements, CE [10, 11]. Each CE processes h-bit words. Then a pipeline stage C_i may have its processor P_i assuming the following word sizes: h, 2h, . . ., n·h. If h = 16, n = 4, P_i ranges from 16 to 64 bits in 16 bit increments, i.e., it assumes 16, 32, 46, 64 bit sizes

A pipeline stage C_i is equipped with the general register set m_i storing temporary results and partitioned into two levels m_i (1) and m_i (2) (Fig. 3). The reason for this is that the same m_i register set may be accessed twice during one time interval: when an operand is fetched and when the temporary result is

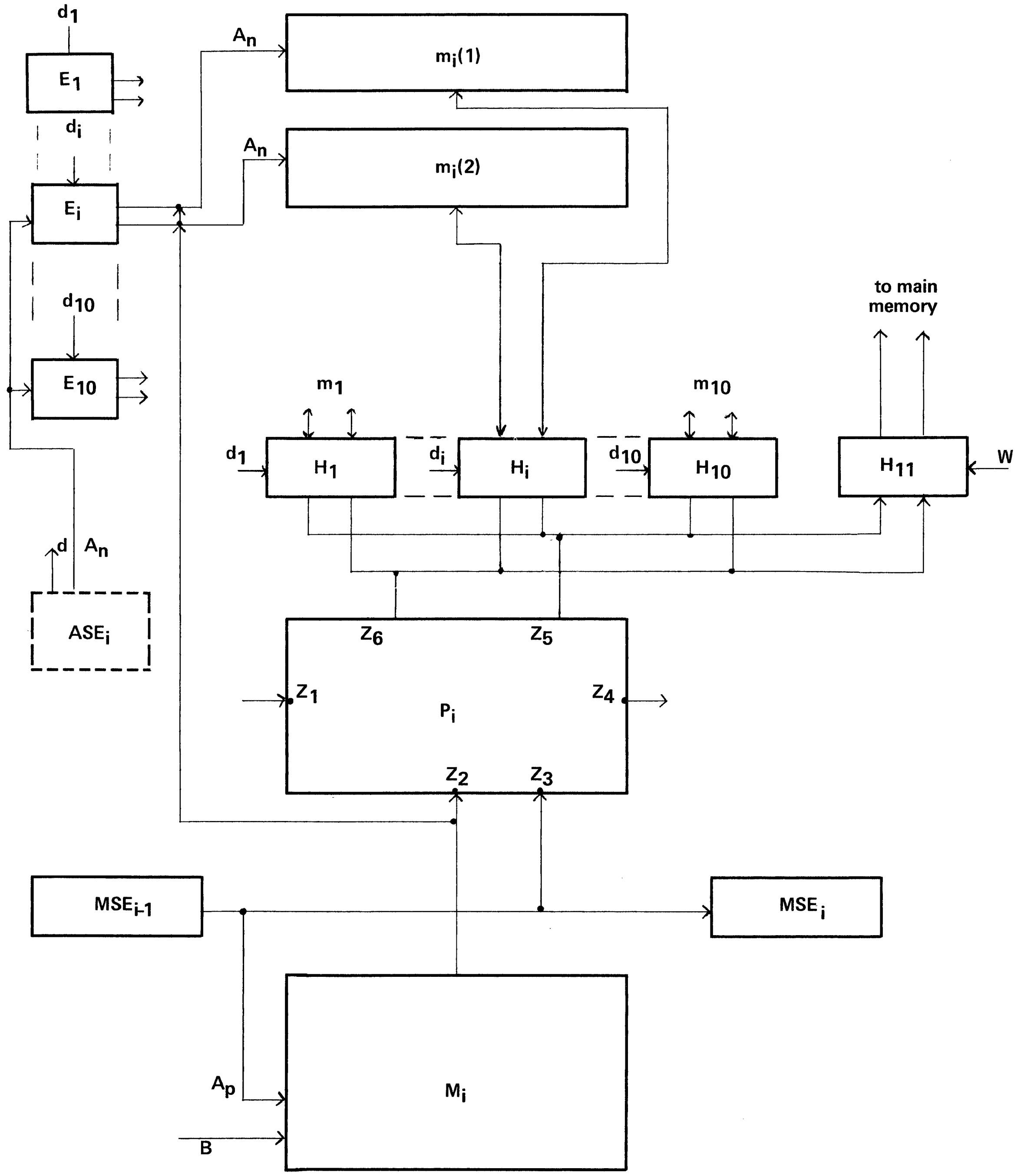


FIGURE 3

Hardware diagram of one stage

written. To perform these accesses in parallel takes two levels of memory so that if one access is performed over one level, then the second one is performed over another level.

Since each pipeline stage C may send its computational result to any m register set the main memory is provided with F + 1 logical circuits $H_1, \ldots, H_{F+1}$ where F is the number of stages (In Fig. 3, F = 10). Each H_j logic (j = 1, . . ., F) but the last one, H_{F+1}, broadcasts an k·h-bit data word in two directions -to or from the respective m_j register set. The last circuit H_{F+1} connects this stage with the main memory, for the case the instruction ends its execution and its computational result has to be sent to the main memory. Selective activation of each H_j circuit connecting the P_i processor with the m_j register set is performed with position code d_j.

Since m_j register set is partitioned into two levels m_j (1) and m_j (2) then communication with m_j(1) is made through Z_5 pins of P_i processor likewize communication with m_j (2) is accomplished through Z_6 pins. Selection of the m_j register set which has to store the computational result obtained in the P_i processor is performed with the position code d stored in the AI instruction.

As was shown above, the AI instruction is available in the ASE_i address element at the time interval the pipeline stage C_i executes the respective PI portion of the instruction. If the C_i stage has to send temporary result to m_j register set of another stage C_j, the position code d_j generated in ASE_i activates broadcast of both the temporary result through H_j circuit and the destination address A_n through E_i logic. To this end, each ASE element is provided with F logical circuits $E_1, \ldots, E_F$ each E_i broadcasting destination address A_n of m_i general register set (i = 1, . . ., F). If the result obtained in the P_i processor has to be written to the local m_i register set belonging to the same stage, the AI instruction stores d_i code (d_i = i) so that it selects E_i and H_i circuits for broadcasting both address and computational result.

The height of the m register set is relatively small, since it stores temporary results. Therefore, most of the times, the final result of each PI instruction must be written to the main memory of the pipeline system. Since PI instruction implements a sequence of w operations it ends its execution in stage C_w. Since w varies in wide ranges any pipeline stage can be the end stage of some instruction. Therefore any pipeline stage must be capable of sending its computational to the main memory.

This is organized as follows. If the PI instruction obtains the final result in stage C_w, this result is sent to the next stage C_{w+1}. The reason for this is that the C_{w+1} stage has empty cell in address B + A_p, where A_p is the relative address stored in PI instruction. It then follows, that one may store in the B + A_p address of data memory M_{w+1} the destination address of the main memory where the final result can be written. This address is fetched from M_{w+1} memory to the R4 register of the P_{w+1} processor

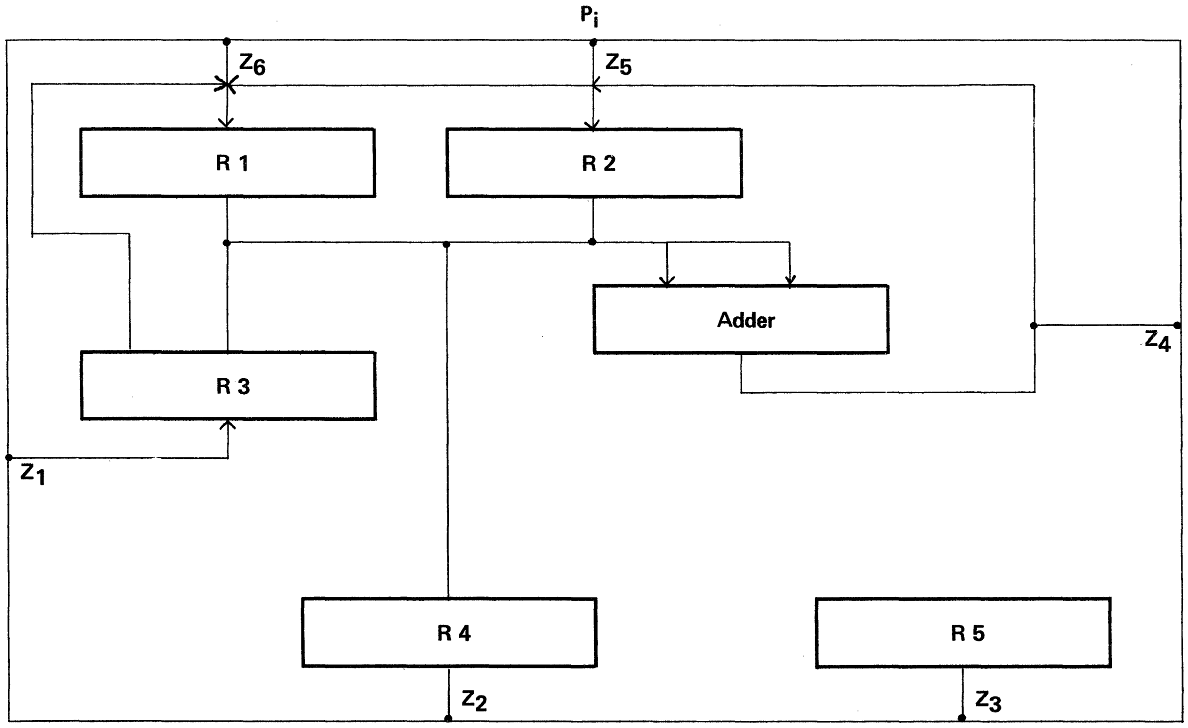


FIGURE 4

A simplified diagram of the P_i processor

(Fig. 4). Since each stage sends its result to the R3 register of the next stage, the P_{w+1} processor having the final result and its main memory address in its R3 and R4 registers will broadcast them to the main memory through logical circuit H_{F+1}. This circuit connects each stage with the main memory. (In Fig. 3, $H_{F+1} = H_{11}$, since F = 10) It is activated by the signal w produced in the last connecting element MSE_w which propagates instruction. This connecting element is recognized by equality d_i=w between its position code d_i and the code w stored in PI instruction which shows how many stages the PI propagates.

COMPUTATIONS IN A PIPELINE STAGE

At each time interval pipeline stage C_i performs two phases of computations concurrently.

(a) basic phase aimed at execution of the operation provided by current PI instruction.

(b) preparatory phase aimed at preparing data words for the next PI instruction in the same pipeline stage.

Let us consider the actions executed in each pipeline stage by each of the phases.

Basic Phase

This phase executes the operation assigned by PI instruction to the C_i stage. The PI instruction is in R5 register of the P_i processor. It was transferred there at the previous time interval (Fig. 4). The operation is performed over two operands stored in registers of the P_i processor. The result of the operation may be sent to any combination of the following destinations:

(1) P processor of the next stage: By passing through Z_4 pins of the P_i processor, the result is written to the R3 register of the next P_{i+1} processor. This occurs when P_{i+1} processor completes its basic phase.

(2) m register set of any pipeline stage: This transfer is performed through the pair of circuits H, E described above and activated by the d position code stored in the AI instruction.

(3) main memory of the pipeline: transfer is made through the H_{F+1} circuit.

Preparatory Phase

In each pipeline stage, the preparatory phase is executed concurrently with the basic phase. Its objective is to prepare operands for the next PI instruction. Let us establish the actions executed in a pipeline stage C_i during preparatory phase.

(1) Writing the next PI instruction: In the beginning of preparatory phase, the preceding MSE_{i-1} connecting element sends the next PI instruction to the P_i processor and MSE_i connecting element of stage C_i. In the P_i processor it is received through Z_3 pins to R5 register (Fig. 4).

(2) Writing the result of the operation executed in the preceding stage: The P_i processor receives this operand to its R3 register

(3) Fetch of the second operand from a data memory M or m attached to the stage: Each stage may fetch the second operand either from M or m memories. To this end it forms the effective address B + A_P, where A_p is brought by PI instruction. The M memory is accessed by this address directly, i.e., the operand is stored in the B + A_P cell. When fetched it is written to R4 register in the processor. The m register set is accessed indirectly because the B + A_P address of the M memory stores the address of m register set. Being fetched from the M memory this address is sent to the m memory. The operand fetched from there is written either to R1 register if it was stored in m (2) level or to R2 register if it was stored in m (1) level of the m register set.

Since the actions (1) to (3) prepare all the necessary information for the next basic phase, upon completion these actions, preparatory phase ends and is followed by the basic phase

SYNCHRONIZATION OF BASIC AND PREPARATORY PHASES

Since each pipeline stage is equipped with the modular control organization, it may generate minimal operation times for all operations it executes. This allows one to obtain a pipeline working with variable rate. If the pipeline is filled with short operations executed during minimal times, it produces the results faster with the rate of a short operation. Therefore implementation of a variable operation time in a pipeline stage is the source of additional speed-up, in comparison with pipelines working with permanent operation time in a stage.

However, introduction of a variable operation time in a stage requires synchronization of basic and preparatory phases in different pipeline stages. Indeed, any next PI instruction and two of its operands can be prepared for execution in the P processor of a pipeline stage only at the moment of time when the current instruction PI ended its execution and sent the result to the next stage. It then follows that for any stage its preparatory phase cannot be shorter than its basic stage.

Furthermore, since preparatory phase in a stage prepares two operands for execution, it requires availability of these operands. Of these two, one operand to be fetched from M or m memories can be made available provided the times of direct accessing the M memory or indirect accessing m register set are smaller than the time of the fastest operation executed in a stage (16 bit addition). As for the second operand to be received from the preceding stage, C_{i-1}, it may be sent to the stage C_i, only when C_{i-1} completes its operation. It then follows, that if C_{i-1} executes shorter operation than C_i, the result fromC_{i-1} appears earlier than C_i completes its operation, i.e., its basic phase. Since the operands may be written to C_i only when it completes its basic phase, the preparatory phase in stage C_i coincides with its basic phase.

Therefore, for two pipeline stages C_{i-1} and C_i executing a shorter and longer operations respectively, the time of preparatory phase in C_i coincides with the time of the basic phase in the same C_i.

If, on the other hand, C_{i-1} stage executes a longer operation than C_i, then the operand from stage C_{i-1} appears after C_i completes its operation. Then, the preparatory phase in C_i is determined by the basic phase in C_{i-1}. Thus it will last longer than the basic phase in C_i. Therefore, for two pipeline stages C_{i-1} and C_i executing longer and shorter operations respectively, the time of preparatory phase in C_i

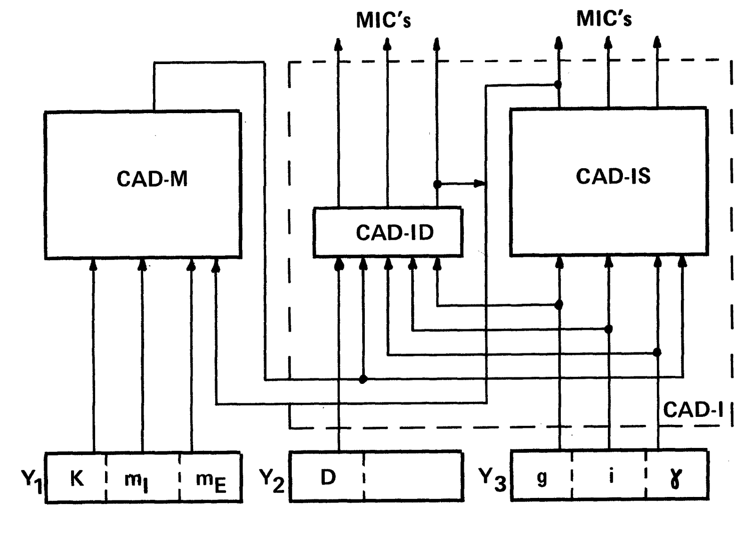


FIGURE 5

Sequencers in a pipeline stage

matches that of the basic phase in the preceding stage C_{i-1}.

Thus we have shown that for the C_i stage, its preparatory phase either coincides with the basic phase of the same stage C_i, or with the basic phase of the preceding phase C_{i-1}. Therefore for each pipeline stage its preparatory phase introduces no additional delay in the rate of pipeline operation which is determined only by consecutive durations of its basic phases.

Such synchronization of phases may be accomplished as follows. As was shown in [12], the modular control organization provides that duration of each PI instruction be determined by the two interactive units CAD-I and CAD-M, where CAD-I functions as either a decoder or sequencer and CAD-M is a sequencer which specifies the time of operation. If PI instruction activates a non-iterative operation in a stage (addition, subtraction, Boolean), then in the CAD-I of this stage decoder CAD-ID is activated (Fig 5). For iterative operation (multiplication, division) executed in a stage, its CAD-I works as a sequencer CAD-IS. The CAD-I functioning is controlled with the following codes: the op code D it receives with the PI instruction and position signal i produced locally by the position code d_i.

Consider now how one may organize a variable time T of operation executed in C_i stage of the pipeline: T is variable, i.e., $T = t_o \cdot b$, where t_o is the time of h-bit addition in one LSI module, b, depends on k and D codes stored in PI instruction. For a processor dependent operation (addition, subtraction), which has to last $T = k \cdot t_o$, the output of CAD-I decoder initiates the CAD-M sequencer which executes a loop having k states. During this time the CAD-I maintains a microcommand MIC which activates operation in the processor. When CAD-M completes its loop, this terminates the CAD-I output. If the operation is independent of the processor size (Boolean, shift, etc.), then it is activated by the CAD-I decoder only. Namely, CAD-M is not initiated and the operation takes time t_o of one small clock-period. If C_i stage executes an iterative operation, CAD-I executes a sequence containing several states. If a state in this sequence has to last the time $k \cdot t_o$, CAD-I initiates the CAD-M and performs transition to the next state under the completion signal issued by the CAD-M.

As follows from this organization, in each pipeline stage the completion of basic phase occurs when CAD-M performs transition to its initial state and issues the completion signal, CS. It then follows, that if C_{i-1} has to delay its basic phase because the next stage C_i executes a longer operation, then this delay can be accomplished if CAD-M in C_{i-1} delays transition to the initial state until CAD-M of the next C_i establishes the state which immediately precedes the initial one. At the next clock period, both CAD-M perform concurrent transitions to their initial state which will mean concurrent end of basic phases in C_{i-1} and C_i.

Such synchronization can be accomplished if any transition of CAD-M in C_{i-1} to the initial state is activated by either a completion signal CS_i generated by the CAD-M in C_i or its immediate predecessor PR_i produced by the state which precedes the initial one, i.e., CS_i v PR_i. This means that in the pipeline each right Processor P_i has to be connected with its next left processor P_{i-1} via one line connection, sending signal CS_i v PR_i produced by CAD-M in P_i. This signal activates transition of each left CAD-M to the initial state. This will accomplish synchronization of shorter and longer basic phases executed in C_{i-1} and C_i respectively.

If C_{i-1} and C_i execute respectively longer and shorter basic phases, then CS_i v PR_i generated in CAD-M of C_i cannot be used for synchronization, since CAD-M in C_i finished its operation much earlier that that in C_{i-1} For this case the time when C_{i-1} completes execution and issues an operand for C_i can be determined by the PR_{i-1} signal generated by CAD-M in C_{i-1}. Indeed, at the next clock period C_{i-1} will send the operand to R_3 register of C_i causing completion of preparatory phase in C_i. This can be accomplished if each left processor P_{i-1} is connected with its next right neighbor P_i via one line connection sending signal PR_{i-1} produced by CAD-M in P_{i-1}. This signal will enable writing of two operands and PI instruction to registers of P_i processor.

This synchronization of phases requires that every pair of neighbors P_{i-1} and P_i be connected with two lines, so that using one line P_i synchronizes P_{i-1} with signals CS_i v PR_i generated in P_i and using another line P_{i-1} synchronizes P_i with signal PR_{i-1} generated in P_{i-1}.

OPERATION OF ADDRESS ELEMENT ASE

Each ASE_i element generates the address A_n and position code d of the m register set which has to receive temporary result produced in stage C_i. Since for every PI instruction each stage C_i may send its result to any m register set, each ASE_i element receiving the AI instruction has to be provided with an opportunity to change the values of a_n and d, stored in the AI instruction. Then each time the AI instruction is stored in ASE_i it may have new values of A_n and d requiring no additional bits in AI to store them. This will allow to have the minimal size in AI (required to

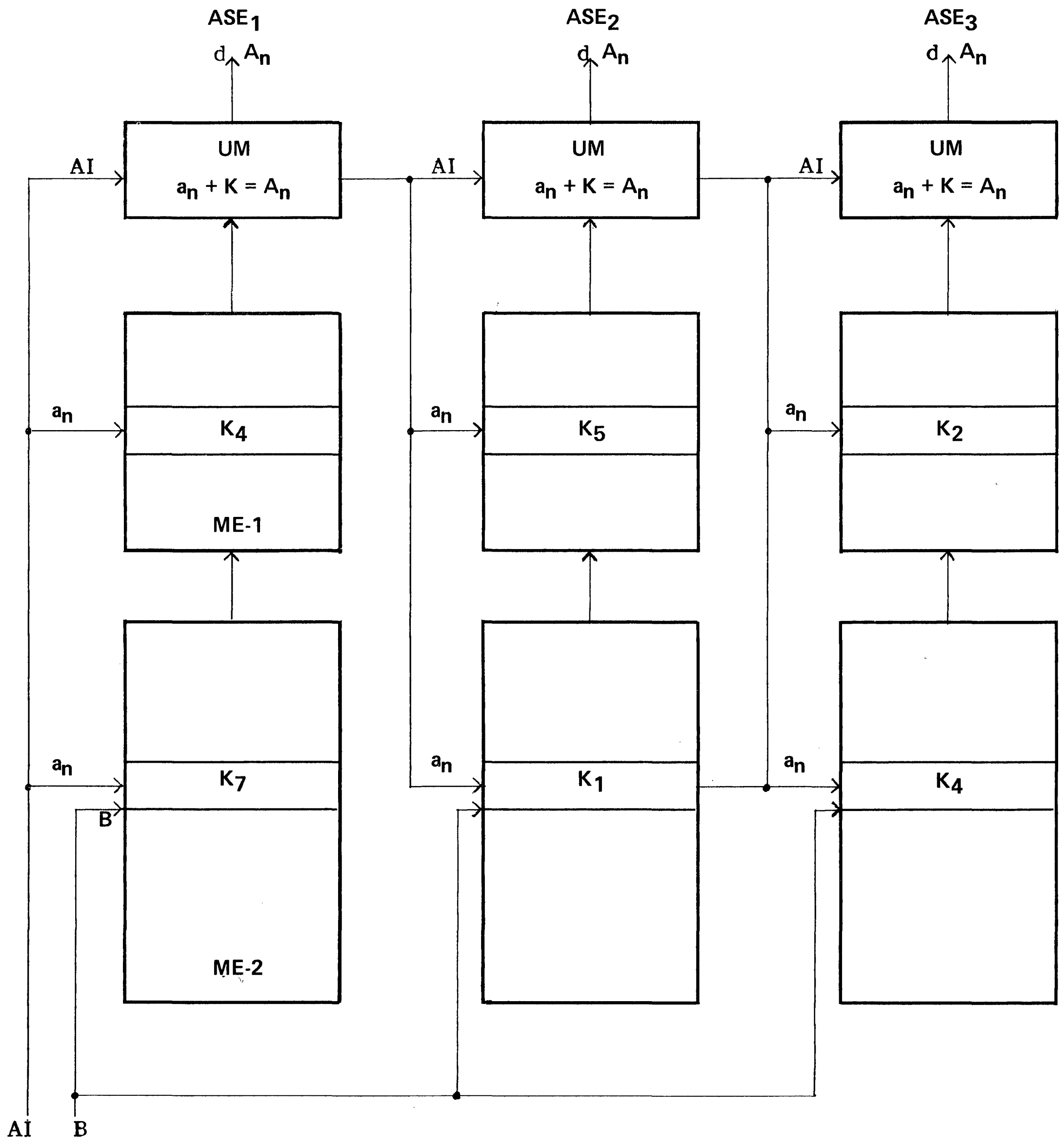


FIGURE 6

Address elements ASE

store one address a_n and one position code d only) and at the same time to maintain this powerful option of communication between pipeline stages.

Let us consider one organization which changes d code nd $_n$ address for each pipeline stage. It provides that each address element ASE contain one microprocessor UM and two levels of memory ME-1 and ME-2 (Fig. 6). The fast and small memory ME-1 is connected with UM; the slower and larger memory ME-2 is connected with ME-1. Thus ME-1 may send its information to UM, likewise ME-2 may send its information to ME-1. ME-1 stores constants that may be added with values stored in AI instruction. By writing different constants k_{i-1}, k_i, k_{i+1}, etc. to the same address a_n of consecutive memories ME-1_{i-1}, ME-1_i ME-1_{i+1} respectively one may achieve variation in d code and a_n address when the AI instruction propagates through consecutive ASE elements. Thus one cell in each ME-1 stores information to modify one AI instruction for the respective ASE. Since the number of cells in ME-1 is small (in as much as ME-1 has to be very fast) to increase the number of consecutive AI instructions which can be modified via a single ME-1, each ME-1 is connected with a larger memory ME-2 designated to replenish content of each cell in ME-1 whenever it is accessed by the AI instruction. Thus a modified content of this cell may now be accessed by a new AI instruction causing no restriction on the number of AI instructions which may modify their d and a_n values.

For example, consider modification of d code and a_n address stored in AI instruction when it propagates through ASE_1, ASE_2, ASE_3 (Fig. 6). When the AI instruction is received in ASE_1, the address a_n it stores is sent to both memories ME-1 and ME-2. The ME-1 fetches constant k_4 which is used to modify a_n and d: $a_n + k_4 \longrightarrow A_n$, $d + k_4 \longrightarrow d$. The new values A_n and d are issued by ASE_1. At the next interval, the same address a_n accesses ME-2 which sends a new constant k_7 to the a_n cell of ME-1. Thus the cell of ME-1 accessed by the AI instruction in ASE_1 is updated. When the C_1 stage completes its basic phase producing PR_1 signal, the AI instruction is transferred with this signal to the next ASE_2 element where it is added with new constant k_5 stored in the same address a_n, etc.

Therefore, by passing through F consecutive ASE elements the same instruction may generate F different meanings of A_n and d. This allows a programmer, for each pipeline instruction containing w stages to form w addresses and position codes, so that each stage activated by the PI instruction may send its result not only to the next stage but also to any m register set.

REFERENCE

1. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "IBM System 360 Model 91, Machine Philosophy and Instruction Handling," IBM Journal of Research and Development, January 1967, pp. 8-24.

2. S. F. Anderson, J. G. Earle, R. E. Goldschmidt and D. M. Powers, "The IBM System 360, Model 91: Floating-point Execution Unit," IBM Journal of Research and Development, Jan. 1967, pp. 34-53.

3. R. M. Russell "The CRAY-1 Computer System" Communications ACM, vol. 21, January 1978, pp. 63-72.

4. W. J. Watson, "The TI ASC—A highly Modular and Flexible Super Computer Architecture," In AFIPS 1972 Fall Jt. Computer Conf., AFIPS Press, Montvale, N.J. 1972, pp. 221-228.

5. C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," ACM Computing Surveys, vol. 9, no. 1, March 1977, pp. 61-102.

6. A. Thomasian and A. Avizienis "A design study of a shared-resource computer system," Proceedings of the 3rd International Symposium on Computer Architecture, 1976, pp. 105-111.

7. S. S. Reddi and E. A. Feustal, "A Restructurable Computer System," IEEE Transactions on Computers, vol. C-27, No. 1, January 1978, pp. 1-20.

8. M. J. Irwin "Reconfigurable Pipeline Systems," Proceedings 1978 ACM Annual Conference, vol. 1, pp. 86-92.

9. R. N. Ibbett and P. C. Capon, "The Development of the MU5 Computer System," Communications of the ACM, vol. 21, no. 1, January 1978, pp. 13-24.

10. S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions," Computer, vol. 11, July 1978, pp. 26-40.

11. S. I. Kartashev and S. P. Kartashev, "Multicomputer System with Dynamic Architecture," accepted for publication in IEEE Transaction on Computers, October 1979.

12. S. I. Kartashev, S. P. Kartashev, and C. V. Ramamoorthy, "Adaptation Properties for Dynamic Architectures," 1979 National Computer Conference, AFIPS Conference Proceedings, AFIPS Press, 1979, vol 48, pp. 543-556.

13. S. I. Kartashev, and S. P. Kartashev, "A Microprocessor with Modular Control as a Universal Building Block for Complex Computers," Proc. 3rd Euromicro Symposium on Microprocessing and Microprogramming, Amsterdam, 1977, pp. 210-216.

A COMPARISON OF THREE TYPES OF MULTIPROCESSOR ALGORITHMS[a]

Harry F. Jordan[1], Maria Scalabrin[2] and Wynne Calvert[1]

[1]Electrical Engineering Department
University of Colorado
Boulder, Colorado 80309

[2]Computer Science Department
Universidade Estadual de Campinas
Brasil

Abstract

In a multiprocessor system with limited communications paths between processors, a heavy communications demand is made by an algorithm requiring input values from all processors in the system. The Finite Element Machine is a special purpose computer designed from 1024 microprocessors communicating by way of explicit word-serial channels. The algorithms for structural analysis using the finite element method which have been proposed for this machine require both maximum and summation over a set of numbers stored one per processor. Three distinct types of sum and maximum algorithm making different use of the communications paths have been formulated and analyzed with respect to execution time. The results give some insight into the best way to use communications paths in a multiprocessor. The study demonstrated the need for a special hardware mechanism to support the global sum and maximum operations in this machine. The design of this hardware unit is also discussed.

Introduction

The Finite Element Machine [1,2] is a special purpose computer architecture developed to support structural analysis using methods based on the theory of finite elements [3]. The machine consists of a large number (1024 is the target design) of microprocessors communicating over a network of parallel "local" channels connecting each processor to a limited number of other processors. The parallel local channels are backed up by a time multiplexed global bus connecting all processors. If a node in a finite element model of a structure is considered to be "connected" to another node when the two correspond to a nonzero stiffness matrix coefficient, then the model forms a graph with the qualitative characteristic that each node connects to only a few others. The Finite Element Machine with processors as nodes and local communications links as edges has the same qualitative characteristic. The idea is to use this qualitative similarity to carry out the solution of finite element model equations in such a way that most of the required interprocessor communications can be performed using the parallel local channels while the time multiplexed bus takes care of the mismatches between the finite element model topology and the topology imposed by the fixed local channels.

The linear equations of the finite element model are solved by iterative methods so that most data interchange between processors is determined by the topology of nonzero elements of the sparse stiffness matrix. In any iterative method, however, a test for convergence must be made which depends on values associated with every node. In particular, a maximum must be computed over 1024 values, one stored in each processor. Rapidly converging iterative methods, such as the conjugate gradient method [4], require not only maximum but also summation over values from all processors. These global computations strain the communications capabilities of a machine tailored to the more limited requirements of iterative updating of values.

This paper takes the architecture of the Finite Element Machine as fixed and proposes three algorithms for performing the global sum and maximum calculations. These algorithms are compared with respect to execution time to measure their suitability for a 1024 processor machine. The fixed machine architecture is then extended by the addition of special hardware to make these computations more efficient.

Description of the Multiprocessor

The Finite Element Machine consists of a large number (~1000) of microprocessors communicating by way of a network of point to point First-In-First-Out (FIFO) communications. The multiprocessor array is not easily characterized as tightly or loosely coupled [5]. The speed and information carrying capacity of the communications network would indicate a tightly coupled system while the lack of shared memory would suggest loose coupling. The communications occur over two separate networks: the local network and the global network. The local network consists of a large number (~8000) of bidirectional unshared connections between two nearest neighbor processors. The global network consists of a single time multiplexed bus connecting all processors. A third interconnecting network is composed of a set of signal flags which are primarily oriented toward processor synchronization. A brief description of the three networks interconnecting the processors is necessary to an understanding of the algorithms to follow. Further details can be found in [2].

The local communications network imposes a fixed interconnection topology on the 16 bit microcomputers making up the array. The current prototyping effort being carried out at NASA

[a]This work was partially supported by NASA under grant number NSG 1489.

Langley Research Center will allow investigation of several interconnection patterns. For the purposes of this paper, however, we will consider only a square array of 32 x 32 processors with each processor having a local communications link to each of its eight nearest neighbors. Processors on the boundaries are connected to those on opposite boundaries in the manner of a toroid so that all processors have a full eight neighbors. The global bus simply connects in parallel to all of the processors in the array. Of course this is only true logically; electronically, the bus and its arbitration network form a tree structure. The structure of the signaling flag network parallels that of the global bus. An abbreviated PMS [6] diagram for the processor array is shown in Figure 1.

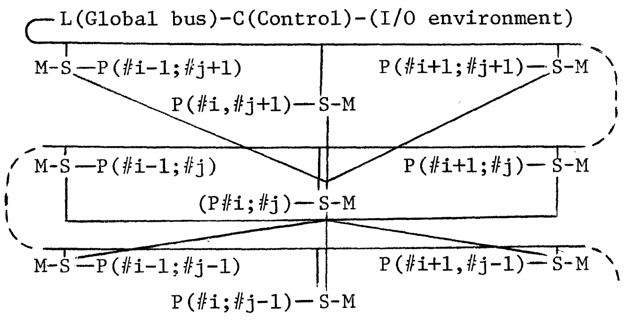
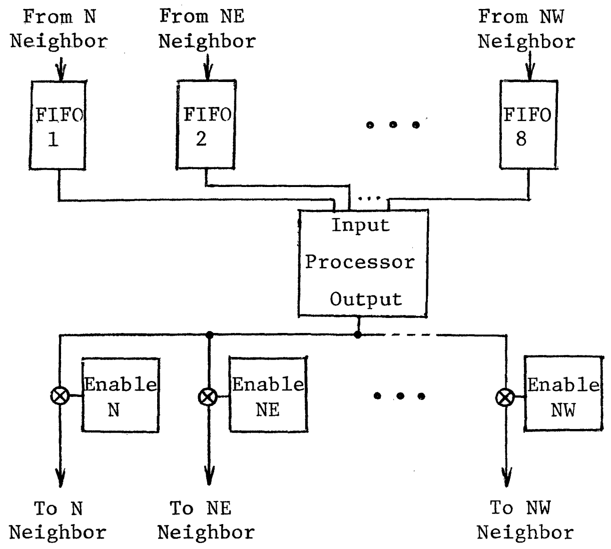


Figure 1: Partial PMS Diagram for the Array

Local communications output is broadcast to all neighbors simultaneously, although neighbors can be selectively disabled from receiving. Information coming in from a neighbor is placed in a hardware FIFO, which the receiving processor may interrogate and empty at its own initiative. Transmission is actually bit serial to conserve hardware but becomes parallel at the processor interfaces. Figure 2 shows a logical block diagram of the local communications interfaces for one processor and lists the primitive operations the processor can perform in connection with these interfaces. The various local neighbors are identified by the eight points of the compass.

The global bus acts as a time multiplexed crosspoint switch, allowing the transmission of one 16 bit word from any source processor to any destination processor. A transaction thus appears on the bus as three items: the source processor number, destination processor number and data word. Since a bus transaction time is short compared to an instruction time a FIFO buffer receives input from the bus. Further, since contention for the bus may delay a transmission, FIFO buffering is also provided for output to the bus. A destination register allows a processor to transmit several successive data words to the same destination without respecifying it. A special processor number serves to identify broadcast data and matches the address of any processor which is enabled for broadcast reception.



a) Local Communications Interfaces

Operations:

Output (word)	- Broadcast word to all enabled neighbors.
Enable (j)	- Enable neighbor j.
Disable (j)	- Disable neighbor j.
Input (j,word)	- Input word from FIFO j and advance it.
Interrupt enable (j)	- Enable FIFO j non-empty interrupt.
Interrupt disable (j)	- Disable FIFO j non-empty interrupt.

Tests:

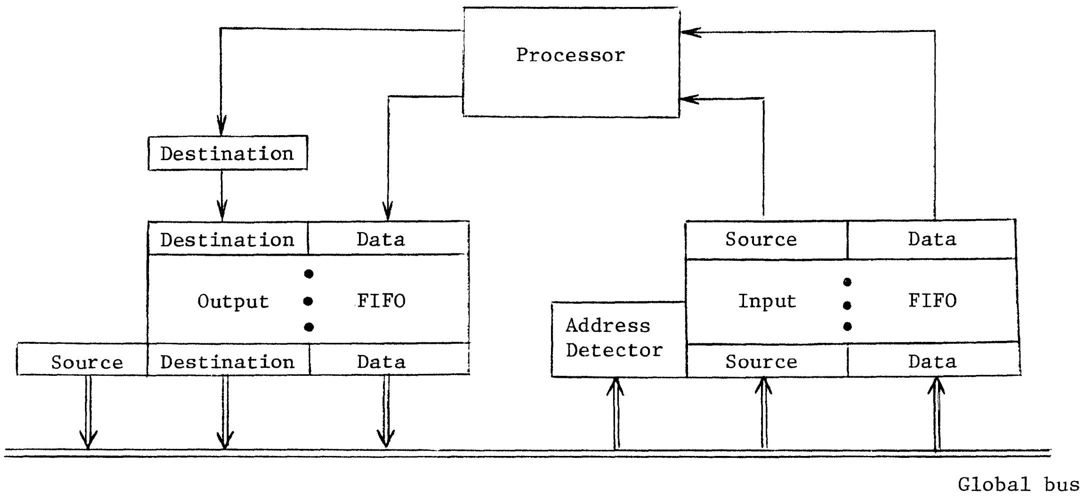
Output busy?	- Is any enabled neighbor's FIFO full?
Input ready (j)?	- Is input FIFO j non-empty?

b) Local Communications Primitives

Figure 2: Local Communications for a Single Processor

Figure 3 shows the block diagram and primitive operations for a processor's global bus communications.

The signal flag network consists of eight single bit variables, or flags, per processor. One of these and the corresponding flags in all other processors form the inputs to a network which forms the combinational functions AND and OR over all its inputs. The OR and AND functions are available to each processor and indicate whether the corresponding flag is set in any other processor or in all other processors, respectively. An enable bit allows a flag to be



a) Global Communications Interfaces

Operations:

Set destination (i)	- Set destination register to i.
Send word (w)	- Put (Destination, w) into output FIFO.
Set broadcast	- Set destination register to broadcast address.
Read source (s)	- Get source address from head of input FIFO.
Read data (w)	- Get data and advance input FIFO.
Receive broadcast	- Sensitize detector to broadcast address.
Ignore broadcast	- Disable detection of broadcast address.

Tests:

Input ready?	- Is input FIFO nonempty?
Output full?	- Is output FIFO full?

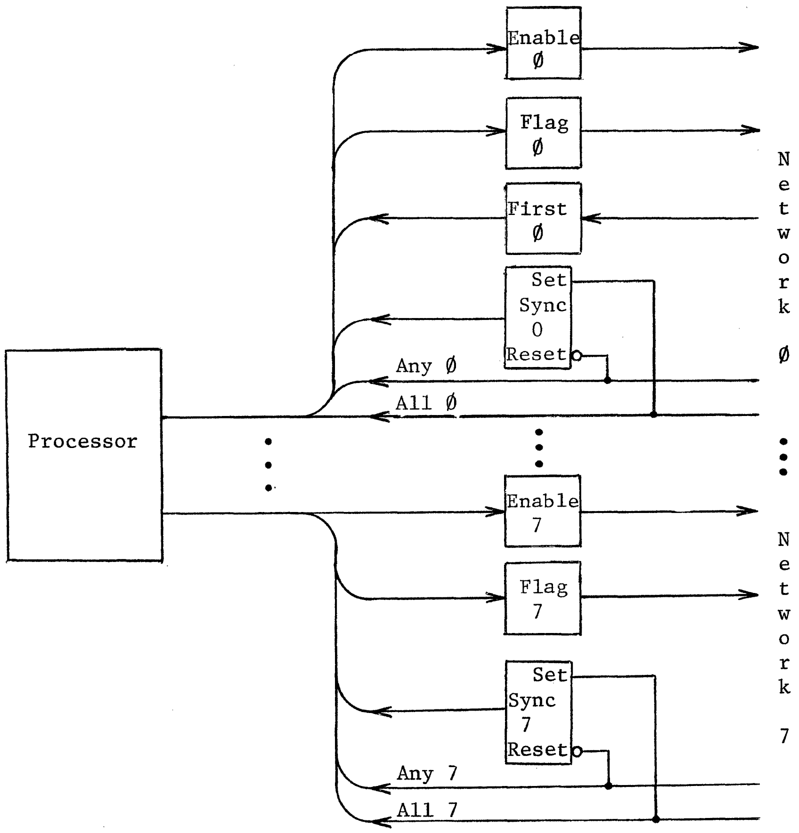
b) Global Communications Primitives

Figure 3: Global Communications for a Single Processor

effectively connected into or isolated from the computational network. There are eight independent and nearly identical networks, one for each of the eight flags available to a processor. Also associated with each flag is a synchronization bit, Sync i, which is set when the AND function becomes true and reset when the OR function becomes false. The only difference between the eight flag networks is that one of them, flag zero, is augmented by a unique selection network used in some algorithms to solve the "multiple hit" problem but not needed below. Whenever a group of enabled processors asynchronously set this flag, one of them will be designated as the "first" to do so. The "first" indicator is thus a separate bit associated with flag zero for each processor. The structure and primitive operations for the signal flag network are shown in Figure 4.

Methods for Performing Array Computations

The finite element method algorithms proposed for the multiprocessor array described above primarily require local computation with input from neighboring processors; but also some calculations involving numbers from all processors in the array are needed. Specifically, the test for convergance in an iterative algorithm requires the computation of $\max_i X_i$ where X_i is a value contained in processor i only and the maximum is taken over all processors. Using the conjugate gradient method to perform the iterative linear equation solution requires the computation of $\sum_i X_i$ where the sum runs over all processors. Again X_i is local to a given processor and in this case it is the product of two local values. The computation performed is



a) Signal Flag Interfaces

Operations:

Connect (k)	- Enable the kth flag.
Disconnect (k)	- Disable the kth flag.
Set (k)	- Set flag k.
Clear (k)	- Clear flag k.

Tests:

Any (k)?	- Is flag k set in any connected processor?
All (k)?	- Is flag k set in all connected processors?
Sync (k)?	- Was All true previously?
First?	- Was this processor's Set (∅) the "first" one?

b) Signal Flag Primitives

Figure 4: Signal Flag Communications for One Processor

actually a vector inner product but only the summation requires calculation involving the whole array.

Consider then the isolated problem of evaluating $\max_i X_i$ over the entire array of nodal processors. At least three distinct methods of evaluation are possible using different aspects of the three processor communication networks. First all values could be sent over the global bus to a single processor which would compute the maximum sequentially. This would require 1024 times the length of a several instruction loop in a microprocessor of the array. A possibly faster method would be to compute maxima over subgroups of the nodal processors, using the local connection network to transmit information without conflict between groups. These subgroup maxima could then be combined using local or global communications to form the maximum over all processors, perhaps using higher level grouping of subgroup results.

Call the first method using one processor for the maximum calculation the central computation method and call the second method distributed computation.

The third method that presents itself might be called cooperative computation and operates using the network of signal flags. This is a tightly synchronized calculation which uses some signal flags for synchronization but also uses a signal flag to compute the maximum value in a bit serial manner. The computation proceeds roughly as follows (processor synchronization is not mentioned explicitly):

1. All processors connect themselves to a signal flag, say Flag k, by setting the Enable k bit. A counter ℓ is initialized to zero.

2. Each processor with its Enable k bit set sets its signal Flag k to the ℓth bit of X_i where the bits are numbered starting with zero at the left.

3. Each processor performs the Any (k) test and records a 1 in the ℓth bit of the maximum if Any (k) is true and a 0 if it is false. If Any (k) is true each processor which still has Enable k set clears it if its Flag k is zero.

4. All processors advance to the next bit of X_i by incrementing ℓ. If there are more bits, the loop is repeated beginning at step 2, otherwise step 5 is executed.

5. At this point all processors which still have Enable k set (always at least one) are associated with an X_i which equals the maximum value. All processors have recorded the maximum so that the result of the computation is already distributed.

Many possible versions of the distributed computation of the maximum are possible depending on the way in which the processor array is divided into groups. We considered what is perhaps the simplest of them in which groups of nine processors are formed initially (so far as possible). One processor receives values from each of its eight neighbors and computes the maximum of the nine values to which it has access. These receiving processors then combine subgroup maxima in pairs using the global bus for communications. The pairwise maxima form a binary tree with the final array maximum appearing at the root node. Of course, an array of 1024 processors is not evenly divisible into groups of nine so some of the groups are smaller.

The initial grouping on the array is indicated in Figure 5. In the first phase of the computation the processors marked A merely transmit a value over the local network to a neighbor. The B processors receive and form the group maximum of these values. The three types of B processors, B1, B2 and B3, compute maxima of 9, 6, and 4

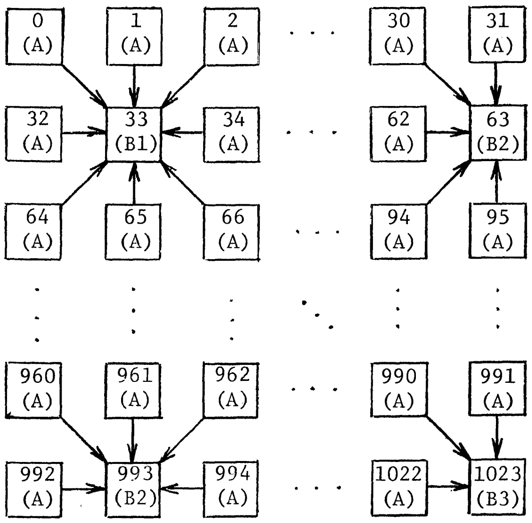


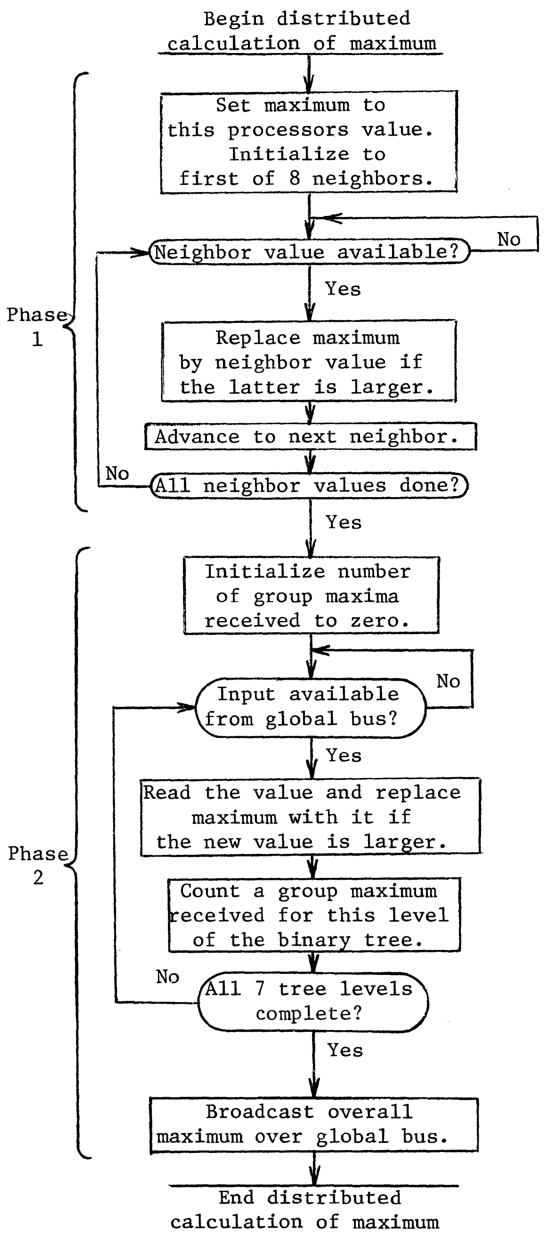
Figure 5: Partioning the Array for Distributed Computation

values respectively. In the second phase of the computation, the 121 B processors are formed into a seven level binary tree with half of the processors remaining at any one level transmitting values over the global bus to another processor and disabling themselves. Thus 61 transmissions will time multiplex the global bus at the first tree level and this number will be successively halved at successive levels. The longest program will be executed by the processor which eventually forms the root of the tree and it is this program which will be used, along with estimates of overhead due to global bus conflicts, to determine the speed of the distributed algorithm.

Comparison of the Methods

The problem is to decide which of the three computational methods (centralized, distributed or cooperative) is most efficient for a 1024 processor version of the finite element machine. To do this, algorithms of each type were formulated and coded. A timing analysis of the code was performed and the overhead due to global bus congestion was estimated and added to the code time. Note that the first two computational methods, centralized and distributed, are also possible ways of calculating $\sum_i X_i$. The cooperative method, using a similar bit serial approach, is possible if one of the signal flag networks is augmented with a parallel counter. This would appear to a processor as an 11 bit port, Count (k), which gives the number of processors having Enable k and Flag k both set. We will take the analysis for $\text{Max}_i\ X_i$ as indicative and use the results to determine whether the addition of a parallel counter is cost effective for the computation of $\sum_i X_i$.

The algorithms for central, distributed and cooperative calculation of $\operatorname{Max}_i X_i$ were timed and compared using hardward parameters from the prototype Finite Element Machine desinged at the University of Colorado and under construction at NASA Langley Research Center. The microprocessor used is a Texas Instruments TMS 9900 with an average instruction time of about 6 microseconds. The time for one transmission over the global bus is



Begin distributed calculation of maximum
Set maximum to this processors value. Initialize to first of 8 neighbors.
Neighbor value available?
No
Yes
Replace maximum by neighbor value if the latter is larger.
Advance to next neighbor.
All neighbor values done?
No
Yes
Phase 1
Initialize number of group maxima received to zero.
Input available from global bus?
No
Yes
Read the value and replace maximum with it if the new value is larger.
Count a group maximum received for this level of the binary tree.
All 7 tree levels complete?
No
Yes
Broadcast overall maximum over global bus.
Phase 2
End distributed calculation of maximum

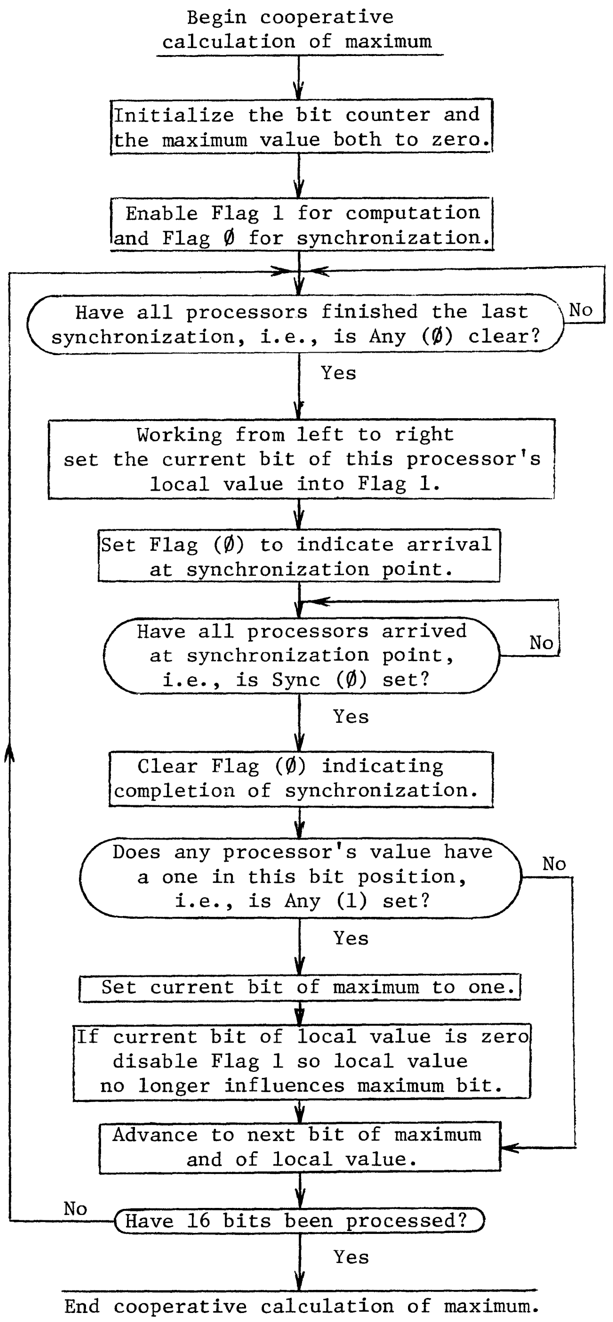
Figure 6: Distributed Calculation of $\operatorname{Max}_i X_i$

taken as 0.5 microseconds. The time for the central computation is determined by the algorithm executed by the single central receiver. This algorithm is a straightforward access and combination of 1024 values. The TMS 9900 program required the execution of 9,219 instructions so that the time for evaluating the maximum would be about 55.3 milliseconds. Since a global bus transmission time is shorter than an instruction time, no transmission overhead need be added to this number.

In the specific distributed calculation described above, there are distinct algorithms executed by distinct groups of processors. Some of them compute maxima over values from their eight neighboring processors and themselves, transmit this local maximum and then wait for a broadcast message containing the overall maximum. This corresponds to the section marked Phase 1 in the flowchart of Figure 6. In Phase 2, one of the processors involved in computing the pairwise maxima will receive values at all seven levels of the binary tree and broadcast the final maximum back to all processors. It is not absolutely necessary that the same processor execute both Phase 1 and Phase 2 of the flowchart, but the overall time will be determined by the juxtaposition of these two program segments in any case. The program code amounts to 175 instructions executed from start to finish or about 1050 microseconds. Added to this will be non-overlapped communications time. The first local network transmission requires 20 microseconds. Subsequent local transmissions overlap with instruction execution. Overlap of global bus transmission with computation time is harder to estimate but it is certainly bounded above by the total number of values transmitted, 120, times the bus transmission time. This gives an upper bound of 60 microseconds for a total of 1.13 milliseconds for the distributed algorithm.

The code for the cooperative algorithm follows the flowchart of Figure 7. The explicitly specified synchronization is an example of so-called barrier synchronization in which all processors must reach a given point before any may proceed. Here it serves to assure that all processors have input a bit into the Flag 1 network before the OR of all the bits is tested. With all processors running at the same speed the overhead due to synchronizing waits will be zero and can thus be neglected even if the processors only run at nearly the same speed. The number of instructions executed in this algorithm is 328 for a time of about 1.97 milliseconds.

As expected, the central computation is the clear loser taking 49 times as long as the distributed calculation. Somewhat unexpected, however, is the result that the crude distributed algorithm which we analyzed is almost twice as fast as the cooperative method. This gain comes at the expense of having to load different programs into different groups of processors while in the cooperative method one program serves for all processors. Certainly the comparison indicates that the



Begin cooperative calculation of maximum
Initialize the bit counter and the maximum value both to zero.
Enable Flag 1 for computation and Flag Ø for synchronization.
Have all processors finished the last synchronization, i.e., is Any (Ø) clear?
No
Yes
Working from left to right set the current bit of this processor's local value into Flag 1.
Set Flag (Ø) to indicate arrival at synchronization point.
Have all processors arrived at synchronization point, i.e., is Sync (Ø) set?
No
Yes
Clear Flag (Ø) indicating completion of synchronization.
Does any processor's value have a one in this bit position, i.e., is Any (1) set?
No
Yes
Set current bit of maximum to one.
If current bit of local value is zero disable Flag 1 so local value no longer influences maximum bit.
Advance to next bit of maximum and of local value.
No
Have 16 bits been processed?
Yes
End cooperative calculation of maximum.

Figure 7: Cooperative Calculation of $\underset{i}{\text{Max}}\, X_i$

addition of a parallel counter to support cooperative calculation of $\sum_i X_i$ would not be cost effective. On the other hand, the fastest method takes a millisecond, which is quite long with respect to the processor speed so hardware support for both $\underset{i}{\text{Max}}\, X_i$ and $\sum_i X_i$ is indicated.

Hardware Support for Global Calculations

As a result of the above analysis a hardware circuit has been included in the finite-element machine to avoid long delays in calculating global sum and global maximum. The configuration of this circuit is a binary tree, in which each tree element accepts an argument pair and passes on the result to the next higher level. The global results, which appear at the apex of the tree, are then broadcast to all the processors. With this configuration, a tree of N levels can serve 2^N processors. For instance, 1024 processors require 10 levels and 1023 tree elements.

The tree elements are serial. Each one accepts two serial arguments and produces a serial result, delayed by one bit period. The sum and maximum functions are calculated alternately, sharing the same tree connections and element circuitry. The entire calculation sequence involves a serial frame of 48 bit periods: 26 for sum, 16 for maximum and 6 unused.

The tree input data consists of a single 16-bit output register at each processor. Both functions are thus calculated from the same data, and they are separately available. The sum requires a double-word input register for the entire result; the maximum requires a single-word register. In both cases, the data are treated as positive, integer values.

The sum-maximum tree is implemented without facilities for process synchronization. Use of the circuit will generally require barrier synchronization, such as that already provided by the signal flags. However, functional synchronization is included, whereby each output register is sampled as a unit. This assures that the results are valid even while different processors are updating their output registers.

The delay for the circuit consists of the tree delay of one bit period for each level, plus the frame delay of 48 bit periods. The delay for 1024 processors, which requires 10 levels in the tree, is thus 58 bit periods. With a bit period of one microsecond, the entire calculation of both sum and maximum is accomplished in 58 microseconds.

Conclusion

The detailed timing analysis for three algorithms for computing $\sum_i X_i$ and $\underset{i}{\text{Max}}\, X_i$ over all processors in a multiprocessor array has been performed. The three algorithms make distinctly different use of the communication pathways in the machine. The analysis shows that $\underset{i}{\text{Max}}\, X_i$ can be computed in 55.3 milliseconds by a centralized algorithm, in 1.13 milliseconds by a distributed algorithm and in 1.97 milliseconds by a cooperative algorithm. The distributed algorithm is the most logically complex, requiring about 6 different programs for different groups of processors.

As a result of this analysis a proposal to support cooperative calculation of $\sum_i X_i$ with special hardware in the Finite Element Machine was discarded. Instead a complete hardware unit to calculate sum and maximum using a bit serial binary tree organization was designed. The unit can calculate both sum and maximum over the same set of 1024 operands in 58 microseconds.

References

[1] H. Jordan, "A Special Purpose Architecture for Finite Element Analysis," Proc. 1978 International Conference on Parallel Processing, (August, 1978), pp. 263-266.

[2] H.F. Jordan and P.L. Sawyer, "A Multi-microprocessor System for Finite Element Structural Analysis," Trends in Computerized Structural Analysis and Synthesis (A.K. Noor and H.G. McComb, Jr., Eds.), Pergamon Press, (1978), pp. 21-29.

[3] O.C. Zienkiewicz and Y.K. Cheung, Finite Element Method in Structural and Continuum Mechanics, McGraw Hill, (1972).

[4] M.R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," J. Res. Nat. Bur. Std., 48, (1952), pp. 409-436.

[5] P.H. Enslow (Ed.), Multiprocessors and Parallel Processing, John Wiley and Sons, (1974).

[6] C.G. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw Hill, (1971).

ON THE MAPPING PROBLEM

Shahid H. Bokhari
Institute for Computer Applications in Science & Engineering (ICASE)
MS 132C, NASA Langley Research Center, Hampton, VA 23665

Abstract--In array processors it is important to map problem modules onto processors such that modules that communicate with each other lie, as far as possible, on adjacent processors. This mapping problem is formulated in graph theoretic terms and shown to be equivalent, in its most general form, to the graph isomorphism problem. The problem is also very similar to the bandwidth reduction problem for sparse matrices and to the quadratic assignment problem.

It appears unlikely that an efficient exact algorithm for the general mapping problem will ever be found. Research in this area must concentrate on efficient heuristics that find good solutions in most cases. A heuristic algorithm that proceeds by sequences of pairwise interchanges alternating with probabilistic jumps is described. This algorithm has been used to solve practical mapping problems on a specific array processor (the Finite Element Machine) with good results. Results for a set of practical problems are tabulated, several of which are illustrated.

I. Introduction

Most arrays of processors are incompletely connected, that is, a direct link does not connect each pair of processors. The reasons for this include (1) the fact that the total number of links in completely connected systems increases as the square of the number of processors--a growth rate that is unacceptable in most cases, and (2) the number of input/output ports on each individual processor increases linearly with the number of processors--this is usually not possible because the number of I/O ports is generally fixed at some constant value.

Suppose a problem made up of several modules that execute in parallel is to be solved on an incompletely connected array. When assigning modules to processors, pairs of modules that communicate with each other should be placed, as far as possible, on processors that are directly connected. We call the assignment of modules to processors a mapping and the problem of maximizing the number of pairs of communicating modules that fall on pairs of directly connected processors the Mapping Problem.

This research was supported by NASA Contracts NAS1-14101 and NAS1-14472 while the author was resident at ICASE.

In this paper we first show that the problem of finding the best mapping is, in general, very difficult. We then describe a heuristic algorithm that has been developed to solve this problem for a specific array processor. We start by giving a mathematical formulation of the problem in Section II. In Section III we show that in its most general form, the mapping problem is equivalent to the graph isomorphism problem, one of the classical unsolved combinatorial problems. We point out the similarities between the mapping problem and the bandwidth reduction and quadratic assignment problems. Exact solutions for neither of these problems exist and they are solved approximately using heuristic algorithms.

In Section IV we describe how the mapping problem arises when solving structural problems on the Finite Element Machine (FEM), an array of processors currently under development at NASA Langley Research Center. In Section V we describe a simple heuristic algorithm that has been implemented and used to find mappings for the finite element machine with very encouraging results. Results for a number of test cases are tabulated, several of which are illustrated.

II. Mathematical Formulation

Let the graph of the problem to be mapped onto the array be denoted $G_p=<V_p,E_p>$, where the nodes or vertices V_p correspond to the set of modules and each edge $(x,y)\in E_p$ denotes that modules $x,y\in V_p$ communicate with each other.

Let the graph of the array processor be denoted $G_a=<V_a,E_a>$, where V_a is the set of processors and the edges E_a represent the interconnection pattern of the processors.

The problem graph G_p may be considered to be a set of vertices V_p and a function $G_p:V_p \times V_p \text{-->}\{0,1\}$, such that $G_p(x,y)=G_p(y,x)$ and $G_p(x,x)=0$ for all $x,y\in V_p$. $G_p(x,y)=1$ is taken to mean that there is an edge between x and y, i.e. the pair $(x,y)\in E_p$.

The graph of the array, G_a, may similarly be considered a set of vertices V_a and a function $G_a:V_a \times V_a \rightarrow \{0,1\}$.

We assume that $|V_p|=|V_a|$. If $|V_p|<|V_a|$, a suitable number of dummy vertices may be inserted into the problem. We do not consider the case $|V_p|>|V_a|$.

A mapping of problem modules onto processors is denoted by the function $f_m:V_p \xrightarrow[\text{onto}]{1-1} V_a$.

The quality of a mapping is determined by the number of problem edges that fall on array edges. We call this number the cardinality of the mapping, denoted $|f_m|$.

The cardinality of a mapping f_m is

$$|f_m| = \frac{1}{2} \sum_{\substack{x \in V_p \\ y \in V_p}} G_p(x,y)*G_a(f_m(x),f_m(y)).$$

This formula arises as follows. $G_p(x,y)=1$ if x and y in the problem graph are connected by an edge. $f_m(x)$ $[f_m(y)]$ represents the processor onto which problem module x [y] is mapped. The expression $G_a(f_m(x),f_m(y))=1$ only if the processors onto which x and y are mapped are connected. Thus the expression inside the summation sign is 1 only if an edge connecting two modules falls on an edge connecting two processors. In summing over all $x \in V_p$ and $y \in V_p$ each processor edge is counted twice, hence the multiplying factor.

To find the best mapping, we must choose a function f_m that has maximum cardinality from among the $(|V_p|)!$ possible functions.

III. Problem Equivalences

In this section we show that the mapping problem, in its most general form (i.e. given arbitrary G_a and G_p), is computationally equivalent to the graph isomorphism problem. We point out the strong similarities between the mapping problem and the bandwidth reduction and quadratic assignment problems.

Graph Isomorphism

Two graphs G_1 and G_2 are said to be isomorphic to each other if there is a one-to-one correspondence between their vertices and between their edges such that the incidence relationships are preserved [1]. This may be stated more formally as follows: two graphs $G_1:V_1 \times V_2 \rightarrow \{0,1\}$ and $G_2:V_2 \times V_2 \rightarrow \{0,1\}$ with $|V_1|=|V_2|$ are isomorphic if there exists a function

$e:V_1 \xrightarrow[\text{onto}]{1-1} V_2$ such that

$G_1(x,y)=G_2(e(x),e(y))$ for all $x,y \in V_1$ [2].

The problem of determining whether two graphs are isomorphic is one of the classical unsolved combinatorial problems. Exact efficient (i.e. polynomial time) algorithms for solving this problem for arbitrary graphs are not known although numerous researchers have attacked this problem [3]. Some researchers have reported success with heuristic algorithms applied to various restricted classes of graphs(see, for example [4]). It appears unlikely that an polynomial time solution to the general problem will ever be found.

We now show that if we had an exact algorithm for solving the general mapping problem, we would also be able to solve the graph isomorphism problem.

If two graphs are isomorphic, they must have the same number of edges and thus a function mapping G_1 onto G_2 and having cardinality equal to the total number of edges must exist. If we had an exact algorithm for solving the mapping problem, we could use it to map G_1 onto G_2 and, if the two were isomorphic, obtain a mapping of cardinality equal to the number of edges. Thus we could answer "yes" or "no" to the question "Are G_1 and G_2 isomorphic?" in polynomial time, for arbitrary G_1 and G_2, if we could solve the mapping problem in polynomial time for arbitrary G_1 and G_2. The mapping problem is therefore computationally equivalent to the graph isomorphism problem and we do not hold much hope for finding an exact polynomial time algorithm for its solution.

Bandwidth Reduction

The bandwidth reduction problem requires the permutation of the rows and columns of a sparse square matrix so as to cluster the non-zero entries as closely as possible about the main diagonal [5]. The mapping problem, as will become clear in the following sections, entails permuting the rows and columns of the adjacency matrix of a problem graph so that it resembles as closely as possible the adjacency matrix of the graph of the array of processors. Arrays of processors that have a regular interconnection pattern (as does the FEM), usually have an adjacency matrix composed mostly of several well-defined bands. The mapping problem for such arrays entails permuting the input matrix so that as many entries as possible fall on the bands. The similarity with bandwidth reduction is obvious.

The bandwidth reduction problem is known to be NP-complete [6]. Many heuristic algorithms for this problem have been developed [5],[7].

The Quadratic Assignment problem

In this problem we are given (1) a set of n objects alongwith a cost matrix in which each entry c_{ij} is a measure of the affinity between objects i and j, and (2) a set of n locations with a distance matrix in which entry d_{st} stands for the distance between locations s and t. A function p that maps objects onto locations is called an assignment. The problem of finding the assignment that minimizes $\sum_{i,j} c_{ij} d_{p(i)p(j)}$ is called the quadratic assignment problem[8]. This problem is exemplified by the task of locating electrical assemblies in given slots so as to minimize the total length of interconnecting wires. No efficient algorithm for the solution of this problem is known.

If the affinity and distance matrices be symmetric and have 0,1 entries, the quadratic assignment problem reduces to the mapping problem.

IV. The Finite Element Machine

The Finite Element Machine (FEM), presently under development at NASA Langley Research Center, is an array of microcomputers interconnected in an "eight-nearest neighbor" interconnection pattern (Fig. 1) [9],[10]. In addition to the nearest neighbor links, which are dedicated to communication between specific pairs of processors, there is a time shared global bus (not shown in Fig. 1) which is used for communication between pairs of nodes that are not adjacent.

The machine is to be used to solve structural analysis problems as follows. The structure is first reduced to a combinatorial graph. The edges of the graph correspond to structural members and the nodes to meeting points of the members (Fig. 2). Each node is assigned to a processor of the FEM and computation proceeds in parallel, as described by Jordan [9]. During the course of the computation, there is communication between pairs of processors only if the structural nodes mapped on them are connected in the physical problem. Thus, should an edge of the physical problem fall on an edge of the FEM, communication proceeds with greatest efficiency via the dedicated nearest neighbor connection. Should this not be the case, interprocessor communication must employ the time-shared global bus with consequent degradation in performance.

Mapping Structures onto the FEM

Fig. 3 shows the adjacency matrix of a 6 X 6 FEM. One possible way of mapping the problem structure of Fig. 2 onto the FEM would be to map node i of the structure onto node i of the FEM. This mapping is indicated in the adjacency matrix of the structure (Fig. 4) by the use of '*'s where an edge of the structure falls on an edge of the FEM, and '0's otherwise. The cardinality of this mapping is 32, while the total number of edges is 80.

We can attempt to increase the cardinality of the mapping by renumbering the nodes of the problem or, equivalently, permuting rows and columns of the adjacency matrix of the problem. The bottom part of Fig. 4 shows an improved mapping, with cardinality 74, obtained by applying the mapping algorithm that will be described in the following section . The permuted row and column labels indicate the renumbering that must be done to the nodes of the problem in order to obtain this improved mapping.

V. MAPPER: A Pairwise Interchange Algorithm

We have developed a heuristic algorithm that accepts as input the adjacency matrix of a problem graph and outputs a permutation of this matrix that matches more closely the adjacency matrix of the FEM.

The algorithm proceeds by sequences of pairwise interchanges, alternating with probabilistic jumps. It starts by accepting the problem matrix and the size of the square FEM onto which it is to be mapped. It then generates the adjacency matrix of the FEM and uses this for comparison while improving the mapping.

Most of the following listing is self explanatory. The function CARDINALITY(MAT) returns the cardinality of the mapping defined by the matrix MAT.

```
program MAPPER;
var MAT, BEST: adjacency_matrix;
    DONE, FLAG: boolean;
begin
   input adjacency matrix of problem, MAT;
   {MAT is taken to be the initial mapping}
   input the size of the FEM, n;
   {the FEM is an n X n array}
   generate adjacency matrix for n X n FEM;
   BEST:=MAT; {the best mapping found so far}
   DONE:=false;
      while not DONE do
      begin{MAIN}

         repeat{SEARCH}
            FLAG:=false;
            for each node do

               begin{AUGMENT}
                  1: examine the pairwise
                     exchange of this node
                     with all other nodes;
                  2: select the one which leads
                     to the largest gain in the
                     cardinality of the mapping;
                  3: if largest gain>=0 then
                     make the exchange;
                  4: if largest gain>0 then
                     FLAG:=true;
               end;{AUGMENT}

          until FLAG=false; {end SEARCH}

          if CARDINALITY(MAT)<CARDINALITY(BEST)
             then DONE:=TRUE
          else
            begin{JUMP}
               BEST:=MAT;
               randomly interchange
               n pairs of nodes of MAT;
            end;{JUMP}

        end;{MAIN}
    output BEST;
end.
```

The block SEARCH of this algorithm attempts to improve the mapping by considering all possible pairwise exchanges of node numberings. The exchange that leads to the maximum increase in cardinality of the mapping is made and the process AUGMENT repeated until no further gains are possible. At this point we leave SEARCH and if the mapping found during this pass through SEARCH is better than the best mapping found so far, a probabilistic jump is applied to the mapping and the algorithm returns to block SEARCH.

Pairwise interchanges are not guaranteed to lead to the best mapping and sometimes lead to mappings that are "dead ends" in that they are not very close to optimal and no pairwise exchange can improve them. The algorithm attempts to leave such dead ends by probabilistically "jumping" to nearby mappings that may permit improvement via pairwise interchanges.

The following is a detailed discussion of various aspects of the algorithm.

1. When carrying out pairwise exchanges, we choose for each node the exchange that leads to the largest gain in cardinality rather than the first gainful exchange encountered. We have found that this strategy leads to mappings that are consistently better than those obtained using the second criterion.

2. We make an exchange even if the largest gain encountered is zero. This has little effect at the outset, when interchanges with nonzero gains are easily found. Towards the end of the algorithm, this criterion helps slide past "dead ends" to mappings which, although they have the same cardinality, may permit further improvement.

3. If the number of nodes on the FEM is N=n X n, then the execution of block AUGMENT will take $O(N^2)$ time.

4. The algorithm will exit block SEARCH if no pairwise exchange leads to an improvement. If the cardinality of the mapping found during this pass through SEARCH is better than the one found during the last pass, the algorithm executes JUMP. Here it tries to break out of the "dead end" from which no pairwise exchange leads to an improvement by probabilistically jumping to a nearby mapping, which, although it will almost certainly have poorer cardinality, may lead to a better mapping upon further application of block SEARCH. A copy of the old mapping is saved in BEST, in case the new mapping is poorer.

5. The probabilistic jump described above needs to be far enough from the current mapping to offer the prospect of improvement, but not so far as to undo all the gains made up to this point. We have found that an interchange of n randomly selected pairs of nodes gives the best results.

6. If the mapping found after attempting to augment from a probabilistically disturbed mapping is poorer, the algorithm terminates. We have found that further probabilistic jumps very rarely lead to improvements.

7. For an N node FEM, the cardinality of a mapping cannot exceed 4N. Each pass through loop MAIN must lead to a gain of at least 1. The time required to execute this loop is dominated by AUGMENT, which takes $O(N^2)$ time. The algorithm thus takes $O(N^3)$ time in all.

VI. Performance of the Algorithm

The algorithm has been implemented and tested on about 20 structural problems of 9 to 49 nodes for FEMS of sizes 4 X 4 to 7 X 7. The results are tabulated in Table I. Some of these cases are illustrated in Figs. 5-7. In most cases the algorithm is able to improve the mapping dramatically.

It is difficult to say just how close to optimal the mappings obtained by the algorithm are, since we have no way of knowing what the best mapping for a specific probem is. In cases where the cardinality of the final mapping is close to the total number of edges, we can be sure that it is very close to optimal. For example, the mapping of Fig. 2 was improved from 32 to 74 as shown in Fig. 4. The final mapping is very close to the total number of edges (80) and must therefore be very near optimal. (For this specific example, it is possible to prove that the optimal mapping is of cardinality 78 [11]). In general, we have found that graphs whose input mappings have cardinalities of around 50 percent of the total number of edges or less can usually be improved dramatically.

To get a better idea of the performance of the algorithm, we mapped random permutations of the FEM onto itself. Since the FEM can be mapped perfectly onto itself, the success of the algorithm in doing so gives us some idea of how well it performs on general problems, for which it is impossible to specify the cardinality of the best mapping. The results of this experiment, in which we fed the algorithm 100 random permutations of 5 X 5 and 6 X 6 FEMS are listed in Fig. 8. Histograms for the initial cardinality, cardinality at the end of the first application of SEARCH and the cardinality at the termination of the algorithm are given (the difference between the latter two illustrates the impact of jumping). The algorithm is seen to perform very well in these experiments, suggesting that the results obtained when the algorithm is run on natural structural problems are also of similar quality.

The run times of the implemented algorithm on a CDC Cyber 175 vary from about 1/3 sec. for 4 X 4 problems to 30 sec. for 7 X 7 problems.

VII. Conclusions

The observed run times of our algorithm are quite acceptable for the current prototype 6 X 6 FEM. However, as the growth rate of time is bounded from below by N^2, the algorithm will probably not be suitable for very large arrays (say 32 X 32). For such arrays, entirely different heuristics will need to be developed.

VIII. Acknowledgements

The author is indebted to Dr. Robert Voigt for his constant encouragement of this research. Discussions with Harry Jordan, Wynne Calvert, David Loendorf and Patricia Sawyer have contributed significantly to this research. The author is grateful to Winifred Stalnaker for providing test problems from the National Transonic Facility, and to Brian Thoma for assistance in preparing the manuscript.

IX. References

[1] N. Deo, Graph Theory with Applications to Engineering and Computer Science, Englewood Cliffs: Prentice-Hall, (1974).

[2] R. M. Karp, "Probabilistic Analysis of a Canonical Numbering Algorithm for Graphs," Computer Science Division, Dept. EECS, U. of California, Berkeley, 1978.

[3] D. Corneil and R. C. Read, "The Graph Isomorphism Disease," Journal of Graph Theory, vol. 1, no. 4, Winter 1977, pp. 339-363.

[4] V. M. Kureichik and T. A. Bickart, "An Isomorphism Test for Homogeneous Graphs," Proc. 1979 Johns Hopkins Conference on Information Sciences and Systems, pp. 60-64.

[5] E. Cuthill, "Several Strategies for reducing the Bandwidth of Matrices," in Sparse Matrices and their Applications, Rose & Willoughby, eds., New York: Plenum press, 1972, pp. 157-166.

[6] R. E. Tarjan, "Graph Theory and Gaussian Elimination," in Sparse Matrix Computations, Bunch & Rose eds., New York: Plenum Press, 1976, pp. 3-22.

[7] N. E. Gibbs, W. G. Poole and P. K. Stockmeyer, "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix," SIAM J. Numerical Analysis, vol. 13, no. 2, April 1976, pp. 236-250.

[8] M. Hannan and J. M. Kurtzberg, "A Review of the Placement and Quadratic Assignment Problems," SIAM Review, vol. 14, no. 2, April 1972, pp. 324-342.

[9] H. Jordan, "A Special Purpose Architecture for Finite Element Analysis," Proc. 1978 Conf. on Parallel Processing, August 1978, pp 263-266.

[10] H. Jordan, M. Scalabrin and W. Calvert, "A Comparison of Three Types of Multiprocessor Algorithms," Proc. 1979 Conf. on Parallel Processing, August 1979.

[11] H. Jordan, personal communication.

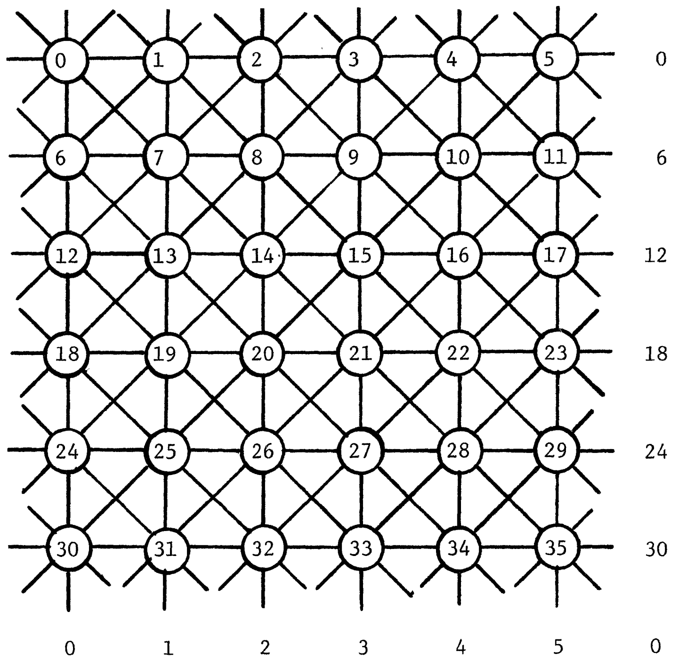


Fig. 1 A 6 X 6 Finite Element Machine (FEM)

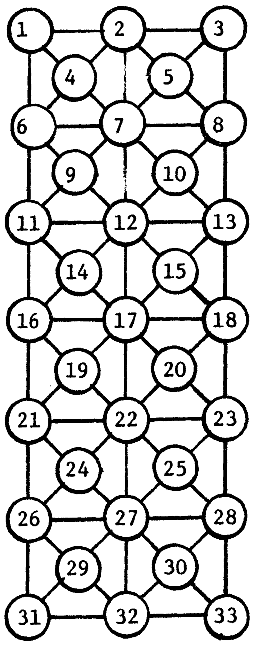


Fig. 2 A 33 node structural problem

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
0		*				*	*	*				*																								
1	*		*				*	*	*																						*	*				*
2		*		*				*	*	*																					*	*	*			
3			*		*				*	*	*																					*	*	*		
4				*		*				*	*	*																					*	*	*	
5	*				*		*				*	*																						*	*	*
6	*	*				*		*				*	*	*				*													*				*	*
7	*	*	*				*		*				*	*	*																					
8		*	*	*				*		*				*	*	*																				
9			*	*	*				*		*				*	*	*																			
10				*	*	*				*		*				*	*	*																		
11	*				*	*	*				*		*				*	*																		
12							*	*				*		*				*	*	*				*												
13							*	*	*				*		*				*	*	*															
14								*	*	*				*		*				*	*	*														
15									*	*	*				*		*				*	*	*													
16										*	*	*				*		*				*	*	*												
17							*				*	*	*				*		*				*	*												
18													*	*				*		*				*	*	*				*						
19													*	*	*				*		*				*	*	*									
20														*	*	*				*		*				*	*	*								
21															*	*	*				*		*				*	*	*							
22																*	*	*				*		*				*	*	*						
23													*				*	*	*				*		*				*	*						
24																			*	*				*		*				*	*	*				*
25																			*	*	*				*		*				*	*	*			
26																				*	*	*				*		*				*	*	*		
27																					*	*	*				*		*				*	*	*	
28																						*	*	*				*		*				*	*	*
29																			*				*	*	*				*		*				*	*
30	*	*				*																			*	*				*		*				*
31	*	*	*																						*	*	*				*		*			
32		*	*	*																						*	*	*				*		*		
33			*	*	*																						*	*	*				*		*	
34				*	*	*																						*	*	*				*		*
35	*				*	*																			*				*	*	*				*	

Fig. 3 Adjacency Matrix for the 6 X 6 FEM shown above.

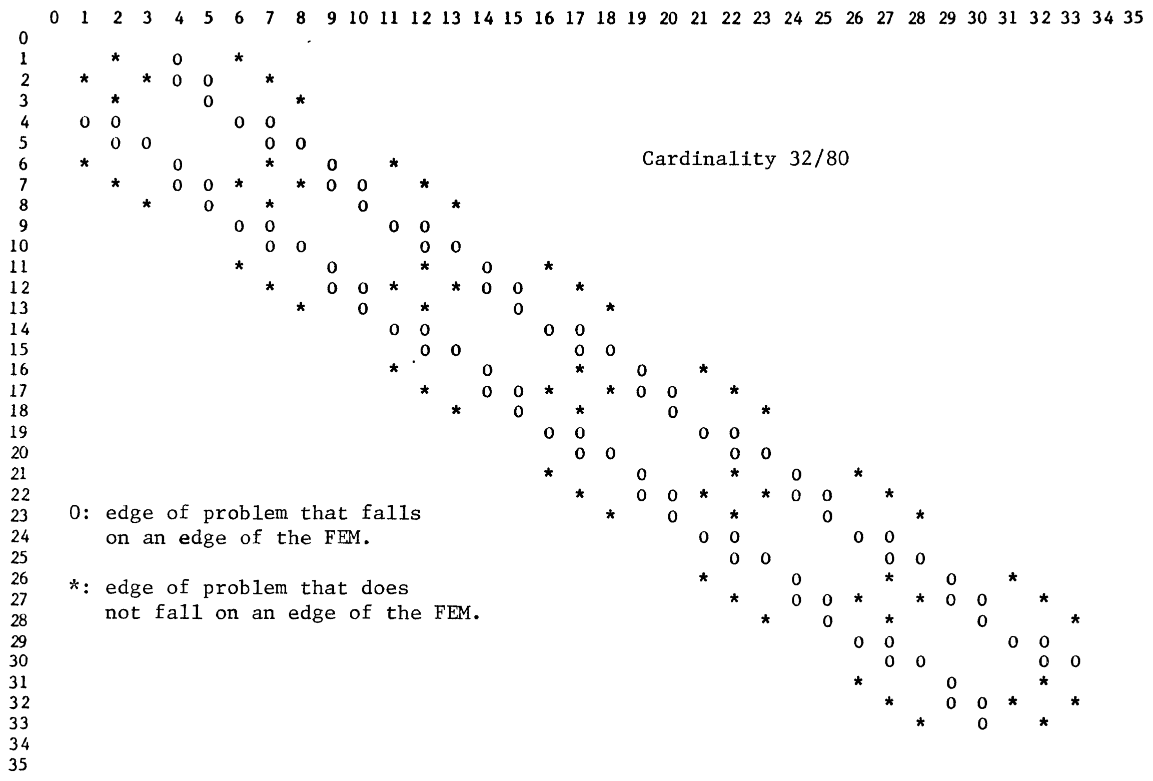

Cardinality 32/80
0: edge of problem that falls on an edge of the FEM.
*: edge of problem that does not fall on an edge of the FEM.

Fig. 4 Initial mapping of the structure shown in Fig. 2 onto the machine of Fig. 1
Above: Initial Mapping. Below: Improved Mapping.

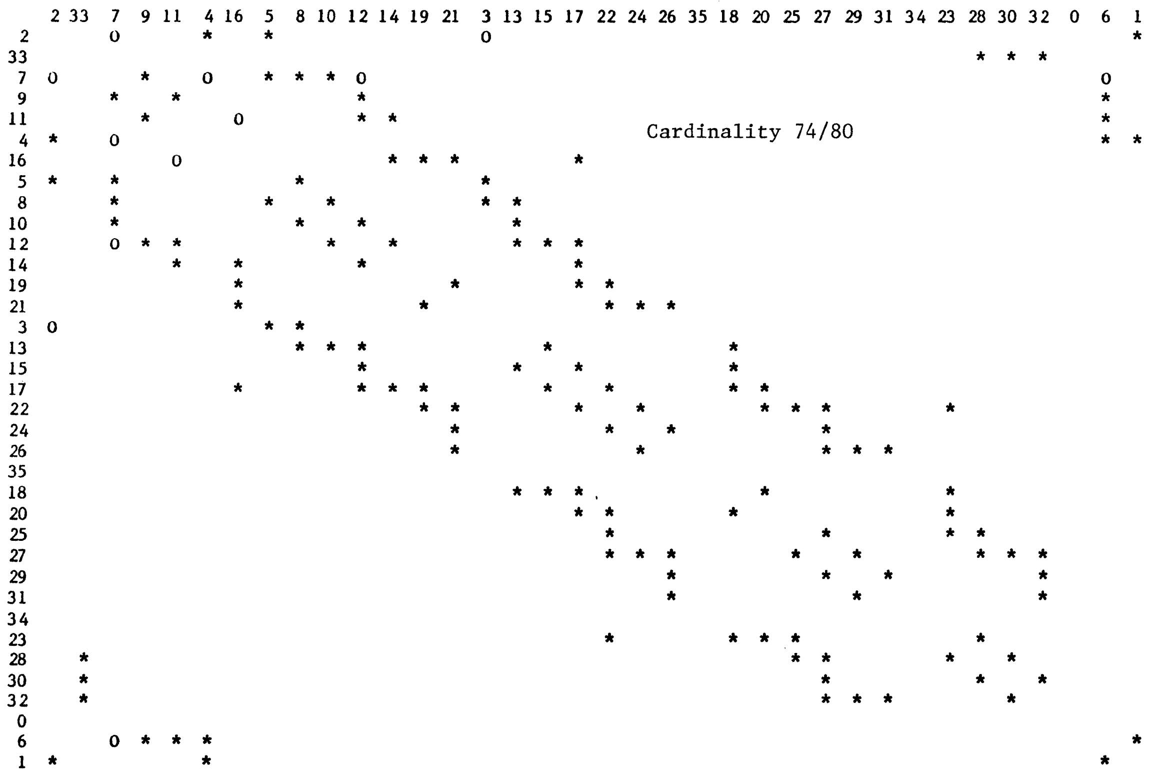


TABLE I

PROBLEM		FEM	TIME	NUMBER OF EXECUTIONS OF		CARDINALITY		EDGES
NAME	NODES	SIZE	MS.	AUGMENT	JUMP	INITIAL	FINAL	
STRUCTURE WITH FREE NODES	33	6X6	11714	14	2	32	74	80
TRUSS	8	4X4	419	5	1	9	15	15
		5X5	1472	5	1	6	15	
		6X6	5083	6	1	6	15	
		7X7	16814	8	1	6	15	
SHIP RADAR TOWER	25	5X5	2912	10	2	44	53	65
SCHWEDLER DOME	6	4X4	321	4	1	6	10	10
+WING BOX	30	6X6	3380	4	1	66	66	78
*NTF-DOWNSTREAM NACELLE	35	6X6	4165	5	1	48	51	58
NTF-NACELLE GUSSET PLATE	39	7X7	29219	14	3	28	49	52
NTF-DOWNSTREAM NACELLE	28	6X6	4224	5	1	37	46	51
NTF-NACELLE BULKHEAD	30	6X6	12650	15	3	27	48	49
NTF-CRADLE	33	6X6	6623	8	1	22	48	52

+The algorithm was unable to improve the given mapping.
*National Transonic Facility.

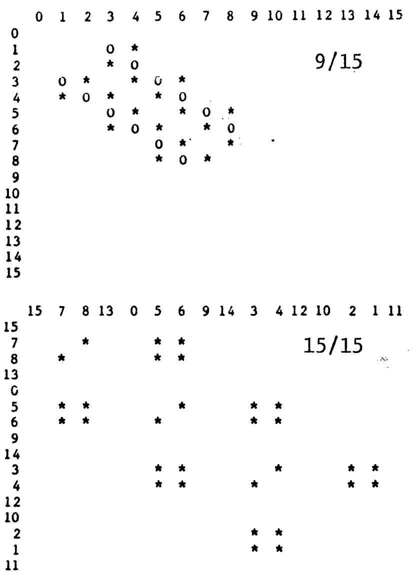


Fig. 5 (Above) Mapping an eight node Truss onto a 4 X 4 FEM

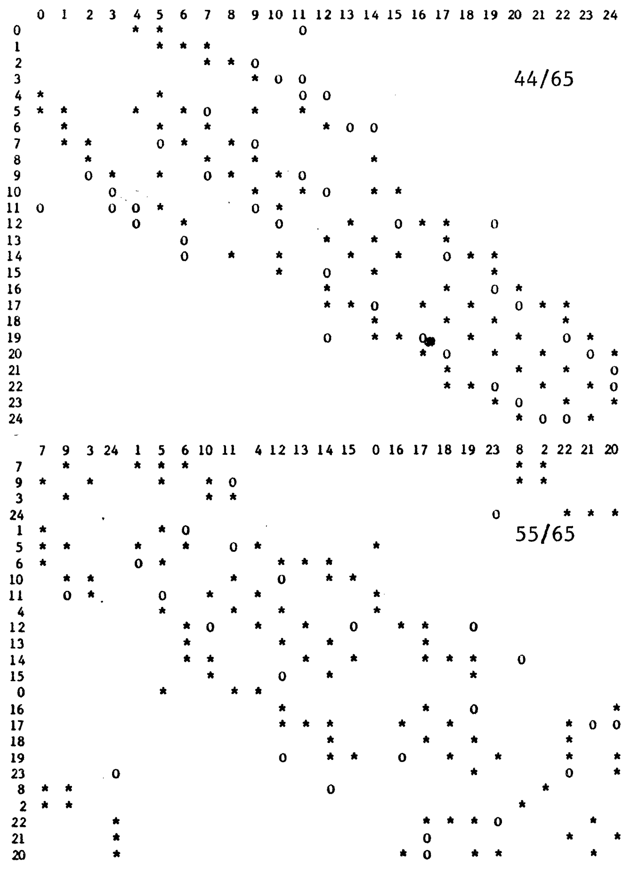


Fig. 6 (Right) 25 node Ship Radar Tower mapped onto 5 X 5 FEM

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
  0     *     o
  1  *     *     o
  2     *           o                                         22/58
  3  o           *     o                          o
  4     o     *     *     o                          o
  5        o     *           o                          o
  6           o           *     o
  7              o     *     *     o
  8                 o     *           o
  9                    o           *     o                                   o
 10                       o     *     *     o                                   o
 11                          o     *           o                                   o
 12                             o           *
 13                                o     *     *
 14                                   o     *
 15           o                                      *     o
 16              o                                *     *     o
 17                 o                                *           o
 18                                               o           *     o
 19                                                  o     *     *     o
 20                                                     o     *           o
 21                                                        o           *
 22                                                           o     *     *
 23                                                              o     *
 24                             o                                               *     o
 25                                o                                         *     *     o
 26                                   o                                         *           o
 27                                                                          o           *     o
 28                                                                             o     *     *     o
 29                                                                                o     *           o
 30                                                                                   o           *
 31                                                                                      o     *     *
 32                                                                                         o     *
 33
 34
 35
```

Fig. 7 NTF-Cradle (33 nodes) mapped onto 6 X 6 FEM. Above: initial ; below: final.

```
     8  9  6  1  0  7 10 11 12  3  4  5 14 13 26 15 16 17 20 29 32 21 18 19 30 31 28 25 22 23 34 27 24  2 33 35
  8                 *     *           *
  9        *           *     *                                                                         *
  6     *           o           *                                 48/58
  1              *                 *                                                                      *
  0           *                 *
  7  *     o           *           *
 10     *           *     *                 *                                         o
 11  *                 *                 *     *
 12     *                                   *
  3        *     *                 *              *
  4           *     *           *     *              *
  5  *                             *                    *                                                 o
 14                       *                 *
 13                    *     *           *
 26                       *                                   *                       o
 15                             *                    *                 *
 16                                *              *     *                 *
 17                                   *              *     *
 20                                                     *                 *                 *
 29                                            *                 *                 *
 32                                                           *                 *
 21                                                                    *                 *
 18                                               *                 *     *
 19                                                  *     *           *                 *
 30                                                                             *                 *
 31                                                                 *           *     *
 28                                                           *                 *  *  *           *
 25                    o                       o                                   *                 *
 22                                                                 *     *                 *
 23                                                        *                             *
 34
 27                                                                          *     *                 *
 24     *                                                                                 *           *
  2           *                       o
 33
 35
```

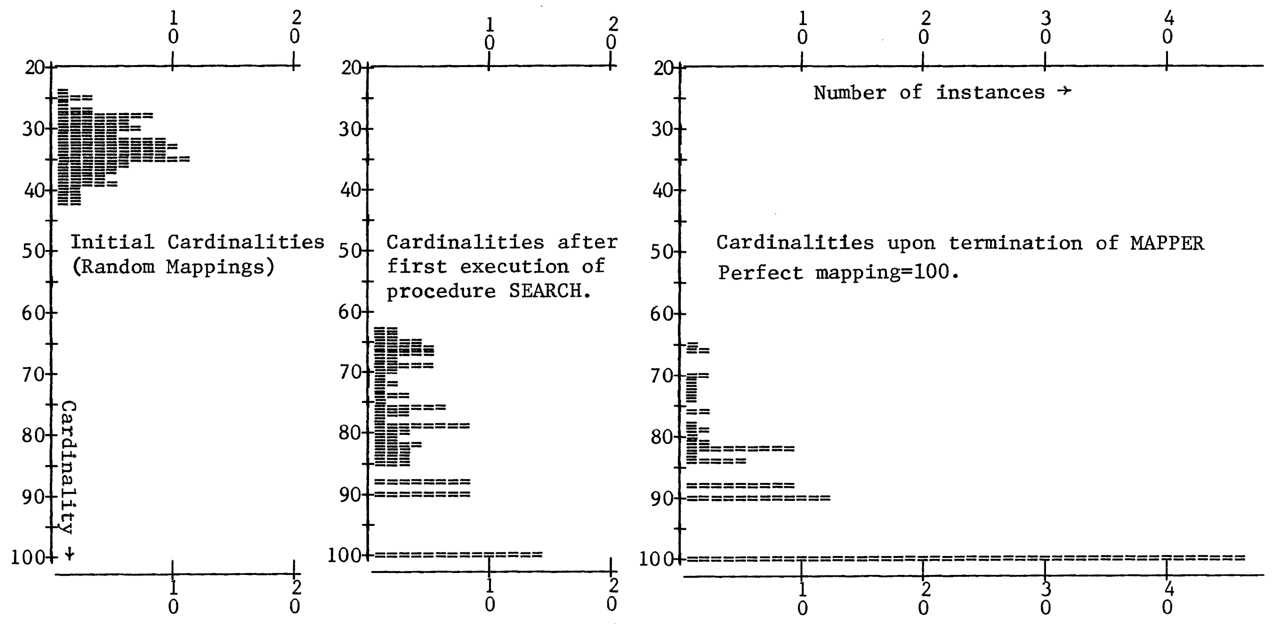
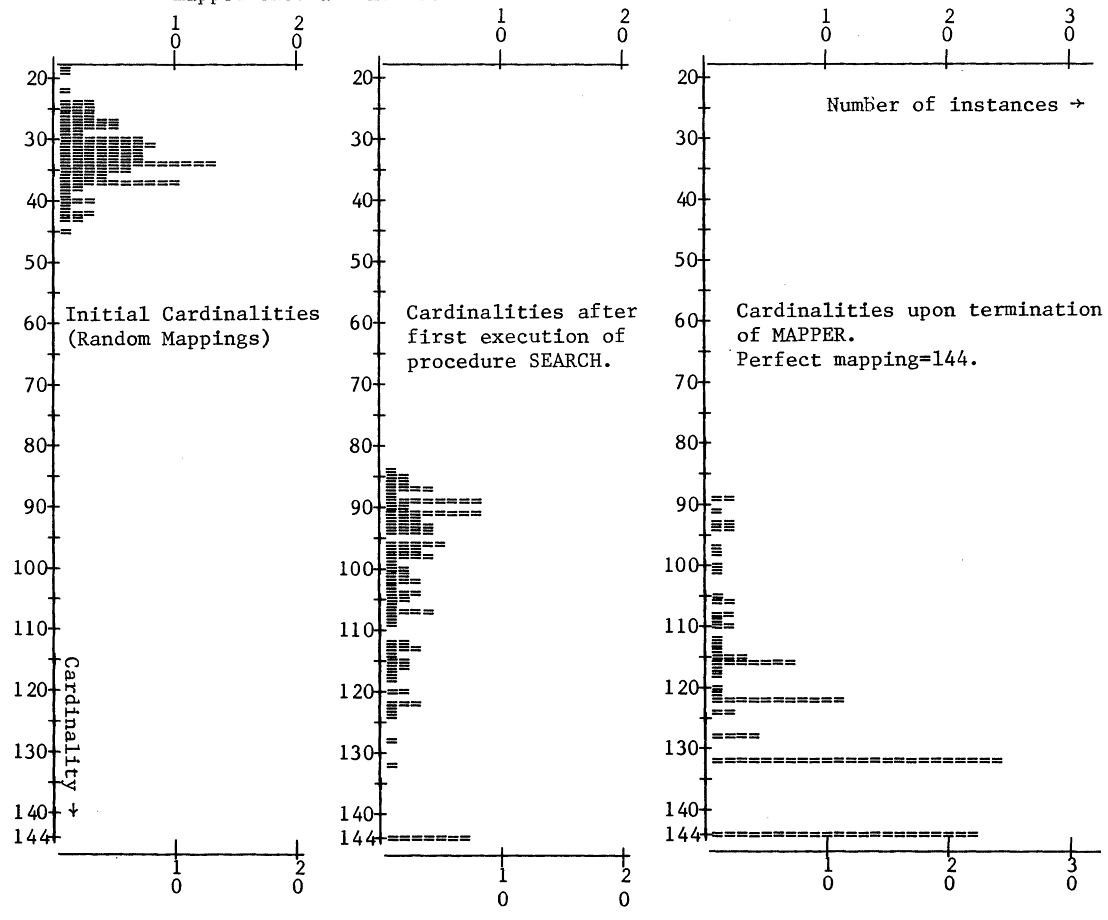


Fig. 8 Histograms showing distributions of cardinalities when 100 randomly permuted FEMs are mapped onto a FEM. Above: 5 X 5 FEMs. Below: 6 X 6 FEMs.



MPP - A MASSIVELY PARALLEL PROCESSOR

Kenneth Batcher
Digital Technology Department
Goodyear Aerospace Corporation
Akron, Ohio 44315

Summary

The processing of satellite imagery requires very fast two-dimensional data processors. For example, the Landsat-D satellite will transmit about a million picture elements (pixels) per second to the ground (on the average). From 100 to 10,000 operations per pixel are required so this satellite alone requires a computer that can perform 100 million to 10 billion operations per second. The Massively Parallel Processor (MPP) is designed to process two-dimensional data such as satellite imagery at high speed.

The main feature of MPP is the large number of processing elements — 16,384 PE's. Since a typical satellite image contains millions of pixels, it is not hard to keep such a large number of PE's busy performing useful work. The PE's are arranged in a 128 x 128 square array with each PE connected to its nearest neighbors on the north, east, south and west. For reliability purposes, an extra 128 x 4 rectangle of PE's is added to the array unit (ARU) and bypass switches are added to the routing network so the failure of any PE can be corrected by simply bypassing the 128 x 4 group containing it.

The length of data items being processed varies widely. Each spectral band of a pixel has a length in the range of 6 to 12 bits. Intermediate results range from 6 bits to over 30 bits. Flags are only 1 bit long. Some items may be in the floating-point format of an attached host computer. The PE's are bit-serial processors so they can accommodate data of any length without waste.

Each PE has a serial adder, a variable-length shift register, a routing-and-logic section, a mask section, an I/O-section and a 1024-bit RAM. The basic cycle time is 100 nanoseconds. Addition, subtraction and searching of integer data occurs with no overhead added to time required to read and write RAM data. Thus, to add two 8-bit arrays and create a 9-bit sum array requires 25 cycles (2.5 microseconds). Multiplication, division and floating-point operations use the shift register. The mask section of each PE is used to turn PE's on and off when required by the application program. The I/O-section shifts input and output data across the array (west-to-east), while other data are processed.

With 16,384 PE's running in parallel, the system is very fast (see Table). Despite the bit-serial nature of the processing, even the floating-point speeds compare favorably with the speeds of several fast number-crunchers.

The nearest-neighbor routing network moves bit-planes north, east, south or west at 100 nanoseconds per step. The edges of the array are either left open or connected to the opposite edges under program control so the array can be given a planar, cylindrical or toroidal topology. An independent I/O-routing network inputs data through a wide port on the west edge and out puts data through another wide port on the east edge. Input and output is overlapped with processing and can transfer data at rates up to 160 megabytes/second.

The array control unit (ACU) has three components. The PE control unit controls the PE's directly. It executes micro-coded subroutines to perform the array arithmetic and logic operations. The I/O-control unit controls array input and output. The main control unit performs fast scalar arithmetic and calls on the other two components for array arithmetic and I/O. Queues allow the operations to be overlapped.

The program and data management unit (PDMU) is a standard minicomputer which executes the program development software (micro-assembler, macro-assembler, linker, loader), allows user interaction, runs diagnostics and manages the flow of data to its peripherals and to a host computer. Imagery data can be transferred between the host (a DEC VAX 11/780) and the array unit at 6 megabytes/second. Images are corner-turned between the pixel format of the host and the bit-plane format of the array unit in a multi-dimensional-access memory.

SPEED OF TYPICAL OPERATIONS

OPERATIONS	EXECUTION SPEED *
ADDITION OF ARRAYS	
8-bit integers (9-bit sum)	6553
12-bit integers (13-bit sum)	4428
32-bit floating-point numbers	430
MULTIPLICATION OF ARRAYS (ELEMENT-BY-ELEMENT)	
8-bit integers (16-bit product)	1861
12-bit integers (24-bit product)	910
32-bit floating-point numbers	216
MULTIPLICATION OF ARRAY BY SCALAR	
8-bit integers (16-bit product)	2340
12-bit integers (24-bit product)	1260
32-bit floating-point numbers	373

*Million Operations per Second

The work reported here was partially funded by NASA-Goddard Space Flight Center under contract NAS 5-25392.

CONCURRENT SEARCH AND INSERTION IN AVL TREES[a]

Carla Schlatter Ellis
Department of Computer Science
University of Oregon
Eugene OR 97403

Abstract -- This paper addresses the problem of concurrent access to dynamically balanced binary search trees. Specifically, two solutions for concurrent search and insertion in AVL trees are developed. The first solution is relatively simple and is intended to allow several readers to share nodes with a writer process. The second solution uses the first as a starting point and introduces additional concurrency among writers by applying various parallelization techniques. Simulation results used to evaluate the parallel performance of these algorithms with regard to the amount of concurrency achieved and the parallel overhead incurred are summarized.

Introduction

Dynamically balanced binary search trees are valuable data structures for implementing symbol tables and directories. This paper deals with the problem of concurrent access to trees built by one of the most widely studied of the balancing techniques, namely, AVL trees. It has been shown [1] that the AVL tree construction is the most efficient method of balancing binary search trees when operations are limited to insertion and searching.

It is not difficult to imagine an application in which concurrent insertion and retrieval of items in a table maintained as an AVL tree would be desirable. For example, a compiler designed to operate in a parallel processing environment might be organized such that several processes require access to the symbol table. In an earlier paper [2] we considered this problem of parallel compilation and found that sharing the symbol table among the proposed parallel processes presented a major conflict. Therefore, investigating the possibility for concurrency in the manipulation of these data structures is important.

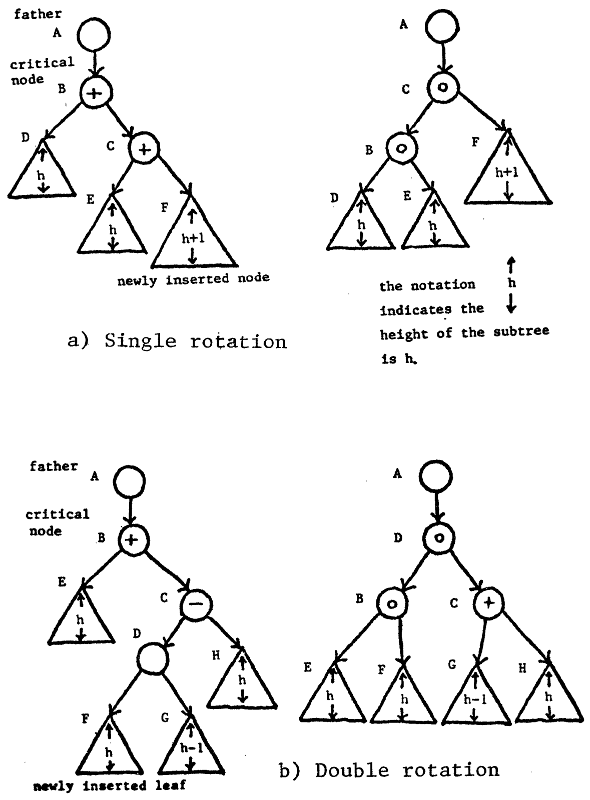
In this paper we present algorithms for concurrent search and insertion in AVL trees. This work is related to similar studies with B-trees [3, 4, 7] and uses the same basic approach of placing locks on nodes of the tree. The presentation begins by defining our notation in terms of the AVL insertion algorithm for a sequential environment. Next, we outline the parallelization techniques applied in the design of two solutions for concurrent search and insertion. Finally, simulation results on the performance of these parallel algorithms are summarized.

Definitions

We assume the reader is familiar with the terminology and operations associated with binary search trees. An AVL tree is defined to be a binary search tree such that for any node n in the tree,

$|\text{height}(T_l(n))-\text{height}(T_r(n))| \leq 1$ (where $T_l(n)$ and $T_r(n)$ denote the left and right subtrees of n).

Detailed algorithms for manipulating AVL trees can be found in [6]. However, we will briefly describe the insertion algorithm in order to establish the terminology and because an understanding of the sequential algorithm is necessary to understand the concurrent algorithms. Each node n consists of four fields: leftson and riteson, pointers to the roots of $T_l(n)$ and $T_r(n)$ respectively or to NIL if the subtree is empty, a data field called key, and bf, which indicates whether the height of the right son is greater than (bf=+1), equal to (bf=0), or less than (bf=-1) the height of the left son. The balanced property of AVL trees is maintained by two transformations on the tree called single and double rotations. The situations which trigger each type of rotation and the modifications made to the tree are illustrated in figure 1.



a) Single rotation

b) Double rotation

Figure 1 Rotations in AVL Tree.

(a) This work was supported by NSF Grant MCS 76-09839.

The algorithm for insertion of a leaf with key k in an AVL tree is as follows:

1) Search the tree to find the appropriate place of insertion and keep a pointer to the last node on the path of insertion with nonzero bf (the root if no such node exists). This is the critical node, cn. Insert the new leaf.

2) Adjust the bf fields of nodes on the insertion path between the cn and the new leaf. For each such node, n, if the path to the place of insertion is to its left, k < key(n), bf(n) is changed to -1; otherwise, bf(n) becomes +1.

3) Rotate if necessary: If bf(cn)=0, the tree has become higher in the direction of insertion and bf must be modified appropriately. If bf(cn) indicates that its higher subtree is in the opposite direction from the direction of insertion (e.g. bf(cn)=+1 and k < key(cn)), bf is changed to 0. If bf(cn) and the direction of insertion coincide (e.g. bf(cn)=-1 and k < key(cn)), a rotation occurs according to figure 1.

Concurrency in AVL Trees

We now consider two solutions that allow a number of processes to operate concurrently on an AVL tree. Both solutions use various locks on the nodes of the tree to selectively exclude other processes.

Locking solution

In the first solution the goal is to allow concurrency between a number of readers and a writer doing an insertion. The approach is straightforward; namely, during its search phase a writer will lock other writers out of those nodes which may be involved in a rebalancing operation. Readers will be locked out of the fewest nodes possible and only during a rotation. Thus readers can share nodes with a writer while it is searching for the place of insertion and the critical node, adjusting bf fields along the insertion path, and determining if a rotation is necessary. The solution uses three types of locks: RHO-LOCKs for readers, ALPHA-LOCKs for excluding other writers along the path from the father of the critical node to the place of insertion, and XI-LOCKs to exclude readers from nodes modified during a rotation.

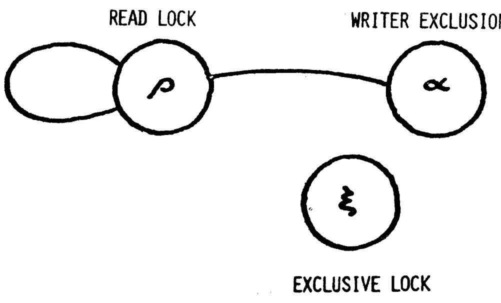
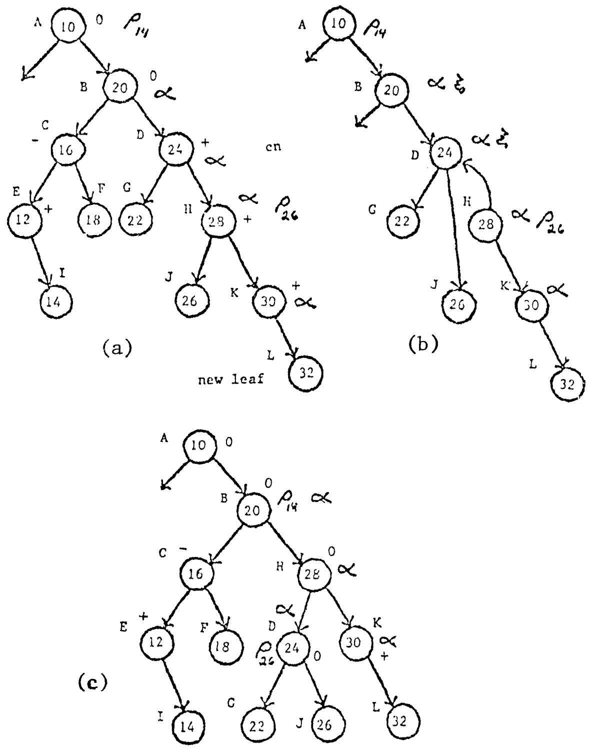


Figure 2 Compatibility graph for locks.

Figure 2 shows the compatibility relations these locks satisfy. An edge between any two nodes in this graph means that two different processes may simultaneously hold these locks on the same node of the tree. Thus a node may be RHO-LOCKed by several readers while it is ALPHA-LOCKed by one writer. However, if a writer holds a XI-LOCK on a node, no other process can hold any other locks on it. A single rotation requires XI-LOCKs on the father of the critical node and the critical node. A double rotation requires an additional XI-LOCK on the son of the critical node which lies on the insertion path. Figure 3 gives an example of concurrent single rotation and read operations.



	writer inserting k=32	reader 1 searching for k=26	reader 2 k=14
	insert	.	.
	adjust balance factors	.	.
		RHO-LOCK H	.
		release RHO-LOCK on D	
			RHO-LOCK A
fig. a			
	XI-LOCK B		
	XI-LOCK D		
			compare 14 with key[A]
			approp. son is B
			wait to RHO-LOCK B
	riteson[D]← J		
	leftson[H]← D		
fig. b			
		determine approp.	
		son is D	
		wait to RHO-LOCK D	
	riteson[B]← H		
	adjust bf		
	release XI-LOCKs		
			RHO-LOCK B
			release RHO-LOCK on A
		RHO-LOCK D	
		release RHO-LOCK on H	.
fig. c		.	
	release ALPHA-LOCKs	.	.
	and terminate	.	
			.

Figure 3 Concurrent single rotation and reading

The algorithms for the reader and writer are given below:

READER

```
    RHO-LOCK pointer to root;
    current <- pointer to root;
    son <- root;
    while son ~=NIL and κ~=κey[son] do
    begin
       RHO-LOCK son;
       release RHO-LOCK on current;
       current <- son;
       /* determine appropriate son */
       if κ < κey[current] then
       son <- leftson[current]
       else son <- riteson[current]
    end;
    release RHO-LOCK on current;
    if son = NIL then fail
    else succeed
```

WRITER

```
    ALPHA-LOCK pointer to root;
    current <- pointer to root;
    father of cn <- current;
    son <- root;
    cn <- root;
    /* search - resulting in path  from father of
    cn to place of insertion remaining locked */
    while son ~= NIL and κ~=κey[son] do
    begin
       ALPHA-LOCK son;
       if bf[son]~=0 then begin
          /*change cn pointer*/
          father of cn <- current;
          cn <- son;
          release ALPHA-LOCKs on ancestors
           of current
       end;
       current <- son;
       determine appropriate son
    end;
    if son = NIL then insert new node with
    κey = κ as appropriate son of current
    else release all ALPHA-LOCKs held by this
    process and terminate
    /* adjust balance fields between cn and
    new node as in sequential insertion */
    if κ<κey[cn] then begin
       direction <- -1;
       r <- current <- leftson[cn]
    end
    else begin
       direction <- +1;
       r <- current <- riteson[cn]
    end;
    retrace path from current to new node
    changing bf appropriately;
    /* rotate if necessary */
    case on bf[cn]
       0: bf[cn] <- direction;
       -direction: bf[cn] <- 0;
       direction: if bf[r] = direction then
          begin
             XI-LOCK father of cn;
             XI-LOCK cn;
             do single rotation;
             release all XI-LOCKs held by
             this process
          end
          else begin
             XI-LOCK father of cn;
             XI-LOCK cn;
             XI-LOCK r;
             do double rotation;
             release all XI-LOCKs held by
             this process
          end
    esac
    release all ALPHA-LOCKs held by this process
```

In this algorithm, a writer ALPHA-LOCKs its path during the search phase so that the nodes along this path from the father of the cn to the place of insertion remain locked for the insertion, rebalancing, and rotation operations. This has the effect of locking subsequent writers out of the entire subtree rooted at the father of the cn rather than just those nodes on the first writer's path.

Claiming solution

The second solution uses the first as a starting point and introduces additional concurrency through the use of various parallelizing techniques. The goal in this next algorithm is to increase concurrency among writers by allowing writers whose restructuring paths (i.e. the path between the father of cn and the newly inserted node) are disjoint to operate concurrently. The solution discriminates against writers that share the same path. Hopefully, the κeys to be inserted by the parallel writer processes will tend to be spread throughout the tree rather than clustered.

The strategy used is summarized below: The writer process will search for the place of insertion using RHO-LOCKs and will place an exclusive claim on the probable father of cn (the cn pointer is set to the last node on the insertion path with nonzero bf whose father is not already claimed by another writer. As we shall see, this may not be the true critical node if another writer shares the insertion path). This node is claimed rather than locked so that another writer may read past it and place a claim within this subtree if another potential cn is found. The new node with κey κ will be inserted while other inserting writers are excluded from the place of insertion. In its rebalancing phase, the writer excludes other rebalancing and rotating writers from nodes on the path from the father of cn to the first new node encountered on the insertion path (note that this new node was not necessarily inserted by this writer) and balance fields between cn and the new node will be adjusted. It is during the restructuring phase that writers which share the same path are penalized: one writer will claim the father of the lowest node with nonzero bf. Other writers will claim nodes higher in the tree and may find a lower cn during their rebalancing operations. Then the cn pointer must be moved down and the bf fields readjusted. Thus writers along a shared path may do useless work that will need to be undone. Finally, rotation will take place if necessary with XI-LOCKs protecting the nodes involved.

This solution requires a modification in the data structure of the previous algorithm. In addition to the key, leftson, riteson, and bf fields, each node will contain one field, the guardian field, which will indicate which process is responsible for the rebalancing and rotation operations associated with the insertion of this node. If those operations have already been taken care of, this field indicates that this is an "old" node. In practice, this could be implemented with a single bit guardian field to signify "old" or "new" and an associative table pairing new nodes with their guardian processes. Or since only three codes are used in the two bit bf field and all new nodes have bf=0, the remaining code could be utilized to indicate a new node, thus eliminating the guardian field altogether. This algorithm also uses a slightly different locking scheme. We will still need RHO-LOCKs for reading and XI-LOCKs for exclusion of other processes during rotation. IOTA-LOCKs will be used to enforce mutual exclusion among writers trying to insert a new node at the same place. The most significant change lies in replacing the ALPHA-LOCK of previous algorithms with an ALPHA′-LOCK and a mark bit that explicitly implement a lock to enforce mutual exclusion among rebalancing and rotating writers. The special feature of this implemented lock is that in addition to the operations of requesting the lock (which implies the process is to wait if the request can not be granted) and releasing the lock, a process will be able to test the mark bit to determine whether or not a node is locked without waiting for a lock request to be granted. The new compatibility relation is shown in figure 4.

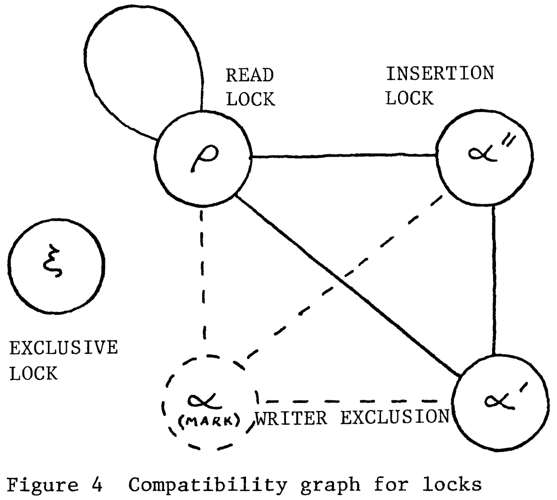


Figure 4 Compatibility graph for locks in Claiming solution.

There are a few key ideas that promote concurrency in this solution. The first important technique is to allow a temporary degradation of the tree structure. Since a writer can search and insert with a nominal amount of interference from other processes, it is possible that the tree could become quite unbalanced (i.e. no longer satisfying the AVL definition) after a number of processes have inserted but not yet restructured (cf. figure 5). The second technique is a relaxation of a process's responsibility to do its own work. Let n_1 and n_2 be two newly inserted nodes such that n_1 is an ancestor of n_2. The restructuring operations associated with n_1 should be done before those associated with n_2, but it is possible that the writer which inserted n_2, process 2, performs its rebalancing phase before process 1. In this solution, the two processes will essentially trade responsibilities with process 2 rebalancing for n_1 and process 1 rebalancing for n_2. Because of the top-down nature of the restructuring pass, this trading must be explicitly done. A message will be sent to process 1 telling it to search for n_2's key during its restructuring pass. The final technique is made possible by the new locking scheme. The new lock has a different effect on searching writers than on rebalancing writers, thus essentially delaying the blocking of one process by another.

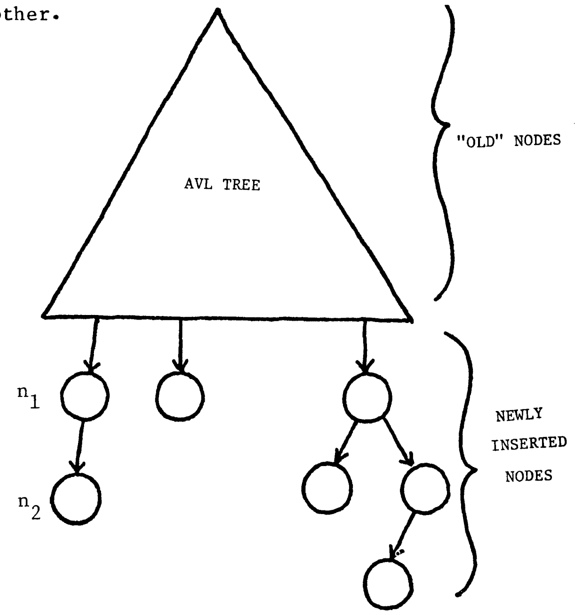


Figure 5 Modified AVL Tree.

Readers in this solution are identical to readers in the Locking solution. The writer algorithm is given below. We first present the procedures called by the main program followed by the main program itself.

```
procedure TRY TO CLAIM;
  begin
    ALPHA′-LOCK current;
    if bf[son] ~= 0 and current is unmarked
    then begin
      mark current;
      cn <- son;
      if claim ~= pointer to root then
      unmark claim;
      claim <- current
    end
    release ALPHA′-LOCK on current
  end
```

```
procedure TRY TO INSERT;
   begin
      IOTA-LOCK current;
      determine appropriate son of current
      again;
      if son = NIL then
         insert newnode with key = k;
      release IOTA-LOCK on current
   end

procedure WAIT TO MARK(node);
/* essentially ALPHA-LOCKing */
   begin
      ALPHA'-LOCK node;
      while node is marked do
      begin
         release ALPHA'-LOCK on node;
         while node is marked do
         ALPHA'-LOCK node
      end;
      mark node;
      release ALPHA'-LOCK on node
   end

WRITER

   RHO-LOCK pointer to root;
   claim <- current <- pointer to root;
   son <- cn <- root;
   /* search and claim potential father of cn */
   while son ~= NIL and k~=key[son] do
   begin
      RHO-LOCK son;
      if bf[son]~=0 and current is unmarked and
      current~=claim then TRY TO CLAIM;
      release RHO-LOCK on current;
      current <- son;
      determine appropriate son;
      if son = NIL then TRY TO INSERT
   end;
   if son ~= NIL then begin
   /* k=key[son] so no insertion necessary */
      if claim ~= pointer to root
      then unmark claim;
      release RHO-LOCK on current;
      terminate
   end;
   release RHO-LOCK on current;
   /* rebalance - mark path from father of cn
   to place of insertion
   and adjust balance factor fields */
   if claim = pointer to root
   then WAIT TO MARK(claim);
   WAIT TO MARK(cn);
   current <- cn;
   while guardian[current] is "old" do
   begin
      if k < key[current] then
      begin
         r <- son <- leftson[current];
         direction <- -1
      end
      else begin
         r <- son <- riteson[current];
         direction <- +1
      end;
      WAIT TO MARK(son);
      while bf[son]=0 and guardian[son]="old"
      do begin
         current <- son;
         if k<key[current] then
         begin
            bf[current] <- -1;
            son <- leftson[current]
         end
         else begin
            bf[current] <- +1;
            son <- riteson[current]
         end;
         WAIT TO MARK(son)
      end
      if bf[son]~=0 then
      /* another process has been rebalancing
      on the path already*/
      begin
         cn <- son;
         claim <- current;
         /* unmark from old claim to node above
         new claim and restore bf fields to zero
         from son of old cn to new claim */
         current <- old cn;
         unmark old claim;
         while current ~= claim do
         begin
            determine appropriate son;
            bf[son] <- 0;
            unmark current;
            current <- son
         end
         current <- cn
      end
      else begin
      /* son hasn't been rebalanced for yet */
         if son is not this process's new node
         then begin
            /* trade new nodes with process
            now responsible for son node*/
            send guardian[son] message to reset
            its k to key[newnode]
            and its newnode pointer to newnode;
            guardian[newnode] <- guardian[son]
         end;
         current <- son
      end
   end
   /* rotate if necessary */
   case on bf[cn]
      0: bf[cn] <- direction;
      -direction: bf[cn] <- 0;
      direction: if bf[r]=direction then begin
         XI-LOCK claim;
         XI-LOCK cn;
         XI-LOCK r;
         do single rotation;
         release all XI-LOCKs held by
         this process
      end
      else begin
         XI-LOCK claim;
         XI-LOCK cn;
         XI-LOCK r;
         XI-LOCK appropriate son of r;
         do double rotation;
         release all XI-LOCKs held by
         this process
      end
```

```
esac
guardian[current] <- "old";
unmark all nodes marked by this process
```

Informal correctness proofs for these solutions can be found in [5].

Evaluation

The primary goal in the design of these parallel algorithms was to increase the amount of concurrency possible between readers and writers and among numerous writers themselves. In the attempt to increase parallelism, a certain amount of parallel overhead was incurred (e.g. locking and unlocking of nodes, extra fields per node). Therefore concurrency and parallel overhead will be the two most important factors to be considered in the evaluation of our algorithms. Results of simulation experiments will be summarized here. For a detailed discussion of the simulation study and a more complete presentation of the results see [5]. Very briefly, the approach is to simulate a fixed number of reader and writer processes executing steps of these algorithms scheduled according to randomly generated execution times which reflect fluctuations due to factors such as memory interference and differences between physical processors.

Among the measures of interest are the average number of concurrently busy processes during an interval of time (where busy means that the process is not waiting to be able to lock a node and is not finished with its operation), the improvement ratio (i.e. time for sequential steps / elapsed time for parallel execution), average path length and longest path (indications of how the tree degrades), and measures of the degree of locking in the tree and the amount of work done in placing and releasing locks.

Figure 6 deals with the amount of concurrency achieved by each solution as the tree grows. The results displayed are based on experiments with 16 readers and 16 writers active in the system. As could be expected, the first solution allows much less concurrency among writers than does the second algorithm. Also not surprisingly, there is a considerable amount of parallel overhead involved in executing these algorithms. One approach to evaluating this overhead requires the processor utilization and the improvement ratio to yield a value x which indicates that the cumulative time to execute all steps of each busy process is about x times the execution time spent doing work that would correspond to steps of a sequential execution. This value is 2.7 for our first solution and 2.5 for the second.

The degradation in the tree structure for the second algorithm is not significant for a randomly chosen set of keys.

The average number of locks placed and released per insertion is used to estimate the overhead cost of locking in the following way:

L=(cost of placing ALPHA-LOCK x average number of ALPHA-LOCKs placed) + (average number of XI-LOCKs placed) + (average number of RHO-LOCKs placed).

Let I be the average insertion path. Then we have for the first solution, L=(3 x I) + 1.1 + 0, and for the second solution, L=(3 x 6.4) + 1.5 + I.

Finally, the maximum number of locks which a

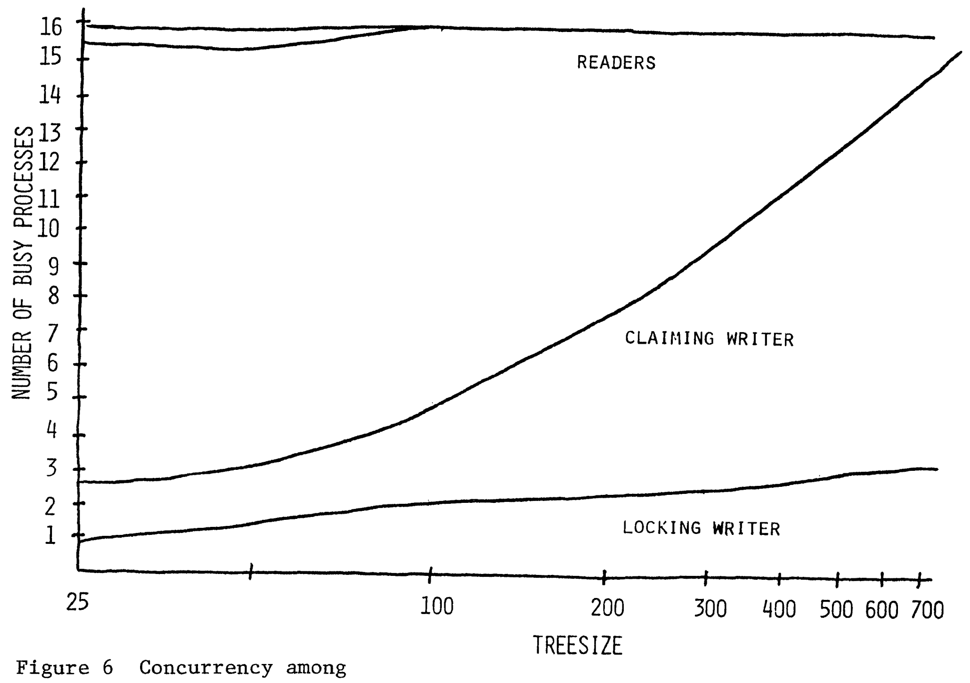


Figure 6 Concurrency among readers and writers.

writer would be expected to hold at some time during its insertion is a measure of potential concurrency. Since there is very little difference between these two solutions with respect to this measure, the concurrency among writers executing the second algorithm can be explained by the delay in locking rather than fewer locks held.

With regard to storage overhead, we compare the requirements of these concurrent solutions with the data structure of the sequential solution. One notable difference is the space which must be devoted to the various locks. In addition, the second algorithm calls for an associative table pairing active writer processes with their newly inserted nodes.

Conclusion

In this paper we have presented algorithms for concurrently searching and inserting in AVL trees. The solutions illustrate parallelizing techniques such as relaxing a process's responsibility to do its own work, allowing limited degradation of the structure, and delaying locking. These techniques should prove to be useful for introducing concurrency in other problems. The measurements presented indicate a reasonable increase in the amount of concurrency achieved by applying these techniques. In spite of the overhead, parallel execution yields a speed-up.

References

[1] Baer, J.-L. and Schwab, B., "A Comparison of Tree-Balancing Algorithms", CACM, Vol. 20, No. 5, (May, 1977), pp. 322-330.

[2] Baer, J.-L. and Ellis, C., "Model, Design, and Evaluation of a Compiler for a Parallel Processing Environment", IEEE Trans. on Software Eng., Vol. SE-3, No. 6, (Nov. 1977), pp. 394-405.

[3] Bayer, R. and Schkolnick, M., "Concurrency of Operations on B-trees", Acta Informatica, 9, (1977), pp.1-22.

[4] Ellis, C., Concurrent Search and Insertion in 2-3 Trees, Dept. of Computer Science, Univ. of Washington, TR-78-05-01, (1978), 38pp.

[5] Ellis, C., Design and Evaluation of Algorithms for Parallel Processing, Ph.D. thesis, Dept. of Computer Science, Univ. of Washington, (June, 1979).

[6] Knuth, D., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison Wesley, (1973).

[7] Miller, R. and Snyder, L., "Multiple Access to B-trees", Proc. Conf. Info. Sci. and Syst., (March, 1978).

A TREE MACHINE FOR SEARCHING PROBLEMS[1]

Jon Louis Bentley[2]
H. T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract -- In this paper we describe a new tree-structured machine (suitable for VLSI implementation) that solves a large class of searching problems. A set of N elements can be maintained on an N-processor version of this machine such that insertions, deletions, queries and updates can all be processed in 2 lg N time units. The queries can be very complex, including problems arising in ordered set manipulation, data bases, and statistics. The machine is pipelined so that M successive operations can be performed in M-1 + 2 lg N time units. In this paper we will study both the basic machine structure and the actual implementation of the machine.

1. Introduction

Very Large Scale Integrated circuitry (VLSI) has been increasing in speed and decreasing in size at an amazing rate over the past several years, and it promises to continue at this rate far into the next decade. In this paper we will describe a tree-structured machine for solving searching problems that is ideally suited for implementation in VLSI. The searching problems that it solves arise in a number of applications areas (including ordered set manipulation, data bases and statistics), and the machine is able to solve all of the problems very efficiently.

Before we describe this machine in detail, it is important to characterize its contribution in general terms. The authors believe that there is a spectrum of impacts that advances in VLSI technology will have on computer architecture. At one extreme, this technology will allow conventional architectures to be implemented as smaller and faster machines -- this will lead to more sophisticated interconnections of conventional machines (see Swan, Fuller and Siewiorek [1977], Browning [1979] and Sequin, Despain and Patterson [1978]). Also at this end of the spectrum will be minor architectural changes that exploit certain features of VLSI; this area has been explored by Sites [1979]. At the other extreme, VLSI architectures have been proposed that are radical departures from the von Neumann tradition (see Backus [1978], Mago [1979] and Wilner [1978]). In this paper we will investigate an approach that lies between these two extremes: a high-performance, special-purpose, non-von Neumann computing device that is designed to be used in conjunction with a typical computer. In general, such devices should be constructed only when they solve a problem satisfying two criteria: the problem should currently consume large quantities of computer time, and the proposed special-purpose device must be much more efficient than conventional ways of solving the particular problem. When such a problem is identified it is reasonable to augment a general-purpose computing system with a special-purpose device for solving the problem; the structure of such a system is depicted in Figure 1. Many such special-purpose devices have recently been proposed; see, for example, Kung and Leiserson [1978].

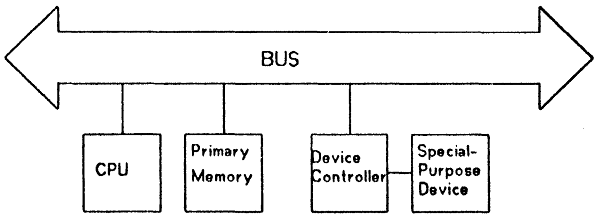


Figure 1. General system structure.

In this paper we will investigate a special-purpose machine for solving searching problems. This machine is described at an abstract level in Section 2, where we will also review some necessary background in searching problems. An architecture (that is, a user's view) of a potential machine is described in Section 3, and issues of implementing that architecture in VLSI are discussed in Section 4. Conclusions are then offered in Section 5.

[1] This research was supported in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551 (monitored by the Air Force Office of Scientific Research) and in part by the National Science Foundation under Grant MCS 75-222-55.

[2] Also with the Department of Mathematics.

2. The Abstract Machine

In this section we will investigate the tree-structured searching machine at an abstract level, apart from the details of architecture or implementation. The general searching problem it solves calls for maintaining a *file* of fixed-format *records*. We must be able to perform the operations of *inserting* a new record into the file, *deleting* an existing record from the file, *updating* records in the file, and *querying* the file to answer questions. Before we examine the general searching problem, we will investigate one searching problem in particular.

That particular problem is called *member searching*. In its abstract form, it calls for maintaining a set of elements so we can determine if a new element is a member of the set. In concrete applications, other information is usually also required. For example, if we find that a particular social security number is a member of a set of social security numbers, then we usually wish to retrieve other information (such as Year-to-Date taxes). We will now investigate how the tree machine solves the abstract member searching problem, and then return in the next section to the complicating issues that arise in applications.

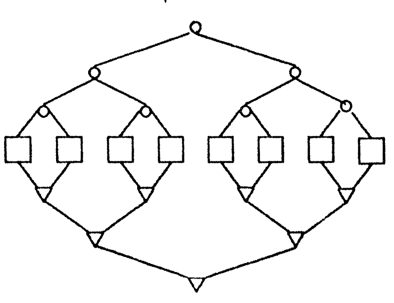


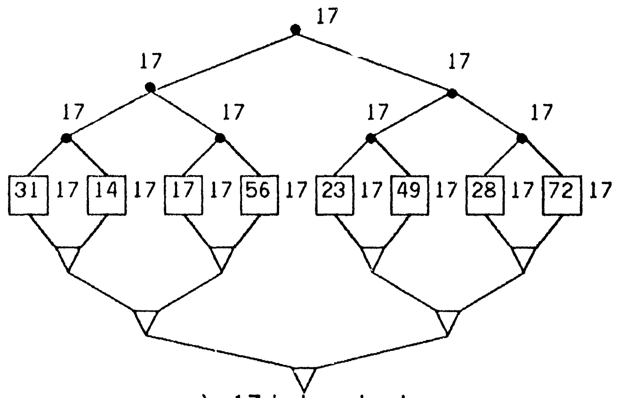
Figure 2. Structure of the tree machine.

The basic organization of the tree-structured searching machine is depicted in Figure 2. There are three kinds of nodes in the machine: circles (which broadcast data), squares (which have limited storage and computation power) and triangles (which "combine" answers to queries). A set of N elements is stored in this machine by placing each element of the set into a distinct square node of the tree. Consider now the problem of performing the member search to answer the query "Is 17 an element of the set?". We accomplish this by inserting 17 into the input node and broadcasting it down the tree -- lg N steps later the value 17 will arrive at all of the squares. This situation is illustrated in Figure 3a. At that point we compare the values stored in each square to 17 and set a bit to one if the value is equal to 17 and zero otherwise; this is shown in Figure 3b. We can now combine the bits together through the bottom portion of the network by letting each triangle compute the logical *or* of its two inputs, as illustrated in Figure 3c. So after a total of 2 lg N time units have passed since the query was posed, a single bit emerges from the output node telling whether or not 17 is an element of the set. We have thus described a procedure for determining whether a given object is a member of the set whose elements are stored in the square nodes.

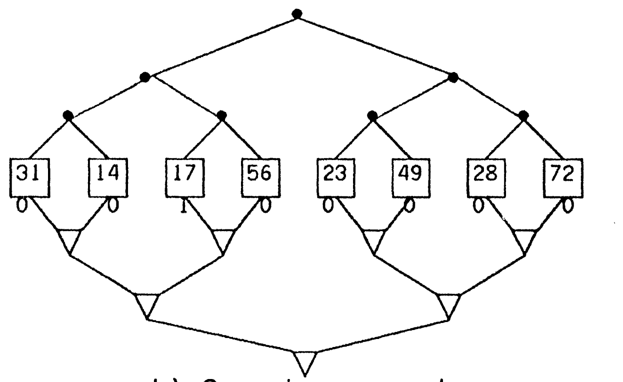
It is important to note that the tree machine has a very regular data flow: the data moves in discrete steps in only one direction (from the input node to the output node). Thus if many successive elements are going to be tested for membership in the set stored in the square nodes, then the process of answering those queries can be pipelined. As the value of the first element to be tested is going down the tree, the next value can follow one step behind, and so on. If M successive tests are performed in this manner, exactly M-1 + 2 lg N time units pass between the entry of the first query at the top of the tree and the exit of the last of the answers at the bottom of the tree.

The tree machine is able to solve many problems besides member searching. For example, if a multiset of elements (that is, a set in which one element can appear many times) were stored in the square nodes of the tree, we might wish to count how many times a given object appears in the set. We proceed exactly as we did for member searching, first broadcasting the given element through the circles to the square nodes. We load a one into each square if it is equal to the given object and zero otherwise, and then combine the answers by letting the triangles *sum* the values of their inputs. Another example is given by nearest neighbor searching. If we wished to find the element of the set that is closest to 17, then we do the following: broadcast 17 through the input node to all squares, subtract the value stored in the square from 17 and take the absolute value of the difference, and finally take the minimum of all those values by having the triangles return the *minimum* of their two inputs. Note that both for member counting and nearest neighbor searching, we can answer a single query in 2 lg N time and a series of M queries in M-1 + 2 lg N time.

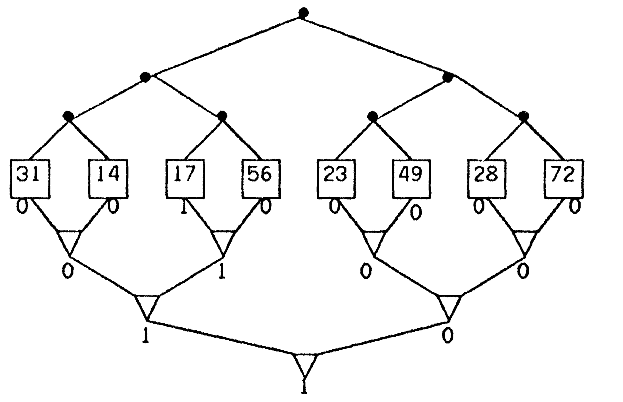
In general, the tree machine can solve any problem that can be phrased as computing some function over every element in the set (such as equality or absolute



a.) 17 is broadcast.



b.) Comparisons are made.



c.) Answer is returned.

Figure 3. A member search.

value of difference) and then combining the values of those functions by some associative, commutative binary operator. For example, the rank of an element X in a set (that is, the number of elements in the set less than X) can be calculated by storing in each square a one if the element is less than X and zero otherwise; the final answer is then computed by having the triangles add their inputs. Other problems defined on totally ordered sets that can be solved by the tree machine include predecessor (what is the greatest element less than the given?), successor (what is the least element greater than the given?), and minimum (what is the least element in the set?). In general, the tree machine can solve all of the "Decomposable Searching Problems" defined by Bentley and Saxe [1979]. That reference contains both an algebraic definition of the class and a list of many particular problems.

The tree machine is also able to answer much more complicated kinds of queries (of the form that arise in data base applications, for instance). Suppose, for example, that every square node of the tree contains a record with ten keys. We might want to know how many records there are in the file with first key equal to a given value, second key at least as great as the third key, the fourth key in a certain range, and so on. This type of query is easily answered: we merely broadcast each of the conditions down to the square nodes, keeping track in each node of whether it has satisfied all the conditions shipped so far. We load a one if all conditions have been satisfied and a zero otherwise, and combine by having the triangles sum their inputs. Many applications call for a list of the satisfying records instead of merely their count, and this can be accomplished by letting the triangles compute the *union* of their inputs. This can be viewed intuitively by observing each triangle independently, and imagining a person "tapping" the entire machine at each time step. As each triangle is tapped, there are three cases to consider: if it has no items in its inputs, it reports that; if it has one item, it returns it; and if it has two items, it returns only one (delaying the other until the next tap). This "tapping" process continues as long as there are elements that have yet to be reported (note that to compute unions in this manner, the pipelining must be "turned off").

Having discussed searching at some length, we will now turn to the issues of maintaining the set of elements stored in the square nodes. A tree machine with N square nodes (where N is a power of two) can store up to N records. A new record can be inserted into the set by placing it in an unused square. We find such a square by having each circle keep track of the number of unused descendants in each of his two sons. When a request comes to the root for a new (unused) position, he passes the request to one of his non-empty sons, and so on. Mechanically, this is accomplished by turning off all of the processors except the one finally chosen as the holder of the new record; this processor is then loaded with the desired data. Note that a single record can be inserted in lg N steps, and a set of M records can be inserted in M-1 + lg N steps.

Another maintenance operation is that of updating a set of records: this can be easily accomplished by broadcasting the conditions that the changed records must meet, turning off all processors that do not meet the conditions, and then making the desired changes. (Although the update set will often have just one element, an example of a "mass update" might be processed on the first of the month: for all salesmen with Month-Of-Starting-Employment equal to This-Month, add one to Years-Of-Service.) To delete a single record we set a flag in its square node saying that it is unused and then adjust the counts in all of the circles above it. This can be accomplished either by pushing information "backward" to the top of the tree (adding one to each counter as you go), or by doing a dummy reinsertion of that element, and modifying the counters on the way down. The time for either of these operations is proportional to lg N. Notice that if a set of elements is to be deleted, this can be accomplished in parallel and all counters can be reset (by pushing the information up the tree) in lg N steps. Although having information go up the tree is handy for deletion, it does complicate the basic design severely; this feature might therefore not be implemented.

So far in our discussion each machine has represented but one set. In some applications, however, a given user might wish to represent many sets, or many users might want to use the machine independently. Either of these can be accomplished so long as the sum of the sizes of the sets is less than N, the number of square nodes. Although we might be tempted to "slice" the machine into sections to accomplish this, there is a much more elegant solution. Namely, a fixed portion of each record is dedicated to a "set identification field", or "SetID". To process an operation on Set 56 (or a set belonging to user 56), we have as a prelude to the operation the sequence "check SetID for equality with 56 and turn off the processor if not equal". (Notice that we are not requiring that all records in all sets be of the same format, but just that they have one field in common.) In an environment with much sharing, this prelude will occur so often that it might be advantageous to provide a single instruction that accomplishes its purpose.

Although so far we have used the tree machine to solve only searching problems, it can be applied to many other problems as well. For instance, it can be used to sort a set of M elements in time proportional to M (as long as M is less than N, the number of square nodes in the tree machine). This is accomplished by making two passes through the M elements: the first inserts the elements into the machine, and the second counts for each element the number of elements less than it (that is, it computes the element's rank, as we saw before). This tells precisely where each element occurs in sorted order (the output is a permutation vector), and it is then trivial to arrange the elements into sorted order. By use of pipelining, both steps run in time linear in M. Note that it was critical to phrase sorting as a counting problem, rather than as extracting the minimum, to make use of pipelining in the second step -- this algorithm essentially implements an N^2 algorithm in N time by using all N processors in parallel. There are many other examples of such speedups for problems that are not *prima facie* searching problems. Two such examples are computing all nearest-neighbor pairs in a k-dimensional point set (which arises in data analysis) and reporting all pairwise design rule violations in a VLSI mask (a design automation task). The application of this machine to the problem of constructing minimum spanning trees has been discussed by Bentley [1979].

This concludes our discussion of the machine at an abstract level, and we can now state the properties that a concrete embodiment of the machine must possess. There must be three kinds of nodes in such a machine: circles, squares and triangles. The triangles must broadcast data and have a small amount of state (namely, remember how many unused squares are descendants of each of their sons). The only processing required of a triangle is incrementing or decrementing by one. The squares, however, must have substantial memory and computation power. Each square must have enough memory to store the largest record in any forseeable application, and enough processing capability to handle the most difficult kinds of queries and updates desired. The circles must be able to combine answers. Most of the "combinators" we desire are very simple to implement; these are *and*, *or*, *min*, *max*, and *plus*. The only complicated combinator is *union*, and we are willing to "turn off" pipelining in the presence of that operator.

3. An Architecture

In Section 2 we investigated the tree-structured searching machine at an abstract level, ignoring many issues of implementation. In this section we will move one step closer to an implementation, and describe a particular architecture (that is, a user's view of the machine) realizing the abstract machine. It is essential that the reader understand that the architecture we will investigate is not proposed as the best possible architecture realizing the abstract machine of the last section. Rather, it is put forth only as evidence that there is *at least one* reasonably efficient architecture for the machine. In Section 4 we will discuss how this architecture can be implemented in VLSI.

The basic structure of the architecture we will investigate is that studied in Section 2 (illustrated in Figure 2). The flow of instructions and data in the machine is exclusively from the input node (at the top of the figure) to the output node (at the bottom) -- we will not have deletions employ any "backwards flow". The machine is based on 16-bit instructions and 32-bit data words (which are interpreted either as integers in two's-complement or as 32-bit vectors). The top data paths in the machine (the son links from circles in Figure 2) are 16 bits wide; the bottom data paths (links to triangles) are 80 bits wide. The entire machine operates synchronously; an operation is (perhaps) performed at each node and data is transmitted from the node to its son on each *major cycle*. Having described the machine at this gross level, we will now examine the circles, squares and triangles individually.

The primary function of the circle on each major cycle is to broadcast what it just received to its sons (on the next cycle). In only three contexts must it perform a more sophisticated operation. As a new element is being inserted, it must decide which way to direct the insertion (to one of its nonempty sons) and then decrement the appropriate counter by one; it then ships a "no-op" to the other son. The no-op is effected by having one bit in the instruction turned off as the 16-bit instruction is passed to the "other" son. To accomplish a deletion we insert an instruction packet of three 16-bit instructions at the root node. The first instruction is the deletion and the next two 16-bit words contain the binary address of the node to be deleted. The circles can tell by looking at the leading bits of the address whether they should increment one of their counters as they see this instruction. The final capability the circles must have is that of passing data to the squares, without interpreting that as an instruction to them; we will return to this issue as we discuss the squares.

While the circles have the simplest architectures of the three units we will see, the squares have the most difficult. The abstract machine requires that the squares be able to store data and to perform enough calculations to answer queries and perform updates. This architecture will accomplish both these tasks by shipping combinations of instructions and data to the machine. We now have to make a fundamental design decision: should the individual squares be special-purpose devices (honed for a particular view of the tree machine's task), or should they be (in some limited sense) general-purpose computing devices? We will choose the latter course, and make each square a "baby" von Neumann computer; it is important, however, to emphasize that this is merely a design decision and not an inherent property of the abstract machine.

Each square will be a small von Neumann-like processor that receives its instructions and data from an external, 16-bit stream. An individual processor contains sixteen 32-bit words of memory, two 32-bit registers, and a vector of eight single-bit data flags. The processor also contains an eight-bit Set Identification number (SetID), and an Instruction Register. The first bit of the Flag vector (Flag[0], abbreviated F[0]) is used as the "Active" bit of the processor; a special "Enable" command turns on all processors, and a processor can conditionally turn itself off by storing a zero in Flag[0]. The basic layout of the machine is shown in Figure 4 (notice that because the machine is rotated 90°, the data flows from right to left rather than from top to bottom).

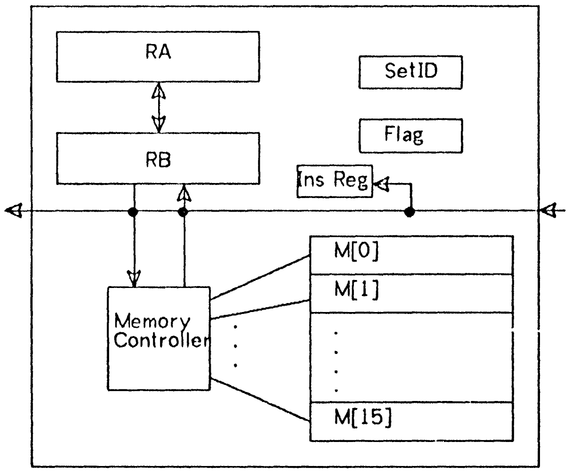


Figure 4. Components of the square.

The 16-bit instruction format for the square processor is shown in Figure 5. The first bit of an instruction processed by the squares is always zero; a one in that bit signifies an instruction that is ignored by the squares but passed on to the triangles. The two Fam bits specify one of the four families to which an instruction can belong (Arithmetic-Logical, Load-Store, Bit or Special), and the Code gives the opcode of the instruction. There is a one bit flag (Bit) in each instruction, and every other instruction has as its arguments either two four-bit addresses (A1 and A2), an 8-bit string (Name) or a five-bit integer (Num). The actual instructions are described by group in an ISP-like language in Table 1. All of the arithmetic-logical instructions are zero-address instructions, combining registers RA and RB and storing the result in RA. The load-store instructions specify one of 16 memory addresses as their operand; the data movement is then between that address and the register RB. All of the bit operations have two addresses; they combine the first and the second operands, storing the result in the first. The only exception to this pattern is the compare (comp) operation; it compares RA with RB and stores in the first bit (A1) whether or not the values are equal and tells which inequality in the second bit (this is just a straightforward encoding of three states into two bits).

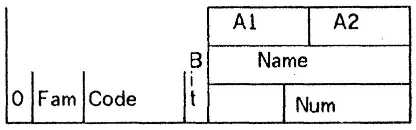


Figure 5. Instruction format.

The only instructions that are not entirely obvious are the special instructions. The *enable* instruction turns on all processors in the tree. The *ins* (insert) instruction turns on precisely one processor, turning off the rest (and decrementing the counters in the circles). The *del* (delete) instruction has no effect on the processors; it only increments the appropriate counters in the circles (the squares must ignore the two following instructions packets, though -- they are just the processor address). The *ship* instruction allows data to enter the RB register from the data/instruction stream. The Flag bit tells whether the next one or two 16-bit packets should be loaded into RB; the data can then be processed as desired. The *chksid* and *setsid* instructions are for manipulating the 8-bit SetID register; the former turns off the processor if SetID is not equal to Num, and the latter loads the SetID field from Num.

To illustrate the operation of the processors we will study two program segments for performing searches. The first segment is for member searching.

```
chksid  ThisSet  // Turn off bad squares
ship    Two      // The next two packets
data_L           //    contain the
data_R           //    comparand
tab              // Put comparand in RA
ldb     KeyAd    // Put key in RB
comp    1,2      // Answer is in F[1]
```

Arithmetic-Logical		
add	→	RA ← RA + RB
sub	→	RA ← RA - RB
neg	→	RA ← - RA
rand	→	RA ← RA ∧ RB
ror	→	RA ← RA ∨ RB
rxor	→	RA ← RA ⊕ RB
rnot	→	RA ← ~ RA
shift Num	→	RA ← RA left shifted by Num
tab	→	RA ← RB
tba	→	RB ← RA
swap	→	RA ↔ RB
Load-Store		
ldb Num	→	RB ← M[Num]
stb Num	→	M[Num] ← RB
Bit		
band A1,A2	→	F[A1] ← F[A1] ∧ F[A2]
bor A1,A2	→	F[A1] ← F[A1] ∨ F[A2]
bxor A1,A2	→	F[A1] ← F[A1] ⊕ F[A2]
bnot A1,A2	→	F[A1] ← ~ F[A1]
comp A1,A2	→	F[A1] ← RA=RB; F[A2] ← RA ≤ RB
Special		
enable	→	F[0] ← 1
ins	→	F[0] ← this processor selected
del	→	(defined in text)
ship Flag	→	(defined in text)
chksid Num	→	F[0] ← SetID = Num
setsid Num	→	SetID ← Num

Table 1. Instruction set for squares.

The search key enters the RB register from the data stream and is then transferred to the RA register. The program then loads the key field of the record into the RB register (KeyAd is an integer identifying which of the 16 memory words holds the key), and makes the comparison. Flag[1] is then one if and only if the record's field is equal to the data shipped in the stream. At this point, the answer can be combined in the triangle network.

The next program that we will examine arises in "nearest neighbor" searching; it computes the distance between the data and the key field of the record. Since we desire the absolute value of the difference of the key and the data, we must have a conditional step in our program.

```
chksid  ThisSet
ship    Two       // RA ← Data
data_L
data_R
tab
ldb     KeyAd     // RB ← Key
comp    2,0       // If Data≤Key,
swap              //     leave processor on
chksid  ThisSet   // Turn all processors on
sub               // RA ← |Key-Data|
```

The crucial step of this program is the comp instruction: if Data is greater than Key then a zero is stored in F[0], which turns the processor off; the swap then interchanges key and data. The next instruction (chksid) turns all appropriate processors back on, and the subtract correctly computes a positive value. The triangles can then be instructed to return the minimum of these values.

The two code segments that we have just seen illustrate many of the aspects of coding the tree machine. Many other examples have been coded, and all of them appear to be fairly efficient. More quantitatively, the ratio of tree machine instructions to "critical" operations in the task clusters very closely around 2.5. This statistic is evidence for the vindication of our design decision to make the squares general-purpose machines, rather than special devices tailored to the searching task domain. (Pursuing that alternative remains an interesting open problem.)

Before ending our discussion of the squares, it is interesting to compare the design of the processor with a more typical von Neumann processor. In some ways, we faced exactly the same problems: the choices of data representation, instruction formatting, operation set, and addressing were all taken from the von Neumann design space as discussed by Blaauw and Brooks [1979]. On the other hand, we avoided many of the issues faced by designers of typical machines; these include instruction sequencing, interrupt handling, and input/output control.

Before we discuss the architecture of the triangle, we must settle one more point about what we want it to do. In most applications that compute the minimum of a set (for instance), we want to know not only what the value of the minimum is but also what element has that value. We therefore have three objects associated with computing the minimum: the operation (minimum), the value, and the name (which is a 32-bit word associated with the value; its address or "key" in many applications). When combining two such objects, we take the value as the minimum of the two values, and the name from the name of the smaller value. The name is thus *inherited* from the minimum. We will also associate names with other binary operators: the name of *maximum* is inherited from the node with greater value; for *plus*, from a nonzero element; for *or*, from a nonzero bit vector (arbitrary if both are zero); and *and* from a zero bit vector.

Having defined the concepts of value, name and inheritance, it is straightforward to describe the architecture of the triangles. They will operate on 80-bit packets: 16 bits of instruction, and 32 bits each of value and name. Computing *min*, *max*, *plus*, *and*, and *or* are all simple. Union is a bit more detailed, but also conceptually straightforward. One aspect that we have not mentioned is the interface between the squares and the triangles; we must include instructions for transferring the contents of the RB register to the name or value field of the triangle immediately beneath it (these could be included in the load-store family). This allows us to give complete programs for answering queries. After computing the answers (as illustrated in the two segments shown above), we load them into the desired fields of the triangles, and combine them as desired.

It is important to emphasize that the architecture we have just seen is not the architecture that the ultimate user of the machine will see. Rather, there will be a hierarchy of functions available to him. At the highest level, he will be able to perform operations on sets (load a set, erase a set, for each element in the set, and so forth); at an intermediate level there are record-handling operations (defining queries or inserting, deleting and updating records); and at the lowest level there are the machine instructions themselves. At the lowest level the user can make very efficient code by knowing the details of the machine; at the higher levels he sacrifices efficiency for clean and easy code.

An important part of the implementation of this architecture is that there be a fairly sophisticated device controller for the tree machine (such as an off-the-shelf microprocessor). This controller will implement the hierarchy of functions mentioned above. This will also reduce the bus activity substantially by having the controller fetch items from main memory and issue instructions to the tree machine; having the CPU itself perform these tasks would lead to a substantial degradation in overall system performance.

4. Discussion of Implementation

In this section we will discuss an implementation of the architecture of Section 3 in VLSI technology. Just as with the architecture, we are not proposing this implementation as the best of all possible, but rather as one that is reasonably efficient. The fundamental description of the implementation is that it is bit-serial. There are two motivations for this: one, to exploit the shift-register technology of VLSI, and two, to use very few pins on packages.

The implementation of both the circles and the triangles described in the last section is straightforward. The squares are also easy to implement bit-serially. The 16-word memory is in fact a parallel shift register, 16 bits wide and 32 bits long. The two registers RA and RB are also shift registers. To load or store a word, RB and the memory shift register are shifted in parallel, and the memory controller of Figure 4 is just a multiplexor (decoding a 4-bit address to one of 16 lines). All of the arithmetic-logical operations are accomplished by putting a single-bit function box between the RA and RB registers, and then shifting the pair through it (all operations require at most one bit of memory). Notice that we have assumed that the squares have 32 minor cycles during each major cycle of the machine. The bit operations are straightforward to implement if the Flag array is just a small RAM. Estimates by experienced VLSI designers indicate that the chip area for the functionality in the square is about equal to the chip area required for the 512-bit memory. Using current technology, it is easy to imagine putting 16 squares on a single chip.

Now that we know how we will implement the individual processing elements (circles, squares and triangles), we must describe how to place them on a chip. The first simplification we will make is to consider them as standard binary trees rather than the "mirrored" binary tree of Figure 1; the unmirroring process is illustrated in Figure 6. We now face the problem of laying out a binary tree on a chip. This problem has been studied at length by Mead and Conway [1979], who suggest the space-economical layout illustrated in Figure 7. The amount of space used in that layout is proportional to the number of processors on the chip. Note that each edge in that layout is realized by two "wires" on the chip -- one for data going to the squares, and one for data coming from the squares.

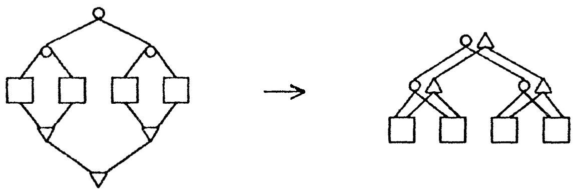


Figure 6. "Unmirroring" the tree machine.

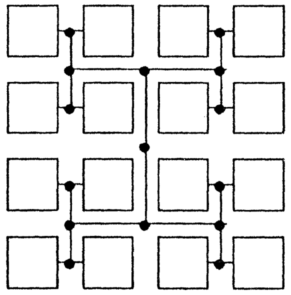


Figure 7. Tree layout on a chip.

Since only a small number of the processors in a tree machine will fit on a single chip, it is important that we discuss the packaging of the chips. The packaging strategy we propose is illustrated in Figure 8. There are two kinds of chips in that figure: the *leaf* chips and the *internal* chips. The leaf chips contain (say) 16 square nodes and 15 circle and triangle nodes. All the communication to a leaf chip is through two wires, so the chip needs only two communications pins (besides power and timing synchronization pins). Notice that this implies that with technological advances in VLSI, we will be able to place many more processors on a square chip; we are not bound by pin limitations. The internal chips would probably be constructed with seven circles and triangles on them; this implies that there is one input-output pair of wires at the top of the chip and eight pairs at the bottom. The total number of pins for this chip is therefore eighteen (plus miscellaneous pins). This chip is therefore pinbound even in today's fabrication technology; unless there are unexpected advances in packaging technology, the internal chips will probably continue to have seven or at most fifteen pairs of circles and triangles.

To get a better feeling for the size of the tree machine, we will briefly consider how one might be built today. Suppose that we put sixteen square nodes on each leaf chip, and seven circle-triangle pairs on each internal chip (both of these are easily accomplished in today's technology). We will now put 64 leaf chips and nine internal chips on a board; this gives us 1024 square nodes. We can then put sixteen of these boards in a small cabinet, giving a tree machine of more

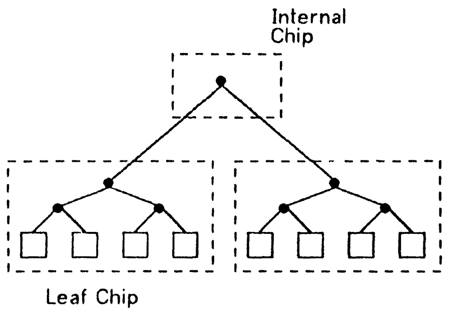


Figure 8. Two kinds of chips.

than sixteen thousand square nodes, each holding a 512-bit record. If we assume that technology continues to double the number of components on a chip every two years, this implies that we can expect a tree machine of one million records to fit in about a cubic foot of space by the end of the 1980's.

These rough (but fairly conservative) estimates indicate that the tree machine might be one reasonable way to exploit the processing power that VLSI will give us. Before we can assert this with confidence, however, we must show that the tree machine is a wiser way to invest resources than other structures for searching. For example, might it be better to put the same resources into a large RAM memory rather than a tree machine? The fact that the functionality of the tree machine is about equal to the chip resources required by the memory shows that if we were to go with a RAM instead of a tree machine, we could get at most twice the memory (and simultaneously lose a lot of functionality). It therefore appears that for searching applications, the tree machine is better suited than the RAM. The detailed comparison of this architecture to its competitors remains an open problem.

5. Conclusions

In this paper we have investigated the tree machine for searching problems on several levels. In Section 2 we studied it in an abstract setting and showed that it can rapidly solve many searching problems, as well as some other problems that do not immediately appear to be searching problems. In Section 3 we saw an architecture (that is, a user's view) of the machine, and in Section 4 we saw that that architecture can be efficiently implemented in VLSI technology. Having studied the machine at these various levels, we will now spend a few moments summarizing the contributions of this work.

This machine can be compared with many other architectures. It is similar to an associative memory in many aspects, but it can perform many more operations than even the most powerful associative memories considered to date (see, for example, Lamb and Vanderslice [1979]). One might consider the square processors as forming a Single-Instruction, Multiple-Data stream (SIMD) computer, but each square is considerably simpler than most SIMD machines proposed to date. The tree machine is also superficially similar to the CASSM computer of Su *et al* [1979], but there are fundamental differences in the two machines at both the architectural and implementation levels. Two other machines to which it might be compared are the tree-structured machines of Mago [1979] and Sequin, Despain and Patterson [1978]. Both of these machines, however, are put forward as general-purpose computing devices, while our machine is much more specialized to the particular problem of searching.

An interesting aspect of the tree machine is what we might call its "computational structure", which is illustrated in Figure 9. That diagram has three interpretations. First, it illustrates the tree machine itself: very small input and output channels, with massive computation going on in between. Second, it describes the searching problem: a small question is asked about a large set, giving a small answer. And finally, the figure illustrates the constraints of working with pinbound VLSI: the number of pins on a chip is very small compared to the number of functional components. The fact that the abstract structure of both the searching problem and the tree machine's solution to it closely model the medium of VLSI indicates that this approach might be very successful.

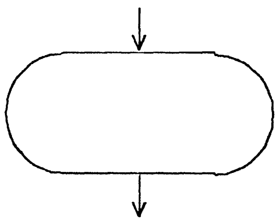


Figure 9. A computational structure.

To summarize the tree machine, the authors feel that this work has three contributions. The first is the abstract tree machine: it gives a number of nice "theoretical" solutions to a large set of problems. The second contribution is the architecture and implementation we have proposed; they indicate that this machine might be a reasonable device to build as further advances in VLSI technology occur. Finally, we feel that the "computational structure" we just investigated provides an example of the kind of argument that will justify special-purpose architectures proposed for implementation in VLSI.

Acknowledgements

The authors would like to acknowledge the careful comments of Dorothea Haken, Dave McKeown, Jim Saxe and Dick Sites and the helpful criticisms received when presenting these ideas in a seminar at Xerox PARC.

References

Backus, J. [1978]. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM 21*, 8 (August 1978), pp. 613-641.

Bentley, J. L. [1979]. "A parallel algorithm for constructing minimum spanning trees," submitted for publication.

Bentley, J. L. and J. B. Saxe [1979]. "Decomposable searching problems," in preparation. (Preliminary version by J. L. Bentley, *Information Processing Letters 8*, 5 (June 1979), pp. 244-251.)

Blaauw, G. A. and F. P. Brooks, Jr. [1979]. *Computer Architecture*, unpublished draft.

Browning, S. [1979]. "Computation on a tree of processors," Caltech Internal Memorandum.

Kung, H. T. and C. Leiserson [1979]. "Systolic arrays (for VLSI)," *Carnegie-Mellon University Computer Science Research Review, 1977-78*, pp. 37-57. To appear in Mead and Conway [1979].

Lamb, S. M. and R. Vanderslice [1978]. "Recognition memory: low cost content-addressable parallel processor for speech data manipulation," presented at the Acoustical Society of America and Acoustical Society of Japan Joint Meeting, Session BB, Honolulu (29 November 1978).

Mago, G. [1979]. "A network of microprocessors to execute reduction languages," to appear in *International Journal of Computer and Information Sciences*.

Mead, C. A. and L. A. Conway [1979]. *Introduction to VLSI Systems*, to appear.

Sequin, C. H., A. M. Despain, and D. A. Patterson [1978]. "Communication in X-tree, a modular multiprocessor system," *ACM 78 Proceedings*.

Sites, R. L. [1979]. "How to use 1000 registers," Caltech Conference on VLSI (January 1979).

Su, S. Y. W., L. H. Nguyen, A. Emam, and G. J. Lipovski [1979]. "The architectural features and implementation techniques of the multicell cassm," *IEEE Trans. Comp. C-28*, 6, (June 1979), pp. 430-445.

Swan, R. J., S. H. Fuller and D. P. Siewiorek [1977]. "Cm*: A Modular Multiprocessor," *AFIPS Conf. Proc. 46*, pp. 637-644.

Wilner, W. [1978]. "Recursive machines," Xerox PARC SSL Internal Memorandum, (January 21, 1978).

FAST EVALUATION OF ARBITRARY DECISION TREES*

Robert H. Kuhn
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Summary

Binary decision trees can be found in several contexts. A logic example is a demultiplexer; a software example is a binary search tree. We show that the evaluation of any decision tree of size n can be speeded up to the Boolean complexity lower bound $\lceil \log n \rceil$.

A decision tree consists of a set of binary decision variables input, a set of result variables output, and a binary tree. If the decision tree is of size n there are n result variables, n-1 decision variables, and the binary tree has n leaf nodes. Each result variable corresponds to a leaf node. Each decision variable corresponds to an internal node so that the left arc incident from the internal node is labeled by the decision variable, the right arc by its complement. Thus, a decision tree is similar to a nested IF...THEN ...ELSE... program.

A decision tree will be denoted by an upper case italic letter: *A*, *B*, ... A Boolean variable is denoted by a lower case italic letter: *a*, *b*, ...

We wish to speed up the serial evaluation of decision trees. A serial evaluation of a decision tree is a traversal of the tree from its root to one of its leaves. When a node is visited, its decision variable is evaluated. If it is true, the left successor node is visited; otherwise, the right successor is visited. Only the result variable of the leaf visited is set true; the others are set false.

For example, Fig. 1 shows a decision tree to search for "dave", "david", or "dan". The decision variables are denoted c_i, meaning the ith character of the pattern is "c". The result variables are denoted r_1 through r_8. Leaf 1 is reached if all of the decision variables d_1, a_2, v_3, and e_4 are true. Then, result variable r_1 is true; results r_2 through r_8 are false. Similarly, r_2 and r_5 are true for $d_1 \cdot a_2 \cdot v_3 \cdot \bar{e}_4 \cdot i_4 \cdot d_5$ and $d_1 \cdot a_2 \cdot \bar{v}_3 \cdot n_3$, respectively. If none of the patterns is matched, one of the other result variables will be true.

For certain decision trees, serial evaluation may take n-1 steps. If the tree has the special form in which there is a chain of n-1 internal nodes, the prefix computation [1] can be used to evaluate the tree in $\lceil \log n \rceil$ time. For arbitrary decision trees, we propose a tree-height reduction transformation which also yields a $\lceil \log n \rceil$ time bound. This transformation is similar to distribution applied to arithmetic expressions.

Definition: Consider a binary tree *T* containing a subtree *A*. The arc path from the root of *T* to the root of *A* is called the factor of *A*. The factor of a decision tree is the product of the Boolean variables or complements labeling the arc path. Let *p* denote the factor of *A* in *T*. ∎

Definition: If a decision tree *A* is conditioned by a Boolean variable *p*, the value of all of the result variables of *A* are false if *p* is false; otherwise, the values of the result variables are determined by evaluating *A*. ∎

Definition: Let the decision tree *T* have a subtree *A*. By distributing *A* from *T* with remainder *R*, we mean: (1) replace the subtree *A* in *T* by a leaf node; call this tree *R*; (2) condition *A* by the factor of *A*, *p*. ∎

After distribution, each of *p*, *A*, and *R* are smaller than *T* and they can all be evaluated in parallel, resulting in a considerable speedup.

Fig. 2 shows an example. Let *A* be the subtree of Fig. 1 rooted at e_4. The factor of *A* is $p = d_1 \cdot a_2 \cdot v_3$; that is, a result of *A* can be true only if $d_1 \cdot a_2 \cdot v_3$ is true. To condition *A* by *p*, we compute $d_1 \cdot a_2 \cdot v_3$ and AND the result with each result variable of *A* as shown. It is easy to verify that result variable r_1, r_2, or r_5 is true in Fig. 2, only when r_1, r_2, r_5 is true in Fig. 1.

It may take 6 steps to evaluate Fig. 1, but it takes at most 4 steps to evaluate Fig. 2. $d_1 \cdot a_2 \cdot v_3$ may be evaluated in 2 steps in parallel with the evaluation of *A*, which takes at most 3 steps.

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS77-27910.

Conditioning A takes only 1 more step. R can be evaluated in at most 4 steps in parallel with the evaluation and conditioning of A. The evaluation time of Fig. 1 can be decreased even further to $\lceil \log 8 \rceil = 3$ steps by recursively distributing A and R.

The proof that a logarithmic speedup results by recursive application of distribution is based on two lemmas similar to those in [2]. The first bounds the size of the tree that can be distributed out (and also the size of the remainder tree). The second bounds the size of the factor.

Lemma 1: Any binary tree, T, of size n (n leaves) where $n \geq 4$ contains a subtree, A, such that $n/4 \leq |A| \leq n/2$. ■

Lemma 2: Any binary tree, T, has a subtree, A, satisfying Lemma 1 such that the factor, p, from the root of T to the root of A contains at most $\lceil n/2 \rceil$ arcs. ■

Lemmas 1 and 2 insure that on each application of distribution there is a linear reduction in the size of: the factor, the tree distributed out, and the remainder tree. Thus, we can prove the following:

Theorem: Any decision tree of size n, $n \geq 4$, can be evaluated in $\lceil \log n \rceil$ gate delays with fewer than $5/4\, n \log n$ AND gates. ■

The gate bound of this theorem is more difficult to establish. It follows from the solution of the gate recurrence equation

$$g(T) = g(p) + g(A) + g(R).$$

The function $g(T)$ may be bounded by the real valued function $\hat{g}(n)$ which satisfies

$$\hat{g}(n) = \max_{n/4 \leq m \leq n/2} (n + \hat{g}(m) + \hat{g}(n-m))$$

where n is the size of T and m represents the size of A. This equation is similar to the information entropy function whose solution is known.

Although the time bound is low, there are decision trees whose time bound is lower ($O(\log \log n)$). That the gate bound is nearly linear for large n may be surprising.

The concept demonstrated above has been studied with regard to implementation and application [3]. Implementation issues investigated include limited fan-out (time: $2\log n$, gates: $2n \log n$) and a programmable evaluation network (time: $3\log n$, gates: $O(n^{\log 3})$). Applications that have been considered are: decision tables, numerical programs containing more conditionals than computations, and searching. We have also investigated the possibility of applying other tree-height reduction transformations to decision trees.

References

[1] R. Ladner, and M. Fischer, "Parallel Prefix Computation," Proceedings of the 1977 International Conference on Parallel Processing (Aug., 1977), pp. 218-223.

[2] R. Brent, D. Kuck, and K. Maruyama, "The Parallel Evaluation of Arithmetic Expressions without Division," IEEE Transactions on Computers, (May, 1973), pp. 532-534.

[3] R. Kuhn, Ph.D. thesis, in preparation, Dept. of Computer Science, University of Illinois at Urbana-Champaign (1979).

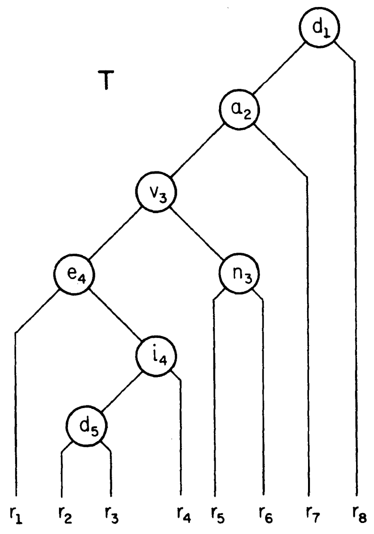


Fig. 1. A decision tree

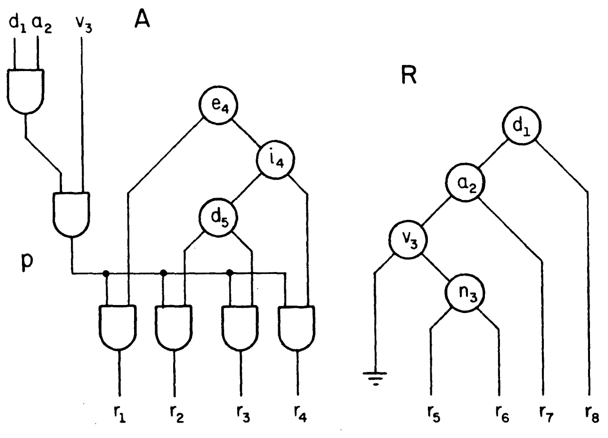


Fig. 2. A distributed decision tree

FAULT-DIAGNOSIS FOR A CLASS OF MULTISTAGE INTERCONNECTION NETWORKS

Chuan-lin Wu and Tse-yun Feng
Department of Electrical and Computer Engineering
Wayne State University
Detroit, Michigan 48202

Abstract -- To study the fault-diagnosis method for the class of multistage interconnection networks a general fault model is first constructed. Specific steps for diagnosing single faults and detecting multiple faults in the interconnection networks such as the indirect binary n-cube network and the flip network are then developed. The following results are derived in this study: (1) independent of the network size, only four tests are required for detecting a single fault; (2) the number of tests are required for locating a single fault and determining the fault type ranges from 4 to max $(12, 6+2\lceil\log_2(\log_2 N)\rceil)$ except for four types of single faults in the switching elements which cannot be pinpointed at the switching element level; (3) only four tests are required for locating a single fault if the switching element is designed in such a way that any physical defect in the switching element causes both outputs of the related switching element to be faulty; and (4) multiple faults can be detected by $2(1+\log_2 N)$ tests.

I. Introduction

The problem of fault-diagnosis in interconnection networks has received little attention in the literature. In this paper we investigate the fault-detection and the fault-location problems of a class of multistage interconnection networks [1]. The class of multistage interconnection networks includes the modified data manipulator [2], the flip network [3], the indirect binary n-cube network [4], the omega network [5], and the baseline network. Since these networks are topologically equivalent, the fault-diagnosis scheme developed for one network can also be used for other networks after applying mapping rules described in [1]. In this paper, we use the baseline network in developing the scheme.

The fault-diagnosis problem is approached by generating suitable fault-detection and fault-location test sets for every fault in the assumed fault model. The test sets are then trimmed to minimum or nearly minimal sets. In Section II we propose a fault model of a switching element and derive a test set for every fault in the fault model. The objective of Sections III and IV is to develop a specific fault-diagnosis scheme for the network constructed of switching elements having direct- and crossed-connection capabilities as shown in Fig. 1. The fault-diagnosis of single faults and the fault characteristics are discussed in Section III. The multiple fault detection problem is then considered in Section IV. The proof details of theorems are omitted because of the page limitation. Interested readers are referred to [6].

II. Fault Model and Test Set of a Switching Element

A. Fault Model

Generally, a switching element with two input lines and two output lines can be considered as a 2×2 crosspoint switching matrix which may have as many as 16 states. Table 1 shows the set S of the 16 states and the related symbolic representation. In our proposed multistage interconnection network, a switching element is designed so that only some of the 16 states are used. We denote these states as valid states. In the flip network and the indirect binary n-cube network, the valid states include S_5 and S_{10}. The valid states in the omega network include S_3, S_5, S_{10}, and S_{12}. The number of valid states which a switching element can assume in order to achieve the network function depends on the physical design of the interconnection network. A fault exists when a switching element is in any one of the 16 states different from a given valid state. A fault in an interconnection network can be located either at a link or in a switching element. All discussion in this paper is confined to solid logical faults. The fault located at a link can be considered to be one of line stuck types, i.e., stuck-at-zero (s-a-0) or stuck-at-one (s-a-1). We use a functional approach to consider fault types in a switching element. For a switching element with n valid state, there are $(16)^n$ possible state combinations in which the faulty switching element can behave. We use the ordered set $\{(s_1, s_2, \ldots, s_n) | s_i \in S, 1 \le i \le n\}$ to describe the state combinations and name each of the state combinations a functional state. As an example, Fig. 1 shows a switching element with two valid states, S_{10} and S_5. Assume the first valid state is S_{10} and the second S_5. The functional states of switching element can be expressed by a functional state set which is an inner product set of S, $S \times S$. There are 256 functional states. The state combination (S_{10}, S_5) is the normal functional state and the other 255 state combinations are faulty functional states of the switching element shown in Fig. 1.

B. Test Set

A test set should be developed for detecting every fault in the fault model described above. By test set, we mean the collection of tests which are needed for detecting all possible faults in the fault model. Faults to be detected and tests for detecting them are listed in Table 2 for a switching element in valid state S_{10}. In Table 2 the detection of the link stuck fault is described in Part I. The superscripts of the link labels indicate whether the fault causes the link stuck at 0 or 1. For example, in the first row, we can see that if we apply an input $(x_1,x_2)=(1,0)$ to the switching element in valid state S_{10}, the normal output will be $(\hat{x}_1, \hat{x}_2)=(1,0)$ and the fault, x_1° or $\hat{x}_1^\circ$, of x_1 or $\hat{x}_1$ sticking at 0 will cause output to be $(\hat{x}_1,\hat{x}_2)= (0,0)$. We then say that the input $(x_1,x_2) = (1,0)$ can detect the fault, x_1° or $\hat{x}_1^\circ$, of the switching element in valid state S_{10}. The detection of switching element faults is described in Part II. For a switching element stuck in S_5, S_{10}-S_5, (see row S_{10}-S_5 of Part II of Table 2), if we apply input $(x_1,x_2) = (0,1)$, the faulty output will be $(\hat{x}_1,\hat{x}_2) = (1,0)$ which is different from the normal output $(\hat{x}_1,\hat{x}_2) = (0,1)$. In Table 2, "-" means logically undefined output and "ϕ" means logically erroneous output where 0 and 1 are the simultaneous inputs. The ouput values of "-" and "ϕ" depend on the circuit implementation of the switching element. However, an arbitrary assignment of 0 or 1 to "-" or "ϕ" would not affect the differentiation between the normal output and the faulty output. Hence whether we can easily design an equipment to detect "-" and "ϕ" would not disturb our development of the test set for various faults.

From Table 2 it can be seen that only two tests $(x_1,x_2) = (0,1)$ and $(x_1,x_2) = (1,0)$, are needed to detect all faults. The test vectors on x_1 and x_2 are 01 and 10, respectively. For an easy reference we define $t = 01$ and $\bar{t} = 10$. The same conclusion can be drawn for a switching element in valid state S_5. The faults, the test inputs and the test outputs of S_5 are shown in Table 3. Similarly, the above techniques can also be extended to detect faulty elements in other networks such as the omega network with additional valid states S_3 and S_{12}.

III. Diagnosis of Single Faults

A. Fault Detection

An algorithm for deriving efficient test sets for the network will be presented. The basic idea is to establish connection paths and to label each link in a path with t (or $\bar{t}$) such that each switching element in the network has its two input lines labelled with the two test vectors (t and $\bar{t}$), respectively. The connection paths are established by putting switching elements into a valid state to be tested. Since only one valid state of the switching element can be tested in a test phase, two test phases are needed for the switching element shown in Fig. 1. In phase 1, we test the valid state shown in Fig. 1(a) for all switching elements in the network and in phase 2, we test the valid state shown in Fig. 1(b). It would be an efficient way that each one of the switching elements has its two input lines labelled with two different test vectors in each test phase. These test vectors appearing on the input lines in each phase constitute a test set which can efficiently test all switching elements in the network. An algorithm for generating such an efficient test set for a network of size N is described as follows:

Step 1: Label the top terminal link in the left side of the network with test vector t = 01.

Step 2: Assume the labelled terminal links are named 0,1,..., and m-1 from top to bottom and the next m unlabelled terminal links are named, from top to bottom, m, m+1,..., and 2m-1, where $1 \leq m < N$ and m is in 2's power. Label terminal link m+i with $\overline{L(i)}$ for $0 \leq i \leq m-1$, where L(i) is the test vector assigned to terminal link i and $\overline{L(i)}$ is the complement.

Step 3: For the unlabelled terminal links in the left side, repeat Step 2 until all N terminals are labelled.

The test set generated by the above algorithm is good for both test phases and an example is given in Fig. 2. The labels on the input lines of the leftmost stage correspond to the required test vectors while the other labels indicate the fault-free response of the network to these test vectors. The fault-free response of the network shown in Fig. 2 assures that each one of the switching elements has its two input lines labelled with two different test vectors. These two different test vectors are exactly the test vectors needed to test each valid state of a switching element (see Section II). It can be seen that to detect single faults in a network four tests (two for each test phase) are necessary and sufficient and the test length is independent of the network size. This fact is restated in Theorem 1.

Theorem 1:

Four tests are necessary and sufficient for detecting single faults in a baseline network constructed of switching elements with two valid states shown in Fig. 2.

B. Fault Location

The problem of locating a single fault can be partitioned into two subproblems: one is to locate the stuck fault at a link and the other is to locate the fault in a switching element.

1. Link stuck fault: A link stuck fault can be located by intersecting the link sets of two faulty paths. A stuck fault at a link can cause one and only one faulty output at an observable terminal in each test phase. Each faulty output should be equal to either 00 or 11 which is different from the normal output. Each link in the network can uniquely be identified by two paths, one from phase 1 and another from phase 2. The method to compute the link set in a path is previously defined [1]. The test set derived for detecting single faults can be used to locate a single stuck fault at a link. However, the link stuck fault may not be distinguishable from some switching element faults which will be discussed later. In spite of this indistinguishability, there exists a one-to-one relationship between a link stuck fault and a faulty output pattern.

Theorem 2:

There is a one-to-one correspondence between the link stuck fault and the faulty output pattern. The faulty output pattern is a necessary condition of the link stuck fault.

Example: Comparing the test outputs to the fault free output of a network of size 16, we observe that the path of link set {7,6,2,0,1} in the phase 2 test lead to the faulty output 00 of output link 6 at the phase 1 test and output link 1 at the phase 2 test. Intersecting the two link sets, we can locate the fault at link 6 of level 1 which is stuck at zero.

2. Switching element fault: A switching element fault can be the result of any one of the 16 states shown in Table 1. Single switching element faults in a network can result in several faulty output patterns. In terms of the response patterns of the detection phases the faults can be classified into four cases as follows:

(1) One-response fault - There is only one faulty output. This faulty output can be a terminal output at either phase 1 test or phase 2 test;
(2) Separated two-response fault - There are two faulty outputs. One of them is a terminal output at phase 1 test and the other is a terminal output at phase 2 test;
(3) Nonseparated two-response fault - There are two faulty outputs. But, both of the faulty outputs are terminal outputs at either phase 1 test or phase 2 test;
(4) Multiple-response fault - There are more than two faulty outputs.

Each case will be considered in the following paragraphs.

Case 1: The set of switching elements involved in a faulty path is not sufficient to locate a single fault at the switching element level since we have to pinpoint exactly one switching element in this set. Additional tests will be necessary to determine the fault location and the fault type. According to Table 2 we find the that $P = \{S_2,S_3,S_8,S_{11},S_{12},S_{14}\}$ is the set of the faulty state which has one faulty output in valid states S_{10}. Thus, if the functional state of a switching element is one of the following: (S_2,S_5), (S_3,S_5), (S_8,S_5), (S_{11},S_5), (S_{12},S_5), and (S_{14},S_5), we have only one faulty output at phase 1 test and no faulty output at phase 2 test. Similarly, according to Table 3 we can find that $Q = \{S_1,S_3,S_4,S_7,S_{12},S_{13}\}$ is the set of the faulty state which has one faulty output in valid state S_5, and any one of the following functional states: (S_{10},S_1), (S_{10},S_3), (S_{10},S_4), (S_{10},S_7), (S_{10},S_{12}), and (S_{10},S_{13}) results in one faulty output at the phase 2 test and no faulty output at the phase 1 test. Therefore, there are 12 fault types of the one-response fault.

Theorem 3:

The fault location and the fault type of the one-response fault can be determined by at most $6+2\lceil\log_2(\log_2 N)\rceil$ or $6+2\lfloor\log_2(\log_2 N)\rfloor$ tests.

The number of tests indicated in Theorem 3 is actually an upper bound of the number of tests for determining the fault location and the fault type at the switching element level. In some cases we may only need to locate the fault at the module level. Then the number of tests needed is less than this upper bound depending on the size of the modules.

Case 2: In this case the faulty switching element has only one faulty output in each valid state. We have described in Case 1 that $P = \{S_2,S_3,S_8,S_{11},S_{12},S_{14}\}$ and $Q = \{S_1,S_3,S_4,S_7,S_{12},S_{13}\}$ are the sets of faulty states which have only one faulty output in valid states S_{10} and S_5, respectively. The possible faulty output combinations of these two sets are depicted in Table 4 in which there are six subcases: A, B, C, D, E, and F. There are 36 functional states in this case, which form the inner product set of P and Q, $P \times Q$. These 36 functional states are classified into six subcases according to the faulty output patterns as shown in Table 4. The classification is shown in Table 5 in which the horizontal caption is for the faulty states of valid state S_{10} and the vertical for the faulty states of valid state S_5. An example of reading Table 5 is described below. Suppose (S_3,S_{13}) is a faulty functional state of (S_{10},S_5). The B at the intersection of column S_3 and row S_{13} in Table 5 implies that the switching element in functional state (S_3,S_{11}) will result in a faulty output of a binary vector in a test phase and a $\phi\phi$ faulty output in another test phase according to Table 4. In each subcase it can be seen that in some examples there are two common switching elements in the two faulty paths. One of these two common switching elements is faulty. Additional test sets should be derived in order to locate the fault within these two questionable switching elements. In some examples there is only one common switching element in the two faulty paths. In these examples the common swithing element is either in the rightmost stage or in the leftmost stage.

Theorem 4:

The fault location and the fault type of the Subcase A fault can be determined by at most 8 tests, independent of the network size. The 8 tests includes 4 test for the detection phase and 4 tests for locating the fault.

Theorem 5:

The fault location and the fault type of the Subcase B fault or the Subcase C fault can be determined by at most 10 tests, independent of network size.

Number of tests needed for locating the faults of Subcases B and C is equal to 4 or 8 and two additional tests are needed for determining the fault type.

Theorem 6:

The fault location and the fault type of the Subcase D fault or the Subcase E fault can be determined by at most 12 tests, independent of network size.

Actually, the total number of tests needed to determine the fault location and the fault type in Subcases D and E is equal to 8 or 12.

Remark: The fault of Subcase F cannot be pinpointed at the single switching element level and it is indistinguishable from a link stuck fault.

Because of the characteristics of the "--" fault in Subcase F, it is impossible to further pinpoint the fault among each questionable switching element and link by applying tests on the input side and observing output on the output side. Hence there exists an ambiguity between the link stuck fault and the Subcase F fault.

Case 3 and Case 4: We can compute the switching element sets of the faulty paths and the intersections of these sets should lead to a unique faulty switching element. In these two cases, only four tests which are developed for the detection phases are required for locating the fault. In Case 3, the faulty switching element has two faulty outputs at one of the test phases. There are 18 fault types in Case 3, which are the inner products of $\{S_0, S_1, S_4, S_5, S_6, S_7, S_9. S_{13}, S_{15}\} \times \{S_5\}$ and $\{S_{10}\} \times \{S_0, S_2, S_6, S_8, S_9, S_{10}, S_{11}, S_{14}, S_{15}\}$. Since the faulty switching elements can be uniquely identified by the switching element set of the two faulty paths in the same detection phase, the number of tests needed is equal to four. Two additional tests may be needed to differentiate ϕϕ and --. In Case 4, there are 189 fault types and the fault type can be any one in the union of the inner product sets of $\{S_0, S_1, S_4, S_5, S_6, S_7, S_{13}, S_{15}\} \times \{S-S_5\}$ and $\{S-S_{10}\} \times \{S_2, S_6, S_8, S_9, S_{10}, S_{11}, S_{14}, S_{15}\}$ where $\{S-S_5\}$ is the set of the states shown in Table 1 excluding S_5 and $\{S-S_{10}\}$ the set excluding S_{10}. Again the faulty switching element can be uniquely identified by the faulty switching element sets of the faulty paths in the detection phases. Furthermore, four, two or zero additional tests may be needed to differentiate ϕϕ and --.

The above discussion can be summarized in the following theorem.

Theorem 7:

The fault location and the fault type of Case 3 or Case 4 can be determined by at most 8 tests, independent of network size.

The single fault diagnosis scheme is good under the assumption that the diagnosis procedure can be repeated in a reasonably short period during which at most a single fault could possibly occur. However, it is well known that many physical faults of a single logical circuit component cannot be represented as a single fault.

IV. Detection of Multiple Faults

Now we consider the detection problem for multiple faults. By a multiple fault, we mean the simultaneous occurence of any possible combination of single faults.

In the single-fault detection problem, we derive tests for every stuck-type fault at the link and functional state fault in the switching element. For the multiple fault case, the test set derived for detecting single faults may fail to indicate the existence of the fault because some faults may be masked by some other faults. The faulty state which can mask a fault such that the fault becomes unobservable is called the masking faulty state. In valid state S_{10}, the masking faulty states are S_5, S_{12}, and S_3, and in valid state S_5, the masking faulty states are S_{10}, S_{12} and S_3. The masking problem can be solved by using distinctive test vectors. Extending the solution to the whole network, we should use N distinctive test vectors for N terminals. The all-zero and all-one vectors should be excluded because these two vectors fail to test stuck-type faults at links. Hence, $1+\log_2 N$ binary bits are needed to form the test vectors for the multiple fault. Two test phases, similar to the two for detecting single faults, are also needed for detecting multiple faults. Concluding the above discussion we have the following theorem.

Theorem 8:

The number of tests for detecting multiple faults is equal to $2(1+\log_2 N)$.

V. Conclusion

In this paper, we have presented a fault model for the network in the class of multistage interconnection networks. Fault diagnosis procedures for the network constructed of switching elements with two valid states have been considered. A diagnosis method for single faults and a detection method for multiple faults are developed. In the diagnosis procedures the control lines of the switching elements in the same stage can be grouped together and activated by the same control signal. The control line grouping of each stage is exactly the control scheme used in the flip network of STARAN [3]. Hence, the diagnosis procedures developed in this paper are good both for the indirect binary n-cube network and the flip network. Extension to the network constructed of switching elements with four valid states is feasible since the test sets for faults in switching elements with four valid states are the same as those we developed for switching elements with two valid states. The problem left is to design diagnosis procedures with minimal or nearly minimal number of tests.

The number of tests which is required under various conditions in the diagnosis procedures developed in this paper is summarized as follows. The number of tests for detecting single faults is equal to four and is independent of the network size. The number of tests for detecting multiple faults is equal to $2(1+\log_2 N)$, where N is the number of terminal links in one side of the network. The number of tests needed for determining the fault location and the fault type of a single fault depends on the fault type and/or the size of the network. The characteristics of single switching element faults are summarized in Table 6. The minimum number of tests needed for determining the fault location and the fault type is equal to four and the maximum max $(12, 6+2\lceil\log(\log N)\rceil)$. For a network size N = 1024 the maximum is equal to 14. There exist four switching element faults (Subcase F) which cannot be pinpointed at the single switching element level and those four are not distinguishable from the link stuck fault. This study provides specific information of fault characteristics for designing an easily diagnosable network.

References

[1] C. Wu and T. Feng, "Routing techniques for a class of multistage interconnection networks," Proc. of the 1978 International Conference on Parallel Processing, pp. 197-205.

[2] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comput., Vol. C-23, No. 3, March 1974, pp. 309-318.

[3] K. E. Batcher, "The flip network in STARAN," Proc. of the 1976 International Conference on Parallel Processing, pp. 65-71.

[4] M. C. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comput., Vol. C-26, No. 5, May 1977, pp. 548-573.

[5] D. K. Lawrie, "Access and alignment of data in an array processor," IEEE Trans. Comput., Vol. C-24, December 1975, pp. 1145-1155.

[6] T. Feng and C. Wu, Interconnection Networks in Multiple-Processor Systems, RADC Technical Report, 1978.

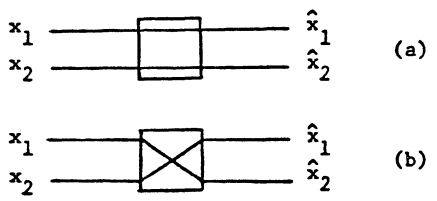


Fig. 1 A switching element. (a) Direct connection. (b) Crossed connection.

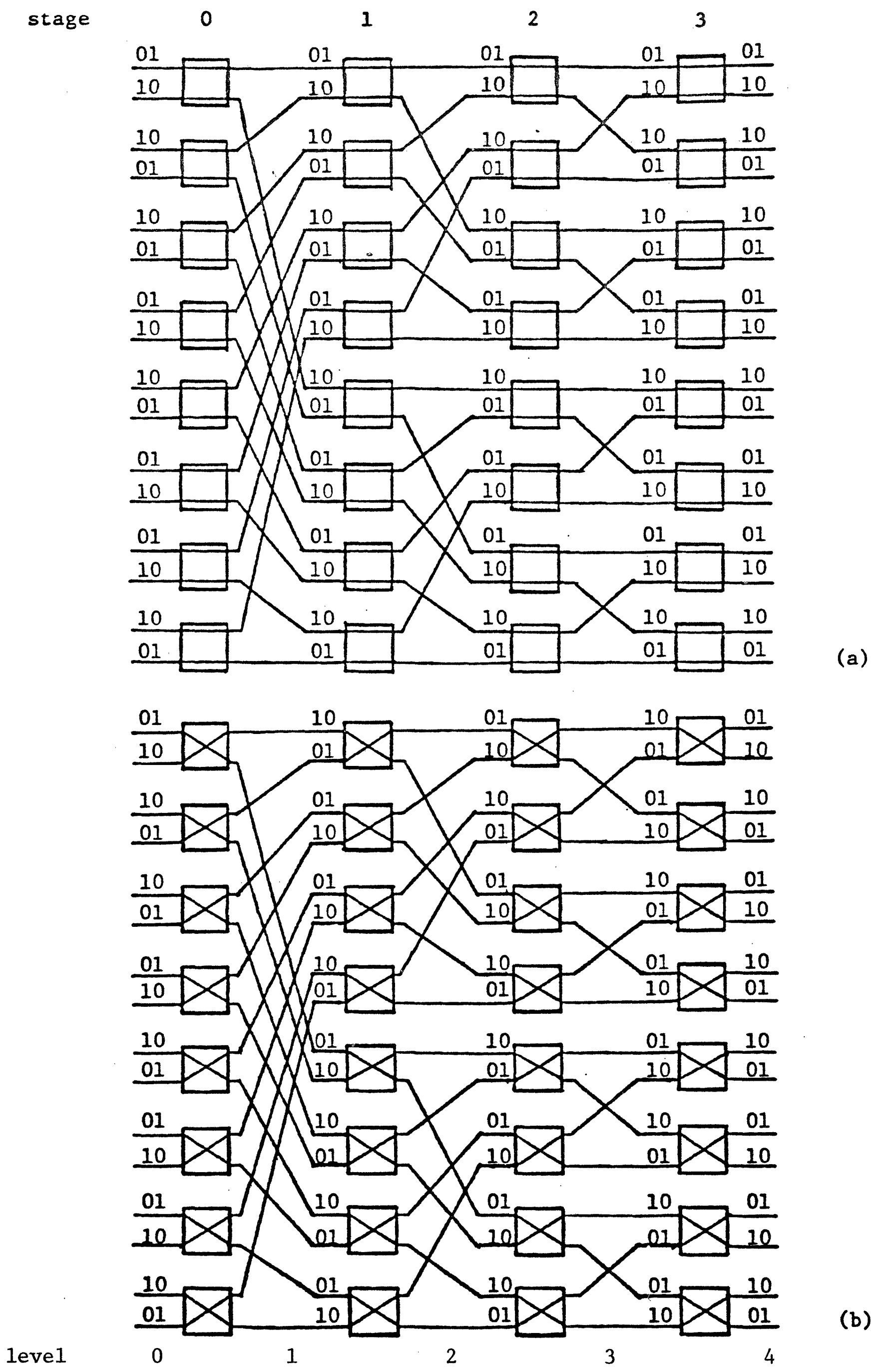


Fig. 2 Fault-free response of a network to the test set.
(a) Phase 1 test. (b) Phase 2 test.

Table 1 Set of the 16 States and the related Symbolic Representation of a 2×2 Switching Element

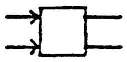
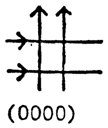
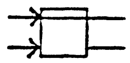
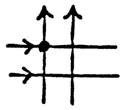
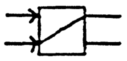
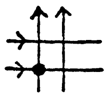
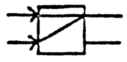
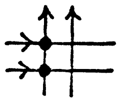
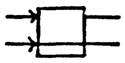
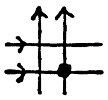
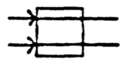
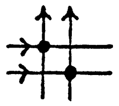
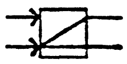
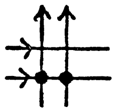
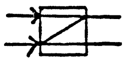
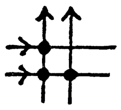
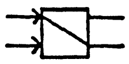
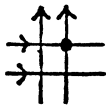
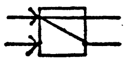
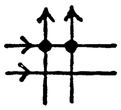
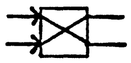
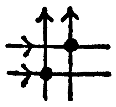
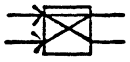
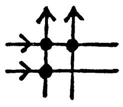
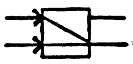
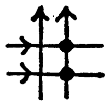
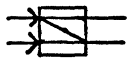
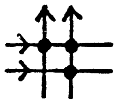
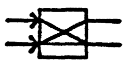
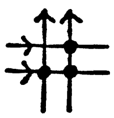
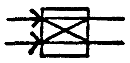
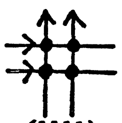
State Name	Switching Box Symbol	Crosspoint Switching Matrix Symbol	State Name	Switching Box Symbol	Crosspoint Switching Matrix Symbol
S_0		 (0000)	S_8		 (1000)
S_1		 (0001)	S_9		 (1001)
S_2		 (0010)	S_{10}		 (1010)
S_3		 (0011)	S_{11}		 (1011)
S_4		 (0100)	S_{12}		 (1100)
S_5		 (0101)	S_{13}		 (1101)
S_6		 (0110)	S_{14}		 (1110)
S_7		 (0111)	S_{15}		 (1111)

Table 2 Faults, Test Inputs and Outputs in Valid State S_{10}.

Fault		Test		Output			
				Normal		Faulty	
		x_1	x_2	$\hat{x}_1$	$\hat{x}_2$	$\hat{x}_1$	$\hat{x}_2$
Part I: Link Stuck Fault	x_1°, $\hat{x}_1^{\circ}$	1	0	1	0	0	0
		1	1	1	1	0	1
	x_1^1, $\hat{x}_1^1$	0	0	0	0	1	0
		0	1	0	1	1	1
	x_2°, $\hat{x}_2^{\circ}$	0	1	0	1	0	0
		1	1	1	1	1	0
	x_2^1, $\hat{x}_2^1$	0	0	0	0	0	1
		1	0	1	0	1	1
Part II: Switching Element Fault	S_{10}-S_0	0	1	0	1	-	-
		1	0	1	0	-	-
		0	0	0	0	-	-
		1	1	1	1	-	-
	S_{10}-S_1	0	1	0	1	1	-
		1	0	1	0	0	-
		0	0	0	0	0	-
		1	1	1	1	1	-
	S_{10}-S_2	0	1	0	1	-	1
		1	0	1	0	-	0
		0	0	0	0	-	0
		1	1	1	1	-	1
	S_{10}-S_3	0	1	0	1	1	1
		1	0	1	0	0	0
	S_{10}-S_4	0	1	0	1	-	0
		1	0	1	0	-	1
		0	0	0	0	-	0
		1	1	1	1	-	1
	S_{10}-S_5	0	1	0	1	1	0
		1	0	1	0	0	1
	S_{10}-S_6	0	1	0	1	-	ϕ
		1	0	1	0	-	ϕ
		0	0	0	0	-	0
		1	1	1	1	-	1
	S_{10}-S_7	0	1	0	1	1	ϕ
		1	0	1	0	0	ϕ
	S_{10}-S_8	0	1	0	1	0	-
		1	0	1	0	1	-
		0	0	0	0	0	-
		1	1	1	1	1	-
	S_{10}-S_9	0	1	0	1	ϕ	-
		1	0	1	0	ϕ	-
		0	0	0	0	0	-
		1	1	1	1	1	-
	S_{10}-S_{11}	0	1	0	1	ϕ	1
		1	0	1	0	ϕ	0
	S_{10}-S_{12}	0	1	0	1	0	0
		1	0	1	0	1	1
	S_{10}-S_{13}	0	1	0	1	ϕ	0
		1	0	1	0	ϕ	1
	S_{10}-S_{14}	0	1	0	1	0	ϕ
		1	0	1	0	1	ϕ
	S_{10}-S_{15}	0	1	0	1	ϕ	ϕ
		1	0	1	0	ϕ	ϕ

Table 3 Faults, Test Inputs and Outputs in Valid State S_5.

Fault		Test		Output			
				Normal		Faulty	
		x_1	x_2	$\hat{x}_1$	$\hat{x}_2$	$\hat{x}_1$	$\hat{x}_2$
Part I: Link Stuck Fault	x_1°, x_2°	1	0	0	1	0	0
		1	1	1	1	1	0
	x_1^1, $\hat{x}_2^1$	0	0	0	0	0	1
		0	1	1	0	1	1
	x_2°, $\hat{x}_1^{\circ}$	0	1	1	0	0	0
		1	1	1	1	0	1
	x_2^1, $\hat{x}_1^1$	0	0	0	0	1	0
		1	0	0	1	1	1
Part II: Switching Element Fault	S_5-S_0	0	1	1	0	-	-
		1	0	0	1	-	-
		0	0	0	0	-	-
		1	1	1	1	-	-
	S_5-S_1	0	1	1	0	1	-
		1	0	0	1	0	-
		0	0	0	0	0	-
		1	1	1	1	1	-
	S_5-S_2	0	1	1	0	-	1
		1	0	0	1	-	0
		0	0	0	0	-	0
		1	1	1	1	-	1
	S_5-S_3	0	1	1	0	1	1
		1	0	0	1	0	0
	S_5-S_4	0	1	1	0	-	0
		1	0	0	1	-	1
		0	0	0	0	-	0
		1	1	1	1	-	1
	S_5-S_6	0	1	1	0	-	ϕ
		1	0	0	1	-	ϕ
		0	0	0	0	-	0
		1	1	1	1	-	1
	S_5-S_7	0	1	1	0	1	ϕ
		1	0	0	1	0	ϕ
	S_5-S_8	0	1	1	0	0	-
		1	0	0	1	1	-
		0	0	0	0	0	-
		1	1	1	1	1	-
	S_5-S_9	0	1	1	0	ϕ	-
		1	0	0	1	ϕ	-
		0	0	0	0	0	-
		1	1	1	1	1	-
	S_5-S_{10}	0	1	1	0	0	1
		1	0	0	1	1	0
	S_5-S_{11}	0	1	1	0	ϕ	1
		1	0	0	1	ϕ	0
	S_5-S_{12}	0	1	1	0	0	0
		1	0	0	1	1	1
	S_5-S_{13}	0	1	1	0	ϕ	0
		1	0	0	1	ϕ	1
	S_5-S_{14}	0	1	1	0	0	ϕ
		1	0	0	1	1	ϕ
	S_5-S_{15}	0	1	1	0	ϕ	ϕ
		1	0	0	1	ϕ	ϕ

Table 4 Faulty Output Pattern in Case 2.

Subcase	Faulty Outputs 1	Faulty Outputs 2
A	01 or 10 Binary Vector	01 or 10 Binary Vector
B	01 or 10 Binary Vector	ϕ ϕ
C	01 or 10 Binary Vector	- -
D	ϕ ϕ	- -
E	ϕ ϕ	ϕ ϕ
F	- -	- -

Table 5 Classification of the Functional States in Case 2.

Valid State S_5 \ Valid State S_{10}	S_{12}	S_3	S_2	S_8	S_{11}	S_{14}
S_{12}	A	A	C	C	B	B
S_3	A	A	C	C	B	B
S_4	C	C	F	F	D	D
S_1	C	C	F	F	D	D
S_{13}	B	B	D	D	E	E
S_7	B	B	D	D	E	E

Table 6 Characteristics of Single Faults

Cases		Fault Types	No. of Fault Types	No. of Tests Needed For Determining the Fault Location	No. of Additional Tests For Determining the Fault Type
Case 1: One-Response Fault		$(S_3,S),(S_{12},S_5),(S_{10},S_3),(S_{10},S_{12})$	4	$4+2\lfloor\log_2(\log_2 N)\rfloor$ or $4+2\lceil\log_2(\log_2 N)\rceil$	0
		$(S_2,S_5),(S_8,S_5),(S_{11},S_5),(S_{14},S_5),(S_{10},S_1),(S_{10},S_4),(S_{10},S_7),(S_{10},S_{13})$	8	$4+2\lfloor\log_2(\log_2 N)\rfloor$ or $4+2\lceil\log_2(\log_2 N)\rceil$	2
Case 2: Separated Two-Response Fault	Subcase A	$(S_{12},S_{12}),(S_{12},S_3),(S_3,S_{12}),(S_3,S_3)$	4	4 or 8	0
	Subcase B & C	$(S_{12},S_{13}),(S_{12},S_7),(S_3,S_{13}),(S_3,S_7),(S_{11},S_{12}),(S_{11},S_3),(S_{14},S_{12}),(S_{14},S_3),(S_{12},S_4),(S_{12},S_1),(S_4,S_4),(S_3,S_1),(S_2,S_{12}),(S_2,S_3),(S_8,S_{12}),(S_8,S_3)$	16	4 or 8	2
	Subcase D & E	$(S_2,S_{13}),(S_2,S_7),(S_8,S_{13}),(S_8,S_7),(S_{11},S_4),(S_{11},S_1),(S_{14},S_4),(S_{14},S_1),(S_{11},S_{13}),(S_{11},S_7),(S_{14},S_{13}),(S_{14},S_7)$	12	4 or 8	0, 2, or 4
	Subcase F	$(S_2,S_4),(S_2,S_1),(S_8,S_4),(S_8,S_1)$	4	(cannot be located at the single switching element level)	(not distinguishable from a link stuck fault)
Case 3: Non-Separated Two- Response Fault		$\{S_0,S_1,S_4,S_5,S_6,S_7,S_9,S_{13},S_{15}\} \times \{S_5\}$; $\{S_{10}\} \times \{S_0,S_2,S_6,S_8,S_9,S_{10},S_{11},S_{14},S_{15}\}$	18	4	0 or 2
Case 4: Multiple-Response Fault		$\{S_0,S_1,S_4,S_5,S_6,S_7,S_9,S_{13},S_{15}\} \times \{S-S_5\}$; $\{S-S_{10}\} \times \{S_0,S_2,S_6,S_8,S_9,S_{10},S_{11},S_{14},S_{15}\}$	189	4	0, 2, or 4

CONCURRENT DIAGNOSIS IN PARALLEL SYSTEMS

L. Simoncini
Istituto di Elaborazione dell'Informazione - C.N.R.
56100, Pisa, Italy

A.D. Friedman
Dept. of Electrical Eng. and Computer Science
The George Washington University
Washington, D.C., 20052

Abstract

The advent of microprocessors as low cost general purpose computing elements has made feasible the implementation of multiprocessor systems containing a large number of cooperating modules. In these systems an early diagnosis of faulty modules will become increasingly important. Previous models of system level fault diagnosis have studied this problem considering that the normal operation of the system is interrupted and the diagnostic phase is started in the whole system.

When the number of modules in the system is large it is unlikely that all of them are busy with computation at all times. Therefore it may be possible to utilize this "slack" by having the non-busy modules perform diagnostics. In this way computation and diagnosis are performed concurrently in real time. In this paper we consider the problem of designing systems in which concurrent computation/diagnosis is possible and we evaluate the potential of such approach.

A generalization of previous models of self--diagnosis to take into account the concept of busy modules is introduced. For systems whose interconnection structure is defined by a special class of regular graphs, the relation between the degree of parallelism P, the maximal number of allowed faults t, the outdegree of each module in the system L and the number of modules n, is derived such that, without interrupting any busy module, it is possible to identify the status of all the non busy modules (either faulty or non-faulty) simultaneously in 1-step. Optimal strategies for scheduling the assignment of modules to be busy (i.e. assignment of tasks to modules assuming homogeneity) among the n modules of the system are given. Finally the characteristics of optimal connections for concurrent computation/ diagnosis are derived.

Introduction

The advent of microprocessors as low cost general purpose computing elements has made feasible the implementation of homogeneous multiprocessor systems containing a large number of cooperating and communicating modules. Such systems have many potential performance improvements including increased availability via fault diagnosis and reconfiguration leading to a potential graceful degradation should failures occur. To this end system level diagnosis of faulty modules will become increasingly important.

Previous models of system level diagnosis [1],[2], [3] have studied this problem considering that the normal operation of the system is completely interrupted when the diagnostic phase, which occurs periodically, is started in the system. When the number of modules in the system is large it is unlikely that all of them will be busy performing computations at all times. Therefore it may be possible to utilize this "slack" by having the non-busy modules perform diagnostics concurrently with computation by the busy modules. In this way computation and diagnosis are performed concurrently in real time.

The problem of concurrent computation and diagnosis has been introduced in [4] and has been studied for the case in which, without interrupting any of the modules which perform computation, it is possible to identify the status of at least one non busy module, therefore being able to start a sequential diagnosis in the system, in which the faulty modules are identified by several repetitions of testing and reconfiguration or repair.

In this paper we consider the problem of designing systems in which concurrent computation and diagnosis is possible and it is required than the status (faulty or fault- free) of all the modules not involved in computation can be identified by a single application of tests between these non busy modules. This is called 1-step diagnosis.

In Section 2 we introduce a generalization of previous models of system level diagnosis to take into account the concept of busy modules. In Section 3 for a special class of systems whose interconnection structures correspond to a type of regular graph (which are relevant from the point of view of easy diagnosis [5]) the relation among the degree of parallelism in the system, the maximal numbers of allowed faults, the outdegree of each module in the system and the number of modules is derived for 1-step concurrent diagnosis. Optimal strategies for the assignment of computation tasks to modules are given. Finally in Section 4 we show that optimal concurrent computation/diagnosis is strongly dependent on the interconnection structure of the system and we derive the characteristics of optimal connections for concurrent computation/diagnosis.

Model for concurrent computation/diagnosis

Following the Preparata Metze Chien model of system level diagnosis [1], we represent a self-diagnosing system as a graph G(V,E) in which the set of nodes V, $|V| = n$, corresponds to a set of homogeneous processor elements and the set E of arcs, $|E| = m$, represents the set of interconnections used for testing among the modules of the system. The test outcome is assumed to be binary: a "0" outcome of v_i testing v_j means that v_i has judged v_j as fault-free, and a "1" outcome means that v_i has judged v_j as faulty. Each module in G(V,E) may be in one of two states: "busy" when that module has been assigned a computational task, or "non busy" when that module is not performing a computational task and can be assigned to perform diagnostic tasks in the system. This is represented in the diagnostic graph G(V,E) by the removal of all the busy modules and the incoming and outgoing arcs from these modules. The subgraph $S_G(V',E')$, $V' \subset V$, $E' \subset E$, which is obtained represents the part of the system in which diagnosis is performed.

The problems which arise by introducing this generalized model are the following:

- The subgraph $S_G(V',E')$ must have certain properties required for diagnosis in order that all failures may be accurately diagnosed concurrently with the performance of computational tasks of the busy modules.
- We must consider the complexity of scheduling algorithm; that is that part which assigns computation or diagnostic tasks to modules. Two types of scheduling system are considered: a scheduling system which assigns the modules to be busy with random strategy, and an intelligent scheduling system which assigns the modules to be busy in an optimal way. The two types of scheduling systems allow the determination of the greatest lower bound, and of the greatest upper bound on the degree of parallelism (percentage of modules involved in computational tasks) which is possible in a system with n modules and consistent with the requirements of concurrent diagnosis.
- Finally we have to derive optimal systems that is systems in which we can maximize the number of busy modules (maximize the degree of parallelism in the system), while mantaining the maximal diagnostic capabilities in the rest of the system.

Concurrent computation/diagnosis in D_{1L} systems

In this section we consider the problems previously introduced for a special class of homogeneous multiprocessor system. These systems have an interconnection structure in which each module is connected to L other modules. If the modules are labelled 0, 1, ..., n-1, the interconnections from module i go to i+1, i+2, ..., i+L (where these module numbers are modulo n). Such system, called D_{1L} systems, have been shown to have useful diagnostic properties[5]. Figure 1 shows a D_{1L} system with n=8 and L=3.

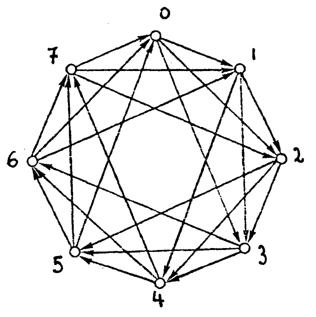


Fig. 1

Concurrent 1-step diagnosis in D_{1L} systems

For these systems, we consider the case in which it is desired that, after the removal of the B busy modules, the subgraph $S_G(V', E')$, V' = n-B is 1-step diagnosable (i.e. all faulty modules can be identified simultaneously by a single test application assuming the number of faulty modules is bounded by a constant t).

Definition: A system S is concurrently 1-step diagnosable if, after the removal of the B busy modules, it is possible in 1-step to identify the status of all the n-B remaining modules, which take part in the diagnostic phase, without interrupting any of the busy modules.

If the scheduling system adopts a random strategy in assigning the busy modules (i.e. assigning computational tasks to modules), Theorem 1 specifies the upper bound on the value t (the maximum number of faults which can always be identified in 1-step) in the subgraph $S_G(V',E')$.

Theorem 1: In D_{1L} systems it is possible to arbitrarily assign B busy modules, $1 \leq B \leq L-1$, so that the subgraph which remains after the removal of the B modules and associated arcs is at least $t = \left\{\min L-B, \left\lfloor \frac{n-B-1}{2} \right\rfloor\right\}$ fault diagnosable in 1-step.

Proof: Removing B nodes and the associated arcs, each of the remaining nodes is tested by at least L-B other nodes. Relabelling the n-B remaining nodes from 0 to n-B-1 in increasing order, the subgraph $S_G(V',E')$ corresponds to a $D_{1(L-B)}$ system. Since D_{1L} systems have been shown to be L fault diagnosable in 1-step if $n \geq 2L+1$ [1,6], it follows that if $L-B \leq \left\lfloor \frac{n-B-1}{2} \right\rfloor$ the subgraph is $t=L-B$ 1-step fault diagnosable, if $L-B > \left\lfloor \frac{n-B-1}{2} \right\rfloor$ the system is $t = \left\lfloor \frac{n-B-1}{2} \right\rfloor$ fault diagnosable in 1-step.

The following example shows that in D_{1L} systems, if $B \geq L$ the diagnostic subgraph obtained may not be diagnosable.

Example: For n=8 and L=3, a D_{1L} system is shown in Fig. 2a. If B=3 and modules 0,1,2 are assigned as busy, then $S_G(V',E')$ is as shown in Fig. 2b.

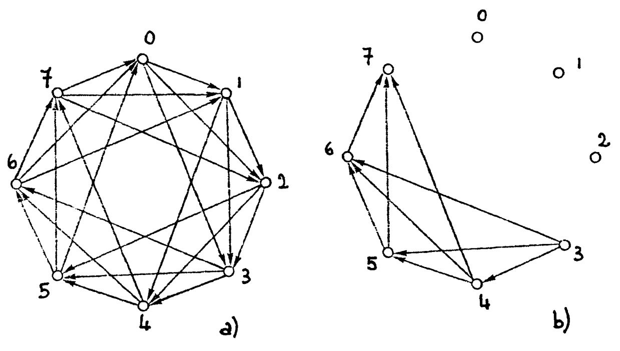


Fig. 2

For this graph, node 3 is not tested by any other nodes and hence a fault in this node cannot the diagnosed. Consequently the original system is not concurrently 1-step diagnosable for B=3.

We can derive a greatest lower bound on the degree of parallellism for D_{1L} systems assuming arbitrary assignment of busy modules where the degree of parallelism P is the percentage of (busy) modules which are assigned to computational tasks. The measure of P is given by $P=\frac{B}{n}$ 100%. The graph in Fig. 3, shows the relation between P and L for a given n and parameter t, which permits concurrent diagnosis, in the worst case of assignment of busy modules.

If "intelligent" assignment of computational tasks to modules is employed the degree of parallelism P consistent with concurrent diagnosis can be greatly increased.

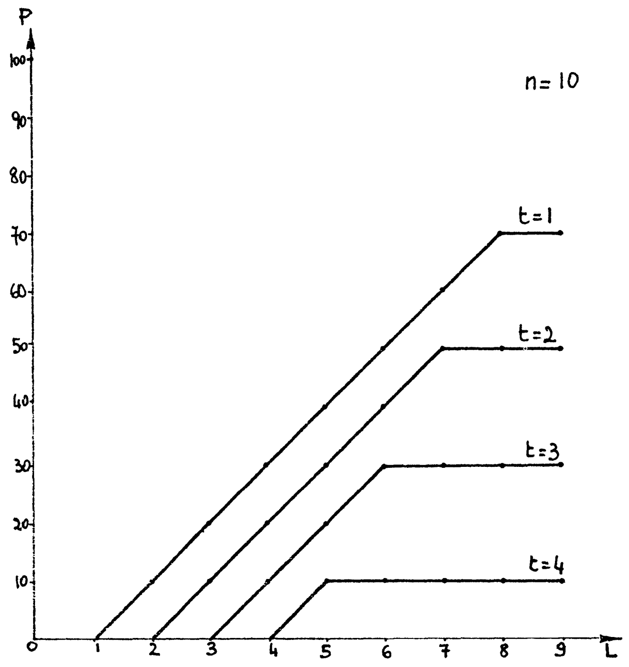


Fig. 3

Theorem 2: In a D_{1L} system, if $n \geq 2L+1$ and k is a positive number such that $kL \leq n < (k+1)L$, for any t, $1 \leq t \leq L$, it is possible to assign $B=k(L-t)+\alpha$ busy modules, (where $\alpha=0$ if $n-kL-t \leq 0$, otherwise $\alpha=n-kL-t$), in such a way that the diagnostic subgraph is 1-step t-fault diagnosable.

Proof: The proof is given through a constructive procedure for assigning B busy modules. If the nodes are labelled 0 through n-1, assign nodes 0 through t-1 as non busy and the nodes t...L-1 as busy; repeat this basic process k times. This leaves a total of n-kL unassigned nodes where $0 \leq n-kL < L$. If $n-kL \leq t$ the remaining n-kL nodes are assigned as non busy otherwise, if $n-kL > t$, nodes kL through kL+t-1 are assigned as non busy and nodes kL+t through n-1 are assigned as busy. With this procedure the n-B non busy nodes may be relabelled from 0 to n-B-1 in increasing order. The resulting diagnostic subgraph $S_G(V'E')$ is a D_{1t} system which from [1, 6] is 1-step t-fault diagnosable.

Theorem 3: In D_{1L} systems, if $L+1 \leq n \leq 2L$, it is possible to assign B=n-2t-1 busy modules in such a way that the system is concurrently 1-step t-fault diagnosable.

Proof: This proof is also given through a constructive procedure which defines the scheduling algorithm of computational tasks to modules.

We assume that the nodes are labelled 0 through n-1 and determine B and t.

If $L-t \geq B$, assign nodes 0 through B-1 as busy and assign the remaining n-B nodes as non busy. If $L-t < B$, assign nodes 0 through L-t-1 as busy, assign nodes L-t through L-1 as non busy, assign nodes L through n-t-2 as busy and nodes n-t-1 through n-1 as non busy. In both cases relabelling the n-B remaining nodes from 0 to n-B-1 in increasing order, the resulting subgraph $S_G(V',E')$ is a D_{1t} design which from [1] [6] is 1-step t-fault diagnosable.

Example: Consider the system D_{14} with n=15 and assume t=2, k=3 and $\alpha=1$. The diagnostic subgraph obtained by the application of the procedure in Theorem 2 is shown in Fig. 4.

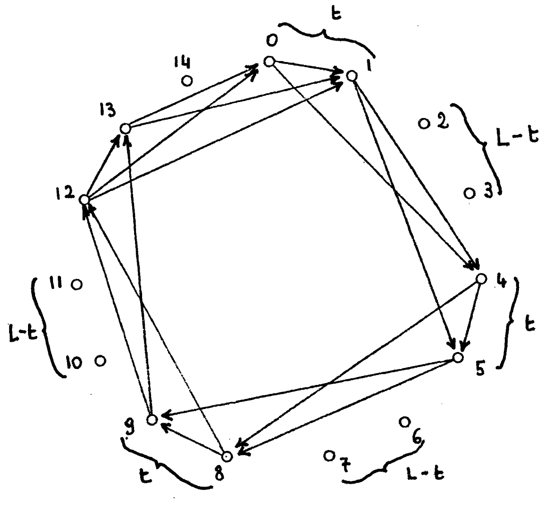


Fig. 4

As a second case consider the D_{16} system with n=10 and assume B=5 and t=2 ($L-t < B$). From the procedure of Theorem 3, the subgraph which is obtained is shown in Fig. 5.

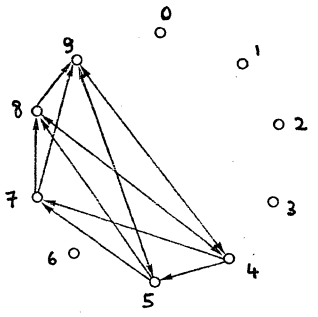


Fig. 5

The degree of parallelism P as a function of L for n=10 and several values of t for this intelligent scheduling algorithm is shown in Fig. 6.

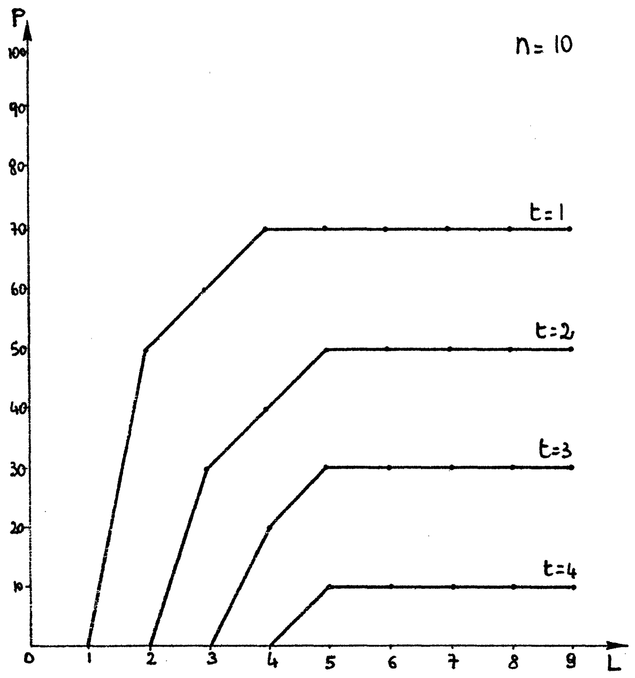


Fig. 6

The graphs in Fig. 3 and Fig. 6 depend strongly or n. If we consider n=100, L=2 and t=1 the degree of parallelism which is obtainable with any possible assignment of busy modules drops to 1% while with an intelligent assignment it remains at about 50%.

The scheduling algorithm described in Theorem 2 and 3 can be shown to be optimal for D_{1L} systems and therefore the graph shown in Fig. 6 represents the greatest upper bound on the degree of parallelism which can be obtained in

D_{1L} system with concurrent 1-step diagnosis.

Optimal connections for concurrent computation/diagnosis

In the previous section we have examined the characteristics of D_{1L} systems which enable concurrent computation and diagnosis. In this section we consider the problem of determining interconnection structures which are optimal for concurrent computation/diagnosis.

Definition: An interconnection structure is optimal for concurrent computation/diagnosis, if it is possible to assign B busy modules, for any $B \leq n-3$, in such a way that the system remains 1-step t-fault diagnosable with $t=\left\lfloor\frac{n-B-1}{2}\right\rfloor$.

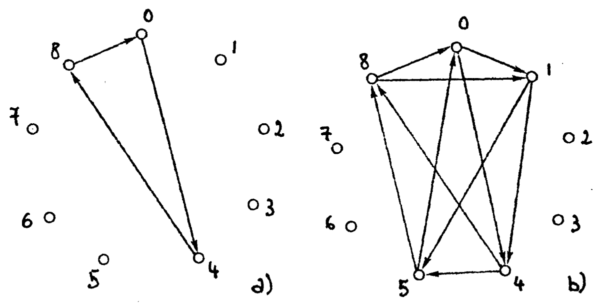
This definition of optimality corresponds to the statement that it is possible to assign the maximal number of busy units so that the diagnostic subgraph s is maximally 1-step diagnosable.

Theorem 4: For $L+1 \leq n \leq 2L+1$, D_{1L} systems define optimal interconnection structures for concurrent 1-step diagnosis.

Proof: From Theorem 3, for $L+1 \leq n \leq 2L$, it is possible to choose B=n-2t-1 busy modules and the diagnostic subgraph obtained by removing the B busy modules is 1-step $t=\left\lfloor\frac{n-B-1}{2}\right\rfloor$-fault diagnosable. For n=2L+1, from Theorem 2 we have k=2, $\alpha=0$ and B=2(L-t) and therefore we can assign B busy modules so that the remaining diagnostic subgraph is 1-step $t=\left\lfloor\frac{n-B-1}{2}\right\rfloor$fault diagnosable.

D_{1L} interconnection structures for n=2L+1 are of particular interest for concurrent 1-step diagnosis. Let us consider a D_{14} system for n=9.

The complete system (i.e. B=0) is 1-step 4-fault diagnosable. If we apply the procedure in Theorem 2 we get for t=1,2,3, the diagnostic subgraphs shown in Fig. 7a, 7b, 7c, respectively, which are respectively D_{11}, D_{12} and D_{13} systems.



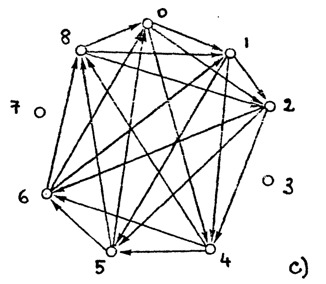


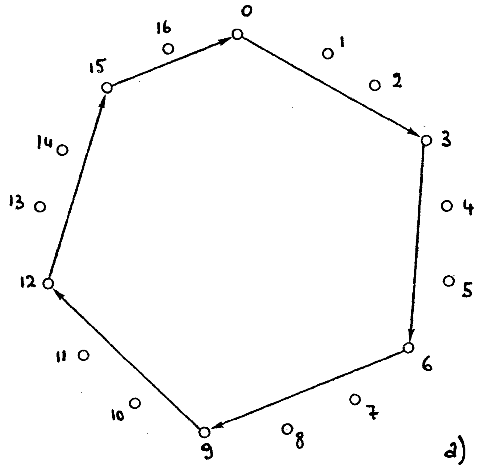
Fig. 7

In general for $n > 2L+1$, D_{1L} systems are not optimal in the sense that it is not always possible to assign the maximal number of busy modules so that the remaining subgraph is maximally 1-step fault diagnosable.

Example: Consider a system with n=17 and L=3. For the D_{13} interconnection structure the node i is connected to i+1, i+2 and i+3 (mod. n)for i=0,...,n-1. The application of the procedure in Theorem 2 for t=1 results in the diagnostic subgraph shown in Fig. 8a

In this case the degree of parallelism is only 64.7% (B/n = 11/17) and the subgraph of Fig. 8a is only 1-step 1-fault diagnosable. No other assignment of more than 11 busy modules for the D_{13} system enables concurrent 1-step diagnosis on the remaining diagnostic subgraph.

Now consider another system with L=3 whose interconnection defined by $i \rightarrow i+1, i+5, i+8$ (mod. n), i=0,..., 16. In this case 14 busy modules can be assigned as shown in Fig. 8b. The degree of parallelism becomes 82.3% and the subgraph in Fig. 8b is 1-step 1-fault diagnosable.



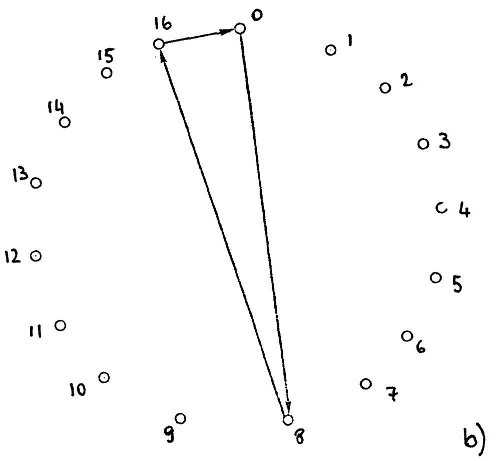


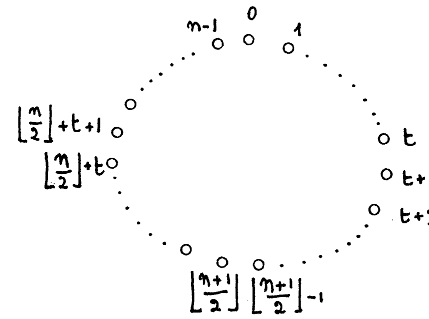
Fig. 8

The example shows that for $n > 2L+1$ D_{1L} systems are not necessarily optimal with respect to concurrent computation/diagnosis.

The following theorem determines an optimal interconnection structure for concurrent 1-step diagnosis.

Theorem 5: The systems defined by the following interconnection structure are optimal for concurrent 1-step t-fault diagnosability: there are up to 2t interconnections from module i defined by $i \rightarrow i+\alpha \pmod{n}$ $i \rightarrow i+\left\lfloor\frac{n+1}{2}\right\rfloor-\alpha \pmod{n}$, for $0 \leq i \leq n-1$, $\alpha = 1,2,\ldots,t$. Nodes 0 through t are assigned as non busy, together with the nodes $\left\lfloor\frac{n+1}{2}\right\rfloor$ through $\left\lfloor\frac{n}{2}\right\rfloor+t$. All the other nodes are assigned as busy.

Proof: Let us refer to Fig. 9.



n-1
0
1
t
t+1
t+2
$\left\lfloor\frac{n}{2}\right\rfloor+t+1$
$\left\lfloor\frac{n}{2}\right\rfloor+t$
$\left\lfloor\frac{n+1}{2}\right\rfloor$
$\left\lfloor\frac{n+1}{2}\right\rfloor-1$

Fig. 9

If n is odd 2t+1 nodes are assigned as non busy. If n is even 2t+2 nodes are assigned as non busy. We show that each non busy node is tested by t other non busy nodes. We show this for n odd; the case of n even is quite similar.

For n odd, the nodes $0 \leq i \leq t-1$ are tested by the i-1 non busy nodes 0 through i-2. They are also tested by the nodes $i+\frac{n+1}{2}-1$ through $\frac{n+1}{2}+t=\left\lfloor\frac{n}{2}\right\rfloor+t$. These nodes are non busy since $i+\frac{n+1}{2}-1 \geq \frac{n+1}{2}$ $(i \geq 1)$ and $i+\frac{n+1}{2}-1 \leq \frac{n+1}{2}+t-1$ $(i \leq t)$. The total number of non busy nodes which test nodes $0 \leq i \leq t-1$ is $\frac{n+1}{2}+t-1-i-\frac{n+1}{2}+1+1+i-1=t$. The node $i=t$ is tested by the non busy nodes 0 through t-1.

The nodes $\frac{n+1}{2} \leq i \leq \frac{n-1}{2}+t$ are tested by the non busy nodes $\frac{n+1}{2}$ through i-2, and by the nodes $i-\frac{n-1}{2}-1$ through t. These last nodes are non busy since $i-\frac{n-1}{2}-1 \geq 0$ $(i \geq \frac{n+1}{2})$ and $i-\frac{n-1}{2}-1 \leq t$ $(i \leq \frac{n-1}{2}+t+1=\left\lfloor\frac{n}{2}\right\rfloor+t+1)$. The total number of non busy nodes which test the nodes $\frac{n+1}{2} \leq i \leq \frac{n-1}{2}+t$ is $t-i+\frac{n-1}{2}+1+1+i-2-\frac{n+1}{2}+1=t$. Therefore each non busy node is tested by t other non busy nodes. Relabelling the 2t+1 remaining nodes of the diagnostic subgraph $S_G(V',E')$ from 0 to 2t in increasing order results in a D_{1t} system which from [1] and [6] is 1-step t-fault diagnosable.
The proof for n even is similar.
In both cases these connections are optimal with respect to the degree of parallelism.

Example: Consider the system with n=17, L=6 and interconnection structure defined by the connections from node i, $i \rightarrow i+1$, $i+2$, $i+3$, $i+6$, $i+7$, $i+8$. For t=3 the diagnostic subgraph resulting from the assignment of Theorem 5 is shown in Fig. 10.

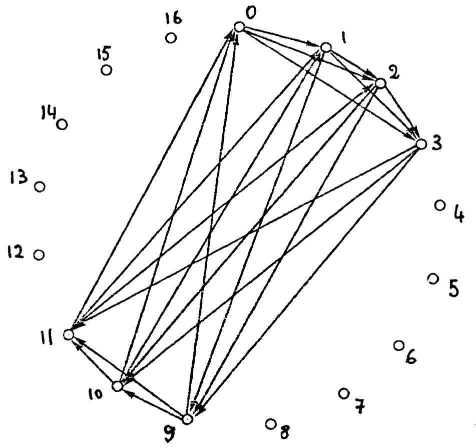


Fig. 10

This subgraph is a D_{13} system with 7 nodes.

Similarly if n=14, L=6 and t=3, the interconnection structure is defined by i i+1, i+2, i+3, i+4, i+5, i+6, and the system is shown in Fig. 11.

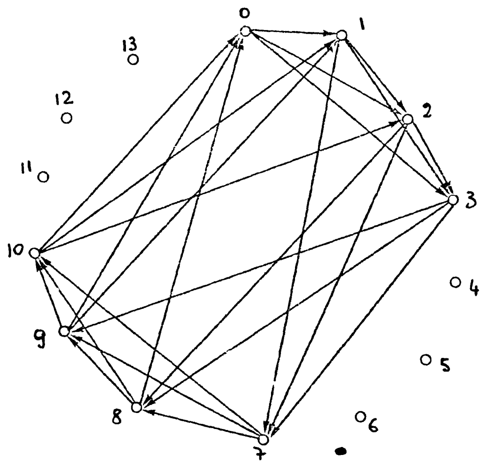


Fig. 11

The diagnostic subgraph is a D_{13} system with 8 nodes.

The interconnection structure described in the Theorem 5 results in concurrent 1-step diagnostic systems with degree of parallelism is given by $P=\frac{n-2t-1}{n}$ 100% for n odd ($P=\frac{n-2t-2}{n}$ 100% for n even). For n=100 the value of P as a function of t (for t=1, 2, ..., 10) is shown in Fig. 12.

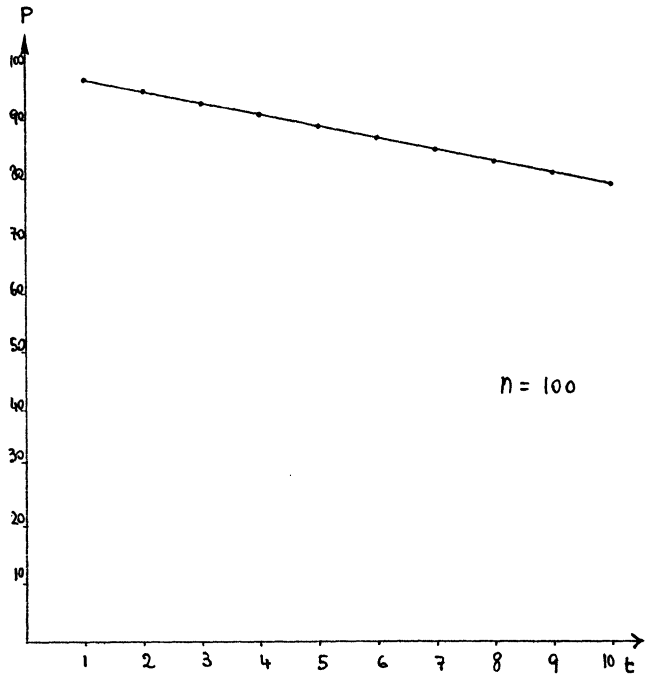


Fig. 12

Table 1 summarizes the values of parallelism obtainable with concurrent 1-step diagnosis for systems with D_{1L} interconnection structure and other optimal interconnection structures. The worst and the best assignment of busy module patterns are also summarized.

Connections	Random scheduling	Worst busy module pattern
D_{1L} ($n \leq 2L+1$) $i \to i+1$, $i+2$, ..., $i+L$ mod.n	$\begin{cases} P=\frac{L-t}{n}\ 100\% \\ L \leqslant n-t-1 \end{cases}$ $\begin{cases} P=\frac{n-2t-1}{n}\ 100\% \\ L > n-t-1 \end{cases}$	B consecutive modules

Optimal Scheduling	Best busy module pattern
$P=\frac{k(L-t)+\alpha}{n}\ 100\%$ $\begin{cases} \alpha=0 & n-kL-t \leqslant 0 \\ \alpha=n-kL-t & \text{otherwise} \end{cases}$ $kL \leqslant n < (k+1)L$, $n \geqslant 2L+1$	Alternating chains of t non busy mod. and L-t busy modules
$\begin{cases} P=\frac{n-2t-1}{n}\ 100\% \\ L+1 \leqslant n \leqslant 2L \end{cases}$	See proof of Theor. 3

Optimal Connections ($n > 2L+1$) $i \to i+\alpha$ $i \to i+\lfloor\frac{n+1}{2}\rfloor-\alpha$ $\alpha= 1,2,...,t$	n odd $P=\frac{n-2t-1}{n}$ 100% n even $P=\frac{n-2t-2}{n}$ 100%	Nodes t+2,... ...,$\lfloor\frac{n+1}{2}\rfloor$ and $\lfloor\frac{n}{2}\rfloor$+t+2,..... ....,n are busy

Table 1

Conclusions

In this paper we have considered the problem of designing homogeneous multiprocessor systems in which concurrent 1-step diagnosis is possible. For systems, whose interconnection structures are defined by regular graphs (called D_{1L} systems), lower bounds and upper bounds on the maximum number of modules involved in computation (degree of parallelism) have been derived and strategies for the assignment of computational tasks to modules have been given. Optimal interconnection structures which enable concurrent 1-step t-fault diagnosability have also been presented. Some more work is needed to take into account the fact that in general required connections between busy modules may be constrained by the algorithms to be executed in the system. A more integrated study of the constraints and the tradeoffs required by computation and diagnosis will be needed in order to more fully exploit the potential performance improvement of multiprocessor systems.

References

1 F.P. Preparata, G. Metze, R.T. Chien: "On the Connection Assignment Problem of Diagnosable Systems", IEEE Trans. Electr. Comp., Vol. EC-16, pp. 848-854, Dec. 1967.

2 J.D. Russel, C.R. Kime: "System Fault Diagnosis: Closure and Diagnosability with Repair", IEEE Trans. on Computer, Vol. C-24, pp. 1078-1089, Nov. 1975.

3 J.D. Russel, C.R. Kime: "System Fault Diagnosis: Masking, Exposure and Diagnosability Without Repair", IEEE Trans. on Computer, Vol. C-24, pp. 1155-1161, Dec. 1975.

4 F. Saheban: "Diagnosis and Reconfiguration in Multiprocessor Systems", Ph.D. Dissertation, University of Southern California, Sept. 1978.

5 L. Simoncini, A.D. Friedman: "Incomplete Fault Coverage in Modular Multiprocessor Systems", Proc. ACM 78 Conference, Washington D.C., pp. 210-216, Dec. 1978.

6 S.L. Hakimi, A.T. Amin: "Characterization of Connection Assignment of Diagnosable Systems", IEEE Trans. on Computer, Vol. C-23, 86-88, Jan. 1974.

This work was supported by the Convention between Selenia s.p.a. and C.N.R. of Italy, and in part by the National Science Foundation under grant NSF MCS77-21569.

RELACS, A DATA BASE COMPUTER

Ellen Oliver*and P. Bruce Berra
Dept. of Industrial Engineering and Operations Research
Syracuse University
Syracuse, New York 13210

Summary

RELACS, Relational Associative Computer System, is a backend data base computer designed to support a relational data model. As shown in Figure 1, there are six main functional blocks: the Global Control Unit (GCU); the Data Dictionary Processor (DDP); the Associative Query Translator (AQT); and the Output Buffer (OB).

Queries to the data base originate with the user and are preprocessed by the host before reaching RELACS. The preprocessor in the host computer parses the query, performs syntactical functions, and determines the attribute and/or relation search order such that the logical relationship(s) of the query is maintained. When the sequence of query and control instructions required for execution is created, it is stored in an area of the host computer memory known as the Transfer Users Memory Area (TUMA) [1]. The TUMA may be implemented as a first-in-first-out queue of jobs or it may include the more sophisticated capability to rank jobs by predetermined priority criteria. In either case, once a job (i.e., query) is placed in the TUMA, the host will notify the GCU of RELACS.

The first step in processing a query is to utilize the DDP [1] which performs data dictionary and data directory functions. The DDP is envisaged as a group of associative array memories interconnected by arrays of specially designed cells. When processing a query, the DDP will first establish whether or not the attribute and/or relation names exist in the data base. If they do, the DDP will verify t' ~ security and access privilege levels of the user fcr the specific attributes and/or relations required. For the case where multiple relations are required, the DDP will establish whether or not a join can be constructed between the relations. Assuming that the user has the correct passwords, the attributes and/or relations exist and the logical relationship(s) can be constructed, descriptor data for all attributes referenced directly or indirectly in the query are output to the AQT.

The AQT performs the function of translating the original sequence of instructions from the user and the descriptor data from the DDP into a set of instructions executable by the AUs. Implementation of the AQT would include associative array memories which contain the query and descriptor data required for translation, a RAM which stores all relation descriptor data and a storage stack or buffer which contains the AU instructions as they are created.

The AUs access the AQT to retrieve a set of instructions which will enable the AU to access the MSD to retrieve the required relation, to perform the search and/or update specified by the query and to execute the necessary output. An associative array memory is the heart of the AU which also includes a comparand array for multiple search arguments, a response array, an output buffer and I/O capabilities. The output from the AU will be either to the MSD (update), to the user via the OB and host computer, or to the other AU (join).

The MSD is a large capacity (> 10^9 bits) storage device with speed and bandwidth requirements compatible with the AU. In addition, the storage structure is a row-column matrix of cells which are addressed as blocks. Each block is equivalent to the associative array memory size.

The OB is provided for those queries which require output from more than one relation. In general, the tuples from one relation will be fully searched by one of the AUs; therefore, when the tuples from more than one relation are searched, there will be output from more than one AU. The OB provides intermediate storage between the AUs and the host computer. In addition, it can provide the capability for collating tuples and/or formatting of data according to the user requirement. From here, the data are returned to the user via the host computer.

The RELACS system was designed with the objective that it support a relational data model yet overcome the weaknesses of earlier systems, namely, the I/O bottleneck due to the relatively slow I/O with respect to faster search times, and the requirement that the entire data base be searched at least once for each query. The DDP provides the capability to precisely locate the relations which must be searched to process the query. I/O time is minimized by utilizing a custom designed MSD which interfaces with the AU such that rapid, parallel data transfers take place. In addition, the AUs provide the capability to perform content based, parallel search operations and a facility to enable the join and/or projection operations.

* Currently at Bell Telephone Labs., Holmdel, New Jersey.

References

[1] Capraro, "A Data Base Management Modeling Technique and Special Function Hardware Architecture," Doctoral Dissertation, Syracuse University, 1978.

[2] Oliver, "RELACS, An Associative Computer to Support a Relational Data Model," Doctoral Dissertation, Syracuse University, 1979.

[3] Special Issue of IEEE on Data Base Machines, Computer, March 1979, Vol. 12, No. 3.

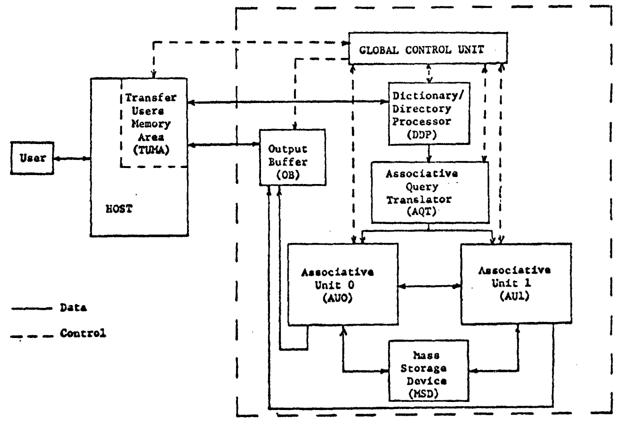

GLOBAL CONTROL UNIT
Transfer Users Memory Area (TUMA)
Dictionary/ Directory Processor (DDP)
User
Output Buffer (OB)
Associative Query Translator (AQT)
HOST
Associative Unit 0 (AU0)
Associative Unit 1 (AU1)
Data
Control
Mass Storage Device (MSD)

Figure 1 RELACS, Relational Associative Computer System, Organization.

PARALLEL RECOGNITION OF PARABOLIC AND CONIC PATTERNS BY BUS AUTOMATA

J. Rothstein and A. Davis
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

ABSTRACT -- Parallel algorithms are described for recognizing parabolas and conics on bus automata. The parabola algorithm implements a conformal transformation by means of which recognition is similar to that of a straight line. It is capable of generalization to other curves. The conic recognition algorithm exploits geometric properties of conics, recognizing all kinds, but does not generalize as the properties are definitive for conics. The procedures and BA architecture are given in detail for conceptually important cases.

I. INTRODUCTION

In a paper given at the 1978 Conference on Parallel Processing [1] it was shown that bus automata, (BA), which are cellular automata augmented by locally controllable intercellular communication channels [2,3,4,5], could recognize patterns by parallel algorithms. Both topological and metric features could be recognized. Recognition of straight lines [6] and their images under conformal transformations was also discussed, the first having been solved earlier while the second depended on implementing a global coordinate transformation. Preliminary discussion was given in [1] of the special case of parabola recognition using this approach. This paper gives an explicit algorithm for doing it, including the all-important coordinate transformation procedure, and the design of a BA which implements it in parallel. It uses both geometric properties of the parabola and a particular conformal mapping taking straight lines into parabolas. A second algorithm was devised which recognizes all conics immediately, using only geometric properties common to the conics. It is both more and less general than the preceeding algorithm, for parabolas are a special case; but the conformal map method is not restricted to parabolas, or even conics in general.

Both algorithms are particular cases of powerful general methods. This is clearly the case for the conformal approach, but the second case is not so obvious. It exploits a particular geometric property. Geometric properties can be specified in many ways, one useful way being those invariant under some group of transformations. Possession of such a property is often the definition of a class of geometric objects. The second algorithm uses a property of the set of diameters of a conic for immediate recognition.

Both kinds of algorithms are included in the general theory of [1], but the subtle interplay of the metric and topological there noted needs many examples for its clarification. An important object of this paper, over and above the intrinsic interest of the algorithms themselves, is to provide such examples.

In the following sections, we refer to the first algorithm as the Conformal Parabola Recognition Algorithm, and the second as the Conic Recognition Algorithm. In Section II, we discuss the theory on which each algorithm is based. Sections III and IV present the algorithms and their BA architectural embodiments. Section V has some comments on the error involved in BA algorithms.

II. THEORY

In this section, we develop the theory required in the design of the two algorithms. First, we consider Conformal Parabola Recognition. The conformal map is discussed as are certain properties of the parabola which allow us to locate the map's singularities. An overview of the algorithm is given, and the discrete approximation used to represent the conformal parabolic net is considered. Next, we take the Conic Recognition Algorithm. Properties of diameters of conic sections are noted, and the flow of the pursuant algorithm is presented.

1. CONFORMAL PARABOLA RECOGNITION

Our notation follows [1] except that we interchange the roles of w and z to avoid the nuisance of dealing continually with the inverse of the function f(z) defined there. The conformal transformation

$$w = f(z) = z^{1/2} \qquad (1)$$

takes the cartesian coordinate system (u,v) where

$$w = u + iv \qquad (2)$$

into the confocal, coaxial, parabolic coordinate system shown in Figure 5.2 of [1] in the z-plane, with cartesian coordinates (x,y) given by

$$z = x + iy. \qquad (3)$$

The two families of parabolas all have the origin as focus and the x-axis as their symmetry axis. The two families are open to the -x direction in one case, to the +x direction in the other, and every member of one family is orthogonal to all members of the other [7, p. 564].

The parabola recognition strategy uses the following intuitive notion. Let the putative parabola be drawn in the z-plane. If we find its focus and axis of symmetry, we can then translate the origin to the focus and rotate the coordinate system so that the new x-axis lies along the axis of symmetry. In this new coordinate system the putative parabola is "parallel" to the parabolas of one family; it either coincides with one of them or lies between two of them, not intersecting either. Its pre-image in the w-plane is either a coordinate line or a straight line lying between two adjacent parallel coordinate lines which never meets them, i.e., it is parallel to them. To within the resolving power of the grid this is a straight line, for wiggles within a cell are undetectable. Parabola recognition is a transform of straight line recognition.

This motivates the following general approach. The z-plane is represented by a data-plane consisting of cells, one to a coordinate square. The putative parabola is stored in bits in the states of cells traversed by it. Control borders and planes are used, in addition, for help in carrying out the four main steps of the algorithm. These are:

1. Locate the directrix of the parabola.
2. From it, orient the cartesian coordinate system, and find the focus.
3. Generate the adjacent grid parabolas.
4. Recognize.

The major problem is step 3, which is the embodiment of the conformal transformation (1), essentially. It is done immediately, with the present algorithm, corresponding to "propagating" the useful portion of the parabolic coordinate system out from the focus, which is a singular point of mapping (1) (branch point). Step 4 is trivial, after step 3 is done.

Finding the directrix depends on the following property of the parabola. If two tangents to a parabola meet at right angles, their point of intersection lies on the directrix [8, p. 129]. Step 1 uses a pair of right angles (represented by busses) each of which is translated until it touches the parabola in two points. The straight line through the corners is the directrix. Translating a line parallel to the directrix until it touches the parabola locates the vertex of the parabola. The vertex lies along the axis of symmetry (which is perpendicular to the directrix) half-way between directrix and focus. Step 2 thus finds focus and axis of symmetry from input data and directrix. The rotated coordinate system has the symmetry axis, and the line through the focus perpendicular to that axis, as its new x and y axes.

The geometrical basis of step 3 is as follows. The parabolic net in the z-plane is the image of the integer net (u,v) in the w-plane under the transformation

$$z = f^{-1}(w) = w^2 \tag{4}$$

As we saw above, the coordinate lines u = const. and v = const. go over to confocal coaxial parabolas with the x-axis their common symmetry axis. We approximate a net parabola by a chain of chords connecting successive lattice points on the parabola (starting from the vertex and following the $\pm y$ branches to infinity). To find the slopes of these chords for the parabola determined by $v = v_o$ we compute the change in x and y as u varies from u_o to $u_o + 1$. The coordinates of its vertex as $(-v_o^2, o)$, by (4) (here $w = o + iv_o$). We have (where $w_o = u_o + iv_o$)

$$f^{-1}(w_o + 1) - f^{-1}(w_o) = (2u_o + 1) + i(2v_o) \tag{5}$$

Along this parabola we obtain constant increments in y ($2v_o$, or on the negative branch, $2v_o$), and successive odd integers in x. (If u_o is fixed and v_o varies, we get the other family, with y increments $\pm 2u_o$, and the parabolas open up in the opposite direction.) The slopes of consecutive chords are

$$2v_o/1,\ 2v_o/3,\ 2v_o/5,\ \ldots,\ 2v_o/(2n + 1),\ \ldots \tag{6}$$

in the upper half-plane, the negatives of these values in the lower half-plane.

Plotting the adjacent parabolas in step 3 thus involves successive chord construction once the vertices have been located. Step 4 checks whether the putative parabola falls between the adjacent ones (or coincides with one of them) or not.

2. GENERAL CONIC RECOGNITION

The Conic Recognition Algorithm uses properties of sets of parallel chords. The algorithm also distinguishes between ellipses, parabolae, and hyperbolae including degenerate or limiting cases. The equation of a conic in polar coordinates (r,θ) is given by

$$r = \ell/(1 - e \cos \theta) \tag{7}$$

where e is the eccentricity, ℓ/e is the distance from focus to directrix, the origin is at the focus, and the ray $\theta = o$ lies along the axis of symmetry. The case $e = o$ is a circle of radius ℓ (the directrix is the "line at infinity"), the range $o \leq e < 1$ corresponds to ellipses, $e = 1$ is a parabola and $e > 1$ gives the hyperbolae. The circle is a degenerate ellipse, the parabola is a limiting case of both ellipses and hyperbolae, two parallel straight lines are a degenerate parabola, and two intersecting straight lines are a degenerate hyperbola. The conics have two foci, coincident in the case of the circle, with the point at infinity one focus of the parabola, and with the midpoint of the line joining the foci called the center. Only when the foci are separated by a finite distance is the center not at infinity. In that case the center is a center of symmetry and the conic is called a central conic. Straight lines through the center which intersect the conic are called diameters. Two diameters d_1, d_2 are conjugate if their slopes (in a coordinate system (x,y) with origin at the center, x-axis along the symmetry axis, y-axis parallel to the directrix) m_1 and m_2 satisfy

$$m_1 m_2 = e^2 - 1 \tag{8}$$

An important theorem is the following.

Theorem 1 The locus of midpoints of chords parallel to a diameter of a central conic is the conjugate diameter.

For the parabola, this theorem becomes:

Theorem 2 The locus of midpoints of a set of parallel chords of a parabola is a line parallel to the symmetry axis.

The above theorems and (8) suggest the following general algorithm. Choose slope m_1 arbitrarily, covering the finite grid with lines of this slope. Find the midpoints of all segments with endpoints in the locus to be recognized. If any line is found to cut the locus more than twice, here or later, the locus is rejected, since no line cuts a conic in more than two points. (If it cuts it once, say lines parallel to the symmetry axis of a parabola choose a different value for m_1.) If this is not the case and the midpoints lie on a straight line of slope, say, m_2, then the grid is again covered, this time with lines of slope m_2. Midpoints are determined as before. If the new locus is a straight line of slope m_1, the locus is a central conic. For a parabola midpoints are at infinity as the lines of slope m_2 cut the parabola at only one point.

Complete justification of the algorithm requires the following. If a symmetric smooth curve (which can have two disconnected portions, i.e., the connection is by way of the point at infinity in this case) satisfies (a) lines through the center define diameters by their intersections with the curve, (b) when tangents to the curve are constructed at those points of intersection they are parallel, (c) the line through the center parallel to the tangents defines another diameter (said to be conjugate to the first one) and (d) the slopes of the diameter and its conjugate are related by eqn. (8), then the curve is a conic. The proof requires only elementary ananlytical geometry and calculus, and is omitted.

To distinguish the parabola from the others we use at least two values for slope m_1. If the lines determined by the corresponding chord midpoints are parallel, then the conic is a parabola. The degenerate parabola consisting of a pair of parallel lines occurs if and only if the two lines of chord midpoints coincide. In the non-parabolic case, the lines intersect at the center. The conic is a hyperbola if the center is in an open region of the plane and an ellipse or circle if it is in a closed region of the plane. In the open case, the degenerate hyperbola consisting of two intersecting straight lines occurs if and only if the center is on the locus being recognized. It is a circle if conjugate diameters are always equal (or always orthogonal).

III. CONFORMAL PARABOLA RECOGNITION

We next present the detailed architecture and then the mode of operation of a BA implementing the conformal recognition algorithm. A BA is an array of finite state automata (cells) linked by a locally controllable switching network permitting communication channels (busses) to be established between widely separated cells [2]. The busses are assembled from links, each of which conducts or not in accordance with the state of the particular cell with which it is associated.

1. ARCHITECTURE

The BA architecture is shown in Figure 1.

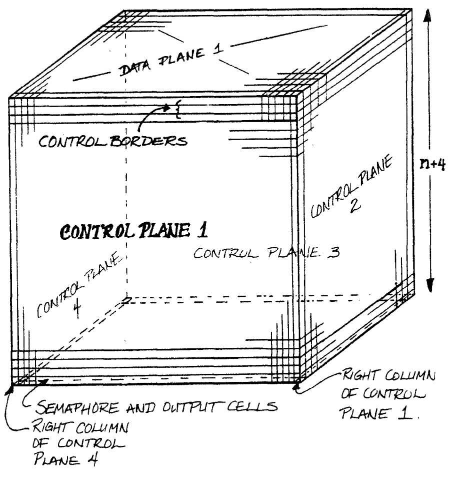


FIGURE 1 - BA Architecture

Three data planes, all n x n arrays of cells, are used, each representing the z-plane. Planes 1 and 2 execute the bulk of the algorithm, while 3 stores input (and intermediate) information. Each data cell has 3 (directed) links to each of its 8 nearest neighbors within its data plane, i.e., those cells sharing either a face or an edge with it. Cells in adjacent data planes, have links to cells with the same row and column coordinates. Four control planes border the edges of the data planes and have "right columns" perpendicular to (the) data planes at their corners (see Figure 1). Each of their control cells has links to its nearest neighbors within its control plane, as well as between planes at right column junctions. The top three rows of the control planes form the control borders. The control border cells are the only control cells linked directly to data cells in the data plane they bound. Finally, certain cells of the bottom row of each control plane are designated as semaphore cells, with one of them designated as the output cell.

In the following discussion, d, s, F, and ℓ will denote directrix, axis of symmetry, focus, and latus rectum (the line segment parallel to d at F with endpoints on the parabola), respectively.

2. CONFORMAL PARABOLA RECOGNITION

The pattern to be recognized is initially encoded (loaded) into the states of the data cells. Other information effectively encoded into the initial state of a cell is: the type of cell (data, control, semaphore or output); and its general location, i.e., data plane 1, 2 or 3, control 1-4, whether the cell is on a control border, right column or other special site. To initiate the algorithm, an external signal is sent to the semaphore cell controlling Step 1.

The synchronization of separate subtasks of the procedure is accomplished by signals sent between semaphores and the rest of the cells. Each semaphore oversees a particular subtask: starting it; awaiting completion; and passing control to the next or to the output cell (halt). Termination signals are sent to waiting semaphores by designated cells at the control borders of the data planes. When the final step is complete, the result, encoded in the termination signal of the last subtask, is displayed at the output cell.

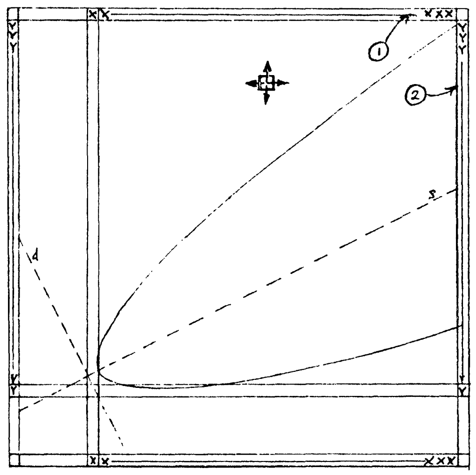
The restricted arrangement was chosen for convenience in describing and implementing the algorithm. Plane dimensions (n) are large enough to contain the entire parabolic piece (including vertex, focus and a piece of the directrix) constituting the curve to be recognized. If we had done everything in a single large plane, the diagrams and opewation would have been complicated by the necessity to construct geometrically complicated bus systems.

2.1 LOCATE THE DIRECTRIX -- Step 1 finds d by using the fact that perpendicular tangents to a parabola meet at d.

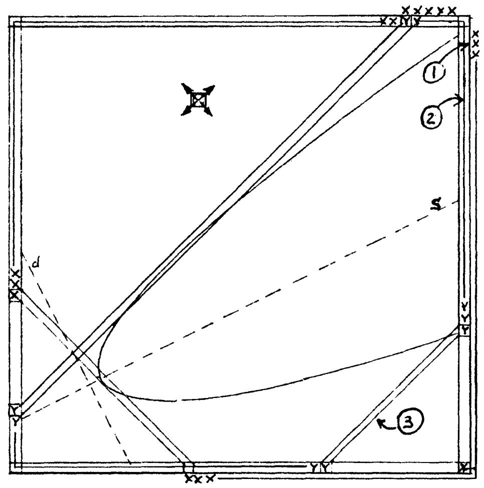
The curve, stored in data plane 3, is loaded into 1 and 2. We examine all candidate vertex positions corresponding to a pair of translatable right angles simultaneously by setting up, in the buasses of data planes 1 and 2 two distinct orthogonal coordinate grids. The slopes of their axes are exactly the slopes of the rays of the right angles. They are chosen as follows: 0(0°) and infinity ∞(90°) in data plane 1; and 1 (45°) and -1 (135°) in data plane 2. Plane 1 sets up its grid by connecting all horizontal and vertical links to form busses, 2 by connecting diagonal links. Each cell of both planes now has coordinate busses running through it (4 busses, + and - for each coordinate).

Consider the horizontal tangent (Figure 2(a)); the other three are similar. To find it, the curve is projected along horizontal busses to the control borders. To do this, each data cell crossed by the locus sends a signal in both horizontal directions along its busses. The "tangent" busses are selected at the control border by those cells which receive a projection signal while one of their nearest control border neighbors does not. False tangents are also identified in this process (Figure 2(b)). They correspond to the situation in which a neighboring data cell of one of the 2 control endcells (at opposite ends of the tangent) is crossed by the locus.

Data planes 1 and 2 concurrently determine tangents using the above method along both their coordinate directions. If the pattern is a parabola, two cases can occur. First, if two



(a) Plane 1: Coordinate Bus System with Horizontal and Vertical Links



(b) Plane 2: Coordinate Bus System with Diagonal Links

1) Lines x = constant project here (and opposite side)
2) Lines y = constant project here (and opposite side)
3) False tangent

FIGURE 2 - Tangent Determination

orthogonal tangents in each plane are found, then two cells on d can be determined (corner cells) as follows. Both corner cells are mapped by projections perpendicular to the data planes to cells in the data plane 1. There we run an immediate algorithm for plotting a line from two distinct points on it (see Appendix I) to represent d in the states of the cells crossed by it.

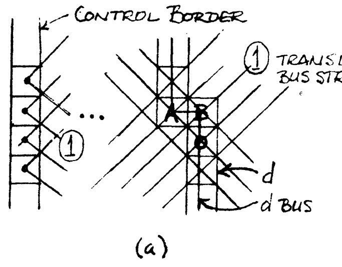
The second case is the special case in which the rays of one of the right angles are parallel to d and s, respectively. Five tangents, two false and parallel, are found. In one plane, we find a corner cell as before. In the other, we find the two false tangents parallel to the symmetry axis, and a true tangent in the other direction at the vertex. Hence d is determined by its slope and a point on it (the slope is that of the true tangent).

<u>2.2 STEP 2: ORIENT GRID AND FIND FOCUS</u> -- In this step, we first rotate the coordinate system (in data plane 1) making d the new y-axis. For uniqueness choose that intersection of d with an edge of the data plane which is closest to a designated edge as the center of rotation, and define the positive direction along d as the one making its slope negative in the old coordinate system. We define the positive direction of the new x-axis (perpendicular to d) as that which translates us from d toward the parabola. The final coordinate system (x',y') will have the origin at F, x'-axis along s, and y'-axis parallel to d (along ℓ). To construct the (x',y') grid we use d as a "template", i.e., d (which is represented by a connected set of cells) is translated parallel to itself, the distinguishable translates being representatives of y' coordinate lines. The translation process also generates representatives of x' coordinate lines.

Translation takes one step. The (x',y') system will be set up in data plane 1, while plane 3 stores the putative parabola and intermediate computations. The translational bus structure is constructed as follows. First, within the data plane, like diagonal links are connected "head to tail" to form a Cartesian grid rotated 45°. At the control border, each cell connects each diagonal input link to the output link (into the data plane) of the opposite diagonal type. Recall that each cell was originally given 3 sets of input and output links. One set propagates template data, the other two will be reserved for the x' and y' coordinate busses.

Since d is a straight line, each cell of d has neighbors with which it forms one of the three configurations shown in Figure 3a. These are: nearest neighbors (1) above and to the right (labelled A); (2) to the left and below (B); or, (3) above and below (O). Type (3) corresponds to O's of the straight line code [6], types (1) and (2), in pairs, to the 1's. To form the y' coordinate lines, cells of d signal along the 45° busses as follows. Each cell signals its label. Data cells receiving them connect links (to form busses parallel to d) correspondingly. If 'A' is received, links at the top are connected to those to the right. If 'B', left links are connected to the links at the bottom. If 'O', links are connected top to bottom.

The x'-coordinate busses are set up by the same signals, after reflection by the control borders. This makes them come in on the 135° busses (Figure 3a). If an 'A' is so received, bottom links are connected to those to the right. If 'B', left links are connected to links at the top. If 'O', links are connected left to right (Figure 3b). Putting all links together in this fashion covers the plane with x'-coordinate busses



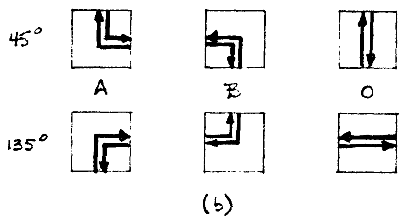


FIGURE 3 - Translation Bussing

in the same way that y'-bus formation covered the plan with representatives of the y'-coordinates previously.

We now copy the "parabola" into data plane 1, and find F. We use the fact that the vertex tangent t is halfway from d to ℓ. The tangent algorithm of step 1 is used; t is parallel to d and thus a y' coordinate line.

We find ℓ as follows. If d, t and ℓ cross a common border, ℓ is easily found by marking off a distance equal to that from d to ℓ (see Figure 4 in which the data and control planes are drawn in one plane). This requires only diagonal and vertical busses as shown.

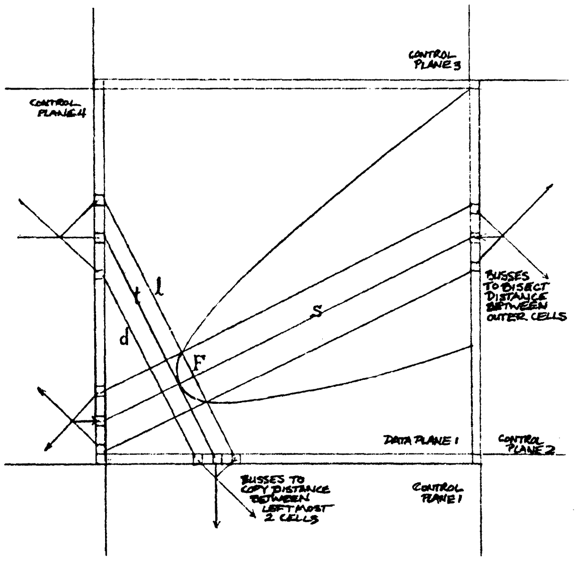

CONTROL PLANE 3
CONTROL PLANE 4
t
l
d
s
F
BUSSES TO BISECT DISTANCE BETWEEN OUTER CELLS
DATA PLANE 1
CONTROL PLANE 2
BUSSES TO COPY DISTANCE BETWEEN LEFTMOST 2 CELLS
CONTROL PLANE 1

FIGURE 4 - Finding Focus F

The following complication can arise. If d hits the control border so that two horizontally connected cells would lie along it (Figure 5a) we must "short out" a column of the control plane beneath one of them (Figure 5b). As this shorting can be done at all control borders when the (x', y') system is constructed, all such problems are handled automatically.

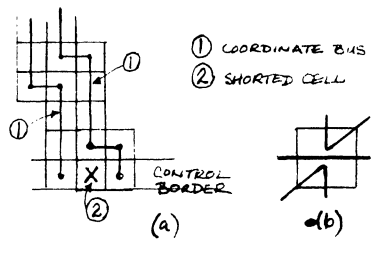


FIGURE 5 - Shorting

If t or ℓ hit the control border perpendicular to that crossed by d, the diagonal busses must effectively "wrap around" the cube faces constituting two adjacent control planes. Figure 6 shows all cases, including the one already discussed. Part (d) illustrates the fact that no special arrangements need be made for these cases either!

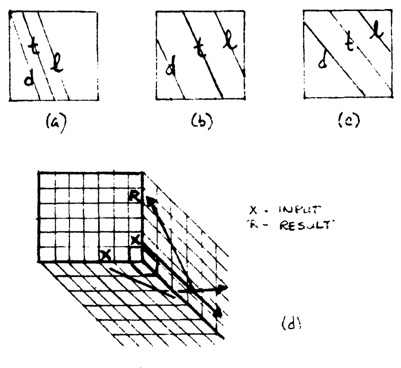


FIGURE 6 - Bus Wrap Around

Finding F is now almost trivial. We merely bisect the segment of ℓ whose endpoints lie on the putative parabola (Figure 4). The x'-line containing F is s and also the x'-axis. The latus rectum is marked on it by this process.

2.3 STEP 3: PLOT THE GRID PARABOLAS -- We must plot adjacent parabolic coordinate lines, between which (or on one of which) the candidate parabola must lie. We compute them in parallel on planes 1 and 2. The data stored in plane 3 is copied into both, and the (x',y') grid, d, etc., also appears in both.

Let the 2 grid parabolas be images under $z=f^{-1}(w)=w^2$ of the lines $v=v_0$ and $v=v_0+1$ in the w-plane. The coordinates of their vertices in the (x',y')-plane are $(-v_o^2,o)$ and $(-(v_o+1)^2,0)$.

Hence, if x_o is the distance from F to d, it follows that

$$v_o^2 \le x_o < (v_o+1)^2 \tag{9}$$

We use (9) and the fact that

$$1+3+5+ \ldots +(2n-1) = n^2, \quad \text{for } n = 0,1,2, \ldots \tag{10}$$

to find the vertices.

All grid parabolas have their vertices to the left of F along s at distances 1, 1+3, 1+3+5, ..., n^2, ... The laying off of the successive segments 1,3,5, ..., (2n-1), ... can be done immediately (see Appendix II). The vertices of the two parabolas sought are then found immediately by comparison with the candidate. It must be remembered that here distances are measured along (x',y')-coordinate lines, but these show up as the same distances along the control borders ((x,y) coordinates) provided the "shorting" described earlier has been carried out. This permits discussion of "oblique" cases with "rectangular" diagrams!

The next step is to find the points at which successive chords meet on the parabolas (see eqn. (6) for their slopes). Starting from the vertex as $(-v_o^2, 0)$ we lay off constant increments $2v_o$ along the y' axis and increments 1,3, 5, ..., along the x' axis. We get $2v_o$ as 1 more than the length of the largest segment (immediately to the right of the vertex), and marking it off along the y' axis immediately. The procedure is similar to the doubling procedure discussed in connection with Figure 4. The x' increments are found immediately as before (Appendix II). The x' coordinate lines at each y' increment "height" and the y' coordinate lines at each x' increment intersect in a network of cells including those sought. The cells in that network are put in a special state by a signal along the coordinate busses from control cells on them. In that state they connect + directed links of x' coordinate busses to + y' directed links "above" s and to -y' directed links "below" s. At a signal from the vertex the network cells undergo a transition to a state indicating they are on the parabola. It can be seen that the signal from the vertex follows a pair of paths which arrive only at the cells stated. See Figure 7.

By using the line routine of Appendix I on the rectangle bounded by path and successive points we obtain the chord approximation of the parabola with vertex at $(-v_o^2, 0)$.

In the other plane, the "parabola" with vertex at $(-(v_o+1)^2, o)$ is formed similarly.

2.4 STEP 4: RECOGNIZE -- Recognition is simple. The grid "parabolas" and the candidate are loaded into one data plane. All data plane links are then connected except those crossed

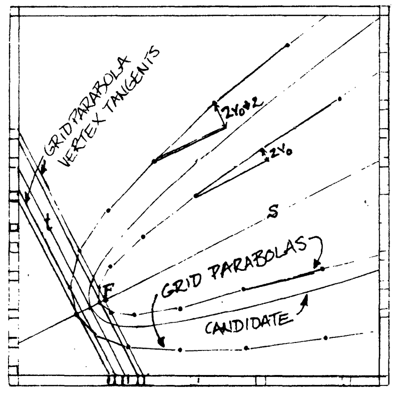


FIGURE 7 - Plot Grid Parabolas

by either of the two grid parabolas. The candidate parabola cells then emit a signal. If any cell of the control border other than those lying between the grid parabolas receives this signal, the candidate is not a parabola, otherwise it is accepted.

IV. CONIC RECOGNITION ALGORITHM

The BA architecture for conic recognition is much simpler than Figure 1. It consists of an n x n data plane surrounded by a control border. Links are as before. The lower left corner cell serves as output and semaphore. Achieving immediacy of part of the algorithm needs a third dimension obtainable by making the square one face of a cube (or square prism).

In IV.1 we present the key subroutine, which given any slope m finds the conjugate slope and corresponding diameter(s). The whole algorithm is sketched in IV.2.

1. CONJUGATE SUBROUTINE

Tagging two data cells specifies the slope m of the line determined by them (Appendix I), and the routine for constructing the (x',y') coordinate system of III2.2 then covers the data plane with parallel lines of slope m. It is easy to check, for any m, whether the candidate is cut by a line in more than two points, using signals from the control border. If the candidate is cut in exactly two points it is easy to find them and the midpoint of the chord they cut off by earlier routines. All the midpoints of all chords of slope m can be found simultaneously and the straight line recognition algorithm of Appendix I applied to them. If it fails the candidate is not a conic. If it succeeds the locus is the conjugate diameter.

2. CONIC RECOGNITION

We must omit detailed discussion of the general algorithm and the many special cases that arise because of lack of space, but this should not impede its comprehension. The procedure is essentially a translation of the geometric terminology of II.2 into BA operations. All talk there of chords, diameters, etc., usually involves little more than the subroutine of 1 above, sometimes more than once, and the algorithms of Appendix I. The other considerations involved often need distinctions to be made of a topological nature, e.g., whether a curve is closed. This is easily done as in [1], and distinguishes ellipses or circles among central conics. Parabolas are distinguished from central conics not only by having this "center" at infinity, and thus never in the data plane, but also in having the "diameter" conjugate to a set of finite parallel chords always intersecting the control border. All the distinctions possible, to the resolution of the BA, can be carried out in analogous fashion, using the appropriate criteria.

V. ERROR CONSIDERATIONS AND CONCLUSIONS

Error analysis for BA algorithms generally and those of this paper specifically are of wider interest than the cases examined, for the general problem faced is no less than that of discrete representation of continuous quantities. Also, errors here go beyond those in numerical analysis, for they apply to patterns (perhaps functionals) in multidimensional spaces rather than to mere numbers. Many considerations like these are implicit or explicit in the topological discussion of [1], and they also apply to the cases at hand.

Conic identification errors arise from limited BA resolving power in many variations. For example, inability to distinguish between an end piece of a long narrow ellipse, a narrow parabola or one sheet of a hyperbola with a small angle between its asymptotes can be purely a question of resolution. Interestingly enough, the "kinks" of oblique straight lines never are a source of error in themselves, i.e., their resolution capabilities are precisely those of horizontal or vertical lines of cells.

The BA resolving power can always be referred back to the number of cells in it. For if we double the size of a "square" BA we can always label the new one with half integer coordinates to give twice the linear resolution of the smaller one.

When we change coordinate systems, e.g., to the confocal parabolic case of II, note that near the singularity many parabolic "squares" fit into one cell and become indistinguishable. Far from the focus, the parabolic cells become very large, containing many cells. The Jacobian of the transformation between systems always measures the local ratio of cells of the two kinds. Error theory with a numerical analysis flavor comes up when these considerations are pursued. The adequacy of chordal and tangent approximations to curves (and specifically the parabola) becomes better and better the farther one gets from the focus (for fixed cell size). It should be possible to develop an approximation theory using straight lines (or planes in the

3D case) which will permit immediacy on a BA for many processes (e.g., curve fitting, integration, etc.). We believe the field of BA error analysis has much promise.

REFERENCES

[1] J. Rothstein, "Topological Pattern Recognition in Parallel and Neural Models on Bus Automata", Proc. 1978 International Conference on Parallel Processing, pp. 95-107.

[2] J. Rothstein, "On the Ultimate Limitations of Parallel Processing", Proc. 1976 International Conference on Parallel Processing, pp. 206-212, Best Paper Award.

[3] J. Rothstein, "Toward an Arithmetic for Parallel Computation", Proc. 1977 International Conference on Parallel Processing, pp. 224-233, Most Original Paper Award.

[4] J. Rothstein, "Transitive Closure, Parallelism, and the Modeling of Skill Acquisition", Proc. 1977 International Conference on Cybernetics and Society, pp. 232-6.

[5] J. M. Moshell and J. Rothstein, "Bus Automata and Immediate Languages", Information and Control, 40, pp. 88-121 (1979).

[6] J. Rothstein and C. Weiman, "Parallel and Sequential Specification of a Context Sensitive Language for Straight Lines on Grids", Computer Graphics and Image Processing, 5, pp. 106-124 (1976).

[7] F. B. Hildebrand, Advanced Calculus for Applications, Prentice-Hall (1962).

[8] I. Todhunter, A Treatise on Plane Co-Ordinate Geometry as Applied to the Straight Line and the Conic Sections, MacMillan & Co., (1858).

APPENDIX I

We show here how to determine a straight line from two points. SEE ERRATUM.

In Figure A1.1(a), all tagged cells are shaded, the two dotted ones being the initially tagged pair. Take the coordinate lines as running through the centers of the cells; four such lines are illustrated going through the two dots, forming a rectangle. The slope of the line is the height of the rectangle divided by its base, here 2/4 or 1/2, in general p/q. We can take $p < q$ with no loss in generality (interchange x and y axes if $p < q$). In effect we have p and q given in stroke notation, and the slope is immediately calculable by the "number theorist" BA of [1]. Essentially the same calculation gives the straight line code [6], from which "tagging" information can be immediately derived and bussed to the appropriate cells. The following is essentially the treatment to appear in John Mellby's dissertation.

The code assigns 0 to a cell which is part of a horizontal run (Figure A1.1(b)), 1 to a pair of cells corresponding to a rise (Figure A1.1(c)). The two 1-cells were labelled A and B in step 2 of the conformal parabola algorithm. Figure A1.2

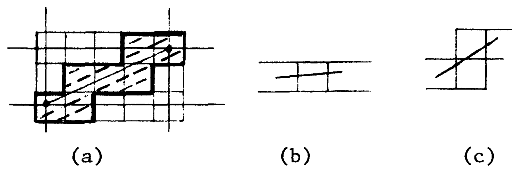


FIGURE A1.1 - Tagged Cells of a Straight Line

indicates how the code is determined from p and q. Marking p and q along an axis as shown is easily done from Figure A1.1 using 45° busses to make q cells along the column of cells containing the right dot.

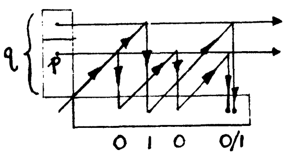


FIGURE A1.2 - Code Determined From p and q

We wish to calculate the code of a line whose slope is p/q. Basically we have to find the cells in which the line will pass vertically to the next cell.

The following observations are elementary. When a line with slope p/q has gone one unit horizontally, it will have gone p/qth of a unit vertically. When it has gone two units horizontally, it will be 2p/q units vertically, and so on. Also when the line has gone one unit vertically, it will have gone q/p units horizontally.

Now note that for each k where kp/q is greater than a unit, and (k-1)p/q is less than that unit, this means that the line has passed a vertical unit. In BA terms, it has crossed vertically into another cell. Also, it has done this in the kth (horizontal) cell. Then the "1" digits of the code are in the k_1, k_2, ... cells where k_i is defined by

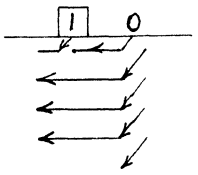
$$k_i \cdot p/q \geq i > (k_i - 1) \cdot p/q.$$

Rothstein and Weiman performed this operation using two shift registers of length p and q. The registers shifted simultaneously; as the p register completed each cycle, it indicated a horizontal move of one cell. (Note that after the first p cycle, the q register had completed p/qth of its cycle. This indicates that the line has gone p/qth of a unit vertically.) Then whenever the q cycle ends indicates a vertical move of one cell.

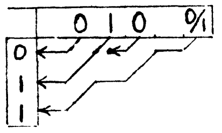
We execute this process immediately using an interval marking algorithm. Note that if a line is marked at an interval of p cells, those cells could indicate the cycle of a p-length shift register. Then to show two registers, we mark the line at intervals of p and q (Figure

A1.2). The end of each p cycle, or the p marking, indicates a horizontal shift, which we will call for now a '0', and the q marking is a vertical shift or a '1'. Whenever we have both p and q markings is a vertical and horizontal shift, and this is the end of one cycle of the code.

To convert this to the line code, note that a '1' in the line code denotes a vertical and horizontal shift, while our '1' is only a vertical shift. Thus we must combine each of our '1's with an '0' after it, so we create a special compression routine which will delete an 0 for each 1 (Figure A1.3(a)). When we compress out the "blanks" (Figure A1.3(b)), it becomes the straight line code for a line of slope p/q. Notice that we can obtain any number of cycles of the code, i.e., generate any length of line we wish.



(a)



(b)

FIGURE A1.3 - Code Compression

Once we have the line code, we can use it to create the line itself on the BA.

Each digit of the code determines the type of transition of the line through that row. However, each separate digit does not in and of itself determine which cells that line runs through. We can create a configuration of busses which will generate the line. In each row all cells create busses connecting them with cells in the adjacent rows, according to that row's code digit. (If the digit is 0 then no rise is indicated so cells in the same row are connected, Figure A1.1(a). If the digit is 1 then cells are connected so that the bus "rises" one cell, Figure A1.1(b)).

With this bus system, the origin cell of the line need only send a signal on these busses to signal the entire line.

The complementary problem to line determination is recognizing whether a pattern of tagged cells actually is a straight line. If we have the pattern representing a line we can determine whether this represents a complete segment of the line's code.

There are two parts to this process. First we can find the endpoints of the pattern and using the line determination routine, find the code of the line which passes through those points. Then we can determine the code of the pattern itself and compare the two. If the codes match, the pattern is a line.

We already know how to determine the code from endpoints. To determine the code of a pattern, each cell in the pattern "examines" its neighbors. Each cell must be a run cell, or one of the two bend cells (Figure A1.4). If the cell has only neighbors to the right and left, it is a run cell, and indicates a '0' in the code. If a cell has a neighbor above it, it is a bend cell and indicates a '1' in the code. If a cell has a neighbor below it, it is a bend cell, but the cell below it will indicate the code.

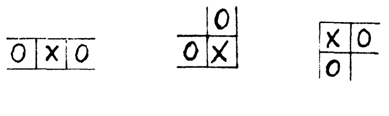


FIGURE A1.4 - Run and Bend Cells (x)

When each cell has determined its own state it sends a signal to the "goal" line showing its state. Then the goal line contains a copy of the code. This process is illustrated in Figure A1.5.

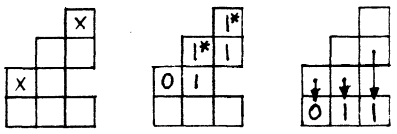


FIGURE A1.5 - Code to Goal Line

APPENDIX II (By John Mellby)

We wish to mark a row of cells at points a_n defined by

$$a_n = 2(\sum_{i=1}^{n} i) - n = n^2.$$

They divide the row into segments of length l_n given by

$$l_n = n^2 - (n-1)^2 = 2n - 1.$$

To accomplish this, we create bus configurations to generate the separate segments, and then concatenate them.

Observe that whenever we send a signal on a diagonal bus, for each cell the signal travels along the diagonal it goes one cell horizontally and vertically. Thus, if the signal travels k cells along the diagonal and we project this upon the marking line we have gone k cells on the marking line (Figure A2.1).

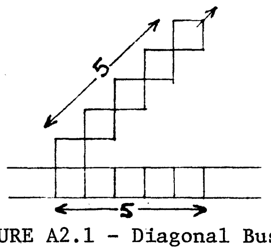


FIGURE A2.1 - Diagonal Busing

The bus configuration to do this will consist of diagonal and vertical busses. The general cell consists of bus connections to travel diagonally and vertically (Figure A2.2(a)). The cells in the ℓ_nth row, receive the signal from the diagonal and transmit it on the vertical (Figure A2.2(b)). Finally, the marking row receives the signal from the vertical bus (Figure A2.2(c)). Then the entire pattern looks like Figure A2.2(d).

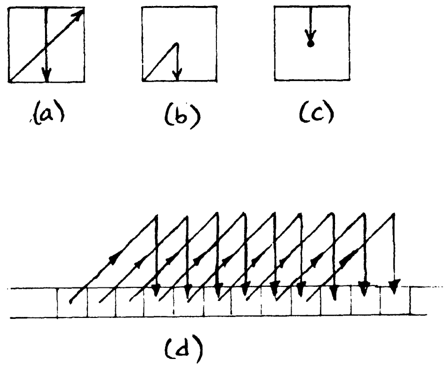


FIGURE A2.2 - Individual Plane Configurations

Each of these segment configurations can be done in a separate plane, then the planes connected (3D BA).

To generate the segments in the correct position on the row, assume the kth segment is in its proper place (i.e., cells $a_{k-1}+1$ through a_k). Then the k+1th segment will start at a_k+1 and will be ℓ_k cells long, ending at a_{k+1}. Since a_1 starts by definition in the proper location we have, "by induction", the segments with the proper length at the proper points along the line.

If we number the plane containing the marking row 1 and each successive plane accordingly, then the ith plane will contain the configuration for marking a segment of length l_i. The beginning of the ith signal comes from the (i-1)th plane, and when the signal ends it will go to the (i+1)th plane. Let the symbol (in the i+1th plane ⊙ indicate a bus coming from the (i-1)th plane and ⊗ indicate a bus going to the (i+1)th plane. Then a cell in the marking line looks like Figure A2.3(a).

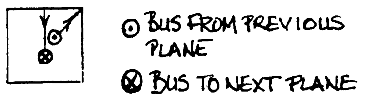


FIGURE A2.3 - Interplane Busses

Now a signal is sent on the first diagonal bus in plane 1. It will mark the a_n cells in the individual planes, then we project these marks onto the marking row in plane 1 and the row is marked as desired.

To create the configurations in each plane, we need only set, in each plane k, the number ℓ_k. Then the configuration of busses in each plane can be constructed based on the number ℓ_k.

To send the numbers (ℓ_i) to the planes, the kth plane receives a signal in the ℓ_kth vertical cell, and sends the signal on in the ℓ_{k+1}th cell. The pattern for this is a steep diagonal (slope 2) in a plane perpendicular to the 1st plane. See Figure A2.4(a). The inidividual cell is as in Figure A2.4(b).

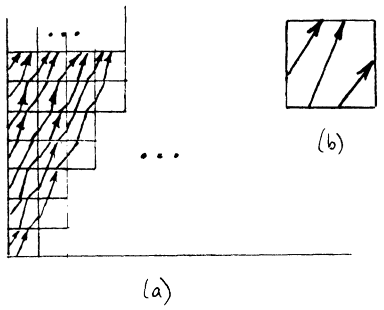


FIGURE A2.4 - Steep Diagonal

In summary, this routine constructs the busses of Figure 2.7 signals ℓ_i in each plane, constructs the individual plane configurations (Figure A2.2), sends signals through these busses and projects the markings onto the marking row. Each of these operations is immediate so that whole routine in immediate.

Note the very close similarity in structure and operation of this BA to the "number theorist" BA described in [1] p. 102, Figure 3.3.

ERRATUM

The first paragraph of Appendix I should read as follows:

We show here how to "tag" the cells along any straight line given two tagged cells on it, i.e., how to determine a straight line from two points on it. Later we also show how to recognize whether tagged cells are a straight configuration.

AN ARCHITECTURE FOR PARALLEL PROCESSING OF "SPARSE" DATA STREAMS

Tom Trilling
Technology Service Corporation
Santa Monica, California 90403

Abstract -- This paper describes a type of special purpose architecture that has been designed for a class of problems involving parallel data streams that contain significant information only at occasional random unpredictable intervals. (Such data streams are termed "sparse," analogous to a sparse matrix.) In these problems, it is important for the processor to be able to sense and analyze these occasional intervals that can be called "bursts of activity." The important feature of the architecture is that a relatively small number of processors are shared among a large number of data streams. This sharing results in a dramatic saving -- sometimes orders of magnitude -- in the number of processors required as compared with conventional architectures in which a processor is dedicated to every data stream.

To illustrate this architecture a configuration designed for a problem in the area of radar target detection is discussed in some detail. A computer simulation of the performance of this architecture has been made, and, as will be shown, the results match the predicted savings. (This paper is abstracted and adapted from my Ph.D. dissertation [1].)

"Sparse" Data Streams

Before describing the processor-sharing architecture, (this term will be used for brevity) it is useful to discuss "sparse" data streams and the type of problems for which the architecture was developed. The term "sparse" need not be defined, but it seems reasonable to apply it to data streams containing significant data less than say five percent of the time. Examples from two quite different areas, medical care and radar target detection, will illustrate this class of problems and the need for this architecture.

Medical Data Example

Many examples exist in the field of medical data, but one that seems especially interesting is the output of electrocardiograms, ECGs, which are used to monitor heart function. An ECG measures electric potentials between various parts of the heart. It is one of the most effective methods of detecting dangerous heart conditions. A normal ECG trace is shown in Figure 1(a) [2]. Figure 1(b) gives a detailed view of a single beat pattern, which contains three segments, known as: the P wave, the QRS complex, and the T wave [2]. Analysis of deviations from this normal pattern is used in the diagnosis of various heart disorders.

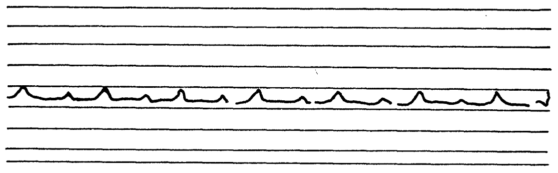


Figure 1(a). The Normal ECG

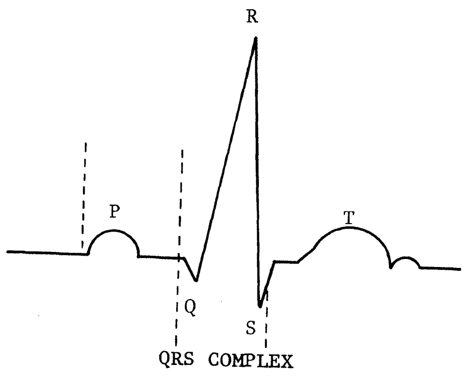


Figure 1(b). Detailed View of a Single Beat

One important type of abnormal pattern is called a PVC, or premature ventricular contraction as illustrated in Figure 1(c) [2]. PVCs are of interest because they are possible indicators of a dangerous condition, but can also occur in people with perfectly sound normal hearts. The critical factor is the frequency, timing, and shape of the PVCs. Automatic processing is capable of monitoring ECGs and issuing an alarm when dangerous patterns occur.

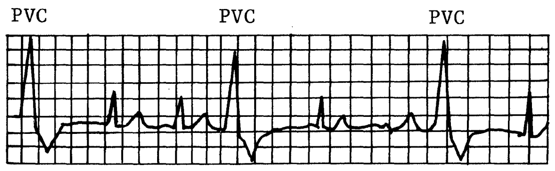


Figure 1(c). Trace with Frequent Premature Ventricular Contractions

Patients in the hospital for reasons other than heart problems do not normally have ECGs monitored, and occasionally serious heart problems develop which are not detected in time, resulting in damage to the heart or even death. Unfortunately, it would be prohibitively expensive to monitor every patient continuously with an ECG, either by the conventional paper recorder, or by dedicating a sophisticated processor to him. However, a data processing system taking advantage of the "sparse" nature of the data stream could monitor many patients with relatively few processors.

Radar Target Detection Example

The following example from radar target detection is similar to problems in a number of related areas such as communication networks, radioastronomy and sonar systems.

A radar system locates targets in space by emitting energy and sensing the reflections of that energy from the targets. The energy reflected by the targets and returned to the radar is usually at a very low power level, often of the same order of magnitude as the thermal noise that is always present. This means that it is often difficult to distinguish between targets and noise. The method generally used is to correlate the results of a number of "looks" at the target, using statistical decision methods to achieve the best possible discrimination between targets and noise.

A two-dimensional scanning radar is illustrated in Figure 2. In this figure, the angular sections, numbered 1, 2, . . ., N, represent the central portions of successive beams transmitted in sequence as the antenna rotates in a clockwise direction. The radial dimension represents range. The range to the target is found by measuring the time interval T_1, between the pulse transmission and the arrival of the return. This time interval, multiplied by the speed of the electromagnetic propagation, c, gives the round-trip distance from the radar to the target.

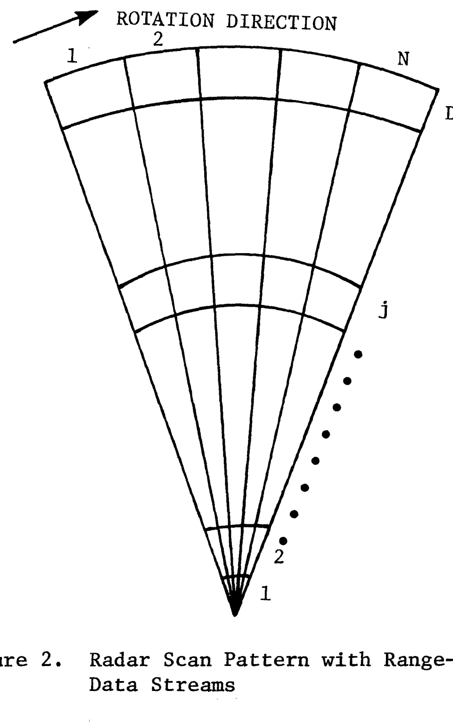


Figure 2. Radar Scan Pattern with Range-Azimuth Data Streams

In the sensing of return energy, the radar receiver acts as an "integrate and dump" circuit over a time equal to the transmitted pulse width, τ. This process imposes a granularity on the range measurement of $c\tau/2$, which has the effect of dividing the surveillance range into what are called range bins. In Figure 2, the range bins are numbered from 1 . . . j . . . D, where D represents the maximum range of the surveillance volume. The number of observations or "looks," N, is dependent on the azimuth width of the transmitted beam. The azimuth data inputs in each of the D range bins are excellent examples of "sparse" parallel data streams because usually only a few if any targets are among the many range bins in space.

There are many statistical decision methods used. A common method is to set an amplitude threshold and to call an input signal exceeding the threshold a "hit" and to call an input below the threshold a "miss." A detection decision is then based on getting a specified density of hits.

The factor common to these examples is that the data streams are "sparse" but that the unpredictable bursts of activity are of great importance and must be sensed and carefully analyzed. However, it would be inefficient to dedicate a processor to each data stream just for those intervals. It would be much more cost effective to share processors among the data streams. This is the motivation for the architecture to be described in the next section.

Processor Sharing Architecture

The processor sharing architecture is shown in a generalized form on Figure 3. This basic design would be modified and adapted for specific applications such as on the specific example that follows.

Basic Design

The major feature of this architecture is that there is a bank of processors that can be assigned to data streams as desired. Only the relatively few data streams having a burst of activity are assigned to a processor. The great majority of data streams are not assigned to processors, and their inputs are not used until there is an indication of a burst of activity. The functional areas that accomplish these objectives are: the input bus, the input sequencing control unit, the ψ evaluator, and the evaluation data routing logic. To understand the operation of this architecture, it is necessary to know both the function of each unit, and the major pathways of information flow.

All data streams feed the input bus. Generally the current input from one of the data streams is selected from the bus according to a signal from the input sequencing control unit. However, in some cases, a multiport bus might be used. The input sequencing and selection, and

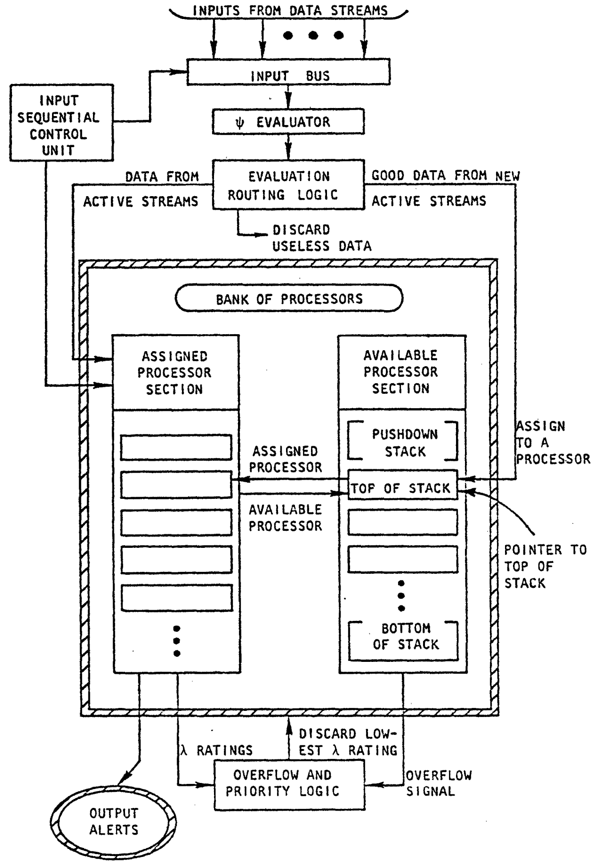


Figure 3. General Architecture for Processing of "Sparse" Data Streams

the resultant sampling rate is very much dependent on the problem.

The ψ evaluator makes a preliminary evaluation of each input from each data stream. Data that appears significant, such as PVC or an input exceeding the threshold, would cause the data stream to be assigned to a processor. As a simplification, it will be assumed that results of the ψ evaluation is one of two values, either +K, data appears significant; or -1, data not significant.

The evaluation routing logic uses the results of the ψ evaluator and the status of the data stream to decide what to do with the input data. There are three possible courses of action: 1) if the input is from a data stream previously designated as active, then a processor would already have been assigned to that data stream. In this case, the ψ evaluation result, whether +K or -1, is sent to the assigned processor, 2) if the input is from a data stream not previously designated as active, but the ψ evaluation is +K, then the data stream becomes designated as active. The top processor from the stack of available processors will be assigned to this data stream, and the evaluation result will be stored by the newly assigned processor. As will be described, this assigned processor now becomes part of the assigned section of the processor bank. In the rare case that there is no processor available for the new active data stream, an overflow signal is generated, activating priority logic. This logic determines which of the data is of least value and should be dropped. (The λ parameter, described shortly may be useful for this decision), and 3) if the input is from a data stream not previously designated as active, which is most often the case for "sparse" data streams, and the ψ evaluation was -1, then the input is simply dropped. (Thus the ψ evaluator screens most of the data inputs, and keeps them from ever being a load on the processor.)

The architecture does not require any particular type of processor. However, it is efficient to have the processor compute a significance parameter, λ. This parameter is used in the data processing decision, and also serves to rank data streams for priority in overflow situations. One such processor, the sequential observer, will be described in the example that follows.

A processor will consider a new input in conjunction with the results of the evaluations of previous inputs from the data stream. The processor has three possible options. First, it can make a decision that some status of interest exists, such as a dangerous heart condition, or a radar target, and issue an alert. Second, it can decide that the data stream is no longer active and drop the data stream. (In this case, the processor becomes unassigned, or available, and is switched to the available processor section as will be described.) Third, if the processor does not have enough information to make a decision with adequate confidence, then it will wait for more data, remaining assigned to the data stream.

As indicated, the bank of processors is divided into two sections. The section indicated on the left contains processors that have been assigned to active data streams. Once assigned to a data stream, a processor will normally remain dedicated to that stream until it either recognizes some status of interest, and issues an alert, or else makes a negative decision, and drops the data stream. When one of these decisions is made, then the processor is transferred to the available processor section. This section, shown on the right contains unassigned, or available processors. These processors are organized in a pushdown stack so that they are available as needed [3]. That is the "top" processor is taken first, and its removal exposes a new "top."

One of the major design problems of the processor sharing architecture is the method of recognizing whether or not a data stream has been assigned to a processor. If the answer is "yes," then it is necessary to be able to locate the assigned processor as promptly as possible, in order to properly route the new data.

The assigned processors will be organized to facilitate both the recognition of assigned data streams, and the location of the assigned processors. In a few problems, the inputs from the data streams might occur at unpredictable intervals and in random sequences. For this case, a content addressable method could be of great benefit [4]. Another possibility would be to order the processors by some parameter so that a binary search could be used. However, rearranging of the processors would be necessary to maintain this order whenever data streams changed from inactive and vice versa. In some cases it might be simplest to just put all the active processors in a block of consecutive locations. This would minimize the search area.

For most applications, all the data streams are sampled at the same rate, in a repetitive sequence. For such applications, a double linked circular list organization of the processors would be the logical choice, because it permits immediate recognition of assigned data streams and gives the location of the assigned processor without any searching [3]. The method by which this is accomplished is better explained with the illustrative example given in the next section.

Specialized Configuration for Radar Target Detection

Figure 4 gives an example of a specific configuration of the processor sharing architecture, one designed for radar target detection. This configuration follows the basic design of Figure 3, but the general functional areas have been adapted to the problem. In this problem the data streams are the consecutive inputs corresponding to the same range bin. However, the receiver data comes in on a single input line. This single input line is thus a very rapid repetitive sequence of inputs from each of the range-azimuth data streams. Each input sample has a duration equal to the effective pulse width of the radar, which is typically of the order of 1 μsec. The ψ evaluation is simply a threshold crossing detector. An input is significant if it exceeds the threshold (i.e., a hit), otherwise, it is not.

A major problem in the design of this configuration is the matching of data stream inputs with their assigned processors. That is, when a hit is received, it is necessary to quickly determine whether this input is from a currently active data stream already assigned to a processor, or if it is the first hit from a previously inactive data stream. The data rate of the single input stream that combines all the data streams is too high for a memory search method to be used. However, there is a method of organizing the processor memory that makes use of the inherent ordering of the data streams to match data streams with their assigned processors without a memory search.

Since the data streams represent consecutive range bins, the inputs are always in order of increasing range. Thus, the range can be used as an identification number for the data streams, and also indicates the input sequencing order. The processor memory is organized to take advantage of this ordering by having all processors stored in order of increasing range of the assigned data

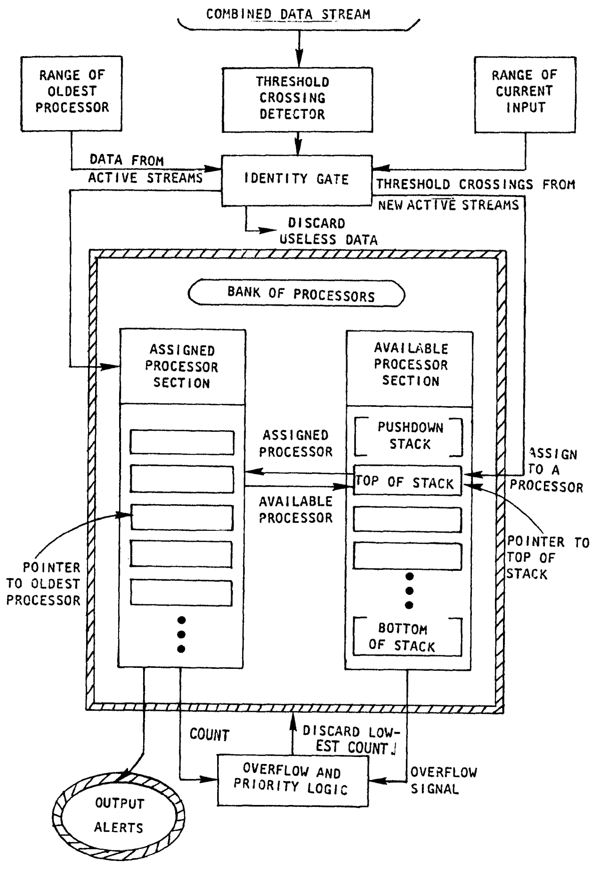


Figure 4. Configuration for Radar Data Streams

streams. A double linked circular list structure is very effective for this purpose. In this structure, all elements contain pointers to the preceding and following elements. Thus, every element contains sufficient information to make an addition or deletion in the list, without having to search the string. This list modification capability is essential in this configuration, because the processors must be kept in range order.

With the processors in range order, the problem of association of new inputs with assigned streams becomes quite simple. The method used is to designate the "oldest" processor as the one longest without an input. The range of this processor is stored in a register as indicated. There is also a pointer that always indicates the location of the "oldest" processor. On each new input, the range of that input is compared with the range of the oldest processor. If the range of the current input is less than that of the "oldest" processor, then data stream corresponding to, the current input has not been assigned to the oldest processor. Furthermore, the data stream has not been assigned to <u>any</u> processor. If the data stream had been assigned to a processor then that processor would have been ahead of the present "oldest" processor and would be the oldest processor now. This association method is a key element in this configuration, simplifying the design and speeding up the processing. Thus, if the current input is a miss, it is discarded because the data stream is not active. If the current input is a hit, the data stream will be assigned to a processor which will be inserted into the list ahead of the "oldest" processor, since its range is less. Of course, when the current range is identical to that of the "oldest" processor, the new data, whether hit or miss, is routed to this "oldest" processor. At this point the next processor in the list becomes the "oldest" processor, and this sequence repeats. This description has been somewhat brief; a detailed explanation is given in Reference [1].

The processors use a digitized form of a statistical decision method called the sequential observer [5]. In essence, this device is an accumulator and a set of decision logic. The accumulator is set to an initial value of zero. On a hit, the accumulator is incremented by an amount K. On a miss, the accumulator is decremented by one. If the accumulator reaches some threshold, T, then a detection decision is made. Should a processor count down to zero, the data stream is considered inactive for the time being and the processor is returned to the available stack. A state diagram for a sequential observer with K = 3 and T = 7 is shown on Figure 5. In this diagram, p is the probability that an input is significant, such as a hit and q is the probability of a miss. If the threshold state, state 7, is reached then a decision is made that the status of interest, such as a target, exists. This type of diagram is useful both for analysis of probability of detecting a condition and for analysis of probability of a false decision due to random input errors such as "noise." Not only is the sequential observer a convenient detection device, but the accumulator value gives an indication of the activity of a data stream (i.e., a λ rating) and is thus useful in the processor sharing architecture in assigning priority for the limited number of processors. That is, if the bottom of the stack of available processor is ever reached, then there is an overflow of active data streams and some data must be sacrificed; and the data stream with the lowest count would be dropped.

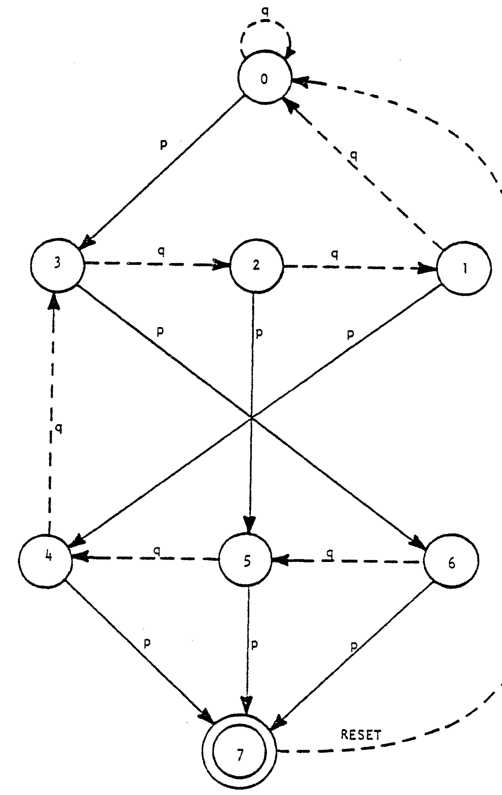


Figure 5. State Diagram for Sequential Observer, with K = 3 and T = 7

Let D be the number of data streams, and P_B be the probability that a data stream is active. Then μ_B and σ_B are given by:

$$\mu_B = DP_B \quad . \tag{1}$$

$$\sigma_B = \sqrt{DP_B\ (1 - P_B)} \quad . \tag{2}$$

If P_B is small, then Equation 2 can be approximated by:

$$\sigma_B = \sqrt{DP_B} \quad . \tag{3}$$

Clearly the number of processors must exceed the mean, μ_B, by some safety factor. If this safety factor is expressed as a probability that a processor will be available when required, then the following equation is obtained:

$$N_p = \mu_B + K_C\ \sigma_B, \tag{4}$$

where N_p is the number of processors required and K_c is the confidence constant expressed in units of the standard deviation.

Performance of the Processor Sharing Architecture

The use of the processor sharing architecture entails acceptance of some sacrifice in recognition performance on the unusual occasions that pseudo-active data streams tie up all the processors. When this condition occurs, active data streams cannot be assigned to a processor.

However, with proper selection of the number of processors, according to the system parameters and requirements, the loss of performance can be reduced to minimal levels, while still achieving a great savings in the number of processors required. For this analysis, it will be assumed that the number of active data streams has a Gaussian distribution, with a mean of μ_B, and a standard deviation of σ_B.

Thus

$K_C = 1$ corresponds to a processor availability probability of 0.84

$K_C = 2$ corresponds to a processor availability probability of 0.977

and

$K_C = 3$ corresponds to a processor availability probability of 0.9987.

The measure of the effectiveness of the processor-sharing architecture is the ratio of the number of processors required to the number of data streams. This parameter, R_p, is given by:

$$R_p = \frac{N_p}{D} \quad . \qquad (5)$$

By substituting Equations 1, 3, and 4 into Equation 5, the following equation for R_p is obtained:

$$R_p = P_B + \frac{K_C \sqrt{P_B}}{\sqrt{D}} \quad . \qquad (6)$$

In Equation 6, the P_B term is a function only of the "sparseness" of the data stream, and is thus not under the control of the digital system. Thus, P_B determines the lower limit on the percentage of processors required. The $K_C \sqrt{P_B/D}$ term is the safety factor, or confidence level, and can be varied according to the system requirements. It will be noted that this term gets smaller in proportion to P_B as D gets larger. Thus, a very high processor availability, such as 0.9987, can be attained with only a modest percentage increase in processors over the minimum level determined by the value of P_B.

The parameter P_B is thus, the critical parameter in the savings achievable by the processor-sharing architecture. This parameter is always "small" for "sparse" data streams, but the meaning of "small" depends on the type of problem. For radar target detection, P_B would usually be less than 0.01, often very much less. For other applications, P_B might be as high as 0.05.

Some examples of the savings achievable by the processor-sharing architecture for $P_B = 0.01$, and $P_B = 0.05$ are given in Tables 1 and 2. In these tables, the first column gives D, the number of data streams. The second and third columns gives μ_B and σ_B respectively. The next three columns give the number of processors required for availability probabilities of 0.84, 0.977, and 0.9987 respectively. The next column gives the ratio, R_p, (i.e., N_P/D) for an availability probability of 0.9987. Of course R_p does approach R_B as D gets large because the data was computed from Equation 6. Additional support for this result is given in Reference [1] based on both analysis and computer simulation. The last column shows the percentage savings achieved by the processor-sharing architecture, for the condition of 0.9987 availability probability, as compared with a conventional architecture using one processor per data stream. It is seen that for $P_B = 0.01$, a savings of 96 to 98.7 percent is achieved. Even for the relatively high $P_B = 0.05$, a savings of 89 to 94.3 percent is still achieved.

Computer Simulation

A detailed computer simulation was run to test the theoretical predictions of savings in processors. The radar example was used because sophisticated mathematic models exist for most aspects of radar detection. In the simulation, the detection performance of the processor sharing architecture with a limited number of processors was compared with that of a system using a processor for every data stream. This reference system used M out of N detection processors which are easily analyzed mathematically, and are approximately equivalent in performance to the sequential observer [1].

A great many cases were run. The volume of data obtained and the complexity of the program is too great to permit more than a brief summary in this paper. More data and a detailed description of the computer program are given in Reference [1]. In the example below, as in all of the cases run, the results were quite constant with the theoretical predictions.

The basic approach of the computer program was to compute the signal strength required for detection by the processor sharing architecture under the specified conditions, and to also compute the signal strength required by the comparable conventional approach with a processor for every data stream. The additional signal strength, if any, required by the processor sharing configuration was used as the measure of loss incurred by the processor sharing. Signal strength was specified as a signal-to-noise ratio (S/N) and measured in decibels (dB). The major input parameters were: the number of data streams, the number of processors, the required probability of detection (P_D), and the acceptable level of false alarms (P_{FA}).

The program first determined the appropriate parameters for the sequential observer. It then computed the stationary probabilities for each state under noise alone, that is the average percentage of the time the machine would be in a given state. (An analytic method for this computation is derived in Reference [1].) These probabilities were used to compute how often there was a processor available when needed by a data stream becoming active. An iterative method was then used to find the S/N required to achieve the desired P_D, with the acceptable P_{FA}. Typically 10-15 iterations reduced the error to an extremely small level. The computation of the required S/N for the reference M out of N detector also used an iterative method. Basically it involved finding the required probability for a single trial that corresponds to a

TABLE 1. PROCESSOR REQUIREMENTS FOR CASE WITH $P_B = 0.01$

D Number of Data Streams	μ_B Mean	σ_B Standard Deviation	N_P Number of Processors			R_p Percentage of Processors for 0.9987 Case	Percentage Saving
			Availability Probability				
			0.84	0.977	0.9987		
100	1	1	2	3	4	4.0	96.0
500	5	2	7	9	12	2.4	97.6
1,000	10	3	13	16	20	2.0	98.0
5,000	50	7	57	64	71	1.4	98.6
10,000	100	10	110	120	130	1.3	98.7

TABLE 2. PROCESSOR REQUIREMENTS FOR CASE WITH $P_B = 0.05$

D Number of Data Streams	μ_B Mean	σ_B Standard Deviation	N_P Number of Processors			R_p Percentage of Processors for 0.9987 Case	Percentage Saving
			Availability Probability				
			0.84	0.977	0.9987		
100	5	2	7	9	11	11.0	89.0
500	25	5	30	35	40	8.0	92.0
1,000	50	7	57	64	71	7.0	93.0
5,000	250	16	266	282	298	6.0	94.0
10,000	500	22	522	544	566	5.7	94.3

desired total probability of the cumulative binomial distribution. That is find p such that:

$$\sum_{j=M}^{M} p^j (1 - p)^{N-j} \frac{N!}{j!(N-j)!} = P_D \quad .$$

Table 3 summarizes the results for a number of runs with various numbers of processors and P_D's, for conditions of 1000 data streams and $P_{FA} = P_{FA} = 10^{-8}$. For these conditions N_P is found to be 10 and σ_B is 3. Thus, for a confidence level of two σ_B, 16 processors would be required. It is seen that the losses decrease with the number of processors, and that with 16 processors the loss does become minimal, agreeing with the prediction.

TABLE 3. S/N LOSS (dB) FOR VARIOUS P_D AND NUMBER OF PROCESSORS. 10 LOOKS, 1000 BINS, $P_{FA} = 10^{-8}$

P_D	Processors								
	8	9	10	11	12	13	14	15	16
.50	1.6	1.0	.7	.5	.3	.2	.1	.1	0
.55	2.1	1.0	.6	.4	.3	.2	.1	.1	0
.60	—	1.2	.8	.5	.4	.2	.1	.1	0
.65	—	1.6	.8	.6	.3	.2	.1	.1	0
.70	—	—	.9	.5	.4	.2	.1	0	0
.75	—	—	1.2	.6	.3	.2	.1	0	0
.80	—	—	—	.9	.4	.3	.2	.1	.1
.85	—	—	—	1.7	.7	.3	.2	.1	.1
.90	—	—	—	—	1.1	.5	.3	.1	0
.95	—	—	—	—	—	—	.5	.2	.1

(— indicates that the P_D cannot be attained.)

(Note, a 0.1 dB loss corresponds to requiring a signal 1.023 times as strong as the reference, a negligible difference in almost all cases.) This example thus corresponds to a savings in processors of 98.4 percent, over the 1000 processors required by a conventional architecture.

Summary of Conclusions

An architecture has been presented for a class of problems involving parallel "sparse" data streams. This architecture achieves a dramatic reduction in the number of processors required, as compared with conventional architectures. Analysis indicated that savings of over 90 percent of the processors would result in typical cases. These predictions were confirmed by a computer simulation for a representative set of problems.

These savings are important not only for the direct savings in hardware cost, but also because they result in an increase in system reliability and a reduction in size, weight, and power consumption. For space and airborne systems these latter factors would be far more important than just a savings of money. It is therefore, recommended that the processor-sharing architecture be considered for as many parallel processing applications as possible.

References

[1] T. Trilling, An Architecture for Parallel Processing of "Sparse" Data Streams, University of Southern California, August 1979.

[2] R. Menard, Introduction to Arrhythmia Recognition, California Heart Association, San Francisco, California, 1968, pp. 4-16.

[3] D. Knuth, Fundamental Algorithms, vol. 1, Addison-Wesley, Reading, Massachusetts, 1975, pp. 270-278.

[4] C. Foster, Content Addressable Parallel Processors, Van Nostrand Reinhold Company, New York, 1976.

[5] I. S. Reed, An Analysis of Sequential Detection by the Sequential Observer, (originally Lincoln Laboratories Report No. 20, 1953, republished as U. S. Document AD-17082).

A MULTIPROCESSOR FOR CONTINUOUS SYSTEM SIMULATION

E. Pearse O'Grady
Department of Electrical and Computer Engineering
Arizona State University
Tempe, Arizona 85281

Summary

A simulation-oriented multiprocessor computer system employing a new concept for interprocessor communication is described. The communication scheme takes advantage of the iterative computing requirements of simulation and related applications and employs parallelism in conjunction with address-mapping memories to realize an efficient, high-speed interprocessor transfer mechanism. Multiprocessor operation consists of a sequence of compute and data-exchange phases. A compute phase is a relatively long period during which all processing elements compute without requiring interprocessor data transfers. A data-exchange phase is a period during which all processing elements are halted and interprocessor transfers are carried out according to a preordained plan.

The multiprocessor includes a host processor, a system control processor with user console, an N x N array of processing elements interconnected through N horizontal buses and N vertical buses, 2 x N bus control processors, and an input/output processor. The host processor is a multiprogrammed general-purpose computer which provides program development and utility functions such as editing, file management, and language translation. The system control processor is a dedicated general-purpose computer which acts as a run-time executive for the simulation system. Its functions are to control problem set-up, execution, and termination and provide an interface between the user and the system during execution of a simulation study.

A processing element is a high-speed bit slice microprocessor featuring simulation-oriented and multiprocessor-oriented operations. The main memory consists of a small high-speed segment, called the transfer memory, and a larger segment called the local memory. The processing element's program is stored in local memory along with local problem variables. The transfer memory holds variables which are involved in interprocessor data transfers. During interprocessor data transfer operations, the processing element is halted and the transfer memory is connected to either the horizontal bus or the vertical bus associated with the processor.

A bus control processor consists of an address sequencer (microprogram-control-unit chip) and a memory which is loaded by the system control processor. The bus control processor fetches bus commands from its memory and either executes them internally or issues them to all processing elements on its bus. There are three types of bus commands specifying either an interprocessor data transfer, a multiprocessor-oriented operation, or an internal bus-control-processor operation.

Processing elements in the N x N array exchange data through N horizontal buses and N vertical buses. A processing element interfaces with the horizontal bus and the vertical bus associated with its row and column through its transfer memory and a memory switch. The memory switch functions as a bus-command decoder and as a three-way switch which connects the transfer memory to the processing element or to one of the two buses. When connected to a bus the transfer memory participates in interprocessor transfers with other processing elements on that bus.

A bus control processor executes an interprocessor transfer by issuing a bus command which specifies a source processor, up to (N-1) destination processors, and one address. A mapping memory in each memory switch translates the address to independent addresses in the source processor and each destination processor, allowing efficient use of limited transfer memory. Transferring a single data item between arbitrary source and destination processors requires at most a horizontal-bus transfer from from the source processor's transfer memory to the transfer memory at the intersection of the source row and the destination column followed by a vertical-bus transfer from the transfer memory at the intersection to the destination processor's transfer memory. During a data-exchange phase, interprocessor data transfers distribute data values to all processors which need them. First the N horizontal buses operate in parallel to perform all horizontal transfers. Then the N vertical buses operate in parallel to perform all vertical transfers. In some problems this scheme can improve effective interprocessor transfer rates by a factor as great as N when compared to a broadcast scheme [1].

References

[1] E.P. O'Grady, "Interprocessor Communication In Multiprocessor Simulation Systems," Proc. Fall 1979 COMPCON, IEEE, (September 1979).

A MODULAR MULTI-MICROPROCESSOR ORIENTED FOR REAL-TIME CONTROLS

M.Coppo and A.Giordana
Istituto di Scienza dell' Informazione, Università di Torino
C.M.D'Azeglio 42, IOIOO Torino, Italy

Summary

Modularity and reconfigurability, besides the use of standard well known components, make the system descrbed in this paper suitable for many applications, expecially those of real-time control [1] . It was designed in particular, as a development system and in-field prototype for automotive engine control. The system has been implemented at a FIAT Research Center (CRF)

Fig.1 gives a sketch of the system which contains a set (form 1 to 6) of (physical and logical) processor modules and a common memory (CM) module,connected together by a common-bus (similar to Intel's MULTIBUS) shared following a rotating priority strategy with hand-shake protocol.

All processor modules are equal except one, which is privileged and will be called the Supervisor module. Each processor module is organized around an internal bus and is composed of the CPU (Intel 8085 in our implementation), two memory segments defined as "exclusive" and "inclusive", and two I/O sets defined in the same way. Inclusive memories (I/Os) have addresses form the byte 8N K to 8(N+1) K for module N (for I/Os, from 32N to 32(N+1), and are addressable (in a fully transparent way, for both memory and I/O operations) by the respective CPU and by the Supervisor (module-1) with the same address set. A segment of 8 K-bytes (32 address for I/O) with addresses from 56 K to 64 K(224 to 255) is the common memory (common I/O) of the system. Exclusive memory (I/O) can have in each module any address not assigned to the common memory (I/O) or to inclusive memory (I/O) of the same module. In our implementation, exclusive memories (I/Os) have addresses from 0 to 8 K (0 to 31) for each module, and 0 to 16 K (0 to 63) for the Supervisor . Inclusive memories plus common memory can be extended if the modules are less than 6; but they are limited to constitute, in their maximal size, a continuous set from 16 K to 64 K. Inclusive I/O facilities can be very useful for fault detection and recovery, while common ones can allow all CPUs to share common resources like bulk memory or arithmetic units. Inclusive memories of all modules can be accessed by the Supervisor which appears as though it were a DMA device. The arbitration logic has been designed to avoid any possible dead-lock condition.

All modules are provided with an addressable switch which, acting on the bus arbitration logic, allows them to hold common memory in an exclusive way. This allows the realization of "Critical Regions" The Supervisor is also provided with a set of addressable switches which allow it to stop any other CPU (possibly all together) and use any inclusive memory segment (or I/O) as if it were the only processor in the system.

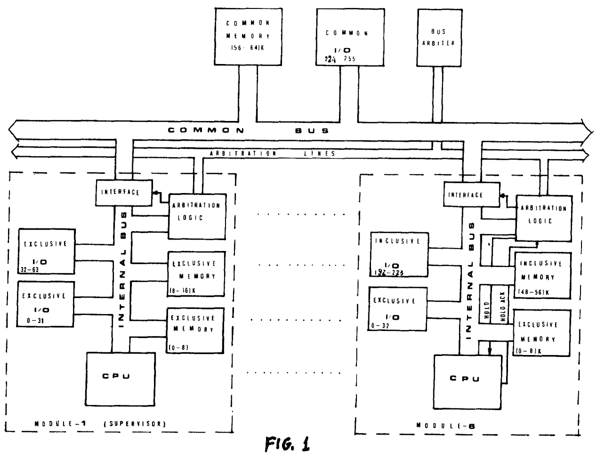


FIG. 1

The system built at CRF has 3 processor modules (including the Supervisor) and a common memory module of 8 K. A measure of this multiprocessor performance is shown in fig.2. Let the common-bus <u>access rate</u> of a program be the ratio between the cycles performed in external memory and the total number of cycles. In fig.2, $\tau_0, \tau, \tau_m, \tau_e$ are the execution times of an (arbitrary) program P, without syncronization requirements executed by processor M1, measured in the following conditions:

τ : external access of P in CM; M2,M3 are idle.

τ_m: as before, but M2 and M3 execute programs with the same access rate as P but relatively randomized

τ_e: as before, but external accesses of P are in inclusive memory of M2(M1 is the Supervisor).

τ_0 is the execution time of P with no accesses via the common-bus. Note that, owing to efficient bus sharing, τ_m/τ_0 is lower than 3 for 100% access rate.

References

[1] M.Coppo,A.Giordana, Multiprocessor Architecture for Microcomputer Controllers, Proc. Int. Conf. on Mini-Microcomputer, Montreal(1979).

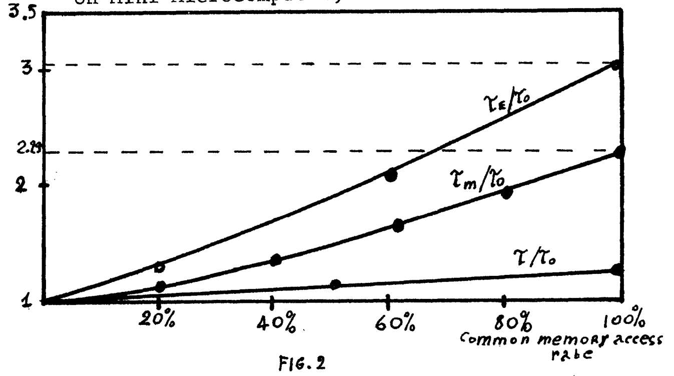


FIG. 2

SOLVING BANDED TRIANGULAR SYSTEMS ON PIPELINED MACHINES[*]

Daniel D. Gajski
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract -- A new algorithm for solution of banded triangular systems on pipelined machines is described. The model of a pipelined machine consists of two functional units, a multiplication pipe and an addition pipe, chained together. Each of the functional units is pipelined in s stages. The time bounds, speedup, efficiency, and utilization for such a model are developed and compared to those of a parallel machine. Furthermore, the performance of the given algorithm is compared to the common "row sweep" algorithm for pipelined machines.

1. Introduction

Banded triangular systems or linear recurrence systems originate from simple DO loops like the following one given in FORTRAN:

```
  DO  10  I = 1, 150
        X(I) = A(I) + B(I) X(I-1) + C(I) X(I-3)
10 CONTINUE
```

There are only a few published methods on how to solve this kind of system on either high-speed supercomputers or smaller array machines intended for the signal-processing market. Although, all of these machines are designed to execute the vector inner product very efficiently, it seems that little attention has been paid to incorporate linear recurrence systems into design.

In this paper an algorithm for the fast solution of linear recurrence systems is presented. The same algorithm in two different implementations can be used on either parallel or pipelined computers.

Parallel linear recurrence solvers have been discussed previously [1-4, 6-8]. The design of special-purpose hardware to solve recurrence problems was discussed in [5]. This paper considers the converse of this problem: Given an ordinary machine with pipelined arithmetic units, what is the speedup and pipe utilization obtained in solving linear recurrence systems?

In Section 2, the definitions and notations used throughout this paper are presented. Instead of using the standard linear-algebra approach, a definition of linear recurrences that facilitates a geometric representation of the algorithm and simplifies the derivation of time bounds is used. Then the Main Theorem (Theorem 1), which forms the basis of our algorithm, is proved.

A model of a pipelined computer is described in Section 3. The model (Fig. 1) is made simple enough to cover a wide spectrum of existing

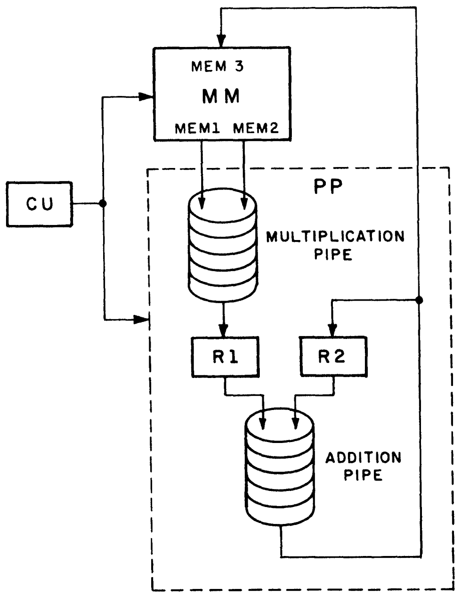


Fig. 1. A model of a pipelined computer

[*]This work was supported in part by the National Science Foundation under Grant No. US NSF MCS77-27910 and MCS77-18642.

machines (from Floating-Point Systems AP-120B to TI ASC, CDC STAR-100 and CRAY-1). It consists of two pipelined arithmetic units, a multiplier and an adder chained together into a multifunction pipe in a static configuration. Each arithmetic unit in the model has s stages and requires one time step for a single operation, although it can deliver s results in one time step in the vector mode.

To compare the performances of different machines, which may be parallel, pipelined or simply sequential, the speedup, efficiency and utilization must be defined using attributes that characterize all machines. Such an attribute is the computational rate r_M (in operations/second) which can be sustained by the machine M in an ideal computation. On the other hand, an algorithm A is characterized by O(A), the number of operations needed to execute A. The computational time of an algorithm A on a machine M is denoted by $T_M(A)$. Note that $T_M(A) \geq O(A)/r_M$. Therefore, for any two machines M_i and M_j and any two algorithms A_k and A_ℓ, we can define speedup $S = T_{M_i}(A_\ell)/T_{M_j}(A_k)$, $E = (r_{M_i} \cdot T_{M_i}(A_\ell))/(r_{M_j} \cdot T_{M_j}(A_k))$, utilization $U = O(A_k)/r_{M_i} \cdot T_{M_i}(A_k)$ and redundancy $R = O(A_k)/O(A_\ell)$. This is a generalization of the notation found in [1].

Section 3 concludes the paper by comparison of results for pipelined and parallel machines. The pipelined machine with two functional units and s stages per unit has almost 2 times better performance for large bands than a parallel machine with s processors where each processor is capable of executing any arithmetic operation in one step. Although the hardware cost of pipelining is considerably smaller than the cost of parallelizing, it is impossible to increase the performance indefinitely by adding extra stages, since the number of gate levels in an implementation of a floating-point add or multiply is fixed. In present-day machines, s is between 2 and 8.

Furthermore, the common "row sweep" algorithm tends to perform better for small s and large m than the algorithm presented in this paper.

2. Main Theorem

Definition 1 An m-th order linear recurrence system R<n,m> is the set of n equations

$$x_i = \underline{a}_i * \underline{x}_{i-1}^T, \qquad 1 \leq i \leq n \tag{1}$$

where * denotes matrix multiplication and where $\underline{a}_i = (a_{i1}, a_{i2}, \ldots, a_{im}, c_i)$ is the vector[(a)] of coefficients, $\underline{x}_{i-1} = (x_{i-1}, x_{i-2}, \ldots, x_{i-m}, 1)$ is the vector of variables with initial vector $\underline{x}_0 = (x_0, x_{-1}, \ldots, x_{-m+1}, 1)$ specified in advance. ■

Furthermore, we will need a set of m+1 unit vectors $\underline{e}_i = (e_1, e_2, \ldots, e_{m+1})$ where for all j, $1 \leq j \leq m+1$

$$e_j = \begin{Bmatrix} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{Bmatrix}. \tag{2}$$

If a row (column) of any matrix is equivalent to a unit vector, it is called a trivial row (column).

Theorem 1

Any R<n,m> system can be written in the form

$$x_i = \underline{b}_i * \underline{x}_0^T \tag{3}$$

where for all i, $1 \leq i \leq n$, $\underline{b}_i = (b_{i1}, b_{i2}, \ldots, b_{im}, d_i)$ is defined recursively as follows:

$$\underline{b}_i = \underline{a}_i * \underline{B}_{i-1} \qquad 1 \leq i \leq n \tag{4}$$

with matrix $\underline{B}_{i-1} = (\underline{b}_{i-1}, \underline{b}_{i-2}, \ldots, \underline{b}_{i-m}, \underline{e}_{m+1})^T$, and $\underline{b}_0, \underline{b}_{-1}, \ldots, \underline{b}_{-m+1}$ equal to $\underline{e}_1, \underline{e}_2, \ldots, \underline{e}_m$.

Proof (by mathematical induction on i)

(Basis)

$x_1 = \underline{a}_1 * \underline{x}_0^T$ by Def. 1

$= (\underline{a}_1 * (\underline{e}_1, \underline{e}_2, \ldots, \underline{e}_m, \underline{e}_{m+1})^T) * \underline{x}_0^T$ by def. of I. matrix

$= (\underline{a}_1 * (\underline{b}_0, \underline{b}_{-1}, \ldots, \underline{b}_{-m+1}, \underline{e}_{m+1})^T) * \underline{x}_0^T$ by def. of $\underline{b}_i$

$= \underline{b}_1 * \underline{x}_0^T$ by def. of $\underline{b}_i$

(Induction step)

$x_{i+1} = \underline{a}_{i+1} * \underline{x}_i^T$ by Def. 1

$= \underline{a}_{i+1} * ((\underline{b}_i, \underline{b}_{i-1}, \ldots, \underline{b}_{i-m+1}, \underline{e}_{m+1})^T * \underline{x}_0^T)$ by hypothesis

$= (\underline{a}_{i+1} * (\underline{b}_i, \underline{b}_{i-1}, \ldots, \underline{b}_{i-m+1}, \underline{e}_{m+1})^T) * \underline{x}_0^T$ by associativity

$= \underline{b}_{i+1} * \underline{x}_0^T$ by def. of $\underline{b}_i$

■

Theorem 1 shows that any set of consecutive variables $\{x_i | j \leq i \leq k\}$ can be computed in parallel, that is, independently of each other, if

(a) Throughout this paper any vector $\underline{x}$ is a row vector and $\underline{x}^T$ is the corresponding column vector.

vectors of parallel coefficients $\underline{b}_k$, $j \le i \le k$, are generated and substituted for $\underline{a}_i$, $j \le i \le k$. The computation of all $\underline{b}_i$ is the overhead that must be paid for parallelism. The following example should clarify this idea.

Let us consider linear recurrence system R<9,2>. Using standard notation of linear algebra R<9,2> can be written as

$$\begin{bmatrix} 1 &&&&&&&& \\ -a_{21} & 1 &&&&&&& \\ -a_{32} & -a_{31} & 1 &&&&&& \\ & -a_{42} & -a_{41} & 1 &&&&& \\ && -a_{52} & -a_{51} & 1 &&&& \\ &&& -a_{62} & -a_{61} & 1 &&& \\ &&&& -a_{72} & -a_{71} & 1 && \\ &&&&& -a_{82} & -a_{81} & 1 & \\ &&&&&& -a_{92} & -a_{91} & 1 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \end{bmatrix}$$

This system can be transformed using (4) into

$$\begin{bmatrix} 1 &&&&&&&& \\ -a_{21} & 1 &&&&&&& \\ -a_{32} & -a_{31} & 1 &&&&&& \\ & -a_{42} & -a_{41} & 1 &&&&& \\ && -b_{52} & -b_{51} & 1 &&&& \\ && -b_{62} & -b_{61} && 1 &&& \\ && -b_{72} & -b_{71} &&& 1 && \\ && -b_{82} & -b_{81} &&&& 1 & \\ &&&&&& -a_{92} & -a_{91} & 1 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ c_9 \end{bmatrix}$$

where

$$[b_{51}, b_{52}, d_5] = [a_{51}, a_{52}, c_5] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$= [a_{51}, a_{52}, c_5]$$

$$[b_{61}, b_{62}, d_6] = [a_{61}, a_{62}, c_6] * \begin{bmatrix} b_{51} & b_{52} & d_5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$= [a_{61}\ a_{51} + a_{62}, a_{61}\ a_{52}, a_{61}\ c_5 + c_6]$$

$$[b_{71}, b_{72}, d_7] = [a_{71}, a_{72}, c_7] * \begin{bmatrix} b_{61} & b_{62} & d_6 \\ b_{51} & b_{52} & d_5 \\ 0 & 0 & 1 \end{bmatrix}$$
$$= [a_{71}(a_{61}\ a_{51} + a_{62}) + a_{72}\ a_{51},$$
$$a_{71}\ a_{61}\ a_{52} + a_{72}\ a_{52},$$
$$a_{71}(a_{61}\ c_5 + c_6) + a_{72}\ c_5 + c_7]$$

$$[b_{81},b_{82},d_8] = [a_{81},a_{82},c_8] * \begin{bmatrix} b_{71} & b_{72} & d_7 \\ b_{61} & b_{62} & d_6 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= [a_{81}(a_{71}(a_{61}\ a_{51}+a_{62}) + a_{72}\ a_{51})$$
$$+ a_{82}(a_{61}\ a_{51}+a_{62}),$$
$$a_{81}(a_{71}\ a_{61}\ a_{52}+a_{72}a_{52})$$
$$+ a_{82}\ a_{61}\ a_{52},$$
$$a_{81}(a_{71}(a_{61}\ c_5+c_6)$$
$$+ a_{72}\ c_5) + a_{72}\ c_5)$$
$$+ a_{82}(a_{61}\ c_5+c_6) + c_8]$$

The above coefficients can be obtained by direct substitution of x_5 into x_6, x_5 and x_6 into x_7 and finally x_5, x_6, and x_7 into x_8. The new transformed matrix allows parallel computation of x_5, x_6, x_7, and x_8 since they do not depend on each other anymore. It seems that nothing has been accomplished since computation of $\underline{b}_i$, $5 \le i \le 8$, must be executed serially, and it requires more time than the original one. However, the generation of $\underline{b}_i$, $5 \le i \le 8$, is independent of any other subset of variables whose intersection with $\{x_i | 5 \le i \le 8\}$ is empty, and therefore, the generation of $\underline{b}_i$'s for any two disjoint subsets of variables can be performed concurrently.

<u>Corollary 1</u>

In any R<n,m> the computation of $\underline{b}_i = \underline{a}_i * \underline{B}_{i-1}$, $1 \le i \le n$, requires at most m(m+1) multiplications and $(m-1)(m+1) + 1 = m^2$ additions. Furthermore, the computation of $x_i = \underline{b}_i * \underline{x}_0^T$, $1 \le i \le n$ requires at most m multiplications and m additions. ∎

However, $\underline{b}_1$ does not require any computation, $\underline{b}_2$ requires only m+1 multiplications and m additions, and so on. The first vector that requires m(m+1) multiplications and m^2 additions is $\underline{b}_{m+1}$. All vectors after $\underline{b}_{m+1}$ (that is, any $\underline{b}_i$, $m+1 \le i \le n$) require the same number of multiplications and additions. Therefore, when computing the total number of operations needed to compute all $\underline{b}_i$, $1 \le i \le n$, in any R<n,m>, two different formulas must be used: one for the case when $n \le m+1$ and the other for $n \ge m+1$.

<u>Corollary 2</u>

In any R<n,m> the computation of all $\underline{b}_i = \underline{a}_i * \underline{B}_{i-1}$, $1 \le i \le n$, requires no more than

$$K_m(n) = \begin{cases} (n - \frac{1}{2}(m+1))\ m(m+1) & \text{if } n \ge m+1 \\ \frac{1}{2}\ n(n-1)(m+1) & \text{if } n \le m+1 \end{cases} \quad (5)$$

multiplications, and

$$K_a(n) = \begin{cases} (n - \frac{1}{2}(m+1))\ m^2 & \text{if } n \ge m+1 \\ \frac{1}{2}\ n(n-1)\ m & \text{if } n \le m+1 \end{cases} \quad (6)$$

additions.

Furthermore, the total number of operations required is equal to

$$K(n) = K_m(n)+K_a(n) = \begin{cases} (n - \frac{m+1}{2})(2m^2 + m) \\ \qquad \text{if } n \ge m+1 \\ \frac{1}{2}\ n(n-1)(2m + 1) \\ \qquad \text{if } n \le m+1 \end{cases} \quad (7)$$

<u>Proof</u>

For all i, $1 \le i \le m+1$, $\underline{B}_{i-1}$ contains only i-1 nontrivial rows, so that $\underline{b}_i = \underline{a}_i \cdot \underline{B}_{i-1}$ requires only (i-1)(m+1) multiplications. Therefore, for all $n \le m+1$

$$K_m(n) = \sum_{i=1}^{n} (i-1)(m+1)$$

$$= (m+1) \sum_{i=1}^{n} (i-1)$$

$$= (m+1)\ \frac{n(n-1)}{2} \quad (8)$$

On the other hand, for $n \ge m+1$, K_m consists of two parts. Using (8), the number of multiplications to compute all $\underline{b}_i$, $1 \le i \le m$, is equal to $\frac{1}{2}(m+1)\ m(m-1)$. Each consecutive $\underline{b}_i$, $m+1 \le i \le n$ requires m(m+1) multiplications by Corollary 1. Therefore, for all $n \ge m+1$

$$K_m(n) = (n-m)\ m(m+1) + \frac{1}{2}(m+1)\ m(m-1)$$

$$= n\ m(m+1) - m^2(m+1) + \frac{1}{2}(m+1)\ m(m-1)$$

$$= n\ m(m+1) - \frac{1}{2}\ m(m+1)^2$$

Now consider $K_a(n)$ when $n \le m+1$. Each nontrivial row of $\underline{B}_{i-1}$, not counting the first, requires m+1 additions and each trivial row only one addition under the assumption that the first row in $\underline{B}_{i-1}$ is nontrivial. If the first row is trivial, then i = 1 and $\underline{B}_{i-1} = \underline{B}_0$ is an identity matrix. No additions are required in computation of $\underline{b}_1 = \underline{a}_1 * \underline{B}_0 = \underline{a}_1$. Therefore, for all $n \leq m+1$

$$K_a(n) = \sum_{i=2}^{n} [(i-2)(m+1) + (m+1) - (i-1)]$$

$$= \sum_{i=2}^{n} m(i-1)$$

$$= \frac{1}{2} mn(n-1)$$

Similarly, $K_a(n)$, $n \geq m+1$, consists of two parts since each $\underline{b}_i$, $m+1 \geq i \geq n$, requires m^2 additions by Corollary 1 and the computation of remaining $\underline{b}_i$, $1 \leq i \leq m$, altogether requires $\frac{1}{2} m^2(m-1)$ additions. Thus

$$K_a(n) = (n-m)\, m^2 + \frac{1}{2} m^2(m-1)$$

$$= n\, m^2 - \frac{1}{2} m^2(m+1)$$

■

3. Pipelined-Processor System

A pipelined-processor model consists of Main Memory (MM), Pipelined Processor (PP) and Control Unit (CU). PP may consist of several pipelined functional units. In our model, we shall assume only 2 functional units: multiplication pipe and addition pipe. Each of them has s stages. These two functional units are connected serially with the multiplication pipe feeding the add pipe through register R1. The results from the add pipe are either stored in MM or fed back into the add pipe through register R2 (Fig. 1).

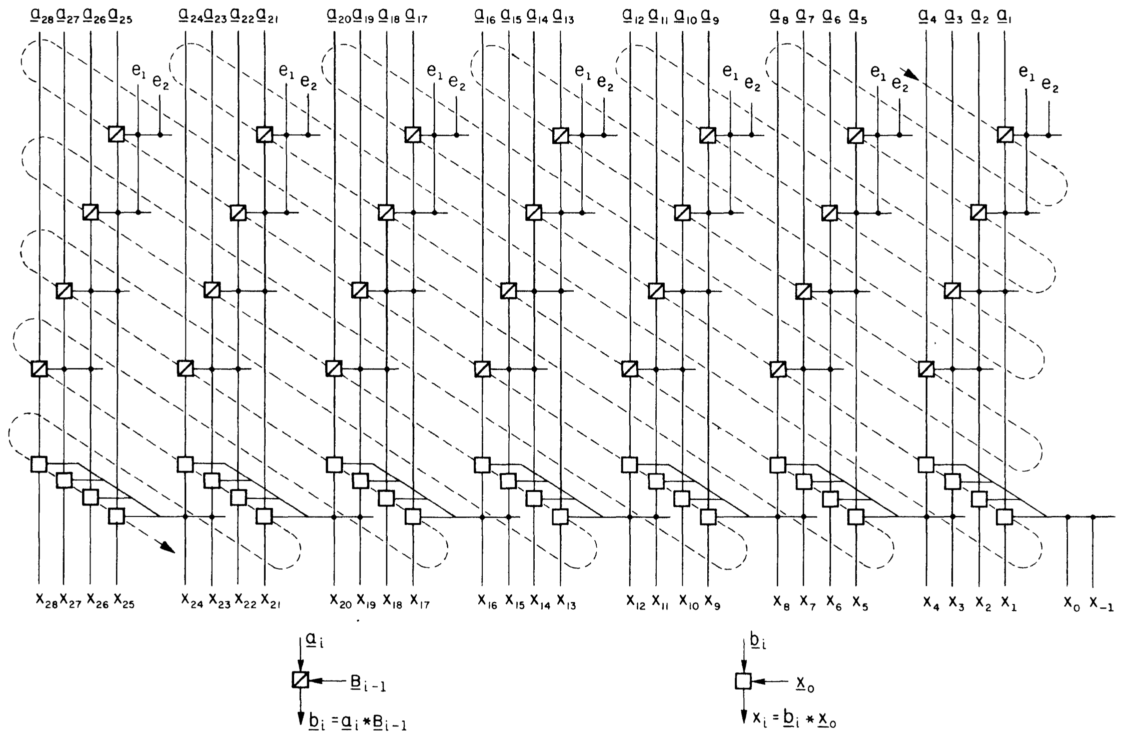
Algorithm 1

Given a linear recurrence system R<n,m>, divide R<n,m> into $\lceil n/p \rceil$ subsystems $R^{(j)}$<p,m>, where $1 \leq j \leq \lceil n/p \rceil$. Execute the following algorithm:

```
1.  begin
2.      for i: = 0 until m-1 do $\underline{a}_{-i}: = (0,\dots,0,x_{-1})$;
3.      for i: = m until $p^2 - 1$ do $\underline{a}_{-i}: =$ null;
4.      for i: = 0 until $p^2 - 1$ do $\underline{a}_{n+i+1}: =$ null;
5.      for j: = 1 until $\lceil n/p \rceil + p$ do
6.          begin
7.              $B_0^{(j)}: =$ identity matrix;
8.              for k: = 1 until p do $\underline{b}_k^{(j-k+1)}: = \underline{a}_k^{(j-k+1)} * \underline{B}_{k-1}^{(j-k+1)}$;
9.              for k: = 1 until p do $x_{p-k}^{(j-p)} = \underline{b}_{p-k+1}^{(j-p)} * \underline{x}_p^{(j-p-1)}$;
10.         end
11. end
```
■

The lines 2, 3, and 4 in Algorithm 1 are used to initialize boundary coefficients which are not included in R<n,m> , although they are referenced by the inner loop in lines 5-9. An application of Algorithm 1 to the system R<28,2> is shown in Fig. 2.

In general, a square with the inverse diagonal symbolizes computation of $\underline{b}_i^{(j)} = \underline{a}_i^{(j)} \cdot \underline{B}^{(j)}$, $1 \leq i \leq 4$, $1 \leq j \leq 7$. The input on the top of the square represents $\underline{a}_i$ and the output on the bottom $\underline{b}_i$. The horizontal input represents the matrix $\underline{B}_{i-1}$, whose rows are vectors $\underline{b}_{i-1}$, $\underline{b}_{i-2}$, ..., e_{m+1}. The dots on the horizontal line indicate the vectors that are rows in $\underline{B}_{i-1}$. The unit vector $\underline{e}_{m+1}$ which is always the last row of any matrix $\underline{B}_{i-1}$ is not connected for clarity. The closest dot to the square represents the top row of $\underline{B}_{i-1}$, that is, $\underline{b}_{i-1}$. The natural ordering of the rows in the matrix is established by proximity to the square. Similarly, the empty square represents the computation of $x_i^{(j)} = \underline{b}_i^{(j)} * \underline{x}_p^{(j-1)}$, with vertical input being $\underline{b}_i^{(j)}$, vertical output $x_i^{(j)}$, and the horizontal input representing vector $\underline{x}_p^{(j-1)}$. The order of fetching from MM and computing coefficients and variables in PP is indicated by dotted arrows. For example, the computation of $\underline{b}_{14}$ requires $\underline{a}_{14}$, $\underline{b}_{13}$ and e_1 to be fetched from MM and entered into PP. It is impossible to proceed with computation of $\underline{a}_{15}$ since $\underline{b}_{14}$ will become available only after a certain number of time steps. Therefore, in order to keep pipelined functional units busy, Algorithm 1 proceeds by computing $\underline{b}_{11}$, $\underline{b}_8$, x_4, x_3, x_2, x_1 and $\underline{b}_{21}$, $\underline{b}_{18}$. At that moment of time, $\underline{b}_{14}$ should be available at the output of PP to be used in computation of $\underline{b}_{15}$. The exact sequence of statements to be executed in this case is given below:



$a_{28}\, a_{27}\, a_{26}\, a_{25}$
$a_{24}\, a_{23}\, a_{22}\, a_{21}$
$a_{20}\, a_{19}\, a_{18}\, a_{17}$
$a_{16}\, a_{15}\, a_{14}\, a_{13}$
$a_{12}\, a_{11}\, a_{10}\, a_{9}$
$a_{8}\, a_{7}\, a_{6}\, a_{5}$
$a_{4}\, a_{3}\, a_{2}\, a_{1}$
$e_1\ e_2$
$x_{28}\, x_{27}\, x_{26}\, x_{25}$
$x_{24}\, x_{23}\, x_{22}\, x_{21}$
$x_{20}\, x_{19}\, x_{18}\, x_{17}$
$x_{16}\, x_{15}\, x_{14}\, x_{13}$
$x_{12}\, x_{11}\, x_{10}\, x_{9}$
$x_{8}\, x_{7}\, x_{6}\, x_{5}$
$x_{4}\, x_{3}\, x_{2}\, x_{1}$
$x_0\ x_{-1}$
a_i
B_{i-1}
$b_i = a_i * B_{i-1}$
b_i
x_0
$x_i = b_i * x_0$

Fig. 2. Solution of R<28,2> on a pipelined processor

$$
\begin{aligned}
b_{14,1} &= a_{14,1}\, b_{13,1} + a_{14,2} \\
b_{14,2} &= a_{14,1}\, b_{13,2} \\
d_{14} &= a_{14,1}\, d_{13} && && + c_{14} \\
b_{11,1} &= a_{11,1}\, b_{10,1} + a_{11,2}\, b_{91} \\
b_{11,2} &= a_{11,1}\, b_{10,2} + a_{11,2}\, b_{92} \\
d_{11} &= a_{11,1}\, d_{10} && + a_{11,2}\, d_{9} && + c_{11} \\
b_{81} &= a_{81}\, b_{71} && + a_{82}\, b_{61} \\
b_{82} &= a_{81}\, b_{72} && + a_{82}\, b_{62} \\
d_{8} &= a_{81}\, d_{7} && + a_{82}\, d_{6} && + c_{8} \\
x_{4} &= b_{41}\, x_{0} && + b_{42}\, x_{-1} && + d_{4} \\
x_{3} &= b_{31}\, x_{0} && + b_{32}\, x_{-1} && + d_{3} \\
x_{2} &= b_{21}\, x_{0} && + b_{22}\, x_{-1} && + d_{2} \\
x_{1} &= b_{11}\, x_{0} && + b_{12}\, x_{-1} && + d_{1}
\end{aligned}
$$

$$
\begin{aligned}
b_{18,1} &= a_{18,1}\, b_{17,1} + a_{18,2} \\
b_{18,2} &= a_{18,1}\, b_{17,2} \\
d_{18} &= a_{18,1}\, d_{17} && && + c_{18} \\
b_{15,1} &= a_{15,1}\, b_{14,1} && + a_{15,2}\, b_{13,1} \\
b_{15,2} &= a_{15,1}\, b_{14,]} && + a_{15,2}\, b_{13,2} \\
d_{15} &= a_{15,1}\, d_{14} && + a_{15,2}\, d_{13} && + c_{15} \\
b_{12,1} &= a_{12,1}\, b_{11,1} && + a_{12,2}\, b_{10,1} \\
b_{12,2} &= a_{12,1}\, b_{11,2} && + a_{12,2}\, b_{10,2} \\
d_{12} &= a_{12,1}\, d_{11} && + a_{12,2}\, d_{10} && + c_{12} \\
x_{8} &= b_{81}\, x_{4} && + b_{82}\, x_{3} && + d_{8} \\
x_{7} &= b_{71}\, x_{4} && + b_{72}\, x_{3} && + d_{7} \\
x_{6} &= b_{61}\, x_{4} && + b_{62}\, x_{3} && + d_{6} \\
x_{5} &= b_{51}\, x_{4} && + b_{52}\, x_{3} && + d_{5}
\end{aligned}
$$

The next theorem is based on observation that there are at most m+1 additions and m+1 multiplications per statement. Multiplication by 1 and addition to 0 are included.

Theorem 2

Given an integer $p > 0$, any linear recurrence system $R<n,m>$ can be solved in $T_s \leq (\frac{n}{p} + p)(m + 2)$ time steps using pipelined functional units with $s = ((m + 2) p - (m + 1))$ stages.

Proof

The proof is based on Algorithm 1. The system $R<n,m>$ is divided into subsystems $R^{(j)}<p,m>$, $1 \leq j \leq \lceil n/p \rceil$. To keep pipelined units busy almost all the time p+1 subsystems are being solved concurrently; that is, different subsystems are using different stages of the pipelined units.

Each $R^{(j)}$ p,m system is solved in two parts: $\underline{b}_k^{(j)} = \underline{a}_k^{(j)} * \underline{B}_{k-1}^{(j)}$, $1 \leq k \leq p$, is computed first and $x_{p-k+1}^{(j)} = \underline{b}_{p-k+1}^{(j)} * \underline{x}_p^{(j-1)}$, $1 \leq k \leq p$, is evaluated afterwards. With p+1 systems computed concurrently, the order of computation is as follows $\ldots \underline{b}_1^{(j)}, \underline{b}_2^{(j-1)}, \ldots, \underline{b}_p^{(j-p+1)}, x_p^{(j-p)}, x_{p-1}^{(j-p)}, \ldots, x_1^{(j-p)}, \underline{b}_1^{(j+1}, \underline{b}_2^{(j)}, \ldots, \underline{b}_p^{(j-p+2)}, x_p^{(j-p+1)}, x_{p-1}^{(j-p+1)}, \ldots, x_1^{(j-p+1)}, \ldots$.

Therefore, at least (k-1) $\underline{b}_i$'s and k x_i's must be computed before their values are used in subsequent computations. Each $\underline{b}_i$ is a vector containing m+1 elements. Each element of $\underline{b}_i$ is computed as a sum-of-products (sop). Similarly, each x_i is a sop. Hence, there are $N = (m+1)(p-1) + p$ sop's altogether. Each sop has m+1 products at most. This implies that multiplication and addition pipes are used m+1 times for each sop. The bottleneck of the pipelined processor is obviously addition pipe since a new product can be added to the corresponding partial sum only after one addition-pipe step, i.e., after s stage delays. Therefore, m+1 addition steps and one multiplication step are needed to obtain p solutions of $R<n,m>$. Total time needed to compute $R<n,m>$ is less than $(\frac{n}{p} + p)(m + 2)$ pipeline processor steps. The number of stages s in the addition pipe must be equal to N. This way, a partial sum for some sop and a new product to be added to that partial sum are generated at the same time by addition and multiplication pipes, respectively, and are ready to be entered into the addition pipe again. ■

As an example, the pipelining of the above sequence of statements starting with computation of $\underline{b}_{14}$ through the pipelined processor model of Fig. 1 is given in Fig. 3. The memory output ports, MEM 1 and MEM 2, deliver two operands on each clock. An empty entry in Fig. 3 denotes the zero operand. The memory input port MEM 3 is independent of the two memory output ports. The critical point is the computation of d_8 which is stored in clock 61 and fetched from memory in clock 62.

It is worth noting the desirability of a large $p \leq \sqrt{n}$ (the function $(\frac{n}{p} + p)(m + 2)$ has a minimum at $p = \sqrt{n}$). However, a large p requires a large number of stages in pipelined functional units. Since the number of gate levels in an implementation of floating-point multiply or add is fixed, the number of stages s will hardly exceed 10-15. In present-day machines s is between 2 and 8.

Secondly, increased s increases the cost of functional units since extra latches or registers have to be added which in turn introduces extra delay, reducing the effect of pipelining.

The problem of finding optimal number of stages s for a given p was answered by Theorem 3. In reality, pipelined processors have fixed s and the problem is in finding minimal T_s. Without loss of generality, we shall assume that all functional units have the same number of stages.

Corollary 3

Given a pipelined processor with s-stage functional units, any $R<n,m>$ can be computed in

$$T_s \leq \min[(\frac{n}{p'} + p')(m + 2), \frac{(m+2)\,p'' - (m+1)}{s} (\frac{n}{p''} + p'')(m + 2)]$$

where $p' = \left\lfloor \frac{s+m+1}{m+2} \right\rfloor$ and $p'' = \left\lceil \frac{s+m+1}{m+2} \right\rceil$.

Proof

For a given p, $N = (m+2) p - (m+1)$. If $s = N$, all stages are busy performing some computation. If $s > N$, exactly $s - N$ stages are idling and waiting for the partial sums to become available at the output of the addition pipe. The time T_s required to compute $R<n,m>$ is still the same, $(\frac{n}{p} + p)(m + 2)$. However, the effectiveness $E_s = T_s/s$ has decreased. On the other hand, if $s < N$ there is no idling, but more pipeline time steps are required to compute $R<n,m>$. Consequently $T_s = \frac{N}{s} (\frac{n}{p} + p)(m + 2)$.

For a given s, it is possible to choose p such that $s \geq N$ ($p \leq (s+m+1)/(m+2)$) or $s \leq N$ ($p \geq (s+m+1)/(m+2)$). In either case a p that minimizes T_s is sought. In the former case, the function $(\frac{n}{p} + p)(m + 2)$ is monotonically decreasing for positive p's, $p < \sqrt{n}$, with the minimum at $p = \sqrt{n}$. Assuming $p \ll n$, the minimum time is obtained for the largest p such that $p \leq (s+m+1)/(m+2)$. When $s \leq N$ the function $\frac{N}{s}(\frac{n}{p} + p)(m + 2)$ is monotonically increasing for

CLOCK	MEM 1	MEM 2	R1	R2	MEM 3
1				$b_{10,1}$	$b_{10,1}$
2				$b_{10,2}$	$b_{10,2}$
3	c_{14}	1		d_{10}	d_{10}
4				b_{71}	b_{71}
5				b_{72}	b_{72}
6	c_{11}	1		d_7	d_7
7				b_{41}	b_{41}
8				b_{42}	b_{42}
9	c_8	1		d_4	d_4
10	d_4	1		x_0	x_0
11	d_3	1		x_{-1}	x_{-1}
12	d_2	1		x_{-2}	x_{-2}
13	d_1	1		x_{-3}	x_{-3}
14	$a_{14,2}$	1			
15					
16			c_{14}		
17	$a_{11,2}$	b_{91}			
18	$a_{11,2}$	b_{92}			
19	$a_{11,2}$	d_9	c_{11}		
20	a_{82}	b_{61}			
21	a_{82}	b_{62}			
22	a_{82}	d_6	c_8		
23	b_{42}	x_{-1}	d_4		
24	b_{32}	x_{-1}	d_3		
25	b_{22}	x_{-1}	d_2		
26	b_{12}	x_{-1}	d_1		
27	$a_{14,1}$	$b_{13,1}$	$a_{14,2}$		
28	$a_{14,1}$	$b_{13,2}$			
29	$a_{14,1}$	d_{13}		c_{14}	
30	$a_{11,1}$	$b_{10,1}$	$a_{11,2}$ b_{91}		
31	$a_{11,1}$	$b_{10,2}$	$a_{11,2}$ b_{92}		
32	$a_{11,1}$	d_{10}	$a_{11,2}$ d_9	c_{11}	
33	a_{81}	b_{71}	a_{82} b_{61}		
34	a_{81}	b_{72}	a_{82} b_{62}		
35	a_{81}	d_7	a_{82} d_6	c_8	
36	b_{41}	x_0	b_{42} x_{-1}	d_4	
37	b_{31}	x_0	b_{32} x_{-1}	d_3	
38	b_{21}	x_0	b_{22} x_{-1}	d_2	
39	b_{11}	x_0	b_{12} x_{-1}	d_1	
40			$a_{14,1}$ $b_{13,1}$	$a_{14,2}$	
41			$a_{14,1}$ $b_{13,2}$		
42			$a_{14,1}$ d_{13}	c_{14}	
43			$a_{11,1}$ $b_{10,1}$	$a_{11,2}$ b_{91}	
44			$a_{11,1}$ $b_{10,2}$	$a_{11,2}$ b_{92}	
45			$a_{11,1}$ d_{10}	$a_{11,2}$ d_9+c_{11}	
46			a_{81} b_{71}	a_{82} b_{61}	
47			a_{81} b_{72}	a_{82} b_{62}	
48			a_{81} d_7	a_{82} d_6+c_8	
49			b_{41} x_0	b_{42} $x_{-1}+d_4$	
50			b_{31} x_0	b_{32} $x_{-1}+d_3$	
51			b_{21} x_0	b_{22} $x_{-1}+d_2$	
52			b_{11} x_0	b_{12} $x_{-1}+d_1$	
53				$b_{14,1}$	$b_{14,1}$
54				$b_{14,2}$	$b_{14,2}$
55	c_{18}	1		d_{14}	d_{14}
56				$b_{11,1}$	$b_{11,1}$
57				$b_{11,2}$	$b_{11,2}$
58	c_{15}	1		d_{11}	d_{11}
59				b_{81}	b_{81}
60				b_{82}	b_{82}
61	c_{12}	1		d_8	d_8
62	d_8	1		x_4	x_4
63	d_7	1		x_3	x_3
64	d_6	1		x_2	x_2
65	d_5	1		x_1	x_1
66	$a_{18,2}$	1			
67					
68			c_{18}		
69	$a_{15,2}$	$b_{13,1}$			
70	$a_{15,2}$	$b_{13,2}$			
71	$a_{15,2}$	d_{13}	c_{15}		
72	$a_{12,2}$	$b_{10,1}$			
73	$a_{12,2}$	$b_{10,2}$			
74	$a_{12,2}$	d_{10}	c_{12}		
75	b_{82}	d_8	d_8		
76	b_{72}	d_7	d_7		
77	b_{62}	d_6	d_6		
78	b_{52}	d_5	d_5		

Fig. 3. Pipeline diagram of R<28,2>

positive p's, and therefore the smallest p, $p \geq (s+m+1)/(m+2)$ requires minimum time to compute R<n,m>. ∎

To obtain a qualitative assessment of Algorithm 1, we shall obtain the time needed to compute R<n,m> serially on a pipelined machine. A simple observation of $x_i = a_{i1} x_{i-1} + \dots + a_{im} x_{i-m} + c_i$ shows that after x_{i-1} has been computed, it should take theoretically at most 1 multiplication and 1 addition to compute x_i since the remainder of the expression could be computed ahead of time. Therefore, as long as the number of stages s is smaller than m/2, the straightforward computation will keep the multiplication and addition pipes saturated and give the best possible speedup asymptotically.

Lemma 1

Any R<n,m> can be computed on a pipelined processor with s-stage functional units in time

$$T_s \leq \frac{n}{s}(1 + \max(2s,m)) + \min(2s,m)$$

using the "row sweep" algorithm.

1 step = s mini clo

Proof

The proof is based on the assumption that a number cannot be written into the memory and read out in the same clock period. First, assume $2s \leq m$. Therefore, all variables are evaluated before they are used, that is, before they are input into the multiplication pipe. Counting a multiplication by 1 as well as addition to 0, the evaluation of each variable requires m+1 multiplications and m+1 additions. Therefore, the time to compute R<n,m> is $t_0 + n(m+1)/s$ where t_0 is the startup time. To calculate t_0, we observe that s variables are evaluated in m+1 time steps. However, the first variable requires m+3 steps and each of s-1 variables can be computed in 2 additional steps afterwards. Hence,

$$t_0 = m + 3 + 2(s-1) - (m+1) = 2s$$

The total computational time

$$T_s \leq 2s + \frac{n(m+1)}{s} \quad \text{if } 2s \leq m.$$

Finally, assume $2s \geq m$. Since $m \leq 2s$, some pipeline stages are idle during computation and it takes at least 2 time steps plus one clock period to evaluate a new variable. The time needed to compute R<n,m> is equal to $t_0 + n(2s+1)/s$ where startup time

$$t_0 = m + 3 + 2(s-1) - (2s+1) = m.$$

Therefore,

$$T_s \leq m + \frac{n(2s+1)}{s} \quad \text{if } 2s \geq m.$$

∎

Two algorithms presented in this paper are compared. For simplicity, the "row sweep" algorithm is denoted by A and the parallel algorithm by A_s. Note that A_s is the parallel algorithm requiring the shortest computational time (determined by Corollary 3) on our pipelined machine M_s, with s stages per arithmetic unit. M_s is compared with a uniprocessor machine M_μ that executes any arithmetic operation in one step. The ideal computational rate of M_s is 2s. For simplicity, we shall denote $T_{M_\mu}(A)$ and $T_{M_s}(A)$ by T and T', and $T_{M_s}(A_s)$ by T_s. Obviously, T = 2mn for any R<n,m>, and $T/T_s = 2mn/T_s$. The speedups $(T/T_s)(1 + \varepsilon_1)$ and $(T/T')(1 + \varepsilon_2)$ for m = 1, 2, 4 and 8 are shown in Fig. 4 with $\varepsilon_1 = \frac{p^2}{n}$ where $p \in \{p', p''\}$ and $\varepsilon_2 = \frac{s}{n} \frac{\min(2s,m)}{1 + \max(2s,m)}$. The values of ε_1 and ε_2 are relatively small for large values of n.

The comparison of speedups in Fig. 4 shows that the parallel algorithms are superior for all $s \geq s_0$, and that "row sweep" is better for $1 \leq s < s_0$. However, the cutoff point s_0 may be too large to be practical for banded triangular systems with medium and large bands. For example, if a practical s = 15 is assumed, then the "row sweep" algorithm has the advantage for all m > 4.

Note that the "row sweep" algorithm has almost the best possible speedup of 2s for all $s \leq m/2$. For all $s > m/2$, the "row sweep" algorithm has approximately a constant speedup slowly approaching the value of m. This behavior is not surprising since the computation time T' is limited at the beginning by the number of stages and later by the operation time, which is kept constant with respect to the number of stages per operation.

The speedup of the parallel algorithms is much smaller than 2s. This can be explained with increased redundancy of A_s with respect to A. The redundancy function (Fig. 5) is a step function with steps corresponding to the set of all positive integers; that is, the first step corresponding to the parallel algorithm with p = 1, the second step to p = 2, and so on. When p changes from, say, k to k+1, there is a substantial increase in the number of redundant operations which is compensated eventually by the increased number of stages s.

The breakpoints in speedup functions in Fig. 4 are easily explained with redundancy functions. For each step there is an optimal number of stages s for which the speedup is maximal. These optimal values of s are indicated by heavy dots in Fig. 4. For any s between the two optimal values s_1 and s_2 corresponding to parallel algorithms A_{s_1} and A_{s_2}, the algorithm with better speedup is used. As the number of stages increases from an optimal value s_1, the algorithm A_{s_1} generates the same speedup since the number of operations stays the same and the operands do not flow faster through the pipelined arithmetic units. Since the number of stages is greater than s_1, some of the stages are idle, waiting for the operands to become available from the bottom of the pipe. The utilization of the machine decreases (Fig. 5). The algorithm A_{s_2} is used when the increased number of stages is capable of compensating for increased redundancy of the algorithm. Since s is smaller than s_2, the number of operations that can be performed independently is greater than the number of stages. There are no idle stages and the speedup is limited to s. As s increases, the speedup increases linearly with s until it reaches its maximum at s_2. The utilization is constant in intervals of the speedup's linear increase.

Another interesting result is a lower-than-expected speedup for banded systems with m = 1. Although the redundancy for m = 1 is low, the utilization is low, too, resulting in overall speedup that is below those for m = 2 and m = 4. The low utilization is the result of a very simple model which requires a multiplication by 1 as well as addition to 0 to be considered as operations, while they were not counted as operations in computation of redundancy. Since the percentage of these operations is high for m = 1, the utilization is very low.

4. Conclusion

A parallel algorithm for solving banded triangular systems on pipelined machines was developed. The algorithm uses extra redundant operations to allow parallel computation in pipelined arithmetic units with s stages. The number of s stages is large enough to compensate for introduced redundancy and to achieve overall speedup with respect to a uniprocessor machine using the natural "row sweep" algorithm. When the "row sweep" algorithm is microprogrammed on the pipelined machine, the comparison shows that the parallel algorithm is superior whenever the number of stages s is greater than s_0. The break-even point s_0 depends on the size of the band m, and increases with m. Even for medium m the break-even point may be too large to be practically implementable, since floating-point operations have a fixed number of gates and, therefore, they can be pipelined only to a certain number of stages.

The final comparison that remains to be made is between pipelined and parallel machines. In other words, what machine organization should we adopt if the only measure of performance is

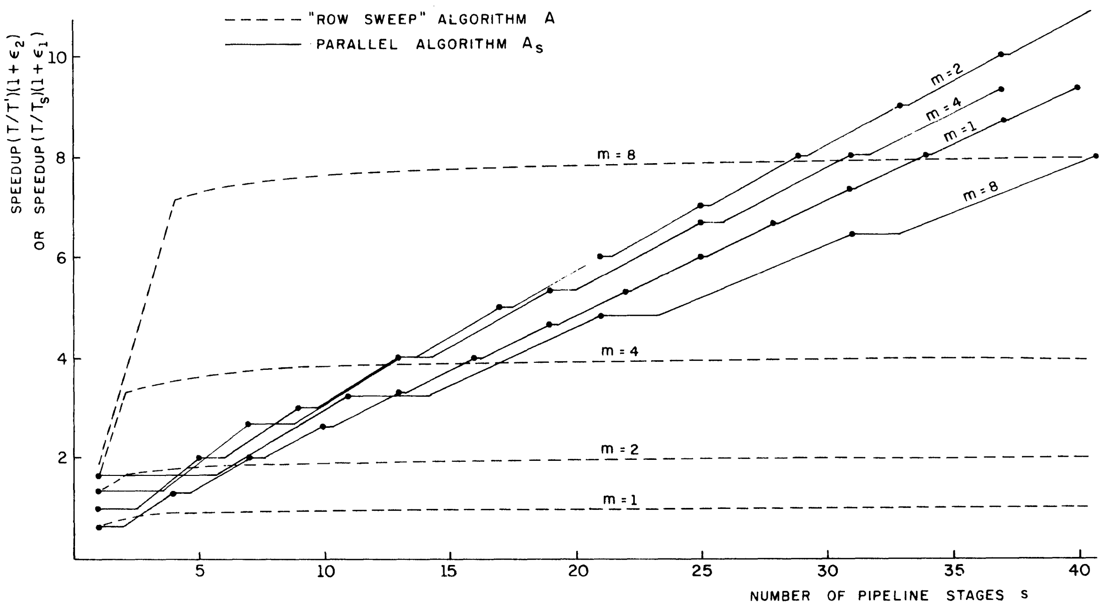


Fig. 4. Speedup for pipelined machine

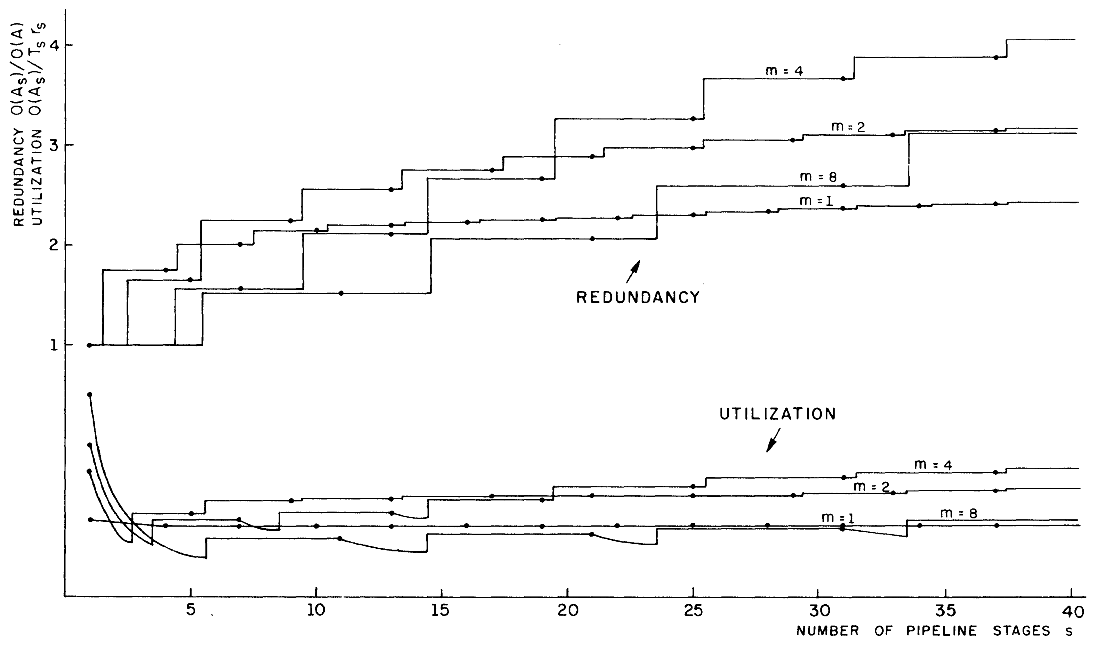


Fig. 5. Redundancy and utilization for pipelined machine

banded triangular system solvers? The pipelined machine with s stages per operation is compared to a SIMD parallel machine. Such a parallel model has p = s memory modules and p processors (arithmetic units). Similarly, to pipelined arithmetic units each processor requires one time step to generate the result from two operands for any arithmetic operation. However, the arithmetic is not pipelined and, therefore, only one result is generated in each time step. It is assumed further that no time is required to communicate data between processors and memories, and that the storage arrangement of data is irrelevant.

The algorithm for a parallel machine was developed in [4] and the time necessary to compute a banded triangular system of size m was established to be

$$T_p = \begin{cases} \frac{n}{p}((2m^2 + 3m) - \frac{m+1}{2p}(2m^2+m)) & \text{if } p \geq m+1 \\ n(m + \frac{1}{2} + \frac{1}{p}(m - \frac{1}{2})) & \text{if } p \leq m+1 \end{cases}$$

As before, two algorithms--the "row sweep" algorithm and the parallel algorithm running on the pipeline machine--are compared to a parallel algorithm on a parallel machine.

The speedups $(T_p/T_s)(1 + \varepsilon_1)$ and $(T_p/T')(1 + \varepsilon_2)$ are plotted in Fig. 6. The pipeline machine with the "row sweep" algorithm is inferior to the parallel machine for m = 1 and m = 2. For all m > 2, the parallel machine eventually

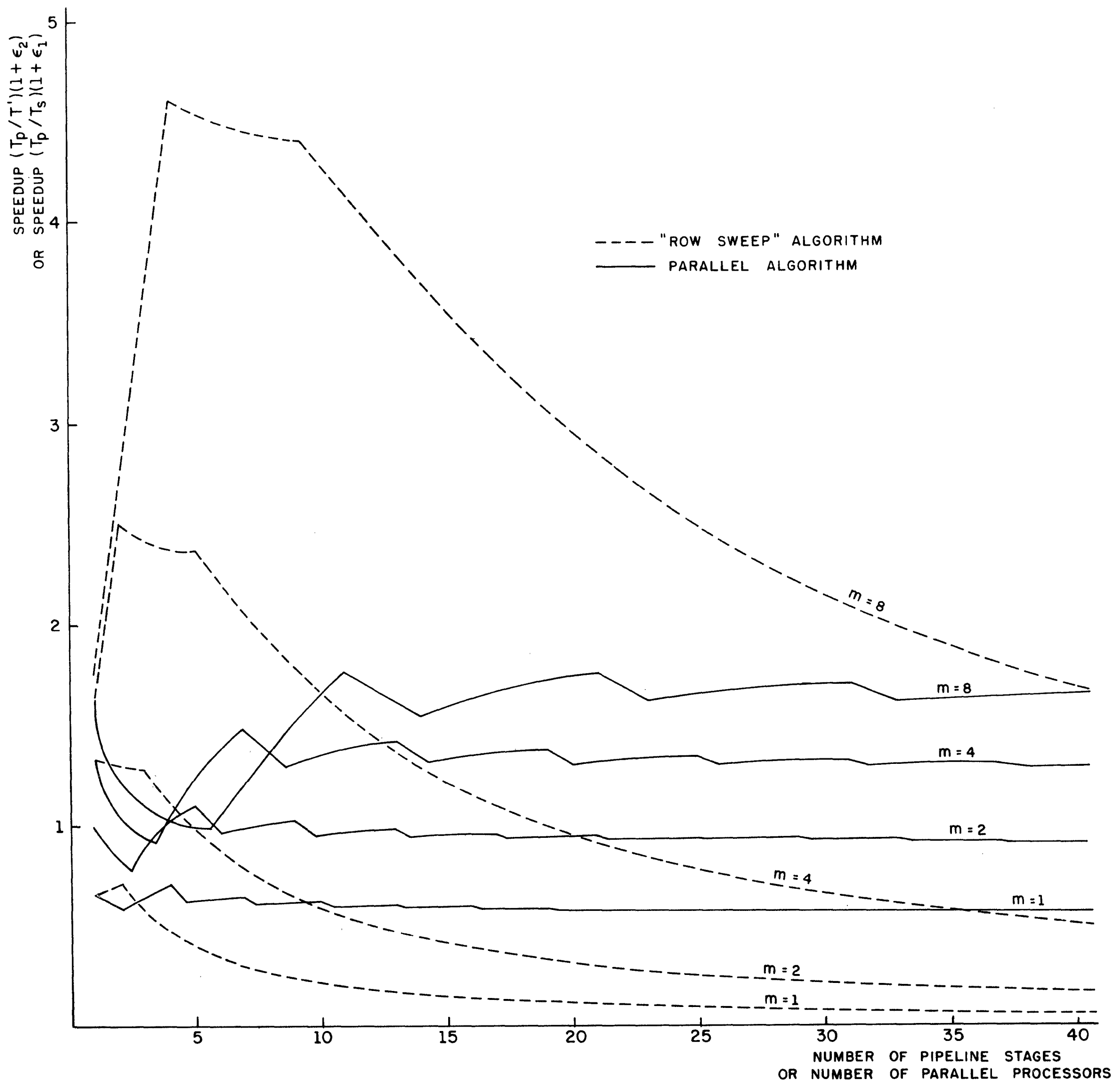


Fig. 6. Speed comparison of pipeline and parallel models

succeeds to outperform the pipelined machine, when the number of processors is large enough. When the parallel algorithm is used on the pipelined machine, the situation is quite different. From m = 1 the parallel machine is 2 times faster than its pipelined counterpart, while for m = 2 they are approximately equal. For all m 2, the pipelined model is performing better, but no more than 2 times better.

It should be mentioned that the results are not very surprising since the ideal computational rate of the pipelined model is proportional to 2s while that of the parallel model is only proportional to p = s. In the pipelined model, we allowed two arithmetic operations (a multiplication and an addition) to be executed at the same time, while only one operation per time step was allowed in each processor of the parallel model.

Acknowledgment

I want to thank Robert Kuhn and David Kuck for helpful suggestions during the development of this paper.

References

[1] D. J. Kuck, The Structure of Computers and Computations, Vol. I, John Wiley & Sons, Inc., (1978), 611 pp.

[2] S. C. Chen, D. J. Kuck, and A. H. Sameh, "Practical Parallel Band Triangular System Solvers," ACM Transactions on Mathematical Software (Sept., 1978), pp. 270-277.

[3] S. C. Chen, and A. H. Sameh, "On Parallel Triangular System Solvers," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing (Aug., 1975), pp. 237-238.

[4] D. D. Gajski, "An Algorithm for Solving Linear Recurrence on Parallel and Pipelined Machines," Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 78-953, (Dec., 1978), 36 pp.

[5] P. M. Kogge, "Maximal Rate Pipelined Solutions to Recurrence Problems," Proceedings 1st Annual Symposium of Computer Architecture (Dec., 1973), pp. 71-76.

[6] D. Heller, "On the Efficient Computation of Recurrence Relations," NASA Langley Research Center, Institute for Computer Applications in Science and Engineering (ICASE), Hampton, VA, (June, 1974).

[7] L. Hyafil and H. T. Kung, "The Complexity of Parallel Evaluation of Linear Recurrences," Journal of the ACM (July, 1977), pp. 513-521.

[8] A. H. Sameh, and R. P. Brent, "Solving Triangular Systems on a Parallel Computer," SIAM Journal of Numerical Analysis (1977), pp. 1101-1113.

A PARALLEL/PIPELINE MULTIPROCESSOR ARCHITECTURE FOR SOLVING SYSTEMS OF LINEAR EQUATIONS

William C. Liles and James W. Demmel

Computer Sciences Division
TECHNOLOGY SERVICE CORPORATION
Santa Monica, California 90403

Summary

Methods have been explored for solving large systems of simultaneous linear equations with positive definite Hermitian coefficient matrices. Our goal was to solve large systems quickly and efficiently, using a parallel/pipeline approach. We summarize here an algorithm based on Gram-Schmidt orthogonalization and its implementation on a simply and regularly connected processor array with simple processors. Given an nxn matrix A and m vectors B_i, $1 \leq i \leq m$, our implementation can solve the m problems $AX_i = B_i$, $1 \leq i \leq m$, in time $O(m + n)$, using $\sim n^2/2$ processors. This method may be used in least squares, regression, and adaptive signal processing problems, and variational problems such as solving self-adjoint differential equations with finite element methods.

Let the system to be solved be AX = B, with A a positive definite Hermitian nxn matrix, B an nx1 vector, and X an nx1 vector of unknowns. If T is any nonsingular nxn matrix, then $A_T = TAT^*$ (T^* = T conjugate transpose) is also positive definite Hermitian and $X = A^{-1}B = T^*A_T^{-1}TB$. By choosing T so that T, A_T^{-1} and T^* are easily computable, we will have simplified our problem. This computation will be accomplished by the Gram-Schmidt orthogonalization process [1] (where <a, b> is a complex inner product):

$$(*) \begin{cases} Z_j^1 \equiv B_j = \text{input} \\ Z_j^{i+1} = Z_j^i - (\langle Z_j^i, Z_i^i\rangle / \langle Z_i^i, Z_i^i\rangle) \cdot Z_i^i \\ Z_j \equiv Z_j^j = \text{output } (Z = TB) \end{cases} .$$

The T provided by (*) is unit upper triangular and $A_T = \text{diag}(\langle Z_i^1, Z_i\rangle)$ is diagonal. In fact, the decomposition $A = T^{-1}A_T(T^*)^{-1}$ is just the Cholesky decomposition of A.

The implementation of T is shown in Figure 1 for n = 4 (arrows indicate direction of data flow). Processor P_{ij} computes

$$Z_j^{i+1} = Z_0^i - w_{ij} Z_i^i \; (w_{ij} = \langle Z_j^j, Z_i^i\rangle / \langle Z_i^i, Z_i^i\rangle) .$$

The n(n - 1)/2 processors work as follows: The processors in row i work simultaneously on intermediate results passed to them by the row above and pass their results simultaneously on to the next row, $P_{i,i+1}$, broadcasting its value to the entire next row. All processors work simultaneously, each row working on the intermediate results of different input vectors being piped into the top of the processor array. The processors are simple, identical, independent of n, and are connected in a simple pattern with fixed unidirectional data paths.

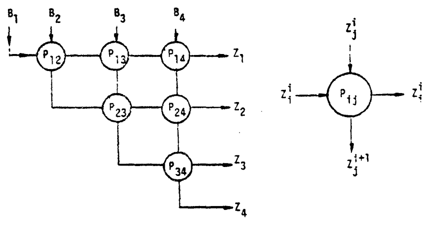


Figure 1. Implementation of T

The w_{ij}'s may be computed in two ways by the array in Figure 1, depending on whether the actual matrix A is known or if A must be computed as a sample covariance matrix of many sample vectors (typical least squares problem). A_{Tii} is available in any processor row i.

Multiplication by T^* may be done by either the unit vector method or the reverse flow method. The unit vector method computes the entries of T^* by passing standard unit vectors through the array and forming the product in vertically connected processors on the output ports of the array. This method solves the m problems $AX_i = B_i$ in time O(mn) with high efficiency E_{uv} = Processor on time/(Total time * Total processors) = $(m/m + 1) \cdot \{[(n + 1)^2 - 2]/(n + 1)^2\}$. The reverse flow method passes the data backwards through the array, performing the computations in reverse. This method requires time O(m + n) but has lower efficiency $E_{RF} = m/[m + 2(n + 1)]$.

Reference

[1] J. R. Rice, "Experiments on Gram-Schmidt Orthogonalization," Math. Comput., (20 April 1966), pp. 325-328.

MACROCELLULAR PIPELINED MULTIPLYING ARRAYS

L.Ciminiera and A.Serra
CENS-Istituto di Elettrotecnica Generale
Politecnico di Torino
corso Duca degli Abruzzi,24-Torino-ITALY

Summary

This paper presents pipelined multiplying arrays, which use macrocells bigger than the gated one bit full adder (GFA) discussed previously in the literature [1] . The introduction of these macrocells decreases the number of latching stages, and, in some cases, increases the gate number of the combinatorial part. Thus the total gate number of the resulting structure must be evaluated for every particular implementation of the cell.On the other hand the row and column number of the array are reduced, because each macrocell performs the same calculation for a larger number of bits than the GFA. Therefore the latency time and throughput are remarkably improved if the delay per stage, D, is opportunely limited; this can be obtained implementing the macrocell with a few gate level. In this paper the cost/operation parameter, as defined in [2] , was selected in order to compare the our proposed arrays with the classical ones. Since the two logic level implementation of a 2x2 bit full multiplier macrocell requires 119 gates , taking into account the results presented in another paper [3], it may be seen that the 48x48 bit maximally pipelined multiplying array, built with these macrocells, has a better cost/operation than the equivalent Guild array,provided that the number of consecutive multiplications, M, is less than 209. Thus, in this case, the solution proposed is better than the equivalent GFA array, in almost all practical situations.

To reduce the complexity of the array a second 2x2 bit full multiplier macrocell, shown in Fig. 1, is introduced. The resulting structure performing 4x4 bit multiplication is shown in shown in fig. 2; the cells enclosed by the dashed line are needed to reduce the output result in a binary number having only one digit for every weight. The two gate level implementation of the two types of cell requires 82 and 14 gates respectively, so that every characteristic of the above mentioned 48x48 bit array is better than the corresponding Guild array [1] . We may define a gain parameter, γ as follows:

$$(1)\quad \gamma = 1-\left[\eta_1(M) \;/\; \eta_2(M)\right]$$

where $\eta_1(M)$ is the cost/operation of the maximally pipelined macrocellular array performing the 48x48 bit multiplication, and $\eta_2(M)$ is the cost/operation of the equivalent maximally pipelined Guild array. Fig.3 shows the behavior of γ vs. M. This curve shows that the macrocellular array improves the cost/operation of the analogous Guild array, for every number of consecutive multiplications. These results confirm that the introduction of macrocells is a new and satisfactory way for implementing pipelined multiplying arrays.

References

[1] J. Deverell, "Pipeline Iterative Arithmetic Arrays", IEEE Trans. Comp. (March, 1975) , pp. 317-322.

[2] J.R. Jump and S.R. Ahuja, "Effective Pipelining of Digital Systems", IEEE Trans. Comp. (September, 1978) , pp. 855-865.

[3] L. Ciminiera and A. Serra, " Pipelined Multipliers Implemented with Macrocells", to be published.

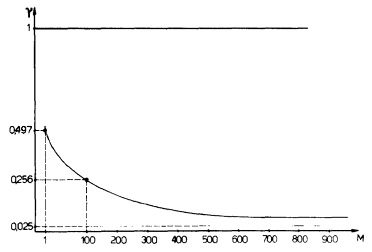


Fig.3 .Cost/operation improvement obtained using macrocells in a 48x48 pipelined array

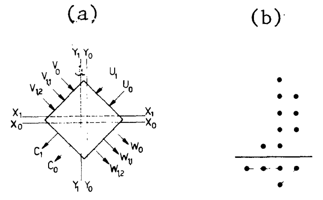


Fig.1.Second type of macrocell:
(a) logic symbol;
(b) dot representation of the arithmetic function.

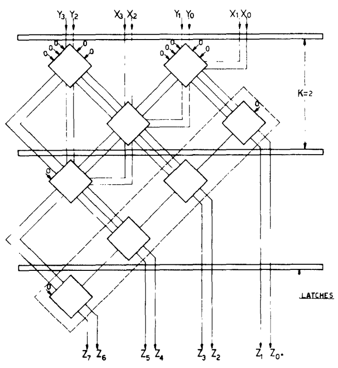


Fig.2.4x4 bit pipelined multiplier built with the second type of macrocells.

MODELING MAXIMUM PARALLEL EXECUTIONS IN PIPELINE EXECUTABLE FORM USING PRECEDENCE EXPRESSIONS

B. I. Dervisoglu
Department of Electrical Engineering
and Computer Science
University of Connecticut
Storrs, CT 06268

Summary

A procedure is described [2] which transforms a computation structure [1] into an execution model in "maximum parallel" form that also allows several executions to be pipelined through the same structure. The procedure produces a set of expressions which are evaluated continuously by a controller to regulate the flow of control through the computation. A computation is initially represented as a sequence of steps, called a program. Each step identifies the source and destination variables for the data values and may contain braching information. Data operations are represented by some unidentified function $\mathcal{F}$. Start and exit steps identify the input and output variables of the program. A sample program is shown below.

```
s: start (A,B)          h: go to (Y)i,j
a: X=F(A)               i: Y=F(Y)
b: T=F(A)                  go to k
c: A=F(B)               j: C=F(B)
d: Y=F(T)               k: Z=F(Y)
e: go to (Y)f,g         l: Z=F(Z,C)
f: T=F(A)               m: W=F(B)
   go to d              t: exit(Z,W)
g: C,Y=F(A,X)
```

The flow of control through the program can be expressed by a control flow expression

$$C=sabcde^{0}fde^{1}g(h^{0}i+h^{1}j)klmt$$

where "+" means OR and superscripts are used to express decision outcome at a k-way branch step. Each loop is represented by listing its steps once. The computation will remain valid as long as the execution of each step is preceded by all previous steps which write data into the source variable(s) and/or read data from the destination variable(s) of that step. An execution is in maximum parallel form iff the steps of the program can be scheduled for execution immediately when the above precedence rule is satisfied. To achieve maximum parallel executions a precedence expression is evaluated for each step. The precedence expression for step c is the Boolean product term $P(c)=s.a.b$ which is obtained examining the control flow expression starting with c and going to the left. Repeated steps are treated in a similar fashion and searching towards the left until either the beginning of the control flow expression or another occurrence of that step is found. Product terms so obtained are OR'ed together, e.g., $P(d)=b+e^{0}.f$. If a loop appears to the left of some step q which must be preceded by a state within the loop then q must be preceded by the entire loop. Thus $P(g)=c.e^{1}$ since g must be preceded by d which is within a loop and e^{1} represents the exit from that loop. Some of the other precedence expressions are $P(h)=de^{1}g$, $P(k)=d.e^{1}.g.(h^{0}.i+h^{1})$. These expressions can be simplified using Boolean algebra and the transitive property of the precedence relationship.

The control flow can be implemented using operators which when "fired" enable execution of a program step and upon its completion sets its own output to logic 1. The operator can be reset when the outputs of the operators it precedes become 1. An operator which is outside a loop but precedes another operator within the loop is reset after the loop terminates. These conditions can be expressed using reset expressions. For example $R(s)=a.b.m$ since the start step s precedes a, b and m, as seen from the simplified precedence expressions for these steps. Also $R(e)=f+g$ and $R(h)=i.1+j.k$ which are obtained taking the sum of the terms obtained for each decision outcome of these k-way branch steps.

For pipelined execution through the program it is necessary to delay resetting the operators until all later steps that read data values from the destination variables of that step have been executed. Such cases can be detected examining the control flow expression from left to right. The resulting delay conditions can be expressed as delay expressions. For example the delay expression for step g is $D(g)=(h^{0}.i+h^{1}).k$. Finally the overall reset conditions are given by $T(q)=R(q).D(q)$ which can be simplified in the usual manner. Some of the final reset expressions for the sample program are $T(s)=(h^{0}+j).m$, $T(b)=a.c.e^{1}$, $T(g)=(i+j).k$.

The control operators should be implemented as bi-stable operators where the operator is set (reset) when its $P(q)$ ($T(q)$) has value 1 and all literals in its $T(q)$ ($P(q)$) have value 0. The particular order in which these expressions are evaluated does not alter the outcome of the computation. Steps scheduled for execution are placed into resource queues which are dynamicly modified according to some priority criteria. Setting the output of a control operator to 1 after the step is executed makes it appear as if its precedence expression has been evaluated just then. This leads to dynamic resource allocation.

References

[1] H.A. Sholl and T.L. Booth, "Software Performance Modeling Using Computation Structures", IEEE TSE (December, 1975), pp. 414-420.

[2] B.I.Dervisoglu, Modeling Maximum Parallel Executions in Pipeline Executable Form Using Precedence Expressions, Univ. Conn., (March 1979), 23pp.

1979 International Conference on Parallel Processing

List of Referees

T. Agerwala
S. Akers
M. S. Anastas
R. L. Anderson
R. S. Arnold
D. Atkins
J. L. Baer
B. Barnes
K. E. Batcher
U. Benerjee
H. Berg
P. B. Berra
D. P. Bhandarkar
S. H. Bokhari
K. W. Bowyer
J. D. Brock
J. T. Cain
D. A. Calahan
W. B. Carson
H. Y Chang
T. C. Chen
Y. Chu
L. A. Conway
G. P. Copeland
G. I. Davida
D. C. Davis
J. B. Dennis
O. I. El-Dessouki
C. S. Ellis
J. Emer
P. H. Enslow
M. W. Evens
T. Y. Feng
W. I. Fletcher
M. J. Flynn
C. C. Foster
M. Freedman
A. D. Friedman
D. P. Friedman
K. S. Fu
D. D. Gajski
O. N. Garcia
S. W. Garrett
J. W. Gault
H. Glass
M. J. Gonzalez, Jr.
K. Gostelow
M. Gouda
S. A. Greibach
M. Hack
S. L. Hakimi
J. P. Hayes
P. Higgins
F. P. Hiner III
G. Holober
E. Horowitz
W. Huen
C. F. Hwang
K. Hwang
O. G. Johnson
H. F. Jordan
J. R. Jump
S. P. Kartashev
R. M. Keller
R. B. Kieburtz
K. H. Kim
W. K. King
D. Klappholz
J. S. Kowalik
A. Krygiel
R. H. Kuhn
G. G. Langdon, Jr.
D. H. Lawrie
A. Liestman
W. C. Liles
G. J. Lipovski
B. Lint
M. T. Liu
P. Look
J. C. Majithia
S. A. Mamrak
K. Mann
P. N. Marinos
D. Marshal
F. P. Mathur
G. W. McDermott
N. H. McDonald
W. C. McDonald
A. Meltzer
G. Mersten
J. M. Michels
E. F. Miller, Jr.
W. L. Miranker
D. Misunas
A. Mukhopadhyay
G. J. Nutt
R. Nutter
E. P. O'Grady
A. E. Oldehoeft
E. I. Organick
D. Padua-Haiek
E. W. Page
A. C. Parker
E. Parrish
J. H. Patel
E. J. Patty
B. Patz
G. R. Peterson
J. L. Peterson
L. J. Peterson
B. Petrasko
D. K. Pradham
C. V. Ramamoorthy
J. Ramanathan
T. R. N. Rao
C. Refnadhas
D. Richards
D. Rine
C. W. Rose
C. Ross
G. E. Rossmann
P. F. Roth
J. Rothstein
J. Rumbaugh
A. Sameh
C. Scheel
P. B. Schneck
C. H. Sequin
B. Shay
J. E. Shemer
H. J. Siegel
D. Siewiorek
T. M. Sigmon
A. Silberschatz
J. B. Slaughter
A. J. Smith
A. D. Snider
S. N. Srihari
E. Stabler
H. S. Stone
M. Stonebraker
N. C. Strole
R. E. Swartwout
G. Tajden
A. Thomasian
K. J. Thurber
W. Troy
K. Trivedi
D. C. VanVoorhis
M. R. Varanasi
M. Vineberg
B. Wah
R. L. Wendt
E. E. Westerfield
D. S. Wise
F. A. Woodworth
C. Wu
S. S. Yau
R. Zaks

1979 International Conference on Parallel Processing

Index of Authors

A DESIGN METHODOLOGY FOR REAL TIME MULTIPLE PROCESSORS SYSTEMS

D. PILAUD, G. SAUCIER

IMAG BP 53X - 38041 GRENOBLE Cedex FRANCE

SUMMARY

A progressive specification method is proposed, since this appears to be the only method currently available for designing a particular software product (set of programs implementing a given application on a particular system). Our aim is to obtain the final product (software, hardware) by a series of specification refinements. Each refinement consists in a transformation from one specification to a more detailed or more precise specification. A refinement may be considered faultless if it does not introduce any design error. To check a specification refinement a mathematical tool or a simulator (the latter giving only partial proof) can be used.

We shall define the initial specifications So as the system representation or model that the designer obtains from the basic information on hand : this model is chosen by the designer. In our practical experience, the initial specifications are obtained in three steps : data graph → data graph labelled by primitives of data acquisition → control monitor, graphically represented by an extent of Petri Nets [PETERSON].

The specifications of a certain level are explained in greater detail by the addition of supplementary information on the system under design, e.g. :

- Temporal information : the designer specifies the implementation time of certain functions ; this means that hardware performance assumptions have been made.
- Architecture : type and number of microprocessors, communications...
- Choice of software : performance of a compiler or interpreter.

The design method consists in making successive reliable refinements from initial specifications. The process of checking that the refining process has not introduced any design error will be called "validation" and will be undertaken at every stage of refinement. A design error is the non-respect of the functional or operational conditions. Validation will generally be achieved by simulation and in special cases, by mathematical analysis.

The first level of validation is called Vo :
On the model described above, it is initially assumed that the hardware resources are unlimited and of infinite power. This means that the function computing time is almost zero. Therefore, the initial aim of the designer is to find errors that are independent from any construction. It is worth noting that the known or suspected "error" situations are defined by the designer. In our approach, the following situations were detected by a simulator :

- evolution cycle of infinite duration,
- locking,
- final state of an evolution cycle depending on the order of transition scruting.

Also coming into this category are the data frequency incoherencies which may be detected by analysis or by simulation.

At the next level of specification S_1 and validation V_1, a performance time is associated with each function, this time being correlated to the choice of a type of microprocessor, and the system is represented by a time-dependent model. The validation can, for example, detect the following :
a) Work loads incompatible with the system : consequently, the designer will have to consider another type of processor, or spread this task over several processors, or again use several processors in parallel. After modification, the validation process (simulation) is repeated. This validation can be also carried out algebraically.
b) Observing response times : in addition to simulation, the validation can again be of the analytical type.
c) Data incoherency : in real time systems, certain output functions depend on external inputs with the following restriction : these inputs must be coherent, i.e. sampled at the same time (same copy). For example, the data sampled at a given moment could all be of the same colour ; the validation condition (simulation) consists in checking that a function consumes only tokens of the same colour.

At the next level of specification S_2 and validation V_2, functions associated with places are defined by their input variables, their output variables and work on local variables. The main validation consists in checking functions and/or predicates calculated in parallel and sharing the same variables. The designer may consider that this is a design error (tasks should only work on local variables) ; the designer may also break up the function and specify (explain) the synchronisation (priority) selected.

The final level of specification S_3 consists in the choice of an architecture : this is a fundamental step in the refinement process since the selected architecture has to be taken into account accurately and the corresponding parameters have to be entered into the model. We shall consider two fundamental choices of architecture (one solution may consist in these two types of choice).

Architecture with specific processor : A physical resource (processor) performs either the monitoring operation (control monitor) or the functions of the system. In this case, an interpretor for this type of specifications (MAS system [3]) has been implemented on a multimicroprocessor system "4M" [5] : one microprocessor (among the four ones) is specifically allocated to the execution of the control monitor evolution.

Decentralized architecture : Some of the physical processors perform both control tasks and functions. The initial net is, for instance, partitionned into connected parts ; every sub-net is associated with a physical processor which performs both the control evolution and the functions computation. This means that a control monitor is stored in the RAM of each microprocessor. This approach was industrially applied for an aircraft system ; the obtained net was made up of four microcomputers which were linked by private communications.
The validation V_3, by means of simulation, consists in checking that the anomalies, checked in the previous steps, do not appear in this more precise description. In addition to the functional anomalies (locking, conflict), we check that the operational constraints (response times, for instance) are still true when ressource restrictions are introduced ; for instance, an upper bound of the task queue can be computed, in the second type architecture and we verify that the system hold these constraints.

The proposed approach is an up-down one which in our sense minimizes the likelihood of design errors. It is obtained by using a general method of successive refinements from initial specifications to final ones.

REFERENCES

[1] J.L. PETERSON : "PETRI Nets", ACM Computing survey, Sept 1977, pp. 223-252.

[2] C. BELLON, G. SAUCIER : "Test, contamination et reprise dans les systèmes distribués à haute disponibilité", RR n° 129, ENSIMAG, aout 1978.

[3] M. ZACHARIADES : "MAS : Réalisation d'un langage d'aide à la description des systèmes logiques", thèse de 3e cycle, sept. 1977, Université de Grenoble.

[4] D. PILAUD, G. SAUCIER : "Conception de systèmes temps réels à très haute sécurité sur microprocesseurs",RR n° 130 ENSIMAG, aout 1978.

[5] HABANNAKEH - MIDANI .H :" onception et réalisation d'une architecture multi-microprocesseur flexible ; application au contrôle de processus industriel", thèse de 3e cycle, Grenoble University, may 1979.