October 1987

Report No. STAN-CS-87-1188
Also Numbered KSL-87-61

2

Experiments with a Knowledge-Based System on a Multiprocessor

by

Russell Nakano and Masafumi Minami

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory
Report No. KSL 87-61

October 1987

Experiments with a Knowledge-Based System on a Multiprocessor

by

Russell Nakano[†] and Masafumi Minami

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

[†]WORKSTATION SYSTEMS ENGINEERING
Digital Equipment Corporation
Palo Alto, California 94301

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, Boeing Contract W266875, and the Workstation Systems Engineering group of Digital Equipment Corporation.

the authors

.their they

Abstract

This paper documents the results ~~we~~ obtained and the lessons ~~we~~ learned in the design, implementation, and execution of a simulated real-time application on a simulated parallel processor. Specifically, ~~our~~ parallel program ran 100 times faster on a 100-processor multiprocessor compared to a 1-processor multiprocessor.

The machine architecture is a distributed-memory multiprocessor. The target machine consists of 10 to 1000 processors, but because of simulator limitations, we ran simulations of machines consisting of 1 to 100 processors. Each processor is a computer with its own local memory, executing an independent instruction stream. There is no global shared memory; all processes communicate by message passing. The target programming environment, called Lamina, encourages a programming style that stresses performance gains through problem decomposition, allowing many processors to be brought to bear on a problem. The key is to distribute the processing load over replicated objects, and to increase throughput by building pipelined sequences of objects that handle stages of problem solving.

The authors were confirmed: this

~~We~~ focused on a knowledge-based application that simulates real-time understanding of radar tracks, called Airtrac. This paper describes a portion of the Airtrac application implemented in Lamina and a set of experiments that we performed. ~~We confirmed the~~ following hypotheses: 1) Performance of ~~our~~ concurrent program improves with additional processors, and thereby attains a significant level of speedup. 2) Correctness of ~~our~~ concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions. (KR)

the

Table of Contents

List of Figures

List of Tables

1. Introduction

This paper focuses on the problems confronting the programmer of a concurrent program that runs on a distributed memory multiprocessor. The primary objective of our experiments is to obtain speedup from parallelism without compromising correctness. Specifically, our parallel program ran 100 times faster on a 100-processor multiprocessor compared to a 1-processor multiprocessor. The goal of this paper is to explain why we made certain design choices and how those choices influence our result.

A major theme in our work is the tradeoff between speedup and correctness. We attempt to obtain speedup by decomposing our problem to allow many sub-problems to be solved concurrently. This requires deciding how to partition the data structures and procedures for concurrent execution. We take care in decomposing our problem; to a first approximation, more decomposition allows more concurrency and therefore greater speedup. At the same time, decomposition increases the interactions and dependencies between the sub-problems and makes the task of obtaining a correct solution more difficult.

This paper focuses on the implementation of a knowledge-based expert system in a concurrent object-oriented programming paradigm called Lamina [Delagi 87a]. The target is a distributed-memory machine consisting of 10 to 1000 processors, but because of simulator limitations, our simulations examine 1 to 100 processors. Each processor is a computer with a local memory and an independent instruction stream.[1] There is no global shared memory of any kind.

Airtrac is a knowledge-based application that simulates real-time understanding of radar tracks. This paper describes a portion of the Airtrac application implemented in Lamina and a set of experiments that we performed. We encoded and implemented the knowledge from the domain of real-time radar track interpretation for execution on a distributed-memory message-passing multiprocessor system. Our goal was to achieve a significant level of problem-solving speedup by techniques that exploited both the characteristics of our simulated parallel machine, as well as the parallelism available in our problem domain.

The remainder of this paper is organized as follows. Section 2 introduces definitions that we use throughout the paper. Section 3 describes the model of the parallel machine that we simulate, and the model of computation from the viewpoint of the programmer. Section 4 outlines a set of principles that we follow in our programming effort in order to shed light on why we take the approach that we do. Section 5 describes the signal understanding problem that our parallel program addresses. Section 6 describes the design of our experiments, and Section 7 presents the results. Section 8 discusses a number of design issues, and Section 9 summarizes the paper.

[1]Each processor is roughly comparable to a 32-bit microprocessor-based system equipped with a multitasking kernel that supports interprocessor communication and restartable processes (as opposed to resumable processes). The hardware system is assumed to support high-bandwidth, low-latency interprocessor communications as described in Byrd et.al. [Byrd 87].

2. Definitions

Using the definitions of Andrews and Schneider [Andrews 83], a *sequential program* specifies sequential execution of a list of statements; its execution is called a *process*. A *concurrent program* specifies two or more sequential programs that may be executed concurrently as *parallel processes*.

We define $S_{n,m}$ *speedup* as the ratio $\frac{T_m}{T_n}$, where T_k denotes the time for a given task to be completed on a k-processor multiprocessor. Both T_m and T_n represent the *same* concurrent program running on m-processor and n-processor multiprocessors, respectively. When we compare an n-processor multiprocessor to a 1-processor multiprocessor, we obtain a measure for $S_{n/1}$ speedup, which should be distinguished from *true speedup*, defined as the ratio $\frac{T^*}{T_n}$, where T^* denotes the time for a given task to completed by the *best* implementation possible on a uniprocessor.[2] In particular, T^* excludes overhead tasks (e.g. message-passing, synchronization, etc.) that T_1 counts.

We define *correctness* to be the degree to which a concurrent program executing on a k-processor multiprocessor obtains the same solution as a conventional uniprocessor-based sequential program embodying the same knowledge as contained in the concurrent program. We call the latter solution a *reference solution*. We use a serial version of our system to generate a reference solution, to evaluate the correctness of the parallel implementation.[3]

MacLennan [MacLennan 82] distinguishes between value-oriented and object-oriented programming styles. A *value* has the following properties:

- A value is read-only.
- A value is atemporal (i.e. timeless and unchanging).
- A value exhibits referential transparency, that is, there is never the danger of one expression altering something used by another expression.

These properties make values extremely attractive for concurrent programs. Values are immutable and may be read by many processes, either directly or through "copies" of values that are equal; this facilitates the achievement of correctness as well as concurrency. A well-known example of value-oriented programming is functional programming [Henderson 80]. Other examples of value-oriented programming in the realm of parallel computing include systolic programs [Kung 82] and scalar data flow programs [Arvind 83, Dennis 85], where the data flowing from processor to processor may be viewed as values that represent abstractions of various intermediate problem-solving stages.

[2]A 1-processor multiprocessor executes the same parallel program that runs on a n-processor multiprocessor. In particular, it creates processes that communicate by sending messages, as opposed to sharing a common memory.

[3]Unfortunately, our reference program is not a valid producer of T^* estimates, and we cannot use it to obtain true speedup estimates. Project resource limitations prevented us from developing an optimized program to serve as a *best* serial implementation.

In contrast, MacLennan defines *objects* in computer programming to have one or more of the following properties:

- An object may be created and destroyed.
- An object has state.
- An object may be changed.
- An object may be shared.

Computer programs often simulate some physical or logical situation, where objects represent the entities in the simulated domain. For example, a record in an employee database corresponds to an employee. An entry in a symbol table corresponds to a variable in the source text of a program. Variables in most high-level programming languages represent objects. Objects provide an abstraction of the state of physical or logical entities, and reflect changes that those entities undergo during the simulation. These properties make objects particularly useful and attractive to a programmer.

Objects in a concurrent program introduce complications. In particular, many parallel processes may attempt to create, destroy, change, or share an object, thereby causing potential problems. For instance, one process may read an object, perform a computation, and change the object. Another process may concurrently perform a similar sequence of actions on the same object, leading to the possibility that operations may interleave, and render the state of the object inconsistent. Many solutions have been proposed, including semaphores, conditional critical regions. and monitors; all of these techniques strive to achieve correctness and involve some loss of concurrency.

Our programming paradigm, Lamina, supports a variation of *monitors*, defined as a collection of permanent variables (we use the term *instance variables*), used to store a resource's state, and some procedures, which implement a set of allowed operations on the resource [Andrews 83]. Although monitors provide mutual exclusion, concurrency considerations force us to abandon mutual exclusion as the sole technique to obtain correctness.

We classify techniques for obtaining speedup in problem-solving into two categories: replication and pipelining. *Replication* is defined as the decomposition of a problem or sub-problem into many independent or partially independent sub-problems that may be concurrently processed. *Pipelining* is defined as the decomposition of a problem or sub-problem into a sequence of operations that may be performed by successive stages of a processing pipeline. The output of one stage is the input to the next stage.

3. Computational model

3.1. Machine model

Our machine architecture, referred to as CARE [Delagi 87a], may be modeled as an asynchronous message-passing distributed system with reliable datagram service [Tanenbaum 81]. After sending a message, a process may continue to execute (i.e. message passing is asynchronous). Arrival order of messages may differ from the order in which they were sent (i.e. datagram as opposed to virtual circuit). The network guarantees that no message is ever lost (i.e. reliable), although it does not guarantee when a message will arrive. Each processor within the distributed system is a computer that supports interprocessor communication and restartable processes. Each processor operates on its own instruction stream, asynchronously with respect to other processors.

In synchronous message passing, maintaining consistent state between communicating processes is simplified because the sender blocks until the message is received, giving implicit synchronization at the send and receive points. For example, the receiver may correctly make inferences about the sender's program state from the contents of the message it has received, without the possibility that the sender program continued to execute, possibly negating a condition that held at the time the original message was sent.

In asynchronous message passing, the sender continues to execute after sending a message. This has the advantage of introducing more concurrency, which holds the promise of additional speedup. Unfortunately, in its pure form, asynchronous message passing allows the sender to get arbitrarily far ahead of the receiver. This means that the contents of the message reflects the state of the sender at the time the message was sent, which may not necessarily be true at the time the message is received. This consideration makes the maintenance of consistent state across processes difficult, and is discussed more fully in Section 4.

3.2. Programmer model

Our programming paradigm, Lamina, provides language constructs that allows us to exploit the distributed memory machine architecture described earlier [Delagi 87b]. In particular, we focused our programming efforts on the concurrent object-oriented programming model that Lamina provides. As in other object-oriented programming systems, objects encapsulate state information as instance variables. Instance variables may be accessed and manipulated only through methods. Methods are invoked by message-passing.

However, despite the apparent similarity with conventional object-oriented systems, programming within Lamina has fundamental differences:

- Concurrent processes may execute during both object creation and message sending.
- The time required to create an object is visible to the programmer.
- The time required to send a message is visible to the programmer.
- Messages may be received in a different order from which they were sent.

These differences reflect the strong emphasis Lamina places on concurrency. While all object-oriented systems encounter delays in object creation and message sending, these delays are significant within the Lamina paradigm because of the other activities that may proceed concurrently *during* these periods. Subtle and not-so-subtle problems become apparent when concurrent processes communicate, whether to send a message or to create a new object. For instance, a process might detect that a particular condition holds, and respond by sending a message to another process. But because processes continue to execute during message sending, the condition may no longer hold when the message is received. This example illustrates a situation where the recipient of the message cannot correctly assume that because the sender responds to a particular condition by sending a message, that the condition still holds when the message is received.

Regarding message ordering, partly as a result of our experimentation, versions of Lamina subsequent to the one we used provide the ability for the programmer to specify that messages be handled by the receiver in the same order that they were sent [Delagi 87c]. Use of this feature imposes a performance penalty, which places a responsibility on the programmer to determine that message ordering is truly warranted. In the Airtrac application, we believed that ordering was necessary and imposed it through application level routines that examined message sequence numbers (time tags) and queued messages for which all predecessors had not already been handled.

In Lamina, an object is a process. Following the definition of a process provided earlier, we make no commitment to whether a process has a unique virtual address space associated with it. Each object has a top-level dispatch process that accepts incoming messages and invokes the appropriate message handler; otherwise, if there is no available message, the process blocks. Sending a message to an object corresponds to asynchronous message-passing at the machine level. A method executes atomically. Since each object has a single process, and only that process has access to the internal state (instance variables), mutual exclusion is assured. An object and its methods effectively constitute a non-nested monitor.

Our problem-solving approach has evolved from the blackboard model, where nodes on the blackboard form the basic data objects, and knowledge sources consisting of rules are applied to transform nodes (i.e. objects) and create new nodes [Nii 86a, Nii 86b]. Brown et. al. used concepts from the blackboard model to implement a signal-interpretation application on the CARE multiprocessor simulator [Brown 86]. Lamina evolved from the experiences from that effort. In addition, lessons learned in that earlier effort have been incorporated into our work, including the use of replication and pipelining to gain performance, and improving efficiency and correctness by enforcing a degree of consistency control over many agents computing concurrently.

4. Design principles

Lamina represents a programming philosophy that relies on the concepts of replication and pipelining to achieve speedup on parallel hardware. The key to successful application of these principles relies on finding an appropriate problem decomposition that exploits concurrent execution with minimal dependency between replicated or pipelined processing elements.

The price of concurrency and speedup is the cost of maintaining consistency among objects. When writing a sequential program, a programmer automatically gains mutual exclusion between read/write operations on data structures. This follows directly from the fact that a sequential program has only a single process; a single process has sole control over reads and writes to a variable, for instance. This convenience vanishes when the programmer writes a concurrent program. Since a concurrent program has many concurrently executing processes, coordinating the activities of the processes becomes a significant task.

In this section, we develop the concept of a dependence graph program to provide a framework in which tradeoffs between alternate problem decompositions may be examined. Choosing a decomposition that admits high concurrency gives speedup, but it may do so with the expense of higher effort in maintaining consistency. We introduce dependence graph programs to make the tradeoffs more explicit.

4.1. Speedup

Researchers have debated how much speedup is obtainable on parallel hardware, on both theoretical and empirical grounds; Kruskal has surveyed this area [Kruskal 85]. We take the empirical approach because our goal is to test ideas about parallel problem solving using multiprocessor architectures. Our thinking is guided, however, by a number of principles describing how to decompose problems to obtain speedup.

4.1.1. Pipelining

Consider a concurrent program consisting of three cooperating processes: Reader, Executor, and Printer. The Reader process obtains a line consisting of characters from an input source, sends it to the Executor process, and then repeats this loop. The Executor performs a similar function, receiving a line from the Reader, processing it in some way, and sending it to the Printer. The Printer receives lines from the Executor, and prints out the line. These processes cooperate to form a pipeline; see Figure 1. By using asynchronous message passing, we obtain concurrent operation of the processes; for instance, the Printer may be working on one line, while the Executor is working on another. This means that by assigning each process to a different processor, we can obtain speedup, despite the fact that each line must be inputted, processed, and output sequentially. By overlapping the operations we can achieve a higher throughput than is possible with a single process performing all three tasks.

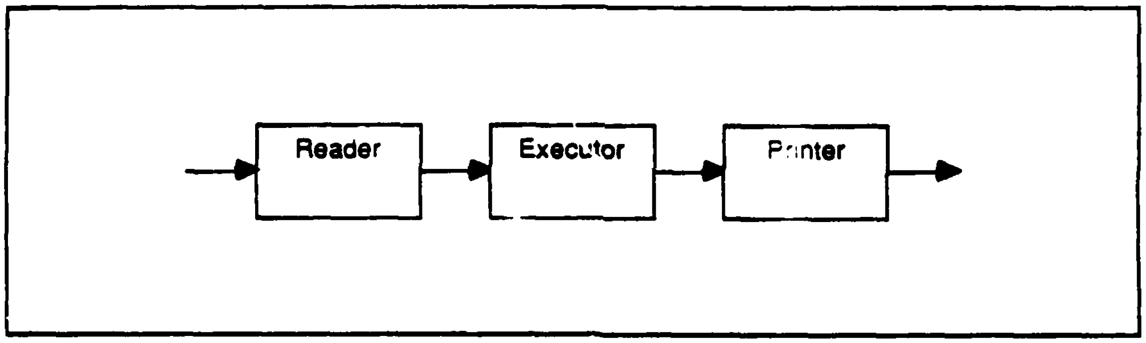

Figure 1. Decomposing a problem to obtain pipeline speedup.

By decomposing a problem in sequential stages, we can obtain speedup from pipelining.

4.1.2. Replication

Consider a variation of Reader-Executor-Printer problem. Suppose that we are able to achieve some overlap in the operations, but we discover that the Executor stage consistently takes longer than the other stages. This causes the Printer to be continually starved for data, while the Reader completes its task quickly and spends most of its time idle. We can improve the overall throughput by replicating the function of the Executor stage by creating many Executors. See Figure 2. By increasing the number of processes performing a given function, we do not reduce the time it takes a single Executor to perform its function, but we allow many lines to be processed concurrently, improving the utilization of the Reader and Printer processes, and boosting overall throughput. This principle of replicating a stage applies equally well if the Reader or the Printer is the bottleneck.

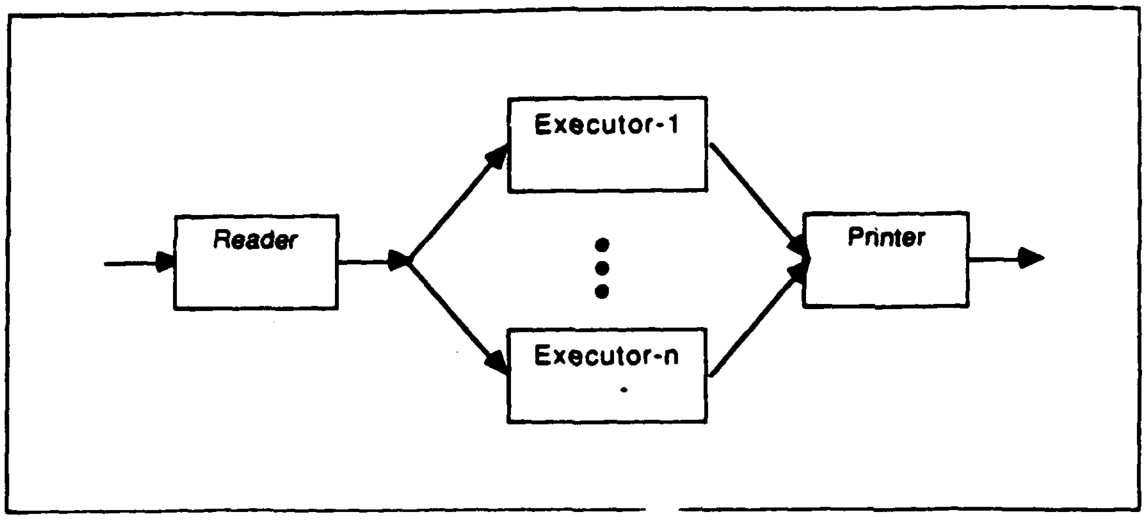

Figure 2. Decomposing a problem to obtain replication speedup.

By duplicating identical problem solving stages, we can obtain speedup from replication.

4.2. Correctness

4.2.1. Consistency

In order to achieve speedup from parallelism, we decompose a problem into smaller sub-problems, where each sub-problem is represented by an object. By doing this, we lose the luxury of mutual exclusion between the sub-problems because of interactions and dependencies that typically exist between sub-parts of a problem. For example, in the Reader-Executor-Printer problem, the simplest version is where a line may be operated upon by one process truly independently; we might want to perform ASCII to EBCDIC character conversion of each line, for instance. We organize the problem solving so that the Reader assembles fixed-length text strings, the Executor performs the conversion, and the Printer does output duties. This problem is well-suited to speedup from the simple pipeline parallelism illustrated in Figure 1. In MacLennan's value/object terminology, a "fixed-length text string" may be viewed as a value that represents the i-th line in the input text; the text string is read-only and it is atemporal. The trick is to :w the ASCII and EBCDIC versions of a text strings as different *values* corresponding to the i-th line; the Executor's role is to take in ASCII values and transform them into EBCDIC values of the same line. As we will see, value passing has desirable properties in concurrent message-passing systems.

In a more complicated example, we might want to perform text compression by encoding words according to their frequency of appearance, where the Reader process counts the appearance of words and the Executor assigns words to a variable length output symbol set. The frequency table is a source of trouble; it is an object which the Reader writes and updates, and which the Executor reads. Unfortunately, the semantics we impose on the text compression task requires that the Reader complete its scan of the input text before the Executor can begin its encoding task. This dependency prevents us from exploiting pipeline parallelism.

As another example, we might want to compile a high-level language source program text (e.g. Pascal, Lisp, C) into assembly code. Suppose we allow the Reader to build a symbol table for functions and variables, and we let the Executor parse the tokenized output from the Reader, while the Printer outputs assembly code from the Executor's syntax graph structures. In the scheme outlined here, the symbol table resides with the Reader, so whenever the Executor or Printer needs to access or update the symbol table, it must send a message to the Reader. Consistency becomes an important issue within this setup. For instance, suppose that the Executor determines on the basis of its parse, that the variable x has been declared global. Within a procedure, a local variable also named x is defined, which requires that expressions referring to x within this procedure use a local storage location. Suppose the end of the procedure is encountered, and since we want all subsequent occurrences to x to refer to the global location, the Executor marks the entry for x accordingly (via a message to the Reader). When the Printer sees a reference to x, it consults the symbol table (via a message to the Reader) to determine which storage location should be used; if by misfortune the Printer happens to be handling an expression within the procedure containing the local x, and the symbol table has already been updated, incorrect code will be generated. The essential point is that the symbol table is an object; as we defined earlier, it is shared by several parallel processes, and it changes. A number of fixes are possible, including distinguishing variables by the procedure they are occur within, but this example illustrates that the presence of objects in concurrent program raises a need to deal with consistency.

Consistency is the property that some invariant condition or conditions describing correct behavior of a program holds over all objects in all parallel processes. This is typically difficult to achieve in a concurrent program, since the program itself consists of a sequential list of statements for each individual process or object, while consistency applies to an ensemble of objects. The field of distributed systems focuses on difficulties arising from consistency maintenance [Cornafion 85, Weihl 85, Filman 84]. Smith [Smith 81] refers to this programming goal as the development of a problem-solving protocol.

The work of Schlichting and Schneider [Schlichting 83] is particularly relevant for our situation: they study partial correctness properties of unreliable datagram asynchronous message-passing distributed systems from an axiomatic point of view. They describe a number of sufficient conditions for partial correctness on an asynchronous distributed system:

- monotonic predicates,

- predicate transfer with acknowledgements.

An predicate is *monotonic* if once it becomes true, it remains so. For example, if the Reader process maintains a count of the lines in the variable `totalLines`, and it encounters the last line in the input text, as well having seen all previous lines, then it might send the predicate P, "`totalLines = 16`," to the Executor and to the Printer. The Printer process might use this information even before it has received all the lines, to check if sufficient resources exist to complete the job, for instance. Intuitively, it is valid to assert the total number of lines in the input text because that fact remains unchanged (assuming the input text remains fixed for the duration of the job). Formally, the Reader maintains the following invariant condition on the predicate P:

Invariant: "message not sent" or "P is true"

In contrast, an assertion that the current line is 12, as in "`currentLine = 12`," changes as each line is processed by the Reader. The monotonic criterion cannot be used to guarantee the correctness of this assertion.

A technique to achieve correctness without monotonic predicates is to use acknowledgements. The idea is to require the sender to maintain the truth condition of a predicate or assertion until an acknowledgement from the receiver returns. In the Reader-Executor-Printer example, the Reader follows the convention that once it asserts "`currentLine = 12`," it will refrain from further actions that would violate this fact until it receives an acknowledgement from the Executor. This protocol allows the Executor to perform internal processing, queries to the Reader, and updates to the Reader, all with the assurance that the current line will remain unchanged until the Executor acknowledges the assertion, thereby signalling that the Reader may proceed to change the assertion. Formally, the Reader and Executor maintain the following invariant condition on the predicate P:

Invariant: "message not sent" or "P is true" or "acknowledgement received"

Note that the each technique has drawbacks, despite their guarantees of correctness. For the monotonic predicate technique, the challenge is to define a problem decomposition and solution protocol for which monotonic predicates are meaningful. In particular, if a problem *decomposition* truly allows transfer of values between processes, then by the semantics of values as we have defined them, values are automatically monotonic. This explains in formal terms why a "data flow" problem decomposition that passes values avoids difficult problems related to consistency. For the predicate acknowledgement technique, we may address problems that do not cleanly admit monotonic predicates, but we lose concurrency in the assert-acknowledge cycle. Less concurrency tends to translate into less speedup. In the worst case, we may lose so much concurrency in the assert-acknowledge cycle that we find that we have spent our efforts in decomposing the problem into sub-problems only to discover that our concurrent program performs no faster than an equivalent sequential program!

Throughout the design process, we are motivated by a desire to obtain the highest possible performance while maintaining correctness. For tasks in the problem whose durations impact the performance measures, we take the approach of looking first for problem decompositions that allow either value-passing or monotonic predicate protocols. Where neither of these are possible, we implement predicate acknowledgement protocols. In the implementation of Airtrac-Lamina, we did not have to resort to heuristic schemes that did not guarantee correctness.

For initialization tasks, the time to *perform initialization* tasks (e.g. creating manager objects and distributing lookup tables) is not counted in the performance metrics, but correctness is paramount. Since initialization requires the establishment of a consistent beginning state over many objects, we use the predicate acknowledgement technique to have objects initialize their internal state based on information contained in an initialization message, and then signal their readiness to proceed by responding with an acknowledgement message.

4.2.2. Mutual exclusion

Lamina objects are encapsulations of data, together with methods that manipulate the data. They constitute monitors which provide mutual exclusion over the resources they encapsulate. These monitors are "non-nested" because when a Lamina method (i.e. message handler) in the current CARE implementation invokes another Lamina method, it does so by asynchronous message passing (where the sender continues executing after the message is sent), thereby losing the mutual exclusion required for nested monitor calls. In return, Lamina gains opportunities to increase concurrency by pipelining sequences of operations.

Within the restriction of non-nested monitor calls, the programmer may use Lamina monitors to define atomic operations. If correctness were the sole concern, the programmer could develop the entire problem solution within a single method on a single object; but this is an extreme case. The entire enterprise of designing programs for multiprocessors is motivated by a desire for speedup, and monitors provide a base level of mutual exclusion from which a correct concurrent program may be constructed.

The critical design task is to determine the data structures and methods which deserve the atomicity that monitors provide. The choice is far from obvious. For example, in the ASCII-to-EBCDIC translator example, we assumed the Executor process sequentially scanning through the string, translating one character at a time. We see that the translation of each character may be performed independently, so a finer-grained problem decomposition is to have many Executor processes, each translating a section of the text line. In the extreme, we can arrange for each character to be translated by one of many replicated Executor processes. Choosing the best decomposition is a function of the relative costs of performing the character translation versus the overhead associated with partitioning the line, sending messages, and reassembling the translated text fragments (in the correct order!). The answer depends on specific machine performance parameters and the type of task involved, which in our example is the very simple job of character translation, but might in general be a time-consuming operation. Unfortunately, the programmer often lacks the specific performance figures on which to base such decisions, and must choose a decomposition based on subjective assessments of the complexity of the task at hand, weighed against the perceived run-time overhead of decomposition, together with the run-time worries associated with consistency maintenance. On the issue of how to choose the best "grain-size" for problem solving, we can offer no specific guidance. However, since the CARE-Lamina simulator is heavily instrumented, it lets the programmer observe the relative amount of time spent in actual computation versus overhead activities.

In addition to providing mutual exclusion, Lamina also encourages the structured programming style that results from the use of objects and methods. In particular, mutual exclusion may be exploited without necessarily building large, monolithic objects and methods that might reflect poor software engineering practice. Since Lamina itself is built on Zetalisp's Flavors system [Weinreb 80], it is easy for the programmer to define object "flavors" with instance variables and associated methods to be atomically executed within a Lamina monitor. This can provide important benefits of modularity and structure to the software engineering effort.

To summarize, Lamina objects and methods may be viewed as non-nested monitor constructs that provide the programmer with a base level of mutual exclusion. The potential for additional concurrency and problem-solving speedup increases as finer decompositions of data and methods are adopted. However, these benefits must be weighed against the difficulties of maintaining consistency between objects in a concurrent program. Two techniques for maintaining consistency have been described, differing in their applicability and impact on concurrency.

4.3. Dependence graph programs

The previous sections have defined concepts relevant to the dual goals of achieving speedup and correctness. This section builds upon those concepts to provide a framework in which tradeoffs between speedup and correctness may be examined. A *dependence graph program* is an abstract representation of a solution to a given problem in which values flow between nodes in a directed graph, where each node applies a function to the values arriving on its incoming edges and sends out a value on zero or more outgoing edges. The edges correspond to the dependencies which exist between the functions [Arvind 83]. A *pure* dependence graph program is one in which the functions on the nodes are free from side effects; in particular, a pure dependence graph program prohibits a function from saving state on any node. (Note that this definition does not preclude a system-level program on a node from handling a function f(X,Y) by saving the value of X if the value of X arrives before the value for Y. Strictly speaking, an implementation of an f function node must save state, but this state is invisible to the programmer.) A *hybrid* dependence graph program is one in which one or more nodes save state in the form of local instance variables on the node. Functions have access to those instance variables.

Gajski et. al. [Gajski 82] summarize the principles underlying pure data flow computation:

- asynchrony

- functionality.

Asynchrony means that all operations are executed when and only when the required operands are available. *Functionality* means that all operations are functions, that is, there are no side effects.

Pure dependence graph programs have two desirable properties. First, consistency is guaranteed by design. As we have defined it, there are only values and transformations applied to those values. There are no objects to cause inconsistency problems. Second, we can theoretically achieve the maximal amount of parallelism in the solution, and if we ignore overhead costs, maximize speedup in overall performance. This follows from the asynchrony principle, which means that in the ideal case we can arrange for each computation on a node to proceed as soon as all values on the incoming edges are available.

Hybrid dependence graph programs allow side effects to instance variables on nodes, thereby making it more convenient and straightforward to perform certain operations, especially those associated with lookup and matching. This immediately introduces objects into the computational model, and raises the usual concerns about consistency and correctness.

We will use dependence graph programs to serve two purposes. First, we depict the dependencies contained within a problem. Second, we explain why we made certain design decisions in solving the Airtrac problem; in particular, we show why we impose certain consistency requirements on the problem solving protocol. A dependence graph serves as an abstract representation of a problem solution, rather than a blueprint for actual implementation. Specifically, we want to avoid the pitfall of using a dependence graph program to dictate the actual problem decomposition. Overhead delays associated with message routing/sending and process invocation degrade speedup from the theoretical ideal if the actual implementation chooses to decompose the problem down to the grain-size typically found in a dependence graph representation. Given an arithmetic expression, for instance, it may not be desirable to define the grain-size of primitive operations at the level of add, subtract, and multiply. This may lead to the undesirable situation where excessive overhead time is consumed in message packing, tagging, routing, packing, matching, unpacking, and so forth, only to support a simple add operation.

Consider the following numerical example from Gajski et. al. [Gajski 82]. The pseudo-code representation of the problem is as follows:

```
input d,e,f
   $c_0 = 0$
for i from 1 to 8 do
   begin
      $a_i = d_i / e_i$
      $b_i = a_i * f_i$
      $c_i = b_i + c_{i-1}$
   end
output a,b,c
```

One possible dependence graph program for this problem is shown in Figure 3. This is the same graph presented by Gajski et. al. They assume that division takes three processing units, multiplication takes two units, and addition takes one unit. As noted in their paper, the critical path is the computational sequence a_1, b_1, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8; the lower bound on the execution time is 13 time units.

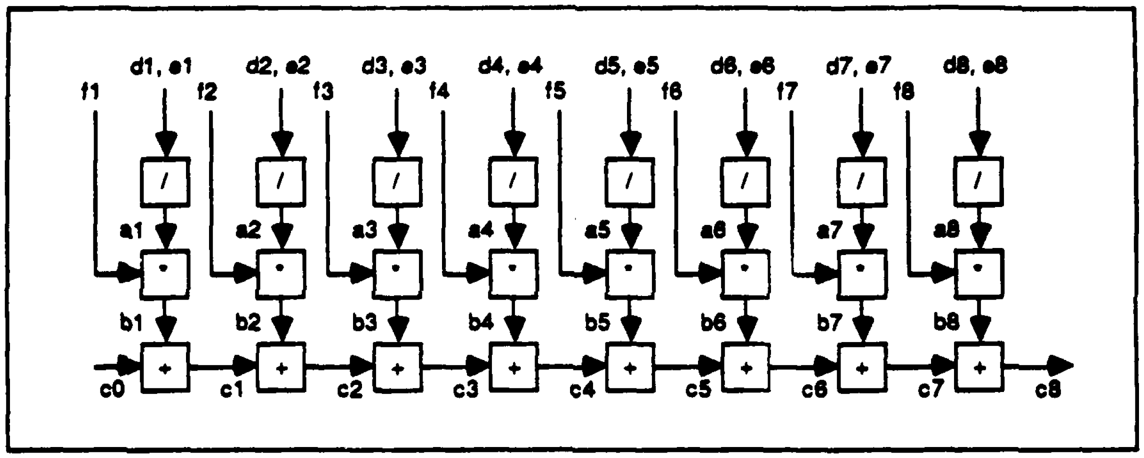

Figure 3. A dependence graph program for a simple numerical computation.

A possible concurrent program implementation would be to assign eight processes to compute the quantities $b_1,...,b_8$, and a ninth to combine the b_i and output $c_1,...,c_8$. Such an arrangement maximizes the decomposition of the problem into sub-problems that may run concurrently, while minimizing the communication overhead. For instance, there is no loss in combining the computation of $c_1,...,c_8$ into a single process because of the inherently serial nature of this particular computation.

Another concurrent program might choose a slightly different decomposition and partition the computation of $c_1,...,c_8$ into, say, three processes: c_1-c_2-c_3, c_4-c_5-c_6, and c_7-c_8. This arrangement uses 11 processes versus the 9 processes in the previous example. While this leads to no improvement in the lower bound of 13 time units for a single computation with d, e, and f, it shows an improvement with repeated computations with different values of the input arrays, d, e, and f. For instance, this allows one computation to be summing on the c_7-c_8 process while another is summing on the c_4-c_5-c_6 process. Depending on the complexity of the computation relative to the overhead costs, it might even be worthwhile to define one process for each of the $c_1,...,c_8$, giving 16 processes overall. This illustrates two points. First, a strictly sequential computation gives an opportunity for pipeline concurrency if many such computations are required. Second, given a dependency graph, many possible problem decompositions are possible.

Gajski et. al. also present a different dependence graph program that is optimized to eliminate the "ripple" summation chain by a more efficient summation network. The dependence graph program for this scheme is shown in Figures 4 and 5. Figure 4 is the "top-level" definition of the program. We use the *convention* of using a single box, `optimized summation`, in Figure 4 to represent the subgraph that performs the more efficient summation. Figure 5 shows the expansion of that box as a graph. Showing a dependence graph program in this way is merely a convenience; one should envision the subgraphs in their fully expanded form in the top-level dependence program definition.

The associative property of addition is used to derive the optimized summation function. For instance, the computation of c_8 is rewritten as follows:

$$\begin{aligned} c_8 &= (((((((c0 + b_1) + b_2) + b_3) + b_4) + b_5) + b_6) + b_7) + b_8) \\ &= (c0 + ((b_1 + b_2) + (b_3 + b_4))) + ((b_5 + b_6) + (b_7 + b_8)) \end{aligned}$$

By regrouping the addition operations, this dependence graph program has more parallelism, and reduces the lower bound on execution time from 13 to 9 execution time units. It is important to realize that the second program is truly different from the first; it cannot be obtained from the first by graph transformations or syntactic manipulations that do not rely on the semantics of the functions on the nodes.

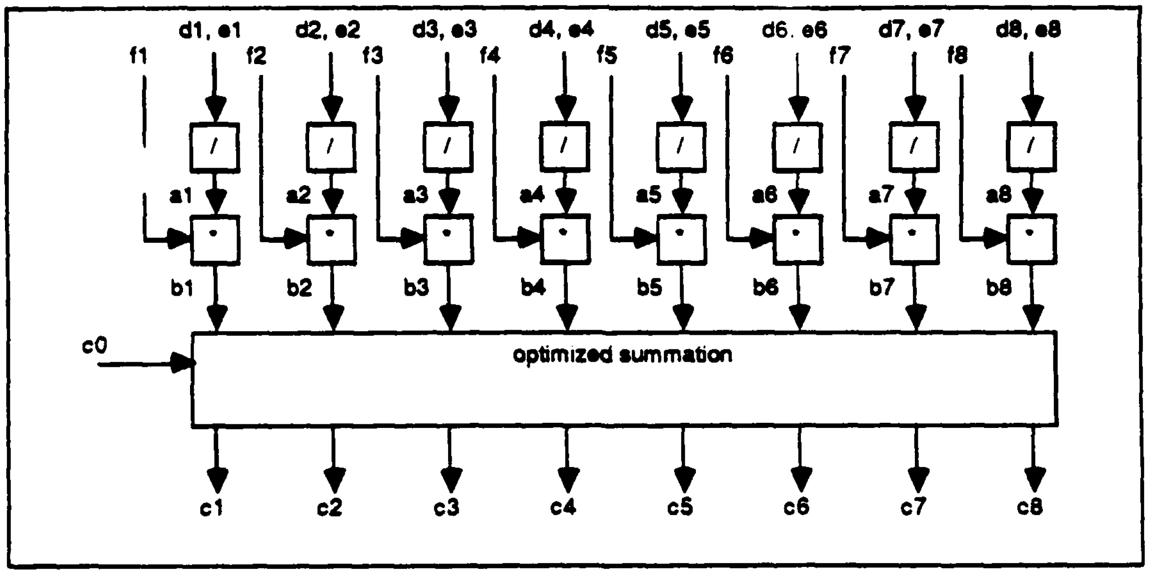

Figure 4. A dependence graph program for the simple numerical computation.

This uses optimization of the recurrence relation using the associative property of addition. This represents the "top-level" definition of the solution. The optimized summation subgraph is shown here a single box, and is shown in expanded form in Figure 5.

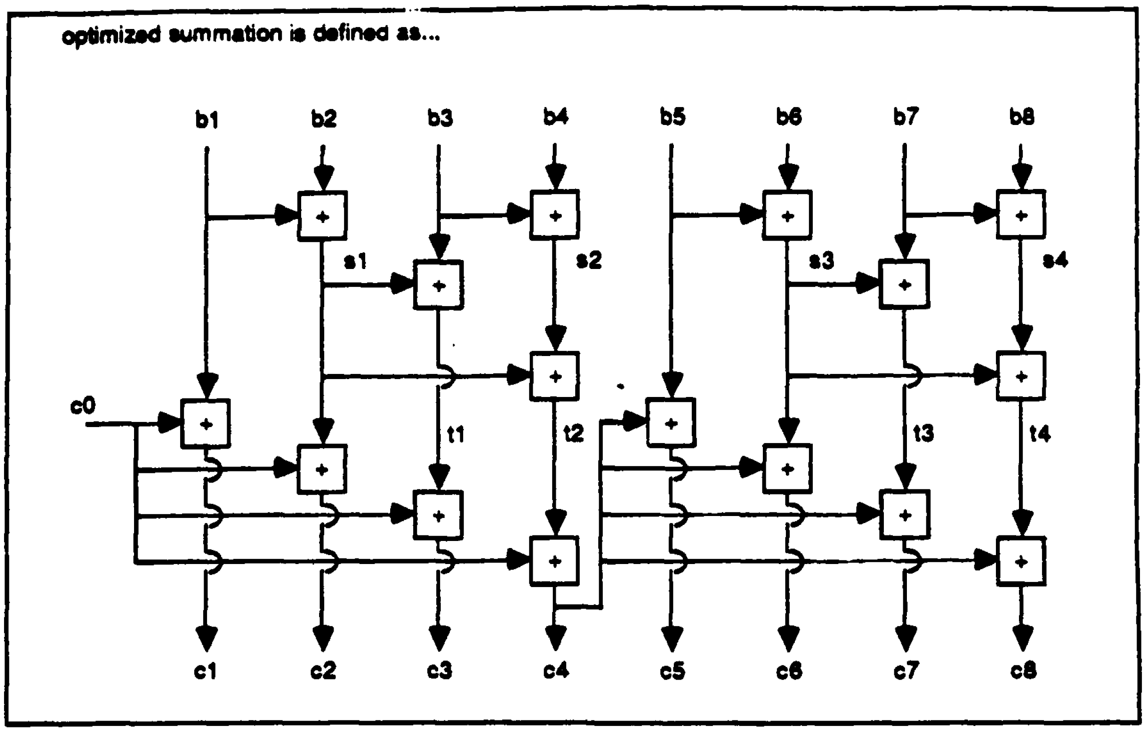

Figure 5. Definition of the "optimized summation" subgraph.

This example highlights several points. First, a given problem may have more than one valid dependence graph program. In the example presented here, the use of knowledge about the underlying semantics of the addition function allows more parallelism. Second, the dependence graph program serves as a intermediate representation from which the solution may be defined for a parallel machine. Third, the dependence graph program does not necessarily make a commitment to the form of the concurrent program. Fourth, for convenience we may describe a dependence graph program as a top-level graph, together with several subgraph definitions.

5. The Airtrac problem

In Airtrac, the problem is to accept radar track data from one or more sensors that are looking for aircraft. Figure 6 depicts a region under surveillance as it might be seen on a display screen at a particular snapshot in time. (Whereas Figure 6 shows many reported sightings, an actual radar would probably show only the most recent sighting.) Locations are designated as either good or bad, where a bad location is illegal or unauthorized, and a good location is legal. The "X" and "Y" symbols represent locations of a good and bad airport, respectively. The locations of radar and acoustic sensors are also shown. The small circles represent track reports that show the location of a moving object in the region of coverage.

Track reports are generated by underlying signal processing and tracking system, and contain the following information:

- location and velocity estimate of object (in x-y plane)

- location and velocity covariance
- the time of the sighting, called the *scantime*
- *track id* for identification purposes.

We would like to answer the following questions in real-time:

- Is an aircraft headed for a bad destination?
- Is it plausible that an aircraft is engaged in smuggling?

By "smuggling" we mean the act of transporting goods from a region or location designated as bad to another bad location. For instance, flying from an illegal airstrip and landing at another illegal airstrip constitutes smuggling.

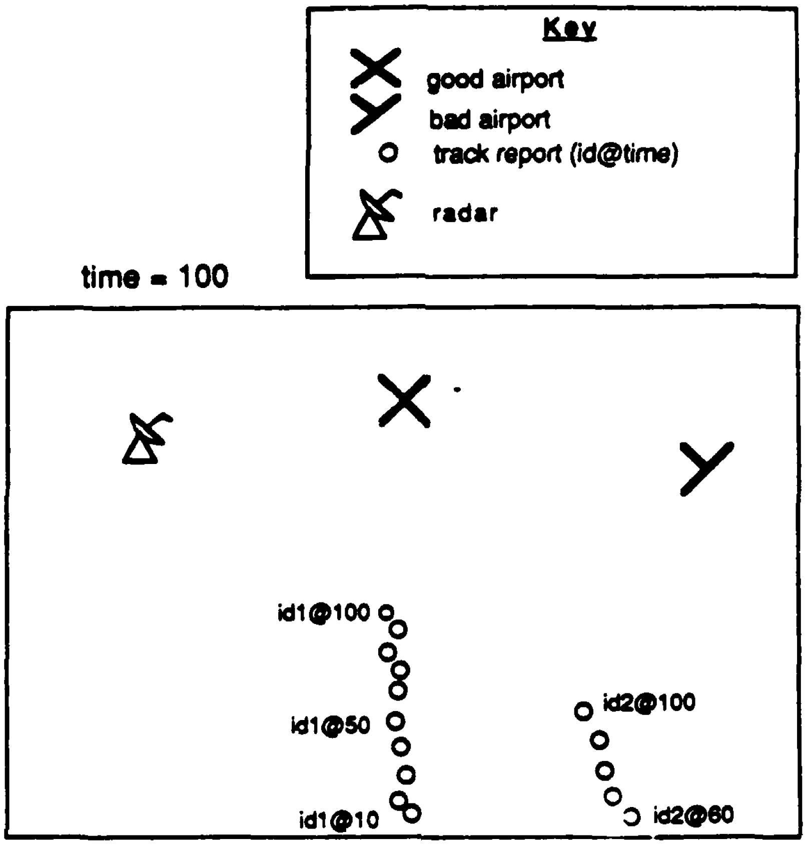

Figure 6. Input to Airtrac.

This shows the inputs that the system receives. The small circles represent estimated positions of objects from radar or acoustic sensors tagged by their identification number and observation time; the goal of the system is to use the time history of those sightings to infer whether an aircraft exists, its possible destinations, and its strategy.

This paper describes our implementation of a solution of a portion of the Airtrac problem. We refer to this portion as the *data association* module. Figure 7 depicts the desired output of the data association step: groupings of reports with the same track id into straight-line, constant-speed sections. These are called *Radar Track Segments*, and have four properties:

- If the Radar Track Segments contains three or more reports, a best-fit line is computed. If the fit is sufficiently good, the segment is declared *confirmed.*
- If a best-fit line has been computed, each subsequent report must fit the line sufficiently closely. If so, the Radar Track Segments remains confirmed. Otherwise, the report that failed to fit (call it the non-fitting report) is treated specially, and the track is declared *broken*.
- A broken track causes the non-fitting report and subsequent reports to be used to form a new Radar Track Segment.

• The last report for a given track id defines that a track is declared *inactive*.

The remaining parts of the Airtrac problem have not yet been implemented as of this writing, but are described more fully elsewhere [Minami 87, Nakano 87].

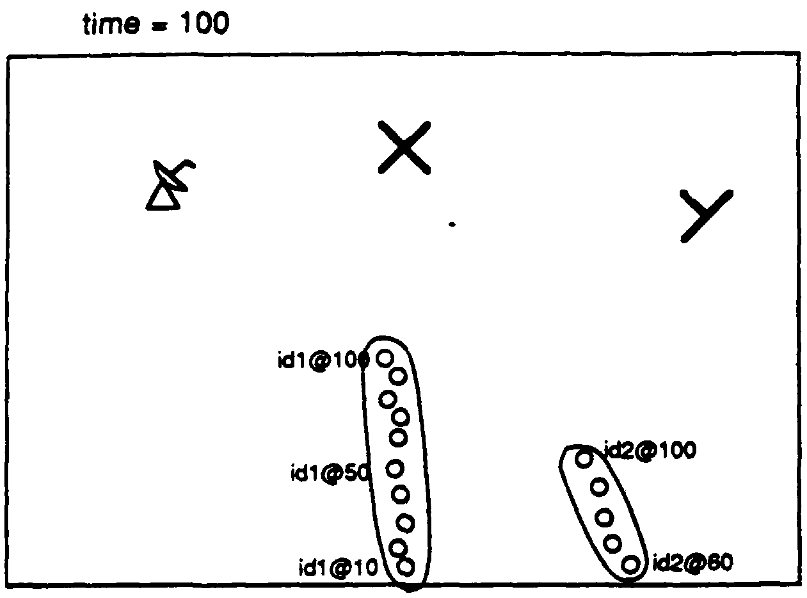

Figure 7. Grouping reports into segments in data association.

This shows the first step in problem solving, grouping the reports into straight-line sections called Radar Track Segments.

5.1. Airtrac data association as dependence graph

Figure 8 shows the Airtrac data association problem as a dependence graph program. On a periodic basis, track reports consisting of position and velocity information for a set of track ids enters the system. Two operations are performed. First, the system checks if a track id is being seen for the first time. If so, a new track-handling subgraph is created. A track-handling subgraph is shown in Figure 8 as a functional box labeled "handle track i," which expands into a graph as shown in Figure 9. Second, the system checks if any track id seen in a previous time has disappeared. If so, it generates an inactivation message for the `handle track` subgraph for the particular track id that disappeared. If the track id has been seen previously, then it is sent to the appropriate `handle track` subgraph.

We distinguish between pure functional nodes, shown as rectangles, and side-effect nodes, shown as rounded rectangles. One use of side-effect nodes is to keep track of which track ids have been seen at the previous time. For instance, by performing set difference operations against the current set of track ids, it is possible to determine the disappeared and new tracks:

```
disappearedTracks = previousTracks - currentTracks
```

```
newTracks = currentTracks - previousTracks
```

One way to implement this scheme is to have the `ids disappeared?` and `id previously seen?` nodes update local variables called `previousTracks` and `currentTracks`, as successive track reports arrive.

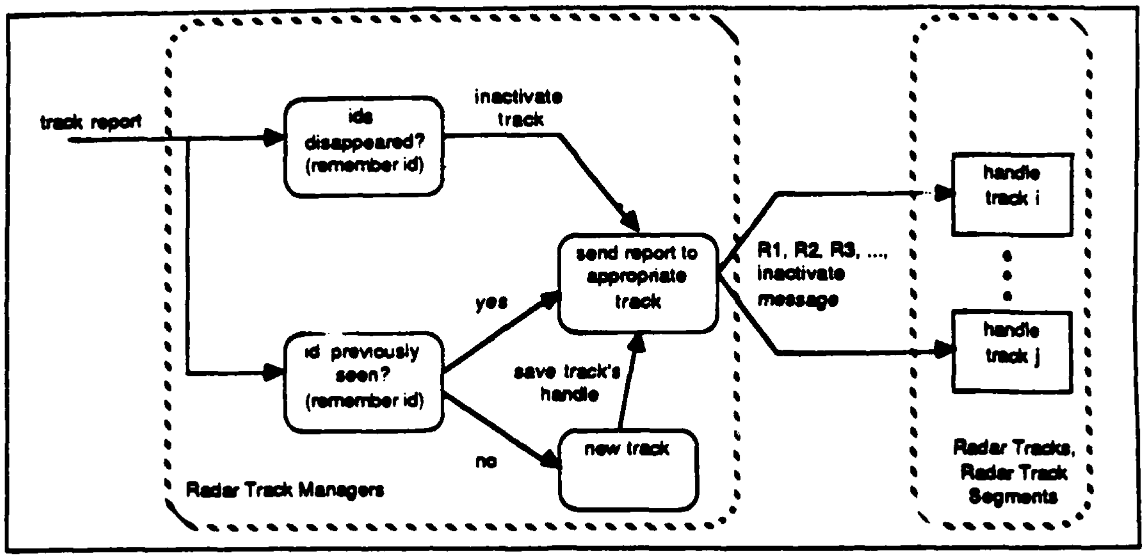
track report
ids disappeared? (remember id)
inactivate track
send report to appropriate track
yes
id previously seen? (remember id)
save track's handle
no
new track
Radar Track Managers
R1, R2, R3, ..., inactivate message
handle track i
handle track j
Radar Tracks, Radar Track Segments

Figure 8. Dependence graph program representation of Airtrac data association.

The dashed boxes indicate the problem decomposition used .n the Lamina implementation.

Besides detecting new and disappeared tracks, side-effect nodes are used to create a new track-handling subgraph, and maintain the lookup table between track id and the message pathway to each track-handling subgraph. `New track` creates a new track handler subgraph. Whenever a new track is encountered, `send report to appropriate track` is notified, so that subsequent reports will be routed correctly. This arrangement requires that one and only one track handler exist for each track id. `Send report to appropriate track` saves the handle[4] to the track handler created by `new track`, sorts the incoming reports, and sends reports to their proper destinations.

In this abstract program, we implicitly assume that only one track report may be processed at a time by the four side-effect nodes in Figure 8. If we allow more than one track report to be processed concurrently, we may encounter inconsistent situations that allow, for instance, a track id to be seen in one track report, but the `send report to appropriate track` node does not yet have the handle to the required track handler subgraph when the next track report arrives. We define the program semantics to avoid these situations.

`Handle track` receives track reports for a particular id, as well as an inactivation message if one exists. It is further decomposed into a subgraph as shown in Figure 9. The

[4]A handle is analogous to a mail address in a (physical) postal system: a Lamina object may use another object's handle to send messages to that object. Since the message passing system utilizes dynamic routing and we assume that an object remains stationary once created, the handle does not need to encode any information about the particular path messages should follow.

nodes in the handle track subgraph pass a structured value between them, called track segments. A *track segment* has the following internal structure:

- report list (a list of track reports, initially empty)
- best-fit line (a vector of real numbers describing a straight-line constant-velocity path in the x-y plane)

Each node may transform the incoming value and send a different value on an outgoing edge. `Add` appends a report to the report list of a track segment. `Linefit` computes the best-fit line, and if the confirmation conditions hold, sends the track segment to `confirm`. `Confirm` declares the track segment as confirmed, and passes the list to `check fit`. If `linefit` fails to confirm, the earliest report in the list is dropped by `drop`, and another `add, linefit` box awaits the arrival of the next report to restart the cycle. The `inactivate` function waits until all reports have arrived before declaring the track inactive. Conceptually, we view the operations of confirm and inactivate as being monotonic assertions made to the "outside world," rather than value transformations to the track segment.

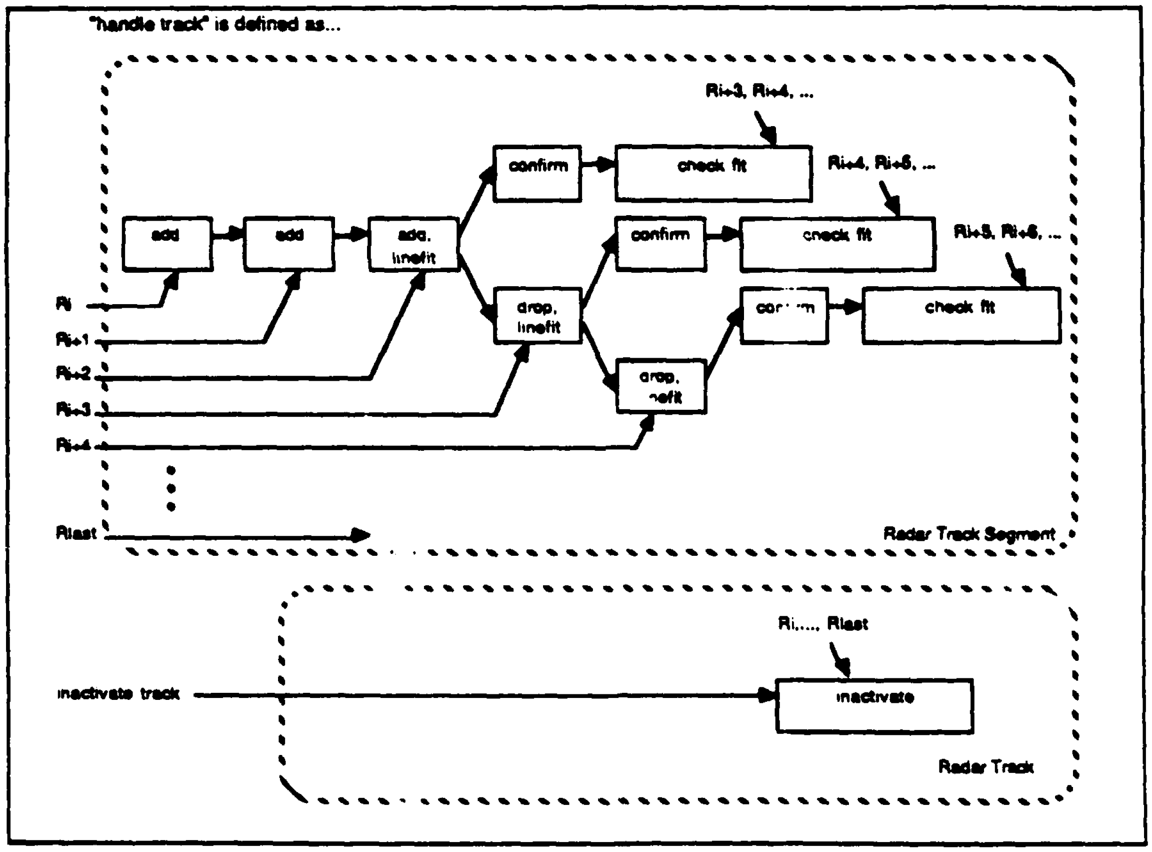

Figure 9. Decomposition of the "handle track" sub-problem.

The dashed boxes indicate the problem decomposition used in the Lamina implementation.

`Check fit` itself is further decomposed into more primitive operations, as shown in Figure 10. The `linecheck` operation is similar to the `linefit` function previously described, except that it compares a new report against the best-fit line computed during the linefit operation: if the new report maintains the fit, the report list is sent to the OK box, and this cycle is repeated for the next report. If the linecheck operation fails, then the track is declared broken, a new track segment is defined. This track segment is sent the report that *failed the linecheck operation, in addition to all subsequent reports for this particular track* id. The track handling cycle is repeated as before.

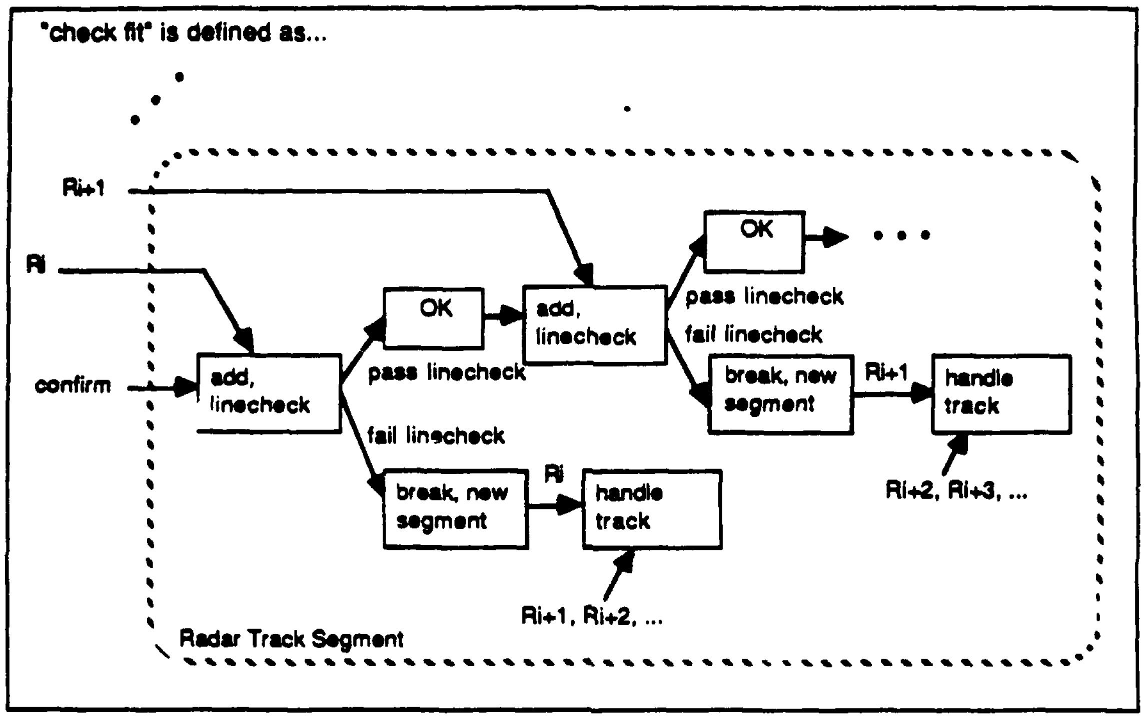

Figure 10. Decomposition of the "check fit" sub-problem.

The dashed boxes indicate the problem decomposition used in the Lamina implementation.

A number of observations may be made about the dependence graph program described in this section. First, the sequence of the reports matters. The graph structure clearly depicts the requirement that the incorporation of the Ri-th report into the track segment by the `add` operation must wait until all prior reports, R1,..., Ri-1, have been processed. This is true for the `linefit`, `linecheck`, and `inactivate` functions. Second, this program avoids the saving of state information except in the operations that must determine whether a given track id has been previously seen, and in the sorting operation where track reports are routed to the appropriate track handler. Except for these, we find that the problem may be cast in terms of a sequence of value transformations. Third, the program admits the opportunity for a high degree of parallelism. Once the track handler for a given track id has been determined, the processing within that block is completely independent of all other tracks. Fourth, the opportunity for concurrency within the handling of a particular track is quite low, despite the outward appearance of the decompositions shown in Figures 8 and 9. Indeed, an analysis of the dependencies shows that reports must be processed in order of increasing scantime. Fifth, unlike certain portions of the dependence graph that have a structure that is known *a priori*, the track handler portions of the graph have no prior knowledge of the track ids that will be encountered during processing, implying that new tracks need to be handled dynamically.

5.2. Lamina implementation

In this section, we express the solution to the data association problem as a set of Lamina objects, together with a set of methods on those objects which embody the abstract solution specification presented in the previous section.

Figure 11 shows how we decompose the Airtrac problem for solution by a Lamina concurrent program. We define six classes of objects: Main Manager, Input Simulator, Input Handler, Radar Track Manager, Radar Track, and Radar Track Segment. Some objects, referred to as *static objects*, are created at initialization time, and include the following object classes: Main Manager, Input Simulator, Input Handler, and Radar Track Manager objects. Others are referred to as *dynamic objects*, are created at run-time in response to the particular input data set, and include the following object classes: Radar Track and Radar Track Segment.

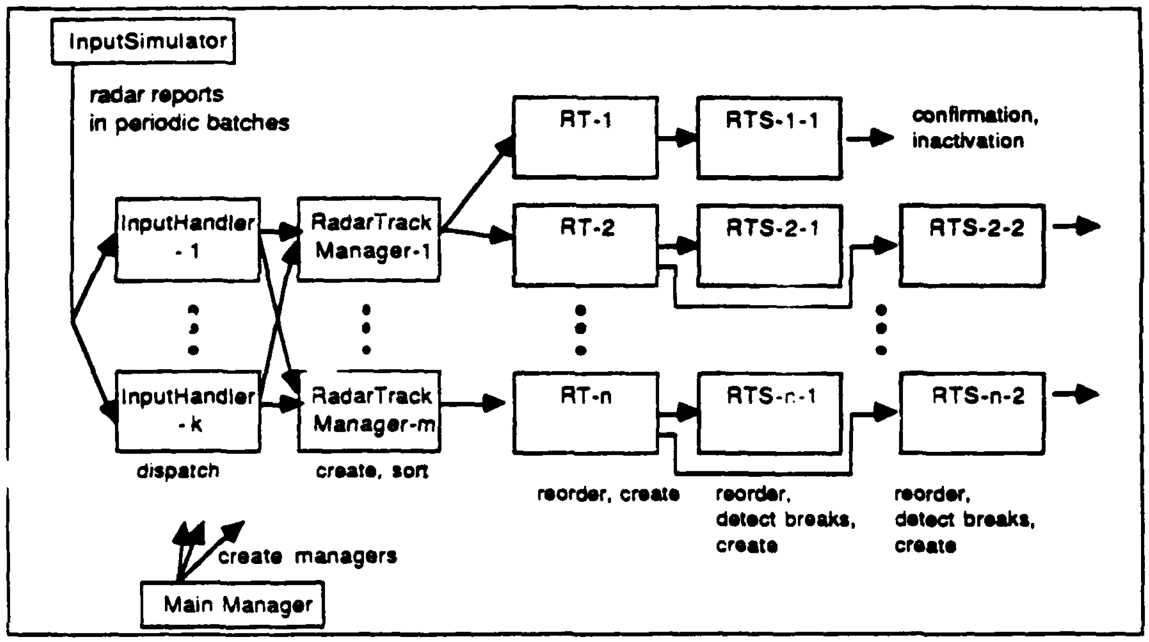

Figure 11. Object structure in the data association module.

Each object is implemented as a Lamina object, which in Figure 11 corresponds to a separate box. The problem decomposition seeks to achieve concurrent processing of independent sub-problems. The Lamina message-sending system provides the sole means of message and value passing between objects. Wherever possible, we pass values between objects to minimize consistency problems, and to minimize the need for protocols that require acknowledgements. For example, we decompose our problem solving so that we require acknowledgements only during initialization where the Main Manager sets up the communication pathways between static objects.

With respect to the dependence graph program, the Lamina implementation takes a straightforward approach. All of the side-effect functions contained in Figure 8, together with some operations to support replication, reside in the Input Handler and Radar Track

Manager object classes. Objects in these two classes are static; we create a predetermined number of them at initialization time to handle the peak load of reports through the system. Replication is supported by partitioning the task of recognizing new and disappeared track ids among Radar Track Managers according to a simple modulo calculation on the track id. Given the partitioning scheme, each Radar Track Manager operates completely independently from the others. Thus, although it needs to maintain a set of objects (e.g. the current tracks, previous tracks), the objects are encapsulated in a Lamina object. Access to and updating of these objects is atomic, providing the mutual exclusion required to assure correctness as specified by the dependence graph program.

Functions in Figures 9 and 10 reside mostly in objects of the Radar Track Segment class, with the inactivation function being performed by objects of the Radar Track class. Objects of these two classes are dynamic: we create objects at run-time in response to the specific track ids that are encountered. For any particular track id, one Radar Track object together with one or more Radar Track Segment objects are created. A new Radar Track Segment is created each time the track is declared broken, which may occur more than once for each track id. Unlike the dependence graph program where we postulate a track segment as a value successively transformed as it passes through the graph, the Lamina implementation defines a Radar Track Segment object with instance variables to represent the evolving state of the track segment. We implement all the major functions on track segments as Lamina methods on Radar Track Segment objects. Again, Lamina objects provide mutual exclusion to assure correctness.

Although nothing in the problem formulation described here indicates why we create multiple Radar Track Segments for a given track, we do so in anticipation of adding functionality in future versions of Airtrac-Lamina. From examination of Figure 10, we see that given any sequence of reports Ri, and any pattern of broken tracks, we obtain no additional concurrency by creating a new Radar Track Segment when a track is declared broken. This is because in the dependency graph program presented here, no activity occurs on one Radar Track Segment after it has created another Radar Track Segment. However, we anticipate that in subsequent versions of Airtrac-Lamina, a Radar Track Segment will continue to perform actions even after a track is declared broken, such as to respond to queries about itself, or to participate in operations that search over existing Radar Track Segments.

Logically, the semantics of the dependency graph program and the Lamina program are equivalent, as they must be. The difference is that the former requires a graph of indefinite size, where its size corresponds to the number of reports comprising the track. The latter requires a quantity of Radar Track Segment objects equal to one plus the number of times the track is declared broken. Although we can easily conceptualize a graph of indefinite size in a dependency graph program, we cannot create such an entity in practice. Because object creation in Lamina takes time, we try to minimize the number of objects that are created dynamically, especially since their creation time impacts the critical path time. A poor solution is to dynamically create the objects corresponding to an indefinite-sized graph as we need them. A better solution is to create a finite network of objects at initialization time, with an implicit "folding" of the infinite graph onto the finite network, thereby avoiding any object-creation cost at run-time. Our Lamina program, in fact, uses a hybrid of these two approaches, folding an indefinite "handle track" graph onto each Radar Track Segment object, and creating a new Radar Track Segment object dynamically when a a track is declared broken. By this mechanism, we model transformations of values between graph nodes by changes to instance variables on a Lamina object. The effect on performance is beneficial. Relative to the first solution, we incur less overhead in message sending between objects because we have fewer objects. Relative to the second solution, we create objects that correspond to track ids that appear in the input data stream as they are needed, which has the effect of bringing more processors to bear on the problem as more tracks become visible.

Both the Radar Track and Radar Track Segment collect reports in increasing scantime sequence. They do so because of the ordering dictated by the dependence graph program, and because the Lamina implementation at the time the experiments were performed did not provide automatic message ordering. Moreover, we know that simply collecting reports in order of receipt leads to severe correctness degradation. For instance, if the scantimes are not contiguous, the scheme by which a Radar Track Segment computes the line-fit leads to nonsensical results because best-fit lines will be computed based on non-consecutive position estimates, leading to erroneous predictions of aircraft movement. To circumvent these problems, we use application-level routines to examine the scantime associated with a report, and queue reports for which all predecessors have not already been handled. These routines effectively insulate the rest of the application from message receipt disorder, and allow the Lamina program to successfully use the knowledge embodied in the dependency graph program.

To indicate the size of the problem, a typical scenario that we experimented with contained approximately 800 radar track reports comprising about 70 radar tracks. At its peak, there is data for approximately 30 radar tracks arriving simultaneously, which roughly corresponds to 30 aircraft flying in the area of coverage.

The correspondence between the Lamina objects in the implementation presented here and the primitive operations embodied in the dependence graph program is shown in the Table 1. The functions described in the dependence graphs are implemented on Radar Track Manager, Radar Track, and Radar Track Segment objects. The Main Manager and Input Simulator perform tasks not mentioned in the dependence graph program. Their tasks may be viewed as overhead: the Main Manager performs initialization, and Input Simulator simulates the input data port. The Input Handler's job is to dispatch incoming reports to the correct Radar Track Manager, thereby supporting the replication of the Radar Track Manager function across several objects. In this way the task of the Input Handler may be viewed as a functional extension of the Radar Track Manager tasks.

Table 1. Correspondence of Lamina objects with functions in the dependence graph program

Lamina object	Corresponding dependence graph program operation
Main Manager	-none- (Create the manager objects in the system at initialization time.)
Input Simulator	-none- (Simulate the input data port that would exist in a real system. This function is an artifact of the simulation.)
Input Handler	-none- (Allows replication of the Radar Track Manager objects; this may be viewed as a functional extension of the Radar Track Manager.)
Radar Track Manager	`ids disappeared?, id previously seen?, new track, send report to appropriate track`
Radar Track	`add, inactivate`
Radar Track Segment	`add, linefit, confirm, drop, inactivate, linecheck, OK, break, new segment`

Table 1 also shows that we decompose the problem to a lesser extent than might be suggested by the dependence graph program, but the overall level of decomposition is still high. We "fold" the dependence graph onto a smaller number of Lamina objects, but we nonetheless obtain a high degree of concurrency from the independent handling of separate tracks. Additional concurrency comes from the pipelining of operations between the following sequence of objects: Input Handler, Radar Track Manager, Radar Track, and Radar Track Segment.

6. Experiment design

Given our experimental test setup, there are a large number of parameter settings, including the number of processors, the choice of the input scenario to use, the rate at which the input data is fed into the system, the number of manager objects to utilize; for a reasonable choice of variations, trying to run all combinations is futile. Instead, based on the hypotheses we attempted to confirm or disconfirm, we made explicit decisions about which experiments to try. We chose to explore the following hypotheses:

- Performance of our concurrent program improves with additional processors, thereby attaining significant levels of speedup.

- Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.

- The amount of speedup we can achieve from additional processors is a function of the amount of parallelism inherent in the input data set.

Long wall-clock times associated with each experiment and limited resources forced us to be very selective about which experiments to run. We were physically unable to explore the full combinatorial parameter space. Instead, we varied a single experimental parameter at a time, holding the remaining parameters fixed at a base setting. This strategy relied on an intelligent choice of the base settings of the experimental parameters.

We divided our data gathering effort into two phases. First, we took measurements to choose the base set of parameters. Our objective was to run our concurrent program on a system with a large number of processors (e.g. 64), picking an input scenario that feeds data sufficiently quickly into the system to obtain full but not overloaded processing pipelines. We used a realistic scenario that has parallelism in the number of simultaneous aircraft so that nearly all the processors may be utilized. Finally, we chose the numbers of manager objects so the managers themselves do not limit the processing flow. The goal was to prevent the masking of phenomena necessary to confirm or disconfirm our hypotheses. For example, if we failed to set the input data rate high enough, we would not fully utilize the processors, making it impossible that additional processors display speedup. Similarly, if we failed to use enough manager objects, the overall program performance would be strictly limited by the overtaxed manager objects, again negating the effect of additional processors.

Based on measurements in phase one, we chose the following settings for the base set of parameter settings:

- 64 processors,

- Many-aircraft scenario (described more fully below),

- Four input handler objects,

- Four radar track manager objects,

- Input data rate of 200 scans per second.

These settings give system performance that suggests that processing pipelines are full, but not overloaded, where nearly all of the processing resources are utilized (although not at 100 percent efficiency), and the manager objects are not themselves limiting overall performance.

The input data rate governs how quickly track reports are put into the system. As reference, the Airtrac problem domain prescribes an input data rate of 0.1 scan per second (one scan every 10 seconds), where a scan represents a collection of track reports periodically generated by the tracking hardware. For the purpose of imposing a desired processing load on our simulated multiprocessor, our simulator allows us to vary the input data rate. With a data rate of 200 scans per second, we feed data into our simulated multiprocessor 2000 times faster than prescribed by the domain to obtain a processing load where parallelism shows benefits. Equivalently, we can imagine reducing the performance of each processor and message passing hardware in the multiprocessor by a factor of 2000 to achieve the same effect, or with any combination of input data rate increase and hardware speed reduction that results in a net factor of 2000.

In the second phase, we vary a single parameter while holding the other parameters fixed. We perform the following set of three experiments:

- Vary the number of processors from 1 to 100.

- Vary the input scenario to use the one-aircraft scenario.

- Vary the number of manager objects.

Figure 12 shows how the many-aircraft-and one-aircraft scenarios differ in the number of simultaneous active tracks. In the many-aircraft scenario, many aircraft are active simultaneously, giving good opportunity to utilize parallel computing resources. In contrast, the one-aircraft scenario reflects the extreme case where only a single aircraft flies through the coverage area at any instant, although the total number of radar track reports is similar between the two scenarios. Although broken tracks in the one-aircraft scenario may give rise to multiple track ids for the single aircraft, the resulting radar tracks are non-overlapping in time.

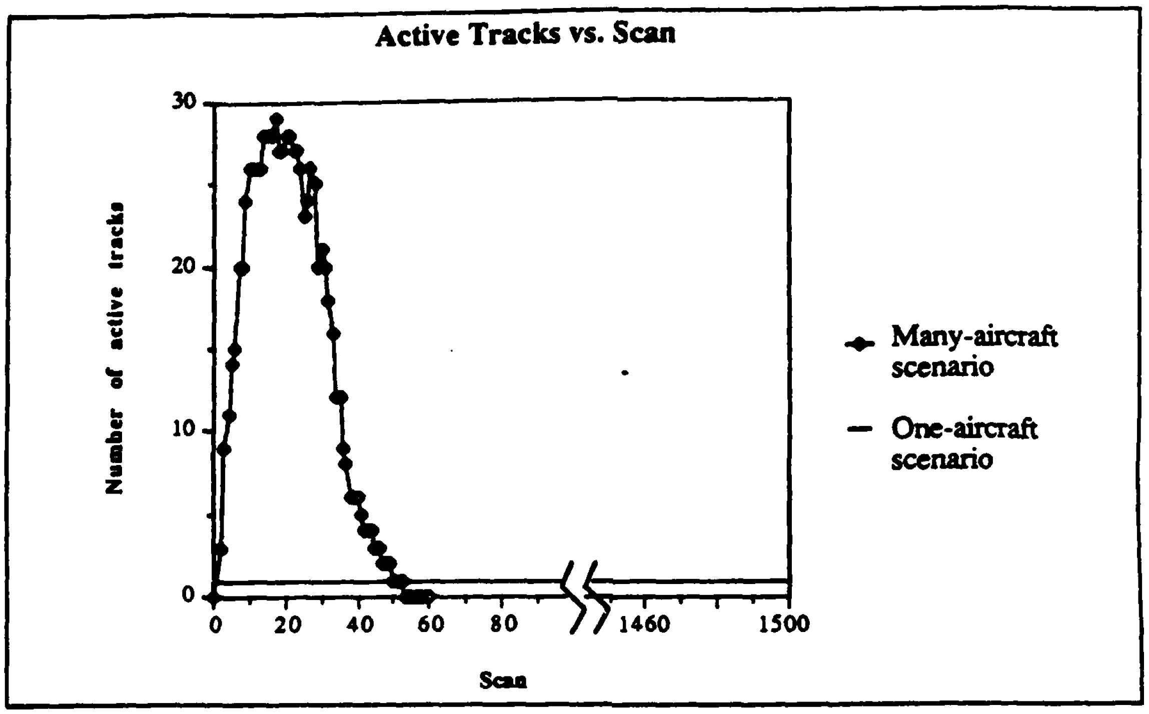

Figure 12. Comparison of the number of active tracks in the many-aircraft and one-aircraft scenarios.

This shows the number of active tracks versus the scan. The scan number corresponds to scenario time in increments of 0.1 seconds.

7. Results

7.1. Speedup

Our performance measure is latency. *Latency* is defined as the duration of time from the point at which the system receives a datum which allows it to make a particular conclusion, to the point at which the concurrent program makes the conclusion. We use latency as our performance measure instead of total running time measures, such as "total time to process all track reports," because we believe that the latter would give undue weight to the reports near the end of the input sequence, rather than weigh performance on all track reports equally.

We focus on two types of latencies: confirmation latency and inactivation latency. *Confirmation latency* measures the duration from the time that the third consecutive report is received for a given track id, to the time that the system has fitted a line through the points, determined that the fit is valid, and it asserts the confirmation. *Inactivation latency* measures the duration from the time that the system receives a track report for the time following the last report for a given track id, to the time when the system detects that the track is no longer active, and asserts the inactivation. Since a given input scenario contains many track reports with many distinct track ids, our results report the mean together with plus and minus one standard deviation.

Figures 13 and 14 show the effects on confirmation and inactivation latencies, respectively, from varying the number of processors from 1 to 100. Boxes in the graphs indicate the mean. Error bars indicate one standard deviation. The dashed line indicates the locus of linear speedup relative to the single processor case; its locus is equivalent to an $S_{n/1}$ speedup level of n for n processors.

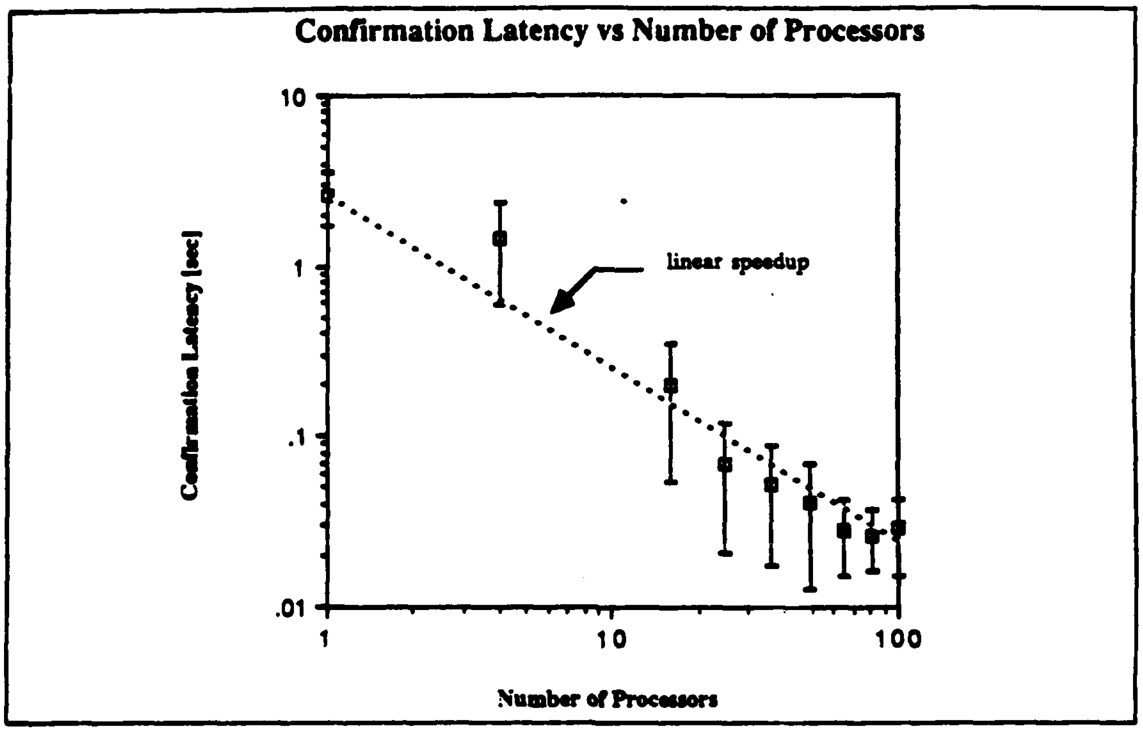

Figure 13. Confirmation latency as a function of the number of processors.

This measures the duration from the time that the third consecutive report is received for a given track id. to the time that the system has fitted a line through the points, and determined that the fit is valid.

The results for both the confirmation and inactivation show a nearly linear decrease in the mean latencies, corresponding to $S_{100/1}$ speedup by a factor of 90 for the confirmation latency, and to $S_{100/1}$ speedup by a factor of 200 for the inactivation latency. The sizes of the error bars make it difficult to pinpoint a leveling off in speedup, if there is any, over the 1 to 100 processor range.

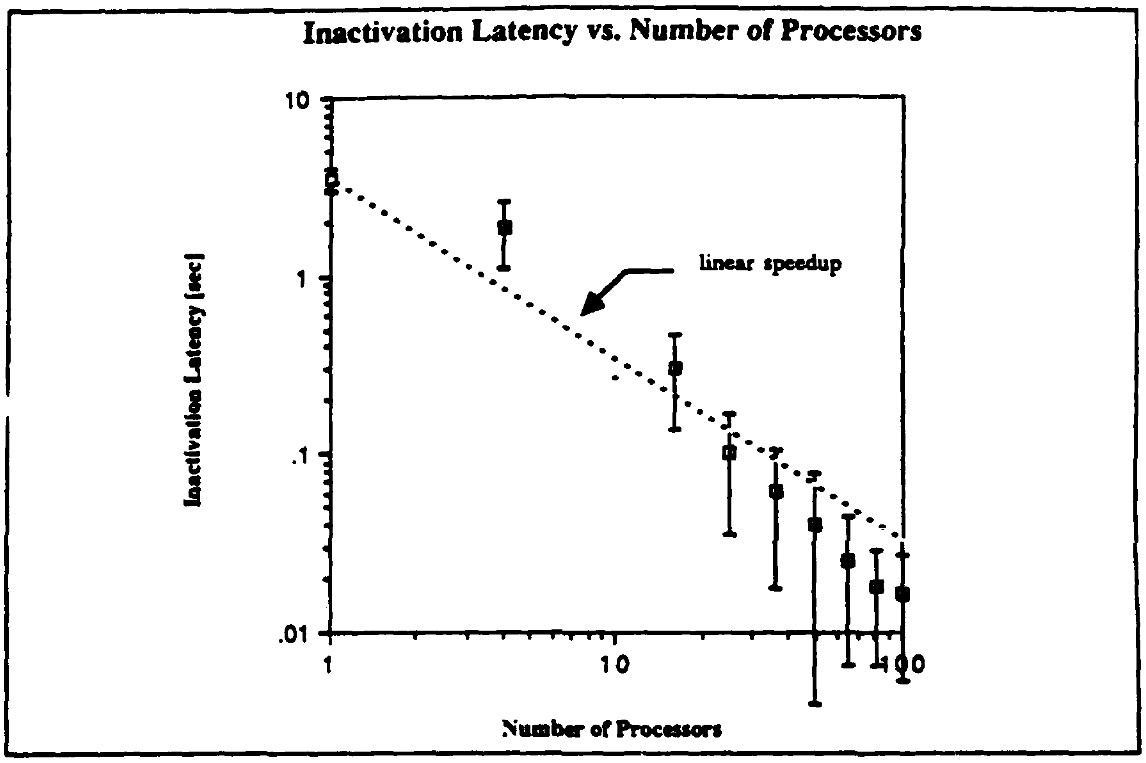

Figure 14. Inactivation latency as a function of the number of processors.

This measures the duration from the time that the system receives a track report for the time following the last report for a given track id, to the time when the system detects that the track is no longer active, and asserts that conclusion.

7.2. Effects of replication

By replicating manager nodes, we measure the impact of the number of manager objects on performance, as measured by the confirmation latency. In one experiment we fix the number of Radar Track Managers at 4 while we vary the number of Input Handlers. In the other experiment we fix the number of Input Handlers at 4 while we vary the number of Radar Track Managers.

Figures 15 and 16 show the results. We plot the confirmation latency versus the number of managers, instead of against the number of processors as done in Figures 13 and 14.

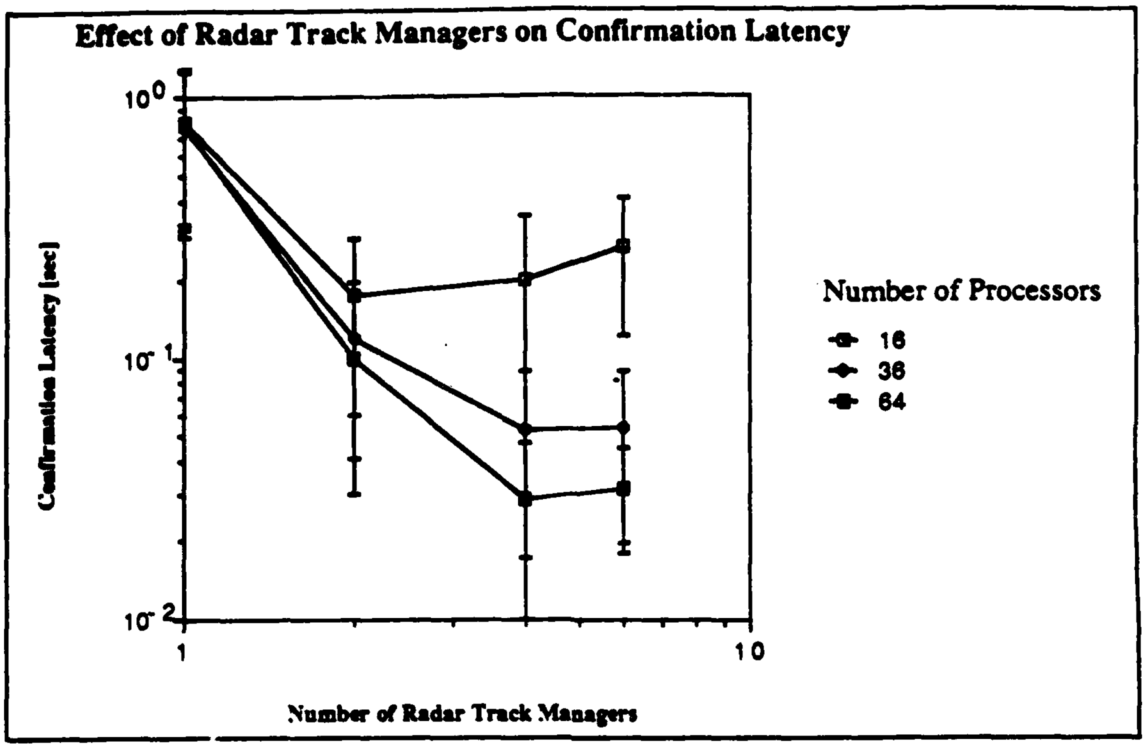

Figure 15. Confirmation latency as a function of the number of radar track managers.

We see that replicating Radar Track Manager objects improves performance; this is because increasing the number of processors does not improve performance in the single Radar Track Manager case, but does in the 4 and 6 Radar Track Managers cases (see Figure 16). Put another way, if we had not used as many as 4 Radar Track Manager objects, then our system performance would have been hampered, and might even have precluded the high degree of speedup displayed in the previous section. Comparing Figures 15 and 16, we also observe that using more Radar Track Managers helps reduce confirmation latency more significantly than using more Input Handlers.

An interesting phenomenon occurs in the 16-processor case. Although the conclusion is not definitive given the size of the error bars, increasing the number of both types of managers from 2 to 4 and 6 increases the mean latency. The likely cause is the current object-to-processor allocation scheme: because each manager object is allocated to a distinct processor, increasing the number of manager objects decreases the number of processors available for other types of objects. Given our allocation scheme (described more fully in Section 8.2), using more managers in the 16-processor case may actually impede speedup.

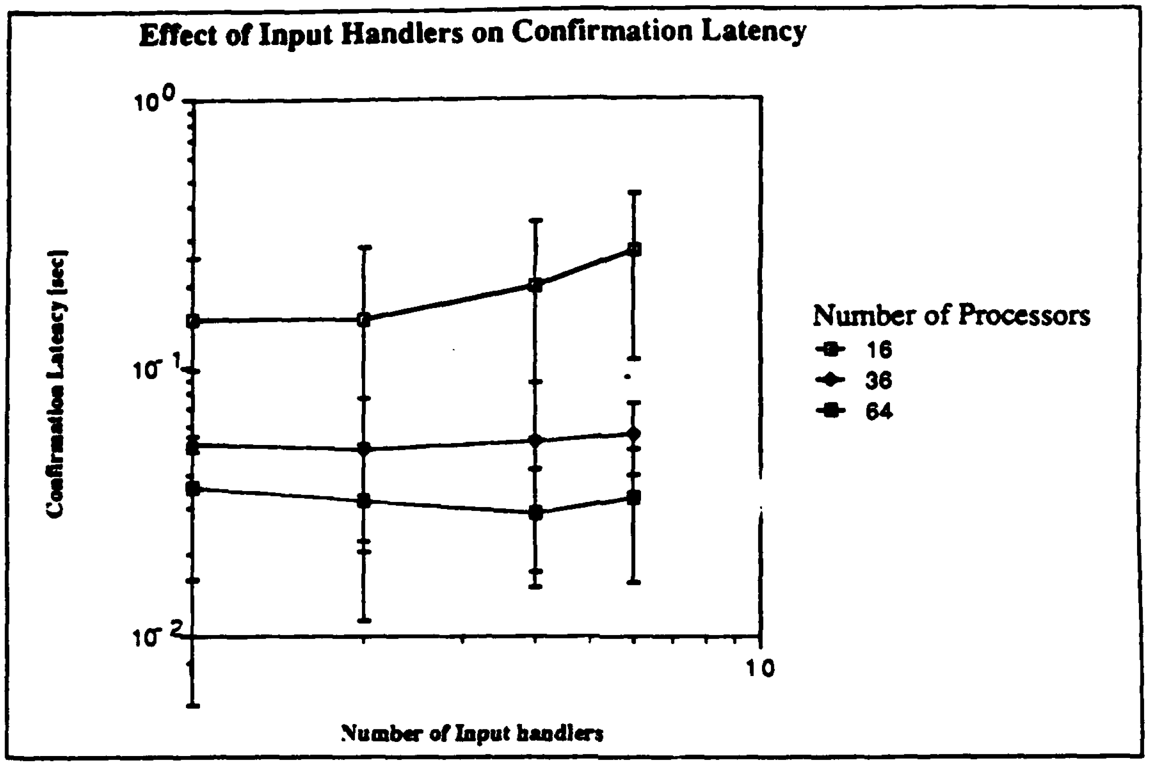

Figure 16. Confirmation latency as a function of the number of input handlers.

The optimal number of manager objects appears to sometimes depend on the number of processors. For Radar Track Managers, 2 or 4 managers is best for the 16-processors array, and 4 or 6 managers is best for the 36 and 64-processor arrays. For Input Handlers, the number of managers does not appear to make much difference, which suggests that Input Handlers are less of a throughput bottleneck than Radar Track Managers. This suggests that in practice it will be necessary to consider the intensity of the managers' tasks relative to the total task in order to make a program work most efficiently. Overall these experiments confirm that replicating objects appropriately can improve performance.

7.3. Less than perfect correctness

Our Lamina program occasionally fails to confirm a track that our reference solution properly confirms. This arises because the concurrent program does not always detect the first occurrence of a report for a given track in the presence of disordered messages. We notice the following failure mechanism. Suppose we have a track consisting of scantimes 100, 110, 120, ..., 150. Suppose that the rate of data arrival is high, causing message order to be scrambled, and that reports for scantimes 110, 120, and 130 are received *before* the report for 100. As implemented, the Radar Track object notices that it has sufficient number of reports (in this case three), and it proceeds to compute a straight line through the reports. When a report for scantime 140 or higher is received, it is tested against the computed line to determine whether a line-check failure has occurred. Unfortunately, when the report for scantime 100 eventually arrives, it is discarded. It is discarded because the track has already been confirmed, and confirmed tracks only grow in the forward direction.

Figure 9 reveals why this error causes discrepancies between the Lamina program and the reference serial program: the handle track operation in the Lamina program is given a different set of reports compared to the reference program, leading to a different best-fit line being computed. To be certified as correct, we require that the reports contained in a confirmed Radar Track Segment must be identical between the Lamina solution and the reference solution.

The lesson here is that message disordering does occur, and that it does disrupt computations that rely on strict ordering of track reports. In our experiments, the incorrectness occurs infrequently. See Figure 17. We believe that with minimal impact on latency, this source of incorrectness can be eliminated without significant change to the experimental results.

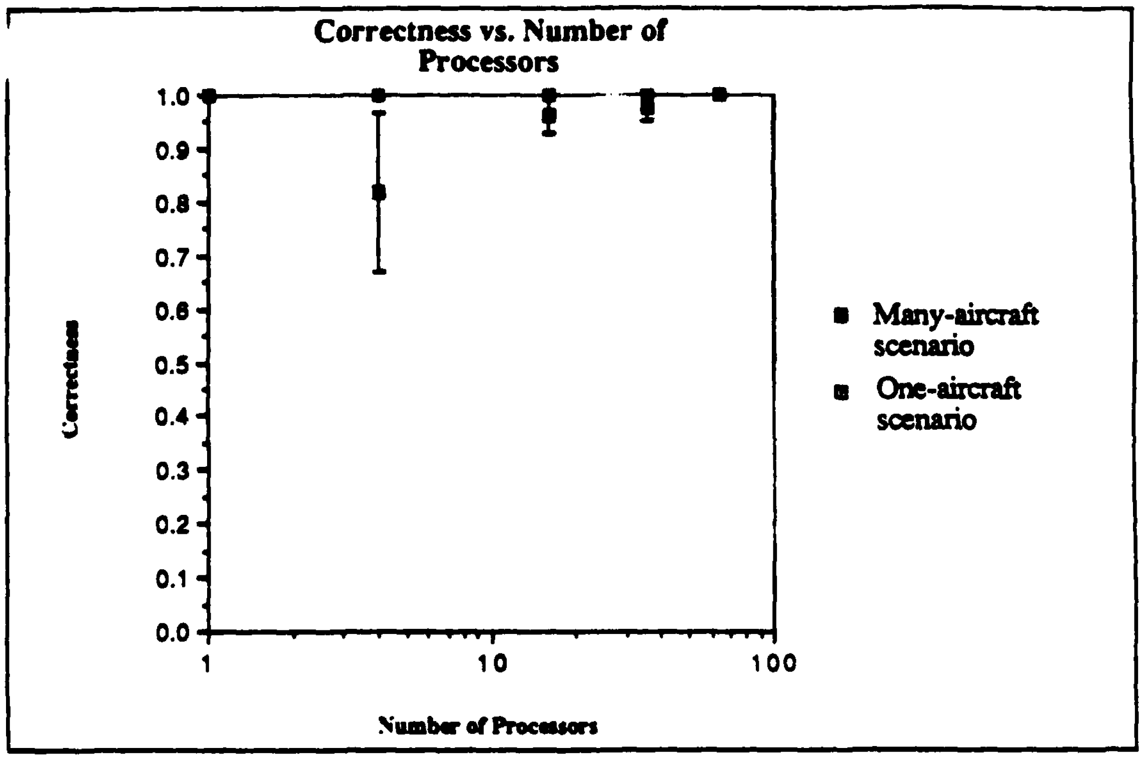

Figure 17. Correctness plotted as a function of the number of processors for the one-aircraft and many-aircraft scenarios.

7.4. Varying the input data set

The results from using the one-aircraft scenario highlight the difficulties in measuring performance of a real-time system where inputs arrive over an interval instead of in a batch. Before experimentation began, we hypothesized that the amount of achievable speedup from additional processors is a function of the amount of parallelism inherent in the input data set. The results relative to this hypothesis are inconclusive. Figure 18 plots the confirmation latency against the number of processors for two input scenarios, the many-aircraft scenario (30 tracks per scan) and the one-aircraft scenario (1 track per scan).

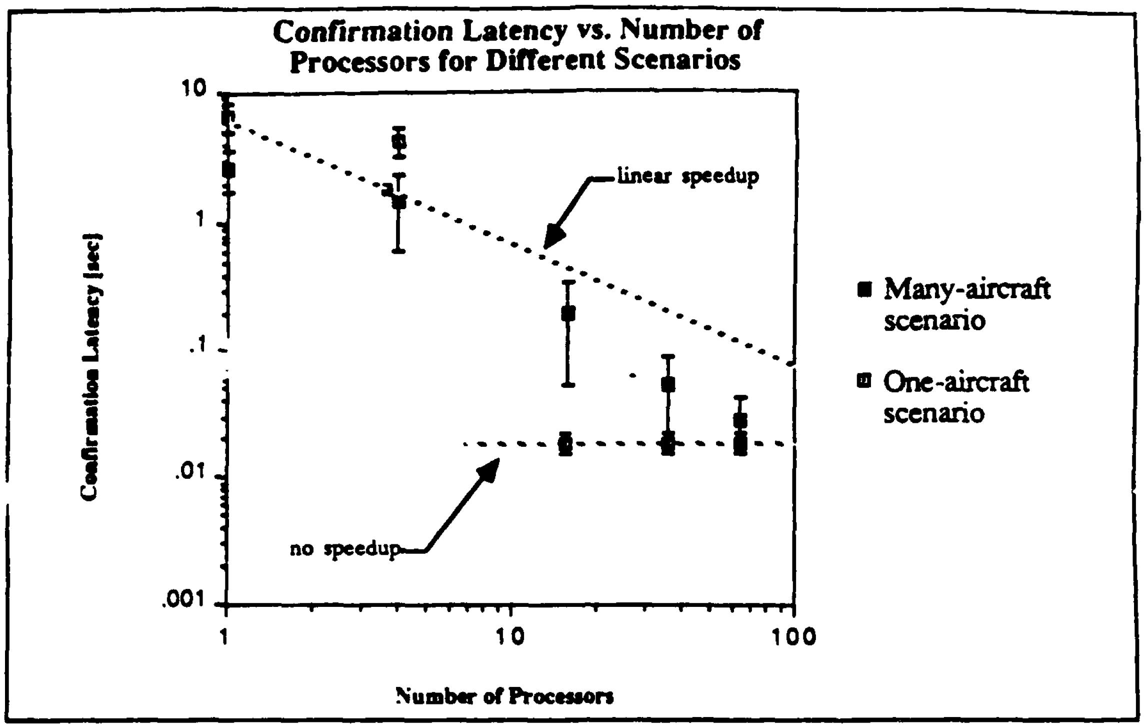

Figure 18. Confirmation latency as a function of the number of processors varies with the input scenario.

The one-aircraft scenario displays two distinct operating modes ɔne in which processor availability and waiting time determines the latency, and anoth:r in which data can be processed with little waiting.

The one-aircraft scenario displays interesting behavior: see Figure 18. While the confirmation latency decreases from the 1-processor to 4-processor case, just as in the many-aircraft scenario, there is distinctly different behavior for 16, 36 and 64 processor cases, where the average latency is constant over this range. The key to understanding this phenomenon is to realize that inputs to the system arrive periodically. The many-aircraft scenario generates approximately 800 reports comprising 70 radar tracks over a 200 millisecond duration. In contrast, the one-aircraft scenario generates approximately 1300 reports comprising 70 radar tracks over an 8 second duration. Thus, although the volume of reports is roughly equivalent (800 versus 1300), the duration over which they enter the system differs by a factor of 40 (0.2 seconds versus 8 seconds). In terms of radar tracks per second, which is a good measu·ᵉ of the object-creation workload, the many-aircraft scenario produces data at a rate of ɔ0 tracks per second, while the one-aircraft scenario produces data at a rate of 8.8 tracks per second. This disparity causes the many-aircraft scenario to keep the system busy, while the one-aircraft scenario meters a comparable inflow of data over a much longer period, during which the system may become quiescent while it awaits additional inputs.

The one-aircraft scenario displays two distinct operating modes: one in which processor availability and waiting time determines the latency, and another in which data can be processed with little waiting. For the 1-processor and 4-processor cases, the system cannot process the input workload as fast as it enters, causing work to back up. This explains why the average confirmation latency for the 70 or so radar tracks is nearly as long as the scenario itself: most of the latency is consumed in tasks waiting to be executed. In contrast, for the 16-processor, 36-processor and 64-processor cases, there are sufficient computing resources available to allow work to be handled as fast as it enters the system. This explains why the average latency bottoms out at 18 milliseconds, and also tends to explain the small variance.

Recalling that this particular experiment sought to test the hypothesis that the amount of achievable speedup from additional processors is a function of the amount of parallelism inherent in the input data set, we see that these experimental results cannot confirm or disconfirm this hypothesis. The problem lies in the design of the one-aircraft input scenario. The reports should have been arranged to occur over the same 20 millisecond duration as in the many-aircraft scenario, instead of over an 8 second duration. Had that been done, the two scenarios would present to the system comparable workloads in terms of reports per second, but would differ internally in the degree to which sub-parts of the problem can be solved concurrently.

The distinction between the one-aircraft and many-aircraft scenarios can be described in Figure 19. This graph is an abstract representation of Figure 12 presented earlier, and plots the input workload as a function of time. The many-aircraft scenario presents a high input workload over a very short duration, while the one-aircraft scenario presents the same total workload spread out over a much longer interval. If we imagine the dashed lines to represent the workload threshold for which an n-processor system is able to keep up without causing waiting times to increase, we see that the many-aircraft scenario exceeded the ability of the system to keep up even at the 100-processor level, but the one-aircraft scenario caused the system to transition from not-able-to-keep-up to able-to-keep-up somewhere between 4 and 16 processors. A more appropriate one-aircraft scenario, then, is one that has the same input workload profile as the current many-aircraft scenario. Such a scenario would allow an experiment to be performed that fixes the input workload profile, which our experiment inadvertently varied, thereby contaminating its results.

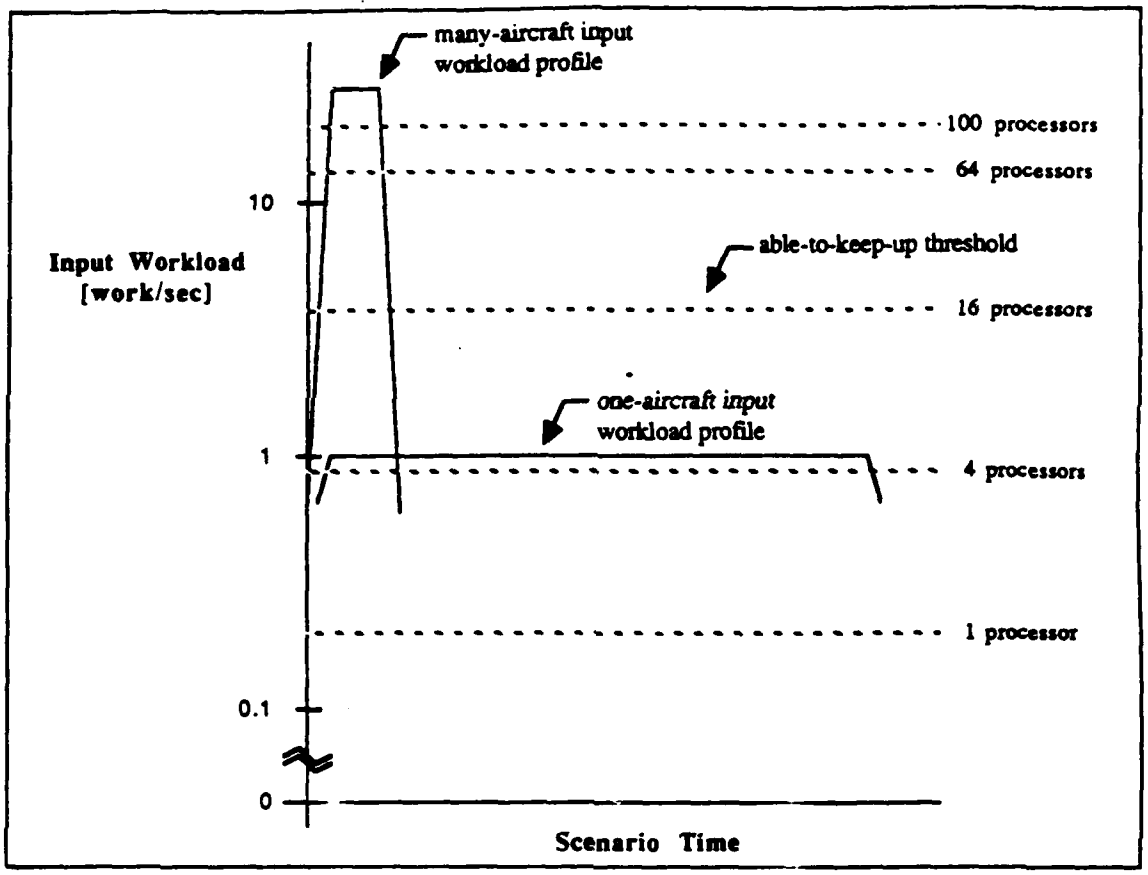

Figure 19. Input workload versus time profiles shown for two possible input scenarios.

The workload threshold above which the work becomes increasingly backlogged varies according to the number of processors.

8. Discussion

This section discusses how we achieved our experimental results using the concepts developed in Section 4. Specifically, we focus on the relationships between problem decomposition, speedup, and achievement of correctness.

8.1. Decomposition and correctness

In this section we analyze the problem solving knowledge embodied in the data association module. We use the dependence graph program to represent inherent dependencies in the problem. This is contrasted with the Lamina implementation to shed light on the rationale behind our design decisions. The goal is to identify the general principles that govern the transition from a dependence graph program to a runnable Lamina implementation.

8.1.1. Assigning functions to objects

We obtained speedup from both independent handling of tracks, and possibly from pipelining within a track, without the necessity to decompose the problem into the small functional pieces suggested in Figures 9 and 10. One might be tempted to believe that a direct translation of the nodes and edges of the dependence graphs into Lamina objects and methods might yield the maximal speedup, but careful study of the dependencies in Figures 9 and 10 reveals that there is very little concurrency to be gained.

In Figure 9, the entire graph is dependent on the arrival of report Ri. For instance, before a track is declared broken, the top-level "handle track" graph requires the arrival of reports R1, R2,...,Rlast. The leftmost add node needs R1, and the remainder of the graph is dependent on this node. The add node to the right of this one is dependent on the arrival of R2, and the remaining right-hand subgraph is dependent on this node. This pattern holds for the entire graph, implying that computation may only proceed as far as consecutive reports beginning with R1 have arrived. Thus, little concurrency may be gained from the "handle track" operation; in particular, no pipelining is possible because the entire graph receives only one set of reports, R1,...,Rlast. Figure 10 is similarly dependent on sequential processing of reports. We conclude that lumping all of the functions of Figures 9 and 10 into a small number of objects does not incur a great expense in concurrency. Given the overhead costs associated with message sending and process invocation, we speculate that one or two objects might yield the best possible design. In fact, our design uses k+2 objects, where k is the number of times a track is declared broken; k is typically fewer than three, giving us fewer than five objects for each "handle track" graph.

The dependence graph program provides several useful insights regarding a good problem decomposition. First, it justifies a decomposition that treats the "handle track" function as primitive function, rather than a finer-grained decomposition. Second, it clearly shows the independence between tracks, suggesting a relatively painless problem decomposition along these lines. Third, it shows the need to maintain consistent state about which tracks have been seen, and those which have not, suggesting a decomposition according to track id number, which is the approach that our Lamina program takes.

8.1.2. Why message order matters

A significant part of the Lamina concurrent program implements techniques to allow a Lamina object receiving messages from a single sender to handle them *as if* they were received in the order in which they were originally sent, without gaps the in the message sequence. By doing this, we incur a performance cost because the receiver waits for arrival of the next appropriate message, rather than immediately handling whatever has been received.

The dependence graphs help to justify such costs because the dependencies imply ordering. Indeed, in preliminary work in a different framework, one author discovered that when no explicit ordering constraints were imposed during Airtrac data association processing, and neither additional heuristics nor knowledge was used, incorrect conclusions resulted in cases when the input data rate was high. The incorrect conclusions arose from performing the line-fit computation on other reports different from the first three consecutive reports. As such, the incorrectness reflected an interaction between message disordering arising in CARE and the particular Airtrac knowledge, rather than the specific problem solving framework. We believe, for instance, that similar incorrect conclusions would arise in a Lamina program that did not explicitly reorder reports.

We emphasize that although the particular problem that we studied showed strong correctness benefits from imposing a strict ordering of reports, this should not be interpreted as a claim that all problems need or require message ordering. As the dependence graphs make strikingly clear, the very knowledge that we implement dictates ordering. Another problem may not require ordering, but require a strict message tagging protocol, for instance. As a general approach, we believe that the programmer should represent the given problem in dependence graph form, preferably explicitly, to expose the required set of dependencies, and let the overall pattern of dependencies suggest the kinds of decompositions and consistency requirements that might prove best.

8.1.3. Reports as values rather than objects

In the dependence graph program we represent reports as values sent from node to node. Similarly, in the Lamina implementation, we use a design where reports are values sent from object to object. This works well because reports never change, enabling us to treat reports as values. The cost of allowing an object to obtain the value of a report is a fairly inexpensive one-way message, where value-passing is viewed as a monotonic transfer of a predicate. This approach works because we know ahead of time which objects need to read the value of a report, namely the objects that constitute the processing pipeline.

Consider a second design where reports are represented as objects. In this scheme, instead of a report being a value passing through a processing pipeline, we arrange for read operations to be applied to an object. Conceptually these are identical problems, the only difference being the frame of reference. In the first case, the datum moves through processing stages requiring its value. In the case being considered here, the datum is stationary, and it responds to requests to read its value. This is attractive when it is not known in advance which objects will need to read its value. The penalty is an additional message required to request the object's value, and the associated message receipt system overhead.

A third design represents reports as objects, but replaces the read message in the previous design with a request to perform a computation, and uses the object's reply message to convey the result of the computation. By arranging a set of reports in a linear pipeline, we can allow the first report to send the results of its computation to the second report, and so forth. This design is the dual of the first design because in this design we send a sequence of computation messages through a pipeline of report objects, whereas in the first design we send a sequence of report value messages through a pipeline of computing objects. The designs differ in the grain-size of the problem decomposition; since our problem has a small number of computations and a large number of reports, the first design yields a small number of computing objects with many reports passing through, whereas the third design yields a large number of objects with a small number of computation messages passing through.

In our design, namely the first design discussed, we choose to represent reports as values sent to successive objects in a processing pipeline because our problem decomposition tells us in advance the objects in a pipeline. Using this design minimizes the number of messages required to accomplish our task, and uses a larger grain-size compared to its dual.

8.1.4. Initialization

Our approach to initialization embodies the correctness conditions of Schlichting and Schneider. Formally, we combine the use of monotonic predicates and predicate transfer with acknowledgement.

During initialization of our application, we create many objects, typically managers. At run-time, these objects communicate among themselves, which requires that we collect handles during creation, and distribute them after all creation is complete. Specifically, the Main Manager collects handles during the creation phase; in essence, each created object sends a monotonic predicate to the Main Manager asserting the value of its handle. The invariant condition may be expressed as follows:

Invariant (asserting own handle): "handle not sent" or "my handle is X"

The Main Manager detects the fact that all creation is complete when each of the predetermined number of objects respond; at this point, it distributes a table containing all the handles to each object. It waits until an acknowledgement is received from each object before initiating subsequent problem solving activity. This is important because if the Main Manager begins too soon, some object might not have the handle to another object that it needs to communicate with. In essence, the table of handles is asserted by a predicate transfer with acknowledgement. The invariant condition is described as follows:

Invariant (distributing table of handles):

"table not sent"

or "problem solving not initiated"

or "all acknowledgements received"

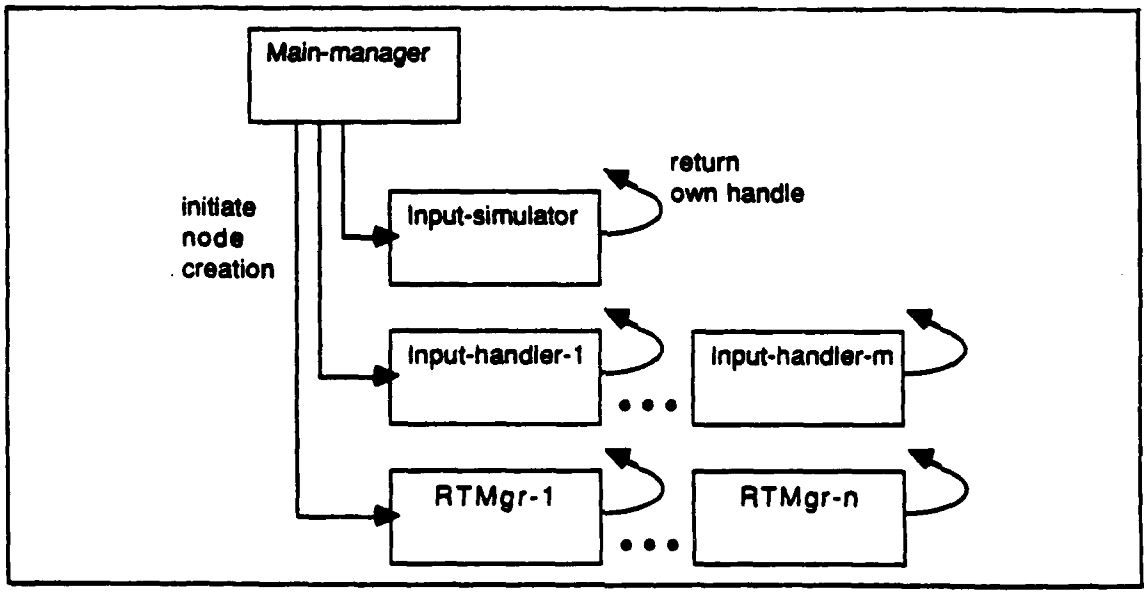

Figure 20. Creating static objects during initialization.

Correctness is crucial during initialization because a missing or incorrect handle, or a missing or improperly created object causes problems at run-time. These problems can compound themselves, causing performance or correctness degradation to propagate. By using an initialization protocol that is guaranteed to be correct, these problems may be avoided.

8.2. Other issues

8.2.1. Load balance

We define *load balance* as how evenly the actual computational load is distributed over the processors in an array over time. Processing load is balanced when each processor has a mix of processes resident on it that makes all the processors equally busy. If a balanced processing cannot be achieved, the overall performance of a multiprocessor may not reflect the actual number of processors available to perform work due to poor load balance. In our experimentation, we discovered the critical importance of a good load balance algorithm.

We encountered two kinds of problems. The first problem deals with where to place a newly created object. Since we want to allocate objects to processors so as to evenly distribute the load, and because we want to avoid the message overhead associated with a centralized object/processor assignment facility, we focused on the class of algorithms that make object-to-processor assignments based on *local* information available to the processor creating the object. The second problem deals with how objects share limited processor resources. It turns out, for instance, that extremely computation-intensive objects can severely impair the performance of all other objects that share its processor.

At one point in our experimentation, for instance, we observed a disappointing value of unity for the $S_{64/16}$ speedup factor, where we instead expected a factor of 4. Moreover, we noticed an extremely uneven mapping of processes to processors: the approximately 200 objects created during the course of problem solving ended up crowded on only 14 of the 64 available processors! The culprit was the algorithm that decided which neighboring processor should be chosen to place a new object. The algorithm worked as follows. Beginning with the first object created by the system, a process-local data structure, called a *locale*, is created that essentially records how many objects are already located at every other processor in the processing array. When a new process is spawned, the locale data structure is consulted to choose a processor that has the fewest existing processes. This scheme works well when a single object creates all other objects in the system; unfortunately in Airtrac many objects may create new objects.

Given the locale for any given process, when the process spawns a new process, we arranged for the new process to inherit the locale of its parent. The idea is that we want the new process to "know" as much as its parent did about where objects are already placed in the array. This scheme fails because of the tree-like pattern of creations. Beginning with the initial manager object at the root of the tree, any given object has inherited a locale through all of its ancestors between itself and the root. Therefore the locale on a given object will only *know* about other objects that were created by the ancestors of the object *before* the locale was passed down to the next generation. Put another way, the locale on a given object will not reflect creations that were performed on non-ancestor objects, or creations that were performed on ancestor objects after the locale was passed down. This leads to extremely poor load balance.

The same problem occurs even if we define a single locale for each processor that is shared over all processes residing on that processor. Unfortunately, that locale will only know about other objects that were created by objects residing on that processor. That is, the locale on a given processor will not reflect creations that were performed by objects that reside on *other* processors.

In contrast, ideal load balance occurs when each object knows about all creations that have taken place in the past over the entire processing array. This ideal is extremely difficult to achieve. First, we want to avoid using a single globally-shared data structure. Second, finite message sending time makes it impossible for many objects performing simultaneous object creation to access and update a globally-shared structure in a perfectly consistent manner.

We changed to a "random" load balance scheme which randomly selected a processor in the processing array on which to create a new object [Hailperin 87]. Running the base case on a 64 processor array with approximately 200 objects, we managed to use nearly all the available processors. Processor utilization improved dramatically.

Random processor allocation gave us good performance. In fact, we can argue from theoretical grounds that a random scheme is desirable. First, we deliberately constrain the technique to avoid using global information that would need to be shared. This immediately rules out any cooperative schemes that rely on sharing of information. Second, any scheme that attempts to use local information available from a given number of close neighbors and performs allocations locally faces the risk that some small neighborhood in the processing array might be heavily used, leaving entire sections of the array underutilized. We are left therefore, with the class of schemes that avoids use of shared information but allows any processor to select any other processor in the entire array. Given these constraints, a random scheme fits the criteria quite nicely and in fact performed reasonably well.

Further experimentation revealed more problems. Manager objects have a particularly high processing load because a very small number of objects (typically 5 to 9) handles the entire flow of data. When a non-manager objects happens to reside on the same processor as a manager object, its performance suffers. For example, a Radar Track object is responsible for creating a Radar Track Segment object, and the time taken for the create operation affects the confirmation performance. Unfortunately, any Radar Track object that happens to be situated on the same processor as a manager object (e.g. Input Handler, Radar Track Manager) gets very little processor time, and thereby contributes significant creation times to the overall latency measure.

Whereas in the random scheme the probability that a given processor will be chosen for a new object is $\frac{1}{n}$ for n processors, our modified random scheme does the following:

- If there are fewer static objects (e.g. managers) than processors, then place static objects randomly, which can be thought of as sampling a random variable *without replacement*. Place dynamically created objects uniformly on the processors that have no static objects, this time sampling *with replacement*.

- If there are as many or more static objects than processors, then place roughly equal numbers of static objects on each processor in the array. Place dynamically created objects uniformly over the entire array, sampling *with replacement*.

This scheme keeps the high processing load associated with manager objects from degrading the performance of non-manager objects. This scheme performs well for our cases. We typically had from 5 to 9 static objects, approximately 150 dynamic objects, and from 1 to 100 processors in the array.

There are other considerations that might lead to further improvement in load balance performance that we did not pursue. These are listed below:

- Account for the fact that not all static objects need a dedicated processor. (In our scheme, we gave each static object an entire processor to itself whenever possible.)

- Account for the fact that a processor that hosts one or more static objects may still be a desirable location for a dynamically created object, although less so than a processor without any static objects. (In our scheme, we assumed that any processor with a static object should be avoided if possible.)

- Relocate objects dynamically based on load information gathered at run-time.

8.2.2. Conclusion retraction

This section explores some of the thinking behind our approach toward consistency, which is to make conclusions (e.g. confirmation, inactivation) only when they were true. This is an extremely conservative stance, and possibly incurs a loss in concurrency and speedup. An alternative approach which might allow more concurrency is to make conclusions that are not provably correct: the programmer would allow such conclusions to be asserted, retracted and reasserted freely until a commitment regarding that conclusion is made. Jefferson has explored this compuational paradigm, known as *virtual time* [Jefferson 85]. The invariant condition describing the truth value of a conclusion P under such a scheme is shown below:

Invariant: "no commitment made" or "P is true"

In essence, this invariant condition says that the program may *assert* that P is true, but there is no guarantee that P is true unless it is accompanied by a *commitment* to that fact. The benefits of such an approach is that assertions may precede their corresponding commitments by some time interval. This interval may be used 1) by the user of the system in some fashion, or 2) by the program itself to engage in further exploratory computation that may be beneficial, perhaps in reducing computation later. In Airtrac-Lamina, we did not investigate the benefits from exploratory computation.

For the user of the system, he or she must decide how and when to act upon uncommitted assertions rendered by the system. On one hand, the user could view assertions as true statements even before a commitment is made, with the anticipation that a retraction may be forthcoming. On the other hand, the user could view an assertion as true only when accompanied by a commitment; this latter approach places emphasis on the commitment, since only the commitment assures the truth of the conclusion.

We decided against using the scheme outlined here. As a technique to allow concurrent programs to engage in exploratory computations, there might be some merit if the power of such computations can be exploited. As a logical statement to the user of the system, such an uncommitted conclusion is meaningless, since it may later be retracted. As a probabilistic statement to the user of the system, a conclusion without commitment might indicate some likelihood that the conclusion is true. However, we believe that a better way to handle probabilistic knowledge is to state it directly in the problem rather than in the consistency conditions that characterize the solution technique. This unclear separation between domain knowledge and concurrent programming techniques steered us away from the approach of making assertions with the possibility of subsequent retraction.

9. Summary

Lamina programming is shaped by the target machine architecture. Lamina is designed to run on a distributed-memory multiprocessor consisting of 10 to 1000 processors. Each processor is a computer with its own local memory and instruction stream. There is no global shared memory; all processes communicate by message passing. This target machine environment encourages a programming style that stresses performance gains through problem decomposition, which allows many processors to be brought to bear on a problem. The key is to distribute the processing load over replicated objects, and to increase throughput by building pipelined sequences of objects that handle stages of problem solving.

For the programmer, Lamina provides a concurrent object-oriented programming model. Programming within Lamina has fundamental differences with respect to conventional systems:

- Concurrent processes may execute during both object creation and message sending.
- The time required to create an object is visible to the programmer.
- The time required to send a message is visible to the programmer.
- Messages may be received in a different order from which they were sent.

The many processes which must cooperate to accomplish the overall problem-solving goal may execute simultaneously. The programmer-visible time delays are significant within the Lamina paradigm because of the activities that may go on *during* these periods, and they exert a strong influence on the programming style.

This paper developed a set of concepts that allows us to understand and analyze the lessons that we learned in the design, implementation, and execution of a simulated real-time application. We confirmed the following experimental hypotheses:

- Performance of our concurrent program improves with additional processors, we attain significant levels of speedup.
- Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.

An inappropriate design of our one-aircraft scenario precluded us from confirming or disconfirming the following experimental hypothesis:

- The amount of speedup we can achieve from additional processors is a function of the amount of parallelism inherent in the input data set.

In building a simulated real-time application in Lamina, we focused on improving performance of a data-driven problem drawn from the domain of real-time radar track understanding, where the concern is throughput. We learned how to recognize the symptoms of throughput bottlenecks; our solution replicates objects and thereby improves throughput. We applied concepts of pipelining and replication to decompose our problem to obtain concurrency and speedup. We maintained a high level of correctness by applying concepts of consistency and mutual exclusion to analyze and implement the techniques of monotonic predicate and predicate transfer with acknowledgements. We recognized and repaired load balance problems, discovering in the process that a modified random processor selection scheme does fairly well.

The achievement of linear speedup up to 100 times that obtainable on a single processor serves as an important validation of our concepts and techniques. We hope that the concepts and techniques that we developed, as well as the lessons we learned through our experiments, will be useful to others working in the field of symbolic parallel processing.

Acknowledgements

We would like to thank all the members of the Advanced Architectures Project, who provided a supportive and stimulating research environment, especially to John Delaney who provided valuable guidance and support throughout this project, and to Bruce Delagi, Sayuri Nishimura, Nakul Saraiya, and Greg Byrd, who built and maintained the Lamina/CARE system. Max Hailperin provided the load balance routines, and also provided insightful criticisms. We would also like to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for their excellent support of our computing environment. Special gratitude goes to Edward Feigenbaum for his continued leadership and support of the Knowledge Systems Laboratory and the Advanced Architectures Project, which made it possible to do the reported research. This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, Boeing Contract W266875, and the Workstation Systems Engineering group of Digital Equipment Corporation.

References

[Andrews 83] G.R. Andrews and Fred B. Schneider, Concepts and notations for concurrent programming *Computing Surveys* **15** (1) (March 1983) 3-43.

[Arvind 83] Arvind, R.A. Iannucci, Two fundamental issues in multiprocessing: the data flow solution, Technical Report MIT/LCS/TM-241, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1983.

[Brown 86]. H. Brown, E. Schoen, B. Delagi, An experiment in knowledge-based signal understanding using parallel architectures, Report No. STAN-CS-86-1136 (also numbered KSL 86-69), Department of Computer Science, Stanford University, 1986.

[Broy 85] M. Broy ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer-Verlag, Berlin, 1985).

[Byrd 87]. G. Byrd, R. Nakano, B. Delagi, A dynamic, cut-through communications protocol with multicast, Technical Report KSL 87-44, Knowledge Systems Laboratory, Stanford University, 1987.

[Cornafion 85] CORNAFION, *Distributed Computing Systems: Communication, Cooperation, Consistency* (Elsevier Science Publishers, Amsterdam. 1985).

[Delagi 87a] B. Delagi, N. Saraiya, S. Nishimura, G Byrd, An instrumented architectural simulation system, Technical Report KSL 86-36, Knowledge Systems Laboratory, Stanford University, January 1987.

[Delagi 87b] B. Delagi and N. Saraiya, Lamina: CARE applications interface, Technical Report KSL 86-67, Working Paper, Knowledge Systems Laboratory, Stanford University, May 1987.

[Delagi 87c] B. Delagi, private communication, July 1987.

[Dennis 85] J.B. Dennis, Data flow computation, Manfred Broy ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer-Verlag, Berlin. 1985) 345-54.

[Filman 84] R.E. Filman and D.P. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software* (McGraw-Hill Book Co., New York, 1984).

[Gajski 82] D.D. Gajski, D.A. Padua, D.J. Kuck, R.H. Kuhn, A second opinion on data flow machines and languages, *IEEE Computer* (February 1982) 58-69.

[Hailperin 87] M. Hailperin, private communication, July 1987.

[Henderson 80] P. Henderson, *Functional Programming* (Prentice-Hall International, Englewood Cliffs, 1980).

[Jefferson 85] D.R. Jefferson, Virtual time, *ACM Transactions on Programming Languages and Systems* 7 (3) (July 1985) 404-25.

[Kruskal 85] C.P. Kruskal, Performance bounds on parallel processors: an optimistic view, Manfred Broy ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer-Verlag, Berlin. 1985) 331-44.

[Kung 82] H.T. Kung, Why systolic architectures?, *IEEE Computer*. (January 1982) 37-46.

[MacLennan 82] B.J. MacLennan, Values and objects in programming languages, *ACM Sigplan Notices* 17 (12) (December 1982).

[Minami 87] M. Minami: Experiments with a knowledge-based system on a multiprocessor: preliminary Airtrac-Lamina quantitative results, Working Paper, Technical Report KSL 87-35, Knowledge Systems Laboratory, Stanford University, 1987.

[Nakano 87] R. Nakano, Experiments with a knowledge-based system on a multiprocessor: preliminary Airtrac-Lamina qualitative results, Working Paper, Technical Report KSL 87-34, Knowledge Systems Laboratory, Stanford University, 1987.

[Nii 86a] P. Nii, Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architectures, *AI Magazine* 7 (2) (1986) 38-53.

[Nii 86b] P. Nii, Blackboard systems part two: blackboard application systems, blackboard systems from a knowledge engineering perspective, *AI Magazine* 7 (3) (1986) 82-106.

[Schlichting 83] R.D. Schlichting and F.B. Schneider, Using message passing for distributed programming: proof rules and disciplines, Technical Report TR 82-491, Department of Computer Science, Cornell University, May 1983.

[Smith 81] R.G. Smith, *A Framework for Distributed Problem Solving* (UMI Research Press, Ann Arbor, Michigan, 1981).

[Tanenbaum 81] A. Tanenbaum, *Computer Networks* (Prentice Hall, Englewood Cliffs, New Jersey, 1981).

[Weihl 85] W. Weihl and B. Liskov, Implementation of resilient atomic data types, *ACM Trans. on Programming Languages and Systems* 7 (2) (April 1985) 244-69.

[Weinreb 80] D. Weinreb and D. Moon, Flavors: message passing in the Lisp machine, Technical Report, Memo 602, Massachusetts Institute of Technology. Artificial Intelligence Laboratory, 1980.