A Parallel Simulation of a Distributed Problem Solving Network

Edmund H. Durfee

Department of Computer and Information Science
University of Massachusetts

September 1984

COINS Technical Report 84-19

Abstract

The research presented describes in detail the implementation of a parallel simulator for the Vehicle Monitoring Testbed (VMT). The VMT simulates a distributed problem solving network, and is used for empiracally evaluating different strategies for organizing networks of loosely-coupled and semi-autonomous nodes which must cooperatively interact to solve a single problem. In implementing a parallel simulator for the VMT, the goal was to both reduce the real time required and to increase the scope of the simulations by splitting the existing simulator (written in Lisp) to run on a network of VAX 11/750s connected by DECNet/ETHERNET-II.

A number of important issues are involved in the implementation of a parallel simulator, such as providing the communication facilities to support cooperation between processes on different machines, maintenance of a global view in a distributed system, testing and debugging in a distributed environment, synchronization between cooperating processes to insure deterministic results, and task allocation among processes to optimize concurrency. This paper describes how these issues were addressed in the parallel simulator for the VMT, and provides experimental data indicating that the parallel simulator results in significant real-time speedup due to the additional CPU power and the combined physical memory available in the network of VAXs.

This research was sponsored, in part, by the National Science Foundation under Grant MCS-8006327 and by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract NR049-041.

This technical report and the research that it represents have been supervised by Victor Lesser.

A PARALLEL SIMULATION OF A DISTRIBUTED PROBLEM SOLVING NETWORK

A Thesis Presented

By

EDMUND HOWELL DURFEE

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

September 1984

Electrical and Computer Engineering

ACKNOWLEDGEMENTS

I would like to thank my advisor, Victor Lesser, for contributing his experience, enthusiasm, and energy to this research. His help in focussing in on the important aspects of this problem was invaluable. I would also like to thank the other members of my thesis committee, Walter Kohler and Jack Stankovic, for contributing their time and ideas to this work.

Dan Corkill possesses both an apparently endless amount of knowledge about computers and the patience to share this knowledge. His conceptual insights and technical expertise were major factors in the timely realization of the parallel simulation. All of the other members of the distributed problem solving group (Jasmina Pavlin, Eva Hudlicka, Joe Walters, Joe Hernandez, and Ed Pattison) have contributed to the successful completion of this research, and I would like to thank them for their assistance (and their patience when I monopolized the machines). I would also like to acknowledge the help of the Research Computing Facility staff, Peter Bates, and Sue Lander.

Finally, I would like to thank my parents and family for their continued support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

"Curtsey while you're thinking what to say. It saves time."
Alice wondered a little at this, but she was in too much awe of the Queen to disbelieve it.
— Lewis Carroll

CHAPTER I

INTRODUCTION

The purpose of this chapter is to motivate the reasons behind the development of parallel simulations, both in general and in the specific context of distributed problem solving networks. An overview of the relevant parts of the particular problem solving network upon which the parallel simulation described in this thesis is carried out, the Vehicle Monitoring Testbed, is also included.

1.1 The Need for Parallel Simulations

As the Red Queen observes in the quote preceding this chapter, one can save time by doing two or more things at once. This is an underlying assumption upon which the need for organizing groups of individuals so that they can cooperatively achieve some mutually beneficial goals is based. In an organization, a number of workers can be applying themselves to separate tasks. If properly coordinated, each worker contributes to the overall goals of the organization, and the organization can attain these goals much faster than if only one worker were to tackle these tasks one after the other.

Similarly, this is the basic assumption behind parallel processing. That is, by dividing a problem into a set of tasks which can be solved in unison, and by assigning each of these tasks to a separate processor, then the amount of time required to solve the overall problem can be reduced. From an external perspective, the organization of processors appears to be doing many things at once, and so, is saving time.

There is an increasing trend toward achieving greater computational power by combining the processing capabilities of a large number of smaller processors, in part due to the recent advances in VLSI and related technologies which have resulted in the decreased price and increased availability of small processors. An important aspect to the realization of improved computational power is that the problems to be solved by these networks of computers must be capable of being divided into subtasks, so that each of the computers has a piece of the problem for which it is responsible. Furthermore, the separate computers must be able to interact in some organized manner so that the results that each produces can be combined to yield an overall solution. If such an organized division of the problem can be achieved, the problem can be solved in reduced time.

As pointed out by Jefferson and Sowizral [JEFF82], one of the most expensive types of computational tasks is simulation. Some simulations can require hours or even days of computer time, and such delays may result in the simulation being of no true value. Consider, for example, a flight simulator that takes five minutes to generate the next image to present to the human aviators. If one of the purposes of the simulator is to train the pilots' reflexes, the particular implementation does not serve this purpose.

The use of parallel processing in the implementation of simulations can result in a better real-time response. Theoretically, one could decrease the run time of a simulation to the serial run time divided by the number of processors used in the parallel simulation (ignoring memory limitations). However, such a result is difficult if not impossible to realize, due to the requirements of the simulation. A simulation is based on a sequence of simulated events occurring in a particular order. Implementation of a simulation on a number of processors running in parallel must insure that the physical distribution of tasks does not alter the ordering of simulated events. Hence, it is likely in a parallel simulation that some processors will have to wait for others in order to keep the simulation time consistent. In other words, there will occur times at which continued computation of one processor could potentially cause errors in the ordering of the simulated events. Mechanisms to insure correct event ordering are summarized in [JEFF82] and [PEAC79].

Hence, there are two major difficulties in the parallel implementation of a simulation that will be addressed in this thesis:

- How the tasks in the simulation should be divided among processors so as to maximize concurrency, thus minimizing run time.
- How the processors should be synchronized so that simulated events occur in the correct order.

Although a number of solutions to each of these problems has been proposed, there does not seem to be any consensus as to the best general solution. Instead, it appears that the methods for task allocation and synchronization are very dependent on the simulation that is to be implemented. It is the purpose of this thesis to explore both of these issues in the specific context of the parallel simulation of a distributed problem solving network.

1.2 Distributed Problem Solving Networks

Problem solving tasks in which the system's input and output are spacially dispersed would require a large amount of communication to route information to a single, powerful computer. In order to reduce the amount of communication as well as the computational demands on such a centralized processor, a distributed network of cooperating, semi-autonomous computer systems can be used. If these computer systems are sufficiently well coordinated, such an arrangement could be quite effective in solving the distributed problem.

The cooperating, semi-autonomous computer systems are refered to as *nodes*, and a distributed problem solving network is composed of a network of these nodes. Although the nodes need not in practice be physically distributed, it is assumed throughout this thesis that they are. Therefore, communication between nodes is limited by the physical properties of the communication channel, such as delay, error rate, and channel capacity.

The physical isolation of nodes means that node activity is loosely-coupled; node interaction is limited. For example, this means that there are times when it may be faster for a node to redundantly derive information rather than wait for another node to supply it. Nodes will not necessarily possess current views of the state of problem solving at the other nodes in the network, and so, a node may have information that is incomplete or inconsistent with the information of another node. Hence, results on a node are considered to be tentative, and nodes exchange tentative partial results in order to cooperatively converge to acceptable results despite incomplete or inconsistent views of the problem. This approach is called functionally accurate, cooperative distributed problem solving [LESS81].

An area of particular interest in distributed problem solving networks is that of imposing some sort of higher level organizational control upon these relatively independent problem solving nodes so as to effectively solve a distributed problem. Each node must have some general concept as to what part of the problem it is responsible for, and how it is expected to interact with its fellow nodes. In order to study the complex issues involved in the specification of an organizational structure for cooperating semi-autonomous nodes, a simulation of such a node network was implemented so as to act as a framework in which such organizations could be studied. This simulation is the Vehicle Monitoring Testbed [CORK83,LESS83].

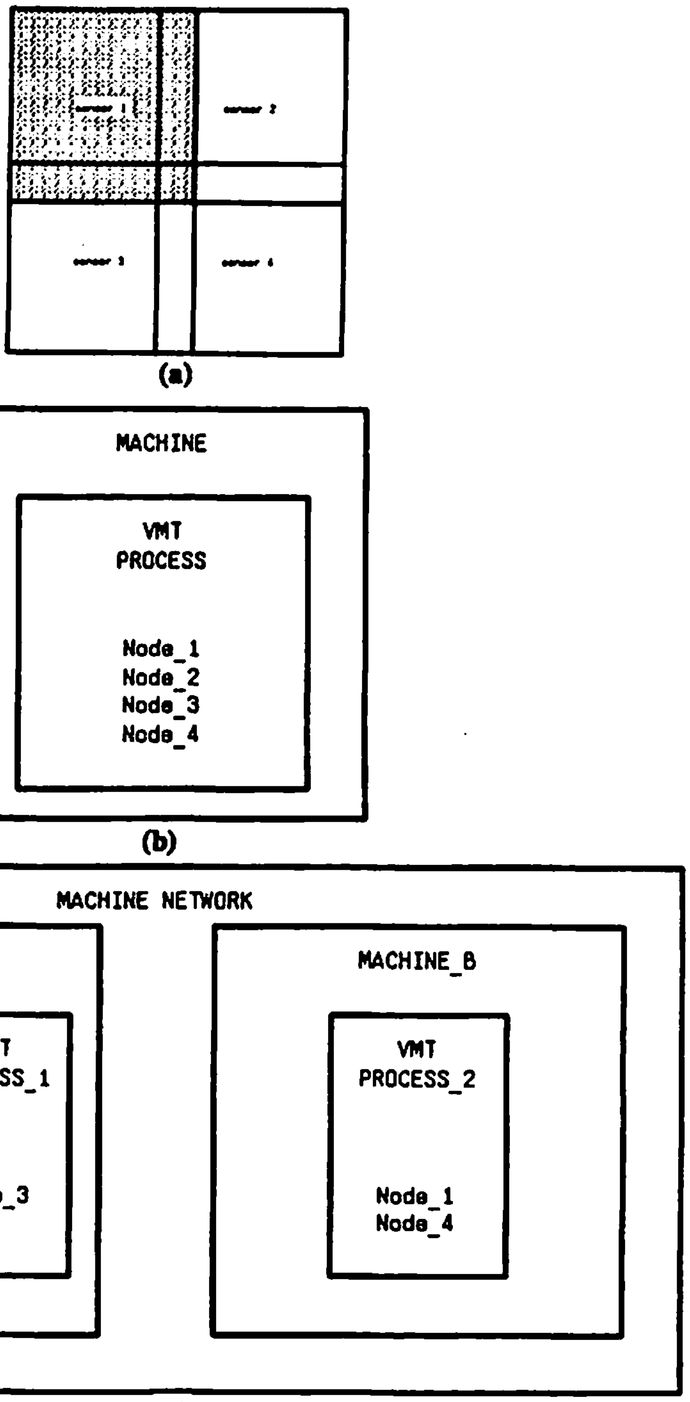
It is beyond the scope of this thesis to explain all of the details of the Vehicle Monitoring Testbed. Rather, the major thrust of the research presented herein involves exploiting the fact that *the tasks of the simulation are already divided among the simulated loosely-coupled nodes.* It is the inherent concurrency of these nodes which will allow a straightforward implementation of a simulator capable of decreasing the run time through parallelism — by assigning nodes to separate processors running in parallel, the tasks of the simulation will be performed in parallel. This thesis will thus address the previously mentioned issues of how to divide the computational demands of the simulation among processors and how to keep these processors sufficiently synchronized such that the order of simulated events remains deterministic.

The basic idea behind this work is simply to modify the simulator so that some number of processes each running an instance of the simulator can interact to cooperatively run an entire simulated environment. Each node in the simulated environment would be assigned to one of these processes, and by distributing the nodes among the concurrently running processes, parallelism will result.

It is important to differentiate between the simulation of the distributed problem solving network and the implementation of that simulation. This research deals principally with the latter; the fact that the simulation is being run on more than one machine should have no bearing on the results of the simulation, but only on the rate of the simulator. In order to further differentiate between the simulation and its implementation, the following terminology will henceforth be used (figure 1):

- **node** — A *simulated* semi-autonomous problem solving system.
- **node network** — A simulated distributed network of communicating, semi-autonomous problem solving nodes.
- **process** — An instance of the VMT simulator.
- **machine** — A *physical* computer capable of supporting one or more processes.
- **network** — A *physical* collection of one or more machines, capable of communicating with each other.

The implementation of the parallel simulation is inextricably linked with the simulation of the node network itself. Hence, the development of the implementation requires an understanding of certain relevant aspects of the VMT. This is the purpose of the next section. Following this, an overview of the rest of the thesis will be presented.



(c)
Figure 1: An Example of the Terminology.

An example node network consisting of four nodes, each responsible for a portion of the total sensed area (a). Note the sensor overlap. In (b) the non-distributed assignment is shown, while (c) presents a possible hierarchical assignment of nodes to processes, processes to machines, and machines within the network.

1.3 Relevant Aspects of the VMT

The Vehicle Monitoring Testbed, as mentioned above, is a simulator for a cooperative distributed problem solving network. The testbed simulates a network of semi-autonomous nodes attempting to identify, locate, and track vehicles moving through a two-dimensional space using signals detected by acoustic sensors. Nodes receive signal data at discrete intervals and attempt to use this data to generate hypotheses as to the movements of vehicles. Each node is a sophisticated problem solving system that can modify its behavior as circumstances change and plan its own communication and cooperation strategies with other nodes. A crucial aspect of the network is that no single node has sufficient local information to make completely accurate processing and control decisions without interacting with other nodes.

Each of the problem solving nodes has the architecture first developed in the Hearsay-II speech understanding system [ERMA80], with extensions to this architecture that enable the nodes to have more sophisticated control and to give the nodes the ability to communicate hypotheses and goals (figure 2). Hypotheses and goals are kept on separate data structures called blackboards, and each of these blackboards is further divided into levels, corresponding with the level of abstraction of the hypotheses or goals contained therein. Currently, there are four levels, ranging from the least abstract (signal data) to the most abstract (patterns of vehicle tracks).

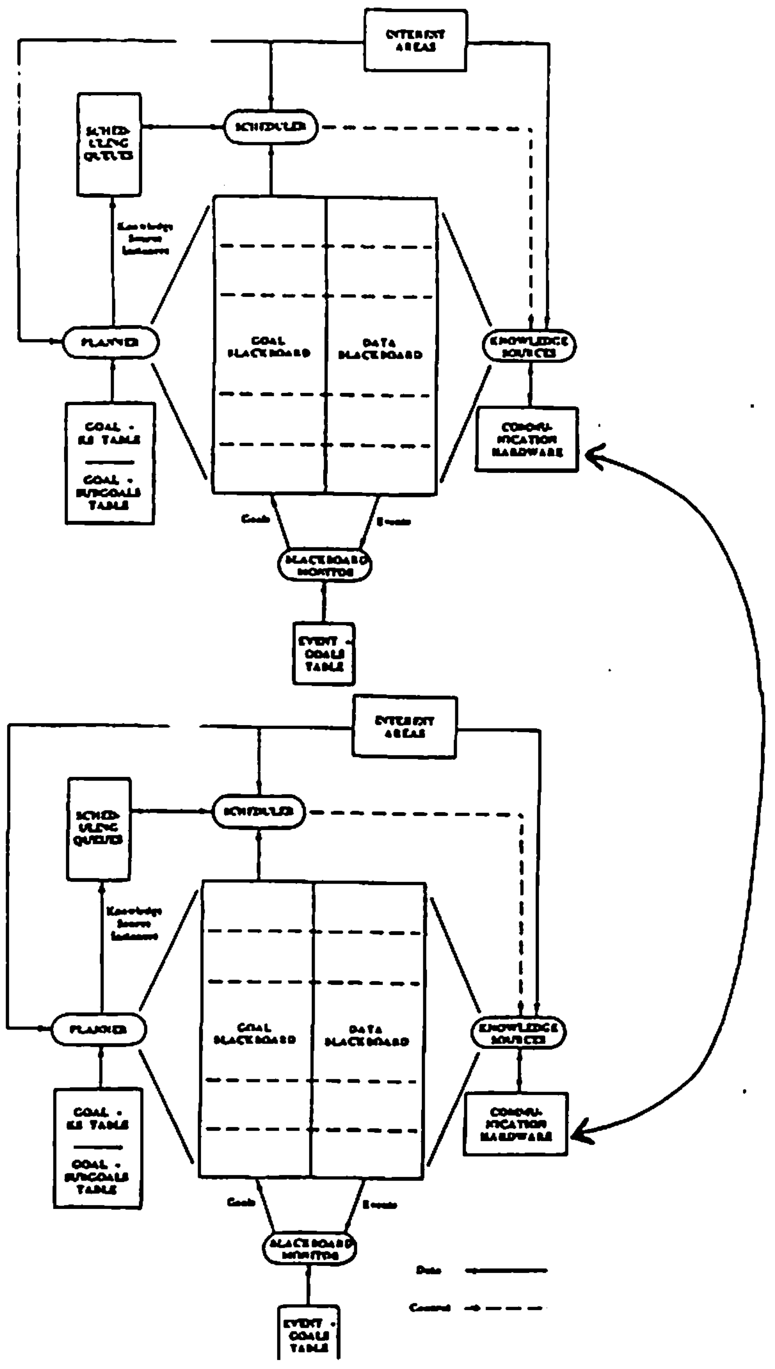


Figure 2: The Problem Solving Architecture.

The problem solving architectures of two nodes, and communication links between them.

A hypothesis is created and inserted onto the data blackboard as a result of the execution of a knowledge source (KS). Examples of KSs include KSs to create signal hypotheses out of sensor data, KSs to group hypotheses from a lower level of abstraction into likely hypotheses at a higher level, KSs to take hypothesized tracks and attempt to extend them forward or backward, and KSs to merge two pieces of a track into a single track. Each created hypothesis has a calculated belief associated with it which acts as an indication as to how accurate that hypothesis is estimated to be.

The data blackboard is monitored, and the appearance of hypotheses on it trigger the creation of goals. These goals represent potential hypotheses that might eventually be created, as indicated by the current state of the data blackboard. Each goal has a rating which is based on a number of factors, including the beliefs of the hypotheses that triggered its creation and the node's perception of its role in the problem solving network.

In turn, the goal blackboard is used by the planner. The planner determines which KSs can be used to satisfy each of the goals, or can create subgoals in order to eventually satisfy a goal. In particular, the planner can execute precondition functions for specific knowledge sources acting on certain data, and these precondition functions will provide some estimate of the utility of the knowledge source in processing that data. Based on this information, the planner creates knowledge source instantiations (KSIs). Each KSI represents a request to execute a KS on specific hypotheses in order to satisfy certain goals, and is rated based on the goals it satisfies as well as the hypotheses it uses. Each node maintains a queue of pending KSIs, kept in order of rating and a node will always execute the highest rated KSI currently on its pending queue.

Therefore, the execution of a KSI results in the creation of some number of hypotheses, which trigger the creation of some number of goals, which in turn cause the creation of some number of KSIs, and the cycle repeats until either a hypothesis which covers the entire solution is created with sufficient belief, or the pending KSI queue becomes empty and the node goes into a state called quiescence. The use of goals and the rating of KSIs serves to focus the attention of the problem solving nodes along the path that appears most promising.

1.3.1 Node Communication.

There are special knowledge sources that exist for communication of hypotheses and goals. As mentioned above, each node has some knowledge as to its global role in the development of the solution, and this information is contained in the *interest areas* data structure. Based on this knowledge, a node knows to what other nodes it should send hypotheses and goals, as well as when to send them. This data structure also contains the information that a node needs as to what nodes it can receive hypotheses and goals from, as well as how to modify their beliefs and ratings, respectively, before incorporating the transmitted information into its own blackboards.

When a node creates a hypothesis which should be sent to some other node (as specified in the interest areas), the planner will create a KSI to invoke the hypothesis sending knowledge source. Similarly, when a goal is created that could be a useful goal for another node (again, specified by the interest areas), a KSI is created to send it. Another reason that a goal might be sent is if the local processing needs a hypothesis lying in the range of another node (to extend a track, for example). In this case, the node instantiates a different type of goal sending knowledge source, which will be perceived at the receiving node as a request for information rather than a suggestion for the focussing of attention.

Conversely, there is a matching set of receiving knowledge sources. When a hypothesis is received, it causes the instantiation of a KS which will incorporate that hypothesis into the data blackboard. The reception of a goal will similarly cause a goal reception KSI to be inserted on the pending KSI queue, and in the case of a goal which requests information, the KS will also stimulate the instantiation of the hypothesis sending KS that will reply to the request. The interest areas play a role in determining if the received hypothesis or goal is of any interest, whether its belief or rating should be altered, and the role of that hypothesis or goal in future calculations. For example, if a node is very locally directed (as specified in the interest areas), it might choose to ignore or rate lowly any received goals. On the other hand, if the node is externally directed, it might drop whatever it's doing in order to satisfy a received goal. The specification of the interest areas may therefore make a great deal of difference in the effectiveness of the network of nodes in deriving the solution.

1.3.2 Node Scheduling.

Because the testbed must simulate a network of loosely-coupled nodes running concurrently, strategies have been developed to interleave node activities in order to achieve apparent concurrency on a single machine. To this end, the initial implementation gave each node access to the processor in a round-robin fashion.

Each node would execute the next KSI on the pending queue, and then relinquish the processor; if the pending queue for that node was empty, the potential knowledge source execution was lost. When each of the nodes had been given a turn, the node network cycle counter was incremented and another cycle was begun. Hence, simulation time was measured in terms of the number of cycles executed. The node network cycles would continue until any node had generated a satisfactory answer. It should be noted that this method of scheduling results in a simulation where all knowledge sources require the same amount of simulated time to execute.

In order to simulate communication, each node had separate queues for transmission KSIs and for reception KSIs. This was intended to simulate separate sending and receiving processors which could operate concurrently with the local problem solving processor. Certain parameters were included in order to simulate bursts of communication, allowing nodes to transmit a certain number of messages over a set number of cycles. Clearly, if the simulation would allow the sending and receiving knowledge sources to take some amount of time relative to the locally running KS, such approximations would be unnecessary. In addition, since a message that was simulated to be sent was merely put into a buffer at the receiving node, and this buffer was emptied each time the receiving node executed, all messages were assumed to take the same amount of time to reach their destinations. It appeared that a much more realistic simulation would involve the specification of delays between pairs of nodes.

1.4 The Remainder of the Thesis

The remainder of this thesis is concerned with the implementation of a parallel simulation of the VMT, and can be broadly divided into these major parts:

- Chapter II is concerned with the implementation of the tools that will enable the user to create the processes to be run in parallel, and the methods by which the processes can communicate.
- Chapter III outlines the changes required to the VMT environment in order to provide for the parallel simulation. The fact that all of the simulated nodes no longer share memory is addressed, and techniques for testing and debugging the parallel simulation are described.
- Chapter IV deals with the methods used to achieve satisfactory synchronization between processes, and will explore the interrelationship between scheduling nodes on a process and synchronization.
- Chapter V briefly investigates the second major issue, that of allocating tasks to processes in order to achieve a well balanced simulation. With that background, experimental results are provided in order to make some conclusions as to the effectiveness of the parallel simulation.
- Chapter VI summarizes the contributions of this research, and considers areas in which future work might be warranted.

"Speak when you're spoken to!" the Queen sharply interrupted her.

"But if everybody obeyed that rule," said Alice, who was always ready for a little argument, "and if you only spoke when you were spoken to, and the other person always waited for you to begin, you see nobody would ever say anything, so that—"

"Ridiculous!" cried the Queen. "Why, don't you see child—" here she broke off with a frown, and, after thinking for a minute, suddenly changed the subject of the conversation.

— *Lewis Carroll*

CHAPTER II

PROCESS CREATION AND COMMUNICATION

A major concern in the implementation of the parallel simulation was to attempt to provide a clear interface between the VMT and the underlying processes and communication routes upon which the VMT was run. Due to the perceived likelihood that the operating systems for the individual machines as well as the network might change over time, it was hoped that these changes would be transparent to the VMT system. That is, the VMT should achieve communication between nodes via a small number of commands (such as send-to-node and read-incoming-messages), the exact implementation of which is handled on a lower level and is transparent to the VMT. In this chapter, the details of this lower level of communication are presented. This level includes the details of establishing the VMT processes that will be run in parallel, providing communication routes between them, and enabling the run to eliminate these processes when they are no longer needed.

2.1 Buffer Processes

The parallel simulation was implemented on a set of Vax 11/750 computers operating on VMS and interconnected by DECNet/ETHERNET-II. Interprocess communication is achieved through the use of mailboxes, as provided by the VMS system service. Associated with each of the VMT processes that are cooperating on the run is a mailbox that will contain the incoming messages to that process. Hence, in order to communicate with another process, a process need merely put a message into the other process's mailbox.

Because the processes are running concurrently it is very unlikely that a process will be reading from its mailbox at exactly the same time that a sending process will be inserting a message. Since we do not want either process to be forced to wait on the other at this level (synchronization between processes occurs on the VMT level), the mailboxes should be asynchronous, so that a process can deposit a mail message without waiting for it to be read, or can return from inspecting its mailbox even if there was not a message in it. Such capabilities are available in in the local operating system, requiring some setting of bits associated with the mailbox object. However, network operating system does not support asynchronous communication; all communication across the network must be done synchronously.

Therefore, it was necessary to provide a means by which the enforced synchronous communication across the network would not cause VMT processes to wait for each other. The means by which this was accomplished lead to the development of the buffer process. Each machine involved in the simulation has a buffer process, and it is the responsibility of the buffer process during a run to wait for messages to be put into its mailbox and transfer them to the appropriate mailboxes of VMT processes on that machine. VMT processes on different machines would thus communicate to each other through the buffer processes on those machines.

Buffer processes are coded in FORTRAN rather than CLisp (a LISP dialect in which the VMT is encoded) for a number of reasons. First, since the relatively simple string manipulations needed for determining the destination of a message and forwarding it are much more optimized in FORTRAN, speed is increased. Second, because CLisp images require much more memory, FORTRAN helps minimize requirements for memory resources better spent on the VMT processes. Finally, because calls to system services can be made directly from FORTRAN, buffer processes could easily take on the added responsibilities for initializing many features of the parallel run, as will be discussed below.

It should be noted that this is certainly not the only way to provide asynchronous communication across the network. For example, the use of asynchronous traps (ASTs) in alerting processes to message arrivals in their mailboxes is another alternative. Even if the receiving process were not ready to make use of that message, it could simply store the message until it was ready. This course was not taken for a number of reasons, the principal reason being that CLisp does not support ASTs. Future changes in the language of implementation used in the testbed may make this choice feasible. The major tradeoffs involved in buffer processes concern whether the increased flexibility of the buffer processes compensates for the extra overhead of creating an extra process per machine.

2.2 Buffer Process Creation

The user desiring to begin a VMT run starts a VMT process on a machine. This process, as the process responsible for the run, is known as the *master* process. Its first act is to read the parallel simulation information from a previously prepared file specified by the user. This file is known as a *distribution file*, and an example of such a file is given in Appendix A. In this way, the master learns what other processes to create, what machine to create each on, what nodes each will be responsible for, and what commands to have each execute.

The first step is to create the buffer processes on each machine, starting with the machine on which the master resides. The master creates a mailbox and then spawns a temporary subprocess, which in turn invokes the DECNet remote task capability to begin a task on the specified machine. This task is passed the name of the master's mailbox. The task then enters the buffer process code which creates a mailbox for the buffer process. Finally, the buffer process inserts the name of this mailbox into the master's mailbox, and a communication link between the two is thus formed.

Note that this scenario was somewhat simplified. For example, in order to pass the mailbox names between the command language and the code, the names had to be passed indirectly using services provided by the system. Also, it is important to realize that only the buffer process on the master's local machine is given the name of the master's mailbox. All other buffer processes are given the name of the buffer process local to the master, which forwards the messages on to the master.

2.3 Detached VMT Process Creation

Once all of the appropriate buffer processes have been created, it is time to create the VMT processes that will work with the master process on the parallel VMT simulation. The master process informs all of the buffer processes as to what VMT processes to create on their respective machines. Each buffer process in turn creates a mailbox for each of its local VMT processes, and then uses the system service to create each process. This system service allows the buffer process to specify the input and output devices for the created process, so that by specifying the proper mailboxes, communication between the buffer process and each of its detached processes is automatically achieved.

Each of the detached processes is told by the buffer process to enter a VMT image. The buffer process can then initialize some of the process's variables, particularly those required to establish communication links with the other VMT processes on the same machine. It is important to note that, since the buffer processes are each doing these things in parallel, significant time savings can occur compared to a scenario in which the master had to create each of the detached VMT processes itself. In addition, the ability for the buffer processes to make direct calls on the system services allows much more flexibility in terms of the specific parameters used to establish the detached VMT processes. This has proven to be quite useful, although the details will not be mentioned here for brevity.

Deletion of all of these processes should also not be overlooked. Again, the master has control of the future of the processes created on its commands. The master can send a message to a buffer process asking it to delete a specified detached VMT process. In addition, the master can also ask the buffer process to delete itself. This is a more important aspect of the simulation than might be obvious. Since the VMT processes are detached, they will not disappear if the master is deleted, and will remain on the system using up resources until stopped explicitly. In a run involving a large number of detached processes, failure to stop them at the conclusion of a run could mean wasted time and resources spent on tracking them down and eliminating them.

Finally, between the creation and deletion phases of a run, the buffer process enters its transfer mode. Each incoming message is examined and the header describing the destination is stripped off. The message is then inserted in the mailbox of the destination process, and the cycle repeats. Headers contain certain symbols which are immediately recognized by the buffer process, and differences in these symbols inform the buffer process as to whether the message is to be transferred to another process, or if the message contains control information for the buffer process itself (such as commands to delete a VMT process, etc.). The simple protocols for these messages will be briefly summarized later in this chapter.

2.4 Sample Stepwise Scenario of Run Initialization

In order to illustrate the previous description, the sequence of events for the simple example from figure 1 will here be described. The steps refer to the numbers in figure 3.

1. MASTER VMT process creates its mailbox and makes a process logical with its name.

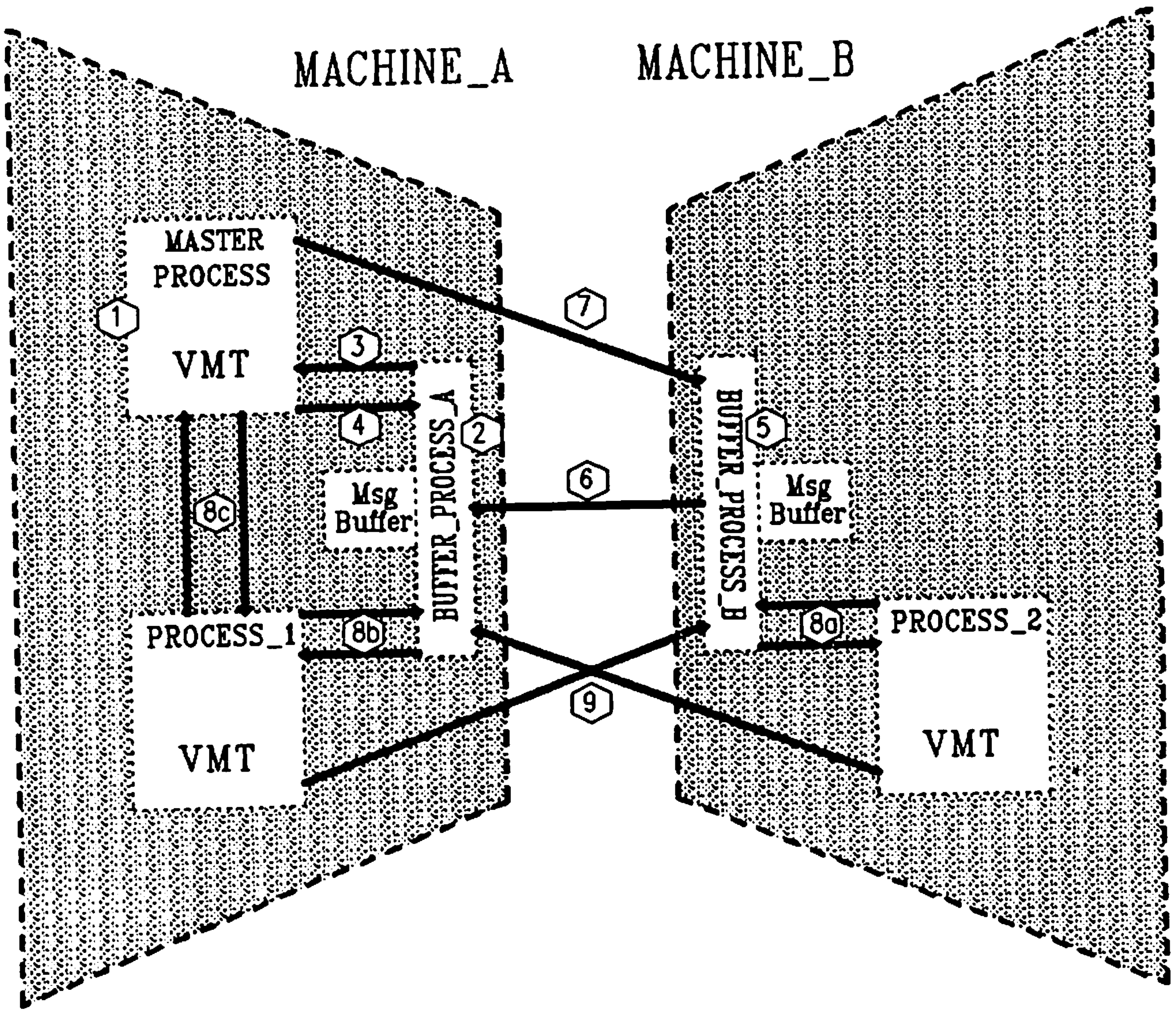


Figure 3: Sample Process Creation Diagram.

Depiction of processes and communication links between them for a possible parallel simulation. Numbers refer to order of events as specified in the text.

2. MASTER spawns process which begins its local buffer process (BUFFER_PROCESS_A) and then spawned process is deleted.
3. BUFFER_PROCESS_A creates its mailbox, translates the previously made logical to find MASTER's mailbox and establishes link.
4. BUFFER_PROCESS_A sends its mailbox name to the MASTER, which then establishes communication link.
5. MASTER spawns process which begins remote buffer process (BUFFER_PROCESS_B). Spawned process then deletes itself. BUFFER_PROCESS_A's mailbox name passed as a logical.
6. BUFFER_PROCESS_B creates its mailbox, and translates logical to establish link with BUFFER_PROCESS_A. Sends its mailbox name to BUFFER_PROCESS_A.
7. BUFFER_PROCESS_A passes BUFFER_PROCESS_B's mailbox name on to MASTER and MASTER establishes communication link with BUFFER_PROCESS_B.
8. MASTER then informs buffer processes of other VMT processes. They concurrently perform: i) BUFFER_PROCESS_B creates PROCESS_2 (8a) and ii) BUFFER_PROCESS_A creates PROCESS_1 (8b) and establishes communication links between MASTER and PROCESS_1 (8c).
9. Finally, MASTER informs each detached process of the buffer processes remote to it, and each establishes links to all buffer processes remote to it.

Deletion of processes is a much simpler task, since the master can now send messages directly to the buffer processes, which can in turn do the deletion. Note that by deleting a process, the communication links involving that process are automatically deleted.

2.5 Shortcomings of the Approach

Clearly, this approach performs its tasks of creation and deletion of the processes needed to perform the parallel simulation. However, consider its performance on the task for which it was originally conceived, the task of transferring messages from processes on remote machines to their destination processes on the local machine. This task is provided for only in a very simple manner. Messages coming into the buffer process are examined to determine the destination, and passed on to the mailbox of the destination process.

However, what occurs if this mailbox is full? The capability of writing asynchronously into a local mailbox depends on there being sufficient room in that mailbox for the entire message to be deposited. If there is not sufficient space, the writing process waits until there is. Thus, if the mailbox is full, communication into that mailbox becomes essentially synchronous because the writing process must wait until at least one of the preceding messages is read.

Consider the scenario where two processes each have full mailboxes, and each is trying to send a message to the other. Each is waiting for the other to read from its mailbox, and as Alice duly points out in the quotation preceding this chapter, if everyone were to wait for someone else to start something, then nothing would ever occur. If there is any cycle of processes trying to write into full mailboxes, the underlying communication mechanism will enter a state of **deadlock**. As a matter of fact, in one of the earlier versions of the code, synchronization required the master to send a message to itself. If its own mailbox was full, the master would put itself directly into deadlock.

One may minimize the probability of deadlock occurance by increasing the size of the mailboxes to such an extent that they never get filled up. However, there is no way to predict what size the mailboxes must be to insure that they will never get full. Different runs of the VMT can produce vastly different amounts of messages. Therefore, another method must be found which prevents the system from entering a deadlocked state.

Another difficult problem with the transfer of messages concerns two facts. First, there is no way to predict the maximum size that a message can attain, so that it must be possible for a message to be broken up into pieces, each sent with the appropriate header. Second, since each process has only one incoming mailbox, it is possible for two or more processes to be inserting messages into the same mailbox at the same time.

Taken together, these two facts indicate potential difficulties. If messages are broken into pieces when they are sent, it is possible for the pieces of two or more messages (being sent by two or more processes) to become interleaved in a mailbox. Since messages to VMT processes have no headers, it is impossible for those processes to determine what pieces go together. Indeed, it is hoped that this should not be a concern for the VMT processes, and instead is a problem for the underlying communication tools to solve.

The problem of interleaved messages is certainly less detrimental than the deadlock problem in the respect that the processes will continue with the data if possible and terminate abnormally otherwise. Hence, at least the processes will terminate eventually. On the other hand, on the unlikely chance that interleaving has occurred and has not caused an unreadable message to be constructed, the effects of an interleaved message might be difficult to detect and could result in a faulty run being accepted as accurate.

Both the possibility of deadlock and of interleaved messages must therefore be eliminated from the underlying communication structure as presented. The majority of the remainder of this chapter deals with the method used to insure that these problems never arise.

2.6 Modification of Buffer Processes

Although there are a number of ways of dealing with the problems of deadlock and interleaved messages, the way adopted in this research involves adding to the buffer process the ability to store and forward messages. Buffer processes will thus have two modes, one for transferring messages directly, and the other for storing incoming messages until those messages are requested by their destination process.

When in a store and forward mode, the buffer process will automatically store any incoming messages. Whenever a process desires the messages intended for it, it must send a control message to its buffer process requesting that it forward the messages. An added modification is that there are two ways that forwarding can be requested. One of the ways requires the buffer process to forward all messages (if any), terminated by a message to inform the receiving process that all messages have been forwarded. The other way requires the buffer process to check to see if there are any messages to be forwarded. If so, forward them, but if not, then do not respond to the requesting process until there is a message for it. Basically, these two methods correspond to read-no-wait and read-wait commands, respectively.

Since buffer processes are only interposed between processes on different machines, this solution does not prevent deadlock if processes on the same machine get into a cycle. A simple way to remedy this is to force all processes to communicate via the buffer processes, regardless of their relationship with the process to which they are sending.

The need for this change has been obviated by a simplification to the buffer processes themselves. It was found that having multiple detached VMT processes on the same machine causes tremendous amounts of page faulting. For example, running two VMT processes concurrently on the same machine causes each process to take much more than twice the amount of time that it takes each to run alone. Therefore, in order to avoid all this thrashing of memory, it is a much better solution to combine all of the nodes from the processes on a particular machine into a single process to be run on that machine. In addition, this causes major simplifications to the buffer processes. If the buffer process were to be able to store and forward messages for numerous processes, it would require the overhead of allocating sufficient space for the storage of each processes messages, as well as the computational overhead of keeping track of the status of the storing and forwarding for each process.

The modification of the buffer processes thus called for adding the additional complication of providing a store and forward capability while eliminating the need for maintenance of state information for each of the detached VMT processes on the machine. In this manner, the overhead of message transference has been kept to a minimum, while the benefits of deadlock prevention and message integrity, as outlined below, have been gained.

2.7 Deadlock Prevention

During initialization and termination of the parallel run, the master has complete control over communications that occur. Either the master is itself sending a message, or has requested another process to send a message to some process. During these phases of the run, all of the other processes are waiting on messages from their mailboxes, and so, are constantly in a reading mode. Therefore, the master knows when it expects a response and will wait for messages in its mailbox whenever appropriate. Since the master always accepts messages coming into it, no process will have to wait on the master's mailbox. Furthermore, unless they are sending messages to the master (which will always be read), the processes are always reading from their mailboxes, so the master will never have to wait for any other process's mailbox. As a result, there can be no deadlock during these phases.

Simply put, because all communication during initialization and termination of the parallel run follows in a deterministic order, there can be no deadlock as long as the master is programmed correctly. In fact, during these phases of the run, it is extremely important for the processes to behave this way, since the processes created by the master must react to messages sent by the master. It is for this reason that the buffer processes have a transfer mode of operation. While in this mode, a buffer process merely acts as a transferring station, assuming that the system behavior is deterministic and there is no need to prevent deadlock. The detached VMT processes need not explicitly request the buffer processes to forward messages since it is done automatically.

However, during the actual run of the VMT, processes can be sending messages to other processes in an unpredictable manner. It is at this time that deadlock looms. For this phase of the run, the buffer processes go into a store and forward mode, in which they will store any incoming messages. The detached processes must explicitly ask the buffer processes to forward these messages.

It can be seen that, if the VMT processes are properly programmed, no deadlock can occur in this mode. Specifically, if a VMT process goes into a reading mode immediately after issuing a forward command to its buffer process, and does not stop reading until the buffer process informs it that all of the messages have been forwarded, then the VMT process's mailbox can never contain messages that will not be read. Similarly, since the buffer process is only forwarding messages when the VMT process is guaranteed to be reading them, the buffer process will never be forced to wait for any extended amount of time for there to be sufficient room in the mailbox of that VMT process. When forwarding, the buffer process is assured of returning in finite time, at which point it will continue reading messages out of its own incoming mailbox. Since the buffer process is guaranteed to read from its mailbox in a finite amount of time, any process sending messages to the buffer process is guaranteed of successfully inserting the messages in finite time.

From this description, it is apparent that there is no possibility of a process having to wait for an infinite amount of time for an event to occur (assuming that the VMT processes are correctly programmed), so deadlock cannot occur. Therefore, the combination of simple message transfer during deterministic phases of the run with a store and forward mechanism during the more complicated phases of the run serves to insure that deadlock will never occur.

A final point to mention concerning deadlock involves the size of the message buffers in the buffer processes for storing the messages to be forwarding. It has already been noted that there is no way to predict the amount of space needed to store all of the incoming messages in a mailbox, and similarly, there is no way of predicting how much storage space should be allocated to the buffer processes. However, there is an important distinction between the two types of space. For mailboxes, there is no way in which a sending process can determine ahead of time whether there is enough storage space to accept the message. The sending process merely tries to insert it, and waits if there is not sufficient room. This is how the deadlock occurs. On the other hand, the buffer process keeps a running tally of what storage space is used and what is available. If there ever occurs a point where there is insufficient space to store an incoming message, the buffer process will know this without going into a deadlock state. Instead, the buffer process could simply throw out the overflow messages, and send some error message on to the VMT process informing it of a buffer overflow. In this way, although the problem of buffer size has not been solved (it is really an empirical problem) and there is no guarantee that messages can always be transferred successfully, errors caused by insufficient buffer space will cause the run to abort, rather than causing deadlock.

2.8 Message Integrity

The problem of insuring that messages will be received correctly at their destinations can be divided into two cases, in a manner similar to the deadlock problem previously discussed. The more simple case, that of deterministic communication, is trivial. Since the master is communicating with the other processes in a serial fashion and no other process could possibly be communicating without the master's knowledge, then correct coding of the master's activities make it impossible for any message interleaving to occur. Because messages sent by a process are received in the same order in which they were sent, errors in order of message pieces can only occur if two or more processes are simultaneously transmitting to the same destination.

In order to protect against message interleaving in the more general case, the buffer process, as it stores messages, must have three pieces of information. It must know the destination of the message (trivial for our modified buffer processes that serve only one detached process). It must know the source of the message, since it wishes to combine parts of a message from the same source together. Finally, it must know when a particular message is complete; that is, it must know when all of the pieces for a message have been received. This information is provided to the buffer process via the message headers and a trailer message and these are described below.

All messages to a buffer process are preceded by a header and have the form %DESTINATION%%SOURCE%$message. DESTINATION is the name of the destination process, SOURCE is the name of the sending process, and %, %%, and

%$ are used simply as delimiters to these fields. As a matter of contrast, all control messages to buffer processes begin with a $, and in this manner the two types of messages are distinguished from each other.

Upon receipt of a message, the header is stripped off, and the message and its source are stored. A list of message sources is also updated if necessary, and a flag with the source is set, marking that a possibly incomplete message is in transit. This flag is checked whenever the destination process requests that its messages be forwarded. The buffer process will only forward messages if none of the incomplete message flags for the sources is set. If one or more are set, then the buffer process will record the forward request, and then continue reading in messages and storing them, each time checking whether they can now be forwarded. As soon as all of the flags are reset, the buffer process forwards the messages to the destination process.

When a process has completed sending a message, it sends a terminator message after it. This has the same form as any other message, but includes a special sequence of characters in the message that the buffer process keys in on. When such a message is received by the buffer process, the flag associated with the source of that message is reset to indicate that the message has been received in full, and the actual terminator message is then discarded.

The buffer process does not allocate separate storage area to each source process, since the number of source processes for a given store and forward cycle cannot be predicted, and statically allocating storage space for each at the start of the run could result in one source overflowing its space while another source has unused space. Therefore, the buffer process has one large block of storage space, and simply places messages into it as they arrive, being careful to associate with each of these messages its source.

When the buffer process forwards messages, therefore, it cannot simply empty the storage buffer in order since pieces of messages from different sources may be interleaved in this storage. However, since a source process cannot interleave with itself, the simple remedy is to step through the source list that has been maintained, and for each source, send the stored messages from that source in the order in which they were received. As previously noted, all of the pieces for a divided message must be present during a forward operation, so the VMT process is guaranteed to receive only complete and correct messages.

2.9 Summary of the Interprocess Level

The tools described in this chapter are those used to implement the underlying process maintenance and interprocess communication needs for the parallel simulation. They interface with the VMT code in such a way that the details of this implementation are all but transparent to the VMT.

In providing the details of methods of creating and deleting processes and establishing communication routes, it is hoped that many of the important considerations involved, as well as the limitations encountered within the framework, have been introduced. Although the tools might have been implemented in other ways, the author hopes that at least the reasons behind his choices are now clear. In addition, the problems of deadlock and message integrity are important in many applications of cooperating and communicating processes. Although the implementation discussed above offers no new solutions, it is hoped that it does indicate how one solution can be implemented in this particular domain that insures against these difficulties without the incurrance of unreasonable overhead.

Finally, it should be noted that the tools indicated here were the first step in the development of the parallel simulation, since without them there could be no communication. Throughout many the changes to the VMT code itself, the underlying tools for process maintenance and communication have remained basically unchanged and have provided a dependable foundation to the remainder of the simulation.

"Of course they answer to their names?" the Gnat remarked carelessly.

"I never knew them do it."

"What's the use of their having names," the Gnat said, "if they won't answer to them?"

"No use to them," said Alice; "but it's useful to the people that name them, I suppose. If not, why do things have names at all?"

— Lewis Carroll

CHAPTER III

CHANGES TO THE VMT ENVIRONMENT

In the design and implementation of simulations of distributed systems, a major concern is that of maintenance of some sort of global perspective or centralized information. By reference to such information, the user can observe the overall state of the simulation. Such information can also be used by the simulation itself in the fulfillment of its objectives.

Because the VMT was originally implemented to be run on a single process, the user of the simulation had direct access to all of the information in the simulation. In addition, since all of the simulated nodes in actuality shared memory, nodes could read and even modify the data structures on different nodes. With careful planning, the use of such techniques could allow the implementor to take some shortcuts in information passing and global maintenance.

A parallel simulation of the VMT requires that nodes be distributed over some number of processes. Although the nodes simulated on the same process will share memory, nodes on different processes will not have this capability. Hence, the communication between nodes can no longer consist of reading and modifying each others data structures, but instead requires nodes to send explicit messages to each other, these messages containing all of the pertinent information. In addition, a single process will no longer have a global view of the simulation; the processes themselves must communicate in order to maintain a relatively current view of the overall state of the simulation.

In this chapter, the changes required to the VMT environment needed to effect the parallel simulation are addressed. Changes to the data structures, to the creation and maintenance of global information, and to the implementation of explicit message passing between nodes have been necessary. Furthermore, the methods by which a run is analyzed and debugged have been required to change, and the techniques used in performing these tasks will also be briefly addressed.

3.1 The Parallel Simulation from an Internal Perspective

In order to enable the VMT to run on a set of parallel processes, a number of changes to the internal organization of data and the methods by which both global and node-specific information are used are necessary. The requisite modifications affect only the internal mechanics of the simulation and are nearly transparent from an external view of the testbed.

3.1.1 Changes to the Data Structures.

Each node in the VMT system has a tremendous amount of information associated with it. Attributes of the node, its queues, its blackboards, its sensor data — all require space in which to be stored. In order to maintain this information, large data structures are created at the start of a run which reserve space for all of the data for each node in the node network. Since nodes are given consecutive integers as names, these data structures typically take the form of arrays whose indices correspond with the node names.

In a parallel simulation, only a subset of the nodes in the entire node network will be resident on a particular process, so there is a tradeoff to be considered in the implementation of the data structures. If each process were to have a data structure with a slot for each node in the node network, then there will be a large amount of space allocated on each process for information about nodes that are on other processes. This space will therefore be wasted. However, if one reduces the size of the data structures so that they only allocate space for the nodes residing on the particular process, then one must also provide a mapping from node names to the array positions in these smaller data structures.

In order to reduce space requirements in the already data intensive VMT, this second course was taken. Each process creates a new data structure that maps each node name into the correct array position, and this is referenced whenever data about that node is needed. In addition, functions that previously made changes to all the nodes by starting at the lowest node and incrementing up to the last node had to be changed so as to step through a list of the nodes resident on the particular process.

3.1.2 Global Data.

Because of the loosely-coupled nature of the simulated nodes in the node network, there are not a large number of global values that must be maintained. As a matter of fact, the only data that changes during the run that must be shared among processes running the parallel simulation concerns the state of the nodes of each process. This data is required in order to synchronize the processes, and will be addressed in the next chapter.

Since all other information common to the processes remains constant, there are two possibilities in the strategy for generating this information. Either each process can generate the information for itself, or else one process can generate the information and then pass the results on to all of the other processes. In actuality, both of these schemes are used, depending on the computational requirements necessary for generating the information.

Many of the important system-wide parameters are contained in the environment file. Since each process reads a copy of the environment file, the majority of the system wide variables are automatically created within each process. However, certain attributes of the environment require a large amount of computation before they can be stored in the appropriate structure.

For example, at the beginning of a VMT run, a structure referred to as the consistency blackboard is created. This contains information as to the form that the solution hypothesis should take, as well as the intermediate hypotheses that are consistent with the solution. A detailed explanation of the uses of this data structure are beyond the scope of this thesis and are addressed in [CORK83]. For the purposes of this thesis, we can assume that the consistency blackboard can be used to increase the perceived intelligence of the knowledge sources, as well as to provide the system with knowledge as to when it has achieved the solution.

There is a large amount of processing required in order to build the consistency blackboard, and fill it with the necessary information. Furthermore, the information in the consistency blackboard is the same for all processes in a parallel simulation, and this information is not modified during the run. This is therefore an occassion where it would be advantageous to have only one process generate all of this data, and this process would simply send the data to the other processes in the simulation.

Another detail in the initialization of the VMT environment thus involved providing the functions needed to have a designated process calculate and send the consistency data to all of the other processes, and have the recipient processes put the information into the appropriate data structures. In this manner, redundant processing is drastically reduced at the cost of some extra communication. Since the amount of information actually transmitted is relatively small, the tradeoff is worthwhile.

Therefore, because the global data required by the system remains unchanged throughout the run (except for the synchronization information described next chapter), setting the global data is done either directly by the process based on the environment file, or else provided to the process by some other process which was given the responsibility for generating the data.

3.1.3 Message Structures.

Another very important change to the system involves the method in which communication between nodes is simulated. In a single process version of the testbed, the message passing mechanism is based on the fact that the nodes can access each other's memory. Hence, when one node wishes to send a hypothesis or goal to another node, it merely places the name of that hypothesis or goal into the receiving node's incoming message buffer. The receiving node can use this name as a pointer to the structure in which the attributes of that hypothesis or goal are stored in order to extract any pertinent information.

As Alice points out in the quote preceding this chapter, a name is useful to the people who name things. The name contains no information in itself — its use is as a mutually understood label for some other object. As noted above, in a single process version of the VMT, nodes can pass each other the names of hypotheses and goals, because the recipient node can use the name as a pointer to the actual hypothesis or goal. However, all nodes do not share memory with all other nodes in the parallel simulation where nodes are distributed among processes. If a node receives only the name of some object, it will not always be able to use that name as a pointer to the object. Because such names no longer have any global significance, it is necessary to implement message passing that would send the information about the hypothesis or goal explicitly to the recipient node.

The messages take the form of lists, each list containing fields from which the recipient node can extract the necessary information to construct its own copy of the hypothesis or goal. For example, a hypothesis message would contain information about the sensed times and locations for the hypothesis, the supporting hypotheses, the belief, and a number of other attributes, including the node which is sending the hypothesis. The sending node builds this message, and then calls on a function to send this message to the specified node. This function in turn either inserts this message into the node's message buffer (if the node is on the same process), or uses the tools outlined in the previous chapter to send the message to the appropriate process, where it will be inserted in the correct node's message buffer.

It should also be noted that messages are not inserted into message buffers in strictly the order in which they arrive, but are instead inserted in order of increasing reception time (as provided by the timestamp which will be explained in the next chapter), with secondary ordering in terms of the sending node. Because the simulation is insured to be deterministic, all messages required by a node during a particular node execution must be in the buffer at the time of that node execution. Since the messages are ordered in the manner described, the message buffer will always have the same messages in the same order no matter what the actual order of message reception was. In this manner, differences in rates for processes will not affect the ordering of messages in a buffer, and the deterministic behavior is maintained.

3.2 The Parallel Simulation from the User's Perspective

As mentioned in the introductory chapter, the parallel simulation of the VMT should result in no changes to the simulation results of the VMT. Since a user is usually interested in the performance of the entire node network, it is important that the output from the parallel simulation be such that an overall view of the problem solving can be achieved. That is, the output from a simulation should not depend on the actual distribution of nodes to processes. Furthermore, there are times when a user wishes to interact with the system in order to uncover data that is not provided in a trace or to detect and diagnose errors in the coding, so there must also be facilities that will allow the user to interact with the detached processes. These matters will now be addressed.

3.2.1 Combining the Distributed Results.

Each of the processes running in parallel produces certain output which the user has requested. Typically, this output might include a trace of the events occurring in the run, data to be used to give a graphical display of the events, and statistics about the run. This output will provide the user with a picture of what happened during the run and why. Because the basic use of the VMT is as a tool to study the interactions between the cooperating nodes, the user will typically desire to compare the activities of nodes during a run, and study their interaction. If the nodes are distributed among a number of processes, then because each process will create its own output, it may be difficult for the user to make the kinds of observations desired.

A more minor issue in the creation of a parallel simulation therefore involves methods by which the distributed results of the simulation can be combined such that the entire node network is represented in a single output file. Basically, these methods involve the interleaving of output from each source in order to achieve some chronological ordering of the overall events in the node network. For example, a trace from a run would typically consist of a sequence of node executions in order of increasing simulation time. In the case of a parallel simulation, trace files generated by separate processes could be interleaved by examining the next node execution in each and printing the one with the earlier simulation time next. In the case of equal simulation times, the node number might be used as a secondary ordering criterion (Appendix B).

In similar ways, other types of output from a set of parallel processes can be combined to form node network-wide output, and this output will accurately describe the results of the simulation. It is important to note that the output files thus produced should in no way be different from those produced by running the same VMT environment distributed in a different way (unless of course the output is collecting statistics about the parallel simulation itself) since a parallel simulation of the VMT should have no effect on the simulated results, but only on the rate of the simulation.

3.2.2 Testing and Debugging the Parallel Simulation.

One of the more challenging aspects of implementing a parallel simulation involves the testing and debugging of the implementation. Due to the distributed nature of the implementation, there will often be bugs that occur on remote processes or are caused by incorrect message passing. Detecting, diagnosing, and repairing such deficiencies in a distributed system is a difficult and time consuming job.

Certain tools and capabilities allow one to make the job of detection and diagnosis much simpler. Some of these tools are provided by the system, and others are functions that had to be created especially for the purpose of debugging. In this section, the more important of these tools and their utility will be discussed.

Log Files.

The creation of remote tasks across the network automatically initializes a file which will act as the default output from the task. By developing the task such that it prints information as to its actions, these actions will be recorded in the file, and can be inspected at a later date if so desired. The log file can thus be a useful tool for analyzing the activities of the remote task after the task has completed.

In the implementation involved in this thesis, each buffer process was a remote task and had one of these log files associated with it. The buffer code was written such that the more important actions performed would be recorded in the log file. For example, the reception of a message, what is done with the message, and when and where messages are passed to are recorded. A part of a log file for a buffer process is provided in Appendix C.

The use of log files provides the user with the ability to inspect the actions of the remote activity once this activity has completed. Such capabilities proved sufficient for debugging the buffer process code. However, debugging of the VMT code is typically more complex, and methods for interactive debugging during a run were desired.

Terminal Allocation.

If a detached VMT process exists on a machine with suitable output devices, then if the process allocates one such device and sends its output to this device, the user can monitor the progress of the detached process directly. For a majority of the debugging of the parallel simulation, this method was used. Instructions for the detached VMT process to allocate the terminal and send its output to it were simply included in the set of commands forwarded by the master to that process, as specified in the distribution file.

This method proved very useful both in debugging the system software, since any error messages were also reported on the allocated terminal's screen, and in testing the synchronization methods, since one could watch the processes lying idle or working depending on the states of the nodes on the other processes. However, just watching the detached process often was not enough. At times it was necessary to find out certain information from the system that was not on the screen, and in order to do this, tools had to be created that would allow the user to interact with the detached processes.

User Interaction.

The interface between the VMT and the underlying communication mechanisms described in the last chapter provided much of the tools needed to enable interaction between the user (working on the master process) and the detached VMT processes. For example, there already existed functions that would allow the user to pass an arbitrary message to a detached VMT process, and this process would evaluate the message and respond accordingly. Hence, if a detached process had a terminal allocated to it, the user could send it messages as to instructions to carry out, and watch the results of these instructions on the allocated terminal.

Other tools were created with which one could interact with a detached process that did not have an allocated terminal. In this case, the command to the detached process would include sending the results of the evaluation of the message back to the master. Furthermore, methods were enabled by which a user could interrupt a detached process, and by which the buffer processes could be instructed to change from a store and forward mode to a transfer mode so that the master could directly communicate with the detached VMT processes.

Interactive testing and debugging of the parallel simulation was thus achieved. Although it has been found that general debugging of the simulation is still best carried out on a single process, for debugging the problems specific to the parallel simulation, the methods above have together formed a useful and effective mechanism for detecting and diagnosing errors.

3.3 Summary

In this chapter, the major changes to the VMT environment that were required in order to implement the parallel simulation were presented. These changes affect both the internal working of the system (data structures, message passing) and the interface that the environment presents to the user. With the background provided by both this chapter and the previous chapter on the underlying communication mechanisms, the questions of how to synchronize the processes and how to allocate tasks to each process can now be investigated.

"Yes, that's it," said the Hatter with a sigh: "it's always tea-time, and we've no time to wash the things between whiles."

"Then you keep moving round, I suppose?" said Alice.

"Exactly so," said the Hatter: "as the things get used up."

"But what happens when you come to the beginning again?" Alice ventured to ask.

"Suppose we change the subject," the March Hare interrupted, yawning.

— Lewis Carroll

CHAPTER IV

PROCESS SYNCHRONIZATION AND NODE SCHEDULING

The distribution of the VMT over a number of processes running in parallel poses problems in terms of insuring that the simulation results are independent of the number of processes used to run the simulation. In the single process case, the simulations are fully deterministic, and hence, reproducible. However, because processes on different machines may run at different rates, the parallel simulation must have some sort of synchronization mechanism built in to it. Since the nodes interact, there must be guarantees that one node not get too far ahead of a node from which it can receive a message, lest it go beyond a simulated time at which it should receive a message. In other words, the implementation must insure that events in the simulation occur in their proper order regardless of the rate differences of the processes involved.

It should be noted that such determinism will inevitably result in some degree of resource waste. Since it is unlikely that at any given time all processes will proceed at exactly the same rate, there is a high probability that processes will occasionally have to wait for other processes to catch up. It is probable that there will be times when processes lie idle while waiting for some set of slower processes, and so, the processing resources of these idle processes are not being used to the fullest advantage.

It is also important to realize that there are alternative methods to the synchronization proposed herein. For example, rather than avoiding synchronization errors by forcing processes to wait for each other, one could instead use a detection and recovery mechanism [JEFF82]. In such a strategy, detection of a synchronization error requires that the process recognize when an event ordering error occurs. Recovery is a more difficult problem, and typically requires that some record of the changes to the system be kept so that the system can backtrack to some appropriate earlier state (before the error occurred) by undoing some of the changes and their effects. The advantage of this strategy is that processes never lie idle, and if the rate of errors is low, then the rate of the simulation can be improved. However, this mechanism requires much more memory, and if the rate of errors is high, then there will be a large amount of backtracking, lowering the rate at which the simulation can be performed. In the VMT, such a strategy is infeasible because the massive amounts of data and the far-reaching (and often subtle) effects of any changes to the data would make records of changes unwieldy and undoing the effects virtually impossible.

As pointed out earlier, the purpose of the parallel simulation is to increase the speed of the simulation in real time while keeping the simulation results unchanged. To achieve the goal of providing a deterministic system, it is therefore necessary to include some degree of synchronization between processes. This chapter focuses on how synchronization between processes can be achieved, and the interdependence of synchronization methods and node scheduling within a process.

4.1 Node Network Cycle Based Synchronization

Recall the scheduling methods of the system expounded in the brief overview of the VMT. The processor's attention steps from one node to the next as time appears to stand still, in a manner reminiscent of how the participants at the Mad Hatter's Tea Party move from chair to chair around the table while it is always tea-time (see quote preceding the chapter). Unlike the March Hare, we have an answer as to what happens when we get back to the beginning — we allow the time (as represented by the number of these cycles) to be incremented and then go around again. Based on this scheduling strategy, a straightforward synchronization method would involve allowing each process to execute a single node network cycle and once the process has finished its cycle, it sends a control message to some specified synchronizing process indicating this fact. The process then waits for a message from that specified process which will allow it to continue on to the next network cycle. This is an example of a tight time-driven centralized synchronization method [PEAC79].

This synchronization method was implemented as a first attempt at creating a parallel simulation. The process that was responsible for the system wide synchronization was the master process since the master process has complete knowledge about the topology of the system. It was the responsibility of the master process to keep track of which processes had finished the current cycle, and when all processes had done so (including itself), it would alert them all to continue on to the next network cycle.

In order to achieve even more parallelism, the user was allowed to specify the possible interactions between processes (i.e. for a given process, what other processes had nodes that could send messages to any node on that process). The master could instruct a process to continue if all of the processes from which it could receive messages had finished, even if there were other processes that had not completed the cycle yet. However, since there was so much interconnectivity between nodes in the simulated environments tested, this latter optimization did not result in any significant gains.

The centralized manner in which this synchronization was achieved certainly had some drawbacks. Because all processes were dependent on a single process, the implementation was not altogether pleasing in terms of robustness in the face of hardware and software failures. In addition, the extra computational demands on the master process resulted in this process often acting as a bottleneck and slowing down the simulation.

The synchronization is also not particularly flexible. In part, this is due to the manner in which communication is simulated, as outlined in a previous chapters. Since communication between nodes is simulated so rigidly, this very strict lockstep type of synchronization is the most logical recourse. Since messages always have a delay of one network cycle, nodes cannot get any more out of step with each other than one network cycle, so the processes' network cycles must increment all together.

Among the positive aspects of this synchronization mechanism, the most attractive is its simplicity. By implementing this mechanism first, the parallel simulation was achieved in such a way that debugging the code was much simpler, and following the events controlling the parallel simulation was practically trivial. Another positive aspect is that the control messages needed to achieve this sychronization were short and predictable. While some synchronization methods might require large amounts of information to be transmitted between processes to enable them to maintain a reasonable global view of the system, the rigidness of this mechanism allowed control to be achieved in a relatively small number of short messages.

The network cycle based synchronization allowed significant decreases in the amount of real time required to achieve a VMT simulation. Typically, in a perfect parallel implementation, one would expect the real time required to run a parallel simulation to equal the real time required to run a serial simulation divided by the number of processes running the simulation. This assumes that all of the processes are busy all of the time. The mechanism described above allowed the real time required to be equal to the serial time divided by about two-thirds of the number of processes.

For example, a run that would require one hour of real time on a serial machine would complete in about forty-five minutes if on two machines, and about thirty minutes if on three machines. Because this mechanism was a stepping stone toward the mechanism to be described later in this chapter, there was not much time devoted to optimizing it and determining through experimentation details of the improvement that this method yields. However, it is safe to say that the node network cycle based synchronization is a mechanism by which significant but suboptimal real-time rate improvements can be realized.

The limitations of the node network cycle based synchronization stem primarily from the way that the node executions are scheduled and internode communication is simulated. By using the round-robin node execution technique, the scheduler might waste time executing nodes that have no processing to do. In addition, by incrementing time after each node network cycle, there is no way to simulate various KS execution times. The assumption that all messages have a delay of one node network cycle is implemented by having messages inserted into buffers that are emptied by the receiving nodes during the next node network cycle. Simulation of more interesting communication issues, such as varying communication delays between nodes, is not possible. In order to achieve a better parallel simulation, it was required that major changes be made to the scheduling of nodes within a process as well as the simulation of communication.

4.2 Changes to the Scheduler

The principal consideration in changing the system involves incorporating the concept of time along with events; there must be the capability of associating with a simulated event the simulated time at which that event occurred. In this manner, it is possible to allow knowledge sources to take a certain amount of simulated time to execute. Instead of each local knowledge source execution being considered as a single event during a node execution, and each node progressing forward in steps of these events, each node will instead have a clock, and synchronization and scheduling of nodes would depend on the interrelationships of these clocks rather than the events occurring on the nodes.

The new implementation will thus allow some nodes to execute a number of knowledge sources in the time that other nodes can execute only one. In addition, transmission and reception events also have times associated with them, and these times can be used to simulate various delays in the communication channel. The modification of the scheduler in this manner changes the system from a node network cycle based implementation to a event-time based system.

4.2.1 The Conceptual Model.

The conceptual model behind the event-time based simulation is one in which each node is actually composed of three processors (figure 4). The majority of the computation occurs on the local processor. This processor is responsible for running all of the local problem solving knowledge sources, and has direct control over the other two processors. These processors are the transmitting and the receiving processors. The transmitting processor has a queue containing the hypotheses and goals which should be sent to other nodes, as specified by the interest areas. The receiving processor has a queue containing received messages that must be processed and incorporated on the appropriate blackboard in shared memory.

Each processor is responsible for generating KSIs, enqueuing them, and executing them. The local processor instantiates local KSIs, the transmission processor send KSIs, and the reception processor receive KSIs. Each processor has access to the shared memory, and so, each can read from and write to the blackboards. However, the activities on the transmission and reception processors are controlled by the local processor. The model therefore simulates a powerful local processor that has both a transmitting and a receiving processor at its disposal. In

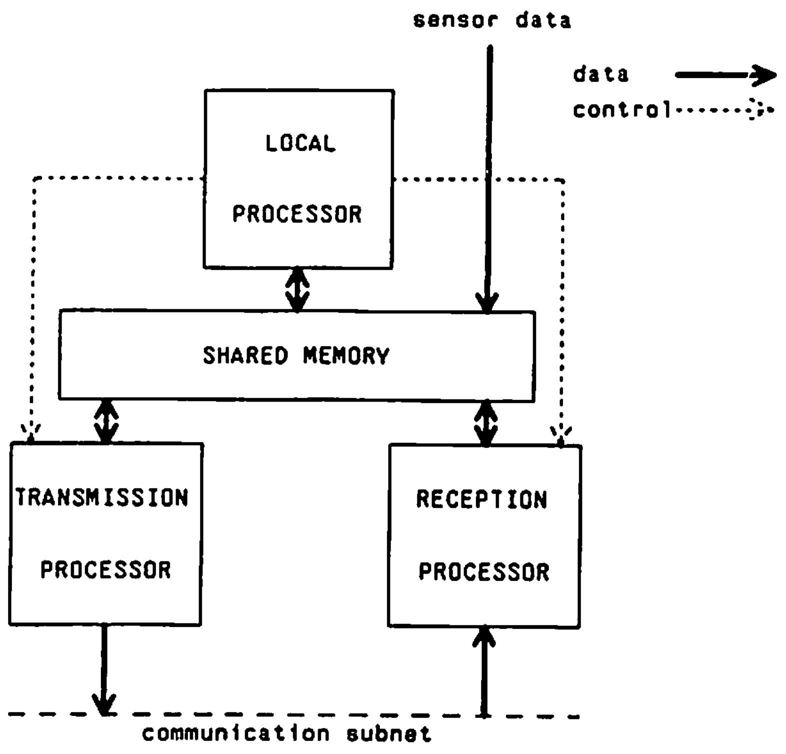


Figure 4: Conceptual Node Architecture.

this way, local processing, and the transmission and reception of messages can all be done concurrently.

Knowledge source executions are considered to be non-interruptable. Because the local processor exerts some control over the other processors, and this control can only be exerted between local knowledge source executions, the transmission and reception processors can be controlled such that events on one of these processors has no effect on the other processor during the execution of a local knowledge source. The reception of a message cannot cause a transmission to occur, or vice versa, without the local processor intervening. Therefore, the activities of each processor during a local knowledge source execution can be fully predicted at the start of the execution. This fact will be used in the simulation's scheduler because, as a result, a node execution can be considered to be composed of a single local knowledge source execution and some number of communication knowledge source executions.

4.2.2 Node, Transmission, and Reception Time.

The simulation must not only keep track of the simulated time of a given node (incremented by the times required to run each local knowledge source), but also the simulated times of the transmission and reception processors; each processor must have a separate clock associated with it. The communication processors may be at somewhat different clock times from the time of the local processor, although all of the times are related, since a node cannot get too far ahead of its reception processor lest it proceed beyond the time that it should incorporate some received data into its blackboards.

The invocation of a knowledge source must always be in the context of which processor is being simulated as running that knowledge source, and the clock time of that processor is modified appropriately. Furthermore, if a processor has no knowledge source to run at a given time, its clock may be advanced to a point in the future representing when the next possible event for that processor may occur.

The use of simulated times allows much more flexibility in the simulation of communication between nodes. Within the context of the run, a delay between each pair of nodes can be specified. At the conclusion of a sending knowledge source, the delay between the sending and receiving nodes is found and added to the completion time of the sending knowledge source. The result is the time at which the message is simulated to arrive at the destination node, and this time value is included in the message. In this manner, all messages are said to be timestamped [CHAN78].

Similarly, a receiving node cannot begin processing a message unless the reception time of that node is at least as advanced as the timestamp of the message. In order to achieve this, messages for a node are placed into that node's message buffer in order of increasing timestamps (reception times), as mentioned in the discussion of message structures. For a given node execution, only those messages with timestamps not greater than the simulated time of the local processor are extracted from the buffer and processed. Therefore, unlike the earlier implementation in which all of the messages in the buffer are processed during a node execution, the modified implementation allows much more flexibility as to when a particular message is perceived to arrive at a node.

4.2.3 Node Executions.

In order to simulate the concurrent processing of a number of nodes in a single process, it is still necessary to interleave parts of the nodes' activities. Each of these small pieces of activity is refered to as a *node execution* and typically consists of some number of communication knowledge source invocations and a single local knowledge source invocation.

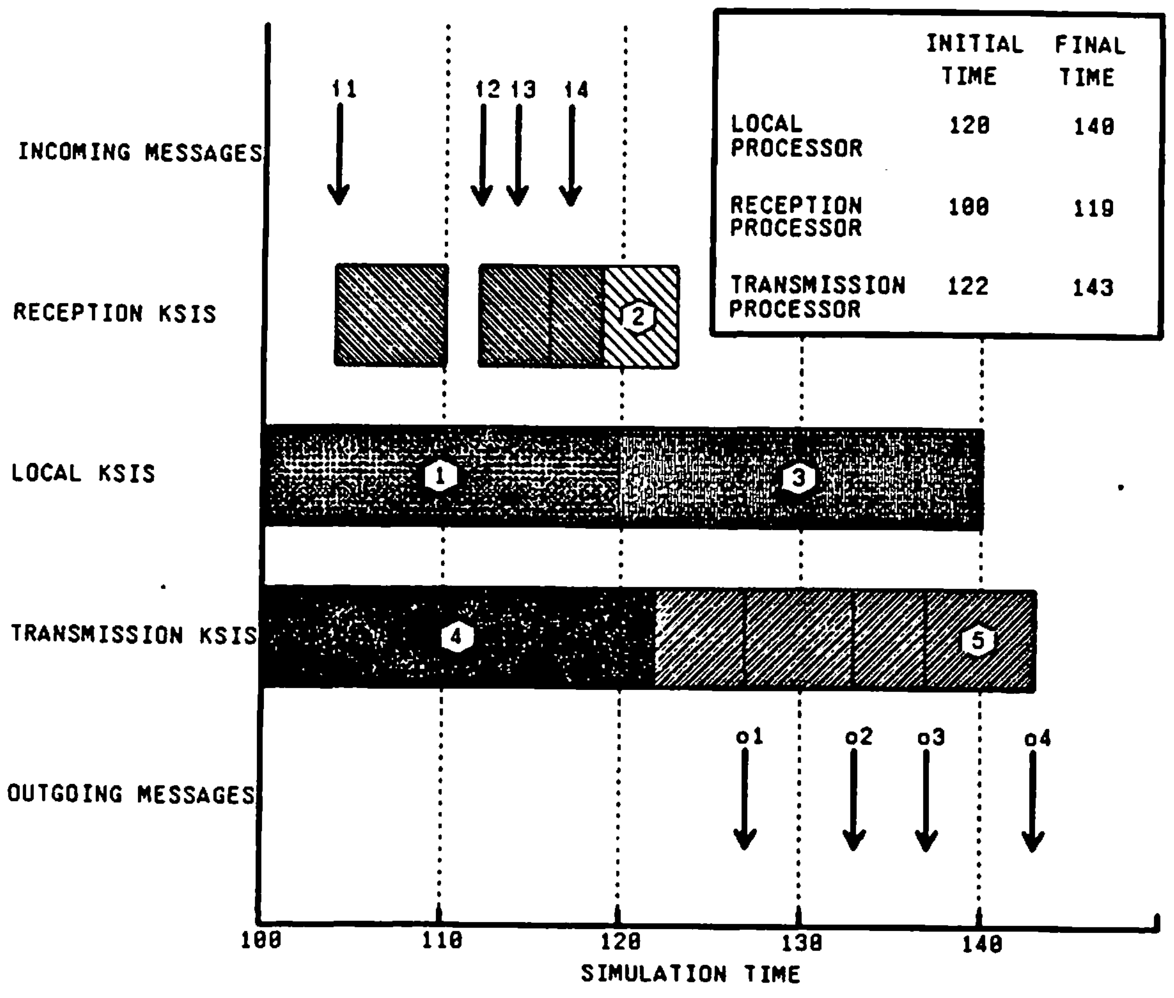
In the event-time based simulation, a major concern is to insure that as long as any node is capable of executing (is not waiting for some event) the process will not be idle. Thus, it is very important that a node execution not be started unless it can also be completed. For this reason, in order to execute a local knowledge source at a given time, it is necessary that all messages that could be incorporated into the blackboard up until that time have indeed been received. If all nodes from which the node in question can receive messages are up to the given time, then, since instantaneous message transmission is not allowed, the node can go on to its next local knowledge source execution.

In order for a node to be executable, the above criteria must be met. A node execution consists of first simulating all of the receptions that are processed up until the local time, since these can affect the blackboards and, hence, the KSI queue from which the local processor will choose its next KSI. Next, the local KSI is extracted from the queue, and the simulated runtime of this KSI is calculated. Based on this time and the current transmission time, transmissions will be simulated until the transmission time exceeds the simulated finish time of the local knowledge source execution. Finally, the local knowledge source is executed, and the local time of the node is advanced to the end time of this knowledge source execution. Note that it is important that transmission knowledge sources be executed before the local knowledge source, since the changes on the blackboard due to the local knowledge source should not affect the transmission knowledge sources. Transmission knowledge sources do not alter the blackboards, so their executions will have no effect on the local knowledge source.

As an example of these concepts, consider a node execution as portrayed in figure 5. The last local knowledge source executed from simulated time 100 to time 120 (1), so that the local node time of this node execution begins at simulated time 120. First, all reception events that can be processed up to and including time 120 are simulated in the reception processor. Note that if the processor is idle, then processing of a received message begins just when the message arrives, but when it is currently processing a message, the incoming messages must wait. The reception time begins at 100, and four messages are received (i1 — i4), three of which can fully processed before time 120. The simulation determines that the data received from the last message will be fully processed only after time 120 (2), and, because data from this KSI should not be on the blackboards before the local KSI execution, this KSI is not executed in this node execution, but remains on the queue so that it can be invoked during the next node execution.

At this point, all reception events that can affect the local and transmission processing during the upcoming local knowledge source execution have been incorporated on the blackboards. The local processor can then select the most highly rated KSI in its queue, and the simulated run time of this is found (3). Beginning at time 122, the time that the last transmission knowledge source of the last node execution completed (4), transmissions that can be invoked before the completion time of the local KSI are then executed. In this example, four such KSIs are invoked, and four outgoing messages (o1 — o4) are produced. Note that the completion time of the last is beyond the completion time of the local KSI (5). Because transmission knowledge sources do not alter the blackboard and the KSI was invoked before the local KSI completed, the non-interruptable nature of knowledge sources allow this to be sent. Indeed, since KSIs are not timestamped, it is possible that, if this KSI were not executed, a more highly rated KSI formed due to the local KSI execution could supplant this KSI at the top of the queue. When the transmission processing was resumed during the next node execution, this KSI would be sent out where the last one left off — at a time earlier than it was actually instantiated.

Finally, the local KSI will be invoked. The hypotheses produced by the knowledge source execution are simulated to appear on the blackboard at the time the knowledge source execution completes. In addition, at this time, all new goals and resulting KSIs are produced and placed in the appropriate data structure. The



	INITIAL TIME	FINAL TIME
LOCAL PROCESSOR	120	140
RECEPTION PROCESSOR	100	119
TRANSMISSION PROCESSOR	122	143

Figure 5: A Graphical Depiction of a Node Execution.

node's local time is altered to reflect the simulated time to run the local knowledge source, and then the node execution is completed.

It is likely that there will not always be sufficient communication KSIs to keep the transmitting and receiving processors completely busy. If the queue for the reception processor is empty before the initial local time is reached, the reception clock is incremented to exactly the initial local time, since we are guaranteed that no messages can be received before that time. Similarly, because the sending queue can only be updated between local node executions, the transmission time can be incremented to the completion time of the local knowledge source if the sending queue becomes empty.

On the other hand, if the local KSI queue becomes empty, the runtime of the local KSI is calculated to be zero. Reception and transmission are carried out as outlined above (receptions up until time local time, transmissions until transmission time exceeds local time). However, since the node has no useful local processing to perform, the node enters an idle state. Similarly, a node that does have local processing to do is referred to as being in an active state. The state that a node is in has an effect on the manner in which scheduling is done, as will be outlined below.

4.2.4 Next Event Calculation.

If a node is in an active state, then the next time that the node will have processing to do is the time of the next local KSI invocation. Local KSIs are simulated to execute one after the other, so that the invocation time of a local KSI is equal to the completion time of the previous KSI. Hence, if a node is active, the next time that a node execution is needed is simply equal to the completion time of the last local KSI execution.

However, if a node is idle, then the calculation of the next event occurring at that node (and, hence, the next time that that node should be executed), is somewhat more complex. In such a case, the simulation time can be *accelerated* to the simulated time of the next event [BRYA79]. Without local processing to be done, the node can only be executed if it has a sensor event, transmission event, or reception event.

If sensor data is provided for some time in the future, then the node must execute at that time. Thus, at any given time, a node can calculate when its next simulated sensor input will occur (if ever), and this is the next sensor event time.

Similarly, if a node has pending KSIs on the transmitting processor's queue, then the node can predict when the next transmission event will occur. Since transmission KSIs require a certain amount of runtime, the time at which the last transmission will be complete is calculable, so that if there are pending transmission KSIs, then the next one is scheduled to be executed when the current transmission is complete.

Finally, if a message is received at a node, this constitutes an event that could stimulate a node execution. Upon receipt of a message, the message has its timestamp inspected to determine the simulated time that it is to arrive. Given this time and the runtime of the knowledge source needed to process this message, the time at which the resulting hypothesis or goal from this message will be incorporated onto the appropriate blackboard can be determined. In this manner, the next time a node execution can be caused by a reception is found.

When a node becomes idle, the next event time for the node is determined as the minimum of the times for any of the types of events described above. If there is no event found of any sort, then the next event time is considered to be at infinite time (never). Of course, during the course of other nodes' executions, it is possible that new reception events for the idle node may occur, and the next event time of that node is changed accordingly. The next event time for a node is used in the scheduling of node executions, as well as in synchronization, as will now be discussed.

4.2.5 Interleaving Node Executions in a Process.

A given process has a list of all of the nodes residing on it. Associated with each node is the next event time of that node, and by comparing these times, the process can determine which node is furthest behind. A general scheduling method is to always execute the node that is furthest behind. In a non-distributed implementation, when all nodes in the network reside on a single process, this method is sufficient to insure a deterministic run, because if no node has time less than the node to be executed, then no message with a timestamp less than the node's time can possibly arrive in the future.

The basic scheduling algorithm for interleaving nodes in a process is simply to execute the node with the earliest next event time. Provisions must also be included to detect if all nodes have no next event (all have infinite next event times). In this case, there is no more processing to be done in the system, and a state of quiescence is reached. The run can then be terminated. This will be addressed in more detail later on in the discussion of synchronization. In addition, if a node creates a hypothesis that meets the criteria for being considered a solution, the run should also be terminated. This is done by assuming that the node broadcasts the fact to the other nodes, with the broadcast messages subject to simulated delay just like any other messages. This will also be considered in more detail in the next section.

4.3 Event-time Based Synchronization

Recall that a node cannot be executed until all messages that could be received prior to the node's local time have indeed been received. Knowledge of the current times of any nodes that can send it a message are thus crucial to a given node. The node cannot proceed until all nodes capable of sending it messages have advanced beyond a certain time. Therefore, the minimum transmission delay between nodes is the maximum amount of time that nodes can get out of synchronization without potential non-determinism.

As an example of this, consider *nodeA*, which can receive messages from *nodeB*, and the minimum message delay is ten time units. *NodeA* currently has executed knowledge sources such that its time is twenty time units, but last it heard, *nodeB* was at nine time units. Because *nodeB* might send a message at its time ten which could reach *nodeA* at time twenty, *nodeA* cannot proceed. *NodeA* must wait either until the message produced by *nodeB* at time ten arrives, or else until it receives news that *nodeB* has executed at time ten or beyond, at which point no message is forthcoming at time twenty.

In a parallel simulation, therefore, it is possible that the node furthest behind on a given process cannot proceed because it is waiting for a potential message from a node that is even further behind on another process. The waiting node is said to be *blocked*. In this case, the simple scheduling strategy of choosing the node furthest behind might not yield an executable node. Hence, the scheduling strategy is modified such that *the node furthest behind that is unblocked is the next node to be executed*. As long as there is a node that can be executed, the process will not lie idle, so that the best use of the processing resources can be attained.

Since there must be some subset of nodes with the furthest behind time at any given point, and since these nodes must be executable (no messages could possibly arrive at an earlier time), there will always be one or more executable nodes in the node network. This is not to say, however, that there will not be a time when a given process has no executable nodes. Rather, there will never be a time when all processes have no executable nodes. Hence, there is no possibility that the system will deadlock.

However, there is a difficulty in the treatment of idle nodes. Although they should be scheduled based on their next event time as outlined above, if a node can receive a message from an idle node, at what time should the node perceive the idle sending node to be? The last event executed by that node may have been far in the past, but the node's time will not be updated until it performs its next execution. Hence, the last executed time is inappropriate. The next event time is also inappropriate, because there is no guarantee that there will be no event sooner than this value; that is, if the node receives a message, its next event time may be changed to an earlier time.

As a solution to this predicament, the concept of *global time* is introduced into the system. The global time is the minimum of all the nodes' next event times, as perceived by a particular process [PEAC80]. Since the process is insured that no node can execute at a time earlier than the global time, then it is guaranteed that that no idle process can have an event earlier than this time. Therefore, in determining whether a node is blocked, any idle node from which it can receive a message is assumed to be at the global time.

The global time is based not just on the state of each node that can communicate with the node in question, but on all of the nodes in the system. It is therefore important for each process to have a relatively timely view of the state of all nodes in the system. There must be a method by which processes trade information concerning the states of their nodes.

4.3.1 Node Update Messages.

Note that the scheduling of actions in the parallel simulation is really quite simple. The process merely schedules the furthest behind unblocked node. The difficulties with this synchronization method lie not in the scheduling of node executions within each process, but instead with the communication of state information about the nodes in the network. Obviously, any messages passed between nodes will act as state information carriers due to the timestamping of the messages. However, the number of messages passed in the node network may often be insufficient to carry all of the necessary information such that each process has a satisfactory understanding of the state of the nodes in all of the other processes. Hence, explicit state information messages must be sent.

When dealing with using the communication of state information between cooperating processes, one must consider the tradeoff between the degree to which each process is kept up to date and the cost of such a strategy in terms of communication overhead. In particular, one cannot fill the communication bandwidth with too many state information messages lest there be insufficient communication capacity to also deal in a timely fashion with actual hypothesis and goal messages. Furthermore, one must consider the overhead involved in processing each of these messages; it is hoped that the synchronization of the processes would require a minimum of added computation.

At the outset of this research, it was anticipated that establishing the best algorithm for achieving these results would be a difficult problem. However, in practice, the best method in the particular domain of a parallel simulation of the VMT, the simplest algorithm seems to be the best.

When considering a parallel simulation of the VMT, one must understand the nature of the problem solving that occurs in each node. In particular, the invocation of a single knowledge source can result in the generation of a large number of hypotheses, which in turn can generate a large number of goals. Briefly put, a single node execution can and usually does encompass a large amount of computation and memory usage and therefore will often take a very large amount of time relative to the amount of time required by the network. For example, it is likely that a node execution for a typical run might require a number of minutes of real time, while the communication of an update message would require an average of a couple of seconds.

When the relative times are considered, it seems hardly likely that update messages occurring after each node execution of each process could come close to clogging up the communication channels. The tradeoff between minimizing congestion while maximizing information flow seems to not be particularly difficult to solve. Even if information passed between processes was maximized, there would be no appreciable congestion as a result. Hence, a strategy involving piggybacking state information to hypothesis and goal messages is inappropriate because of the additional computation required to determine how to send an update. Since congestion is not an issue, it makes more sense to unthinkingly send explicit updates than to incur the overhead of deciding whether piggybacking can be achieved.

Furthermore, the fact that the calculation of global time depends on an adequate view of the states for all of the nodes in the network means that there is not much benefit in sending updates only to processes that can receive messages from the node being updated. The extra computational overhead involved in determining which of the processes could receive a message from the node does not warrant the relatively small amount of bandwidth that this would save, and, as mentioned above, the state of a node may be important in an indirect way in a process's calculation of global time.

In addition, since the amount of resources spent on an update message are so low relative to the resources required to execute a node, it is unlikely that aggregating update messages over a number of node executions and sending this aggregation out will save much resources. A process that withholds an update for a node in order to combine in with subsequent updates could force a process waiting for that update to become idle. The resulting loss of computation time would not justify the modest savings in bandwidth.

Finally, updating methods involving the interaction between processes are not viable in the current implementation environment. In order to for an interactive strategy to achieve the desired results, the response time for a particular request for update information must be kept at a minimum. Since interrupt mechanisms are not currently supported, this means that processes would have to frequently poll their mailboxes for incoming update requests. The overhead of such a large amount of polling would degrade the performance of the simulator.

Therefore, the most feasible method for processes to inform other processes as to a change in the state of one of their nodes is simply to broadcast a message with the appropriate information to all of the other processes. It should be noted that a similar conclusion might not be reached when trying to determine a strategy for another parallel simulation, and a number of strategies are presented in [PEAC80]. The success of this simple strategy rests on the fact that in the application of the VMT, the looseness of the connectivity of the system results in processes being able to do a large amount of computation without interacting. Also, if a process does not receive update information in a timely manner, much potential computation time could be wasted. For these reasons, the best strategy is to simply broadcast updates whenever a local node is executed.

4.3.2 Blocked Processes and Termination.

If a process has no node which is not blocked, then the process itself is said to be blocked. That is, until another process sends it some information, the process will lie idle. However, if some exceptional situation such as a hardware or software failure has occurred, it is possible that the awaited information will never arrive. The system must be capable of recognizing this kind of a situation and dealing with it in an appropriate manner.

During a run, the synchronization is carried out by the broadcasting of update messages as outlined above. The control is fully distributed — there is no single process responsible for the synchronization of the system. However, in the case where exceptions occur, it is important that one process be responsible for detecting if the system is in a state in which the run should be terminated. This responsibility falls to the master process since this is the process from which the run began.

For all processes other than the master, if the process becomes blocked, it responds in a very simple manner. The process simply waits for a message to enter its mailbox and processes the message. If the message allows a node to proceed, then the process is no longer blocked. If the process is still blocked, it repeats this procedure until is is not blocked. Hence, processes other than the master are purely dependent upon stimulus from an external source.

On the other hand, the master process must be capable of initiating events in the case of some exception. To this end, when the master process enters a blocked state, it sets a timer. The master then waits for a message to enter its mailbox much like any other process. If a message arrives before the timer expires, then the master processes the message and proceeds either to an unblocked state or repeats its blocked state procedure. However, if the timer expires before a message arrives, then the master forces an error, warning the user that an exception has occurred.

Any hardware or software error that causes a process to crash will thus eventually cause the master to issue an error message (since there will be no update messages from that process, eventually all other processes will have to wait for it). If the master itself crashes, this will be immediately apparent to the user. Hence, if the run terminates abnormally, the user will be warned of that fact and the master will allow control to return to the user.

It should be noted that, besides hardware and software crashes causing the system to become blocked, it is possible for all processes to become blocked simply because there are no more events to occur in the system. This situation is known as *quiescence*.

If the master becomes blocked, and as far as it knows, all nodes have their next event at infinity, then it begins a test for quiescence. The master sends to all processes a request for them to determine whether they perceive the system as quiescent also. If any respond negatively, then the master treats the situation simply as blocked, sets its timer, and goes into its blocked mode. However, if all processes agree with it, then the master waits a small amount of time, and sends a second round of quiescence test requests. In this way, if some message were in transit during the first quiescence check, it will have arrived before the second and the second quiescence test will fail (in the case where another message is in transit, there must have been an intervening state update message which will cause the master to start the quiescence procedure from the beginning). If all processes still believe that the system is quiescent, then the master sends all processes a quiescence message, which allows the processes to end the run, reporting that the solution could not be found.

4.3.3 Termination due to Solution Generation.

As mentioned before, the consistency blackboard allows the system to determine if a given hypothesis is consistent with the solution, or is the solution itself. Whenever a node generates a solution hypothesis, the process on which that node resides broadcasts the an explicit solution-found message to all other processes stating the name of the solution hypothesis and the node that found it.

When all solution hypotheses have been generated, the system can terminate the run. In order to insure that runs terminate in a deterministic manner, the implementation simulates each node as being required to receive the message. Hence, a particular node will enter a finished state at the time which it is simulated to have received the message informing it that the solution had been found. This reception is based on the simulated delay between nodes, and it is assumed that the message took the shortest path to the node.

Therefore, it is possible for activity to continue in the system even after a solution is found, because the solution must be propagated among the nodes. Once a node has been simulated to have received the solution message, the node is no longer eligible to be executed. When all nodes in a process are thus unable to execute, the process terminates the run, alerting the user as to what solutions had been found.

It should be noted that whether the run terminates normally or abnormally, the set of processes involved in the run remain. This allows the user to specify post-run processing and to interrogate processes as to certain characteristics of the run. Separate commands exist for terminating the processes involved in the run. These may be invoked at any time before the master process is ended, but must eventually be executed or else the detached processes on other machines will not be deleted.

4.4 Summary

The implementation of a parallel simulation where each process must simulate concurrent processing of the nodes on that process while remaining synchronized with the other processes running in parallel to it is a complex problem. In this chapter, the implementation to two solutions of this problem were presented.

The first involved centralized synchronization where a single processor has the responsibility for maintaining the correct ordering of events in the simulation. The interrelationship between this synchronization mechanism and the fact that simulated time was measured in terms of events (network cycles) was investigated, and indicated the need for an event-time based strategy.

The second implementation of the synchronization was based on this more flexible representation of simulated time. The modifications resulted in the alteration of how nodes on a process were scheduled as well as how synchronization was performed. In particular, the synchronization mechanism was distributed among the nodes, so that both a more equal distribution of the synchronization responsibilities as well as a potentially more robust system resulted. In addition, more interesting representations of the knowledge sources and the communication capabilities were achieved.

With the successful implementation of the synchronization mechanism, one of the two major goals of this research has been achieved — a parallel simulation in which the simulated events will occur in the correct order regardless of the rate differences between the processors. The second major goal, that of establishing some load allocation criteria, will next be addressed.

"You don't know how to manage Looking-glass cakes," the Unicorn remarked. "Hand it round first, and cut it afterwards."

This sounded nonsense, but Alice very obediently got up, and carried the dish round, and the cake divided itself into three pieces as she did so. "Now cut it up," said the Lion, as she returned to her place with the empty dish.

— Lewis Carroll

CHAPTER V

LOAD BALANCING AND EXPERIMENTAL RESULTS

The previous chapters have supplied the details of how the parallel simulation of the VMT is implemented so that the results of the simulation are independent of the specific node to process allocation. It is the purpose of this chapter to investigate what kind of improvements in the real-time rate of a simulation can be achieved with the implementation outlined, and how these improvements are dependent upon the assignment of nodes to processes.

The scheduling mechanism outlined last chapter insured that if there was a node residing on a process that could be executed, then the process would not be idle. In order to maximize the processor utilization, it is important to distribute nodes to processes such that the amount of time that processes spend blocked is at a minimum.

It could be argued that if the number of nodes that were assigned to each process was increased, then the probability that all of the nodes would be blocked would be reduced. However, one of the major motivations for implementing the parallel simulation was the observation that, as the number of nodes simulated on a process increases, the time required to run the simulation increases at a disproportionately faster rate due to the demands placed on the memory by the data-intensive VMT. Gains in utilization of processing resources is at the cost of slower simulations. This is similar to trying to maximize the throughput of some communication channel by flooding it with messages — throughput increases but only at the cost of increased response times.

Distributing the simulation among machines can therefore decrease real-time needs both by allowing computations to be performed in parallel and by increasing the total memory available to the simulation. In order to best balance the requirements of the simulation, one must assign the nodes to VMT processes so as to divide the memory and computational needs in the best possible way. This is known as load balancing, and will be investigated in the next section.

5.1 Load Balancing

Much of this thesis has been devoted to outlining methods by which computational parallelism can be increased, and little attention has been paid to the ways in which memory needs can be distributed. This is because the memory needs of nodes are roughly equivalent since they each have similar data structures. Therefore, the strategy for balancing memory requirements is simply to assign approximately the same number of nodes to each process.

Computational requirements, on the other hand, are more difficult to balance because of the variations in a node's computational needs over the course of a run. In order to maximize the use of the computational resources, it is important to minimize the amount of time processes lie idle due to having all of their nodes blocked (figure 6). Therefore, nodes should be distributed among processes so that the overall computational needs of all processes *at any given time* are nearly equal.

Furthermore, the simulated run time of executing a particular KS need not reflect the actual real-time requirements needed by that KS. For example, the real time necessary to extend a track is very dependent on the length of that track, but the simulated run time of such a knowledge source need not take this into account. Concurrency of simulated events thus does not imply real-time concurrency, although these concepts are intertwined because of the dependence of real-time processing to the synchronization of simulated time.

Because the computational needs of a node can vary over time, a static assignment of nodes to processes might often result in sections of the run in which the load is poorly balanced. However, dynamically moving nodes among processes might introduce high overhead. Each of these strategies will be considered in the next sections.

5.1.1 Dynamic Load Balancing.

As the processing needs of nodes change over time, there may be times when the initial allocation of nodes to processes results in some processes being inundated with work while others lie idle. If the simulation could recognize such a situation, then by having a busy process transfer some node or nodes to an idle process, the processing load of the simulation would be better balanced, so that better utilization of the processing resources would result.

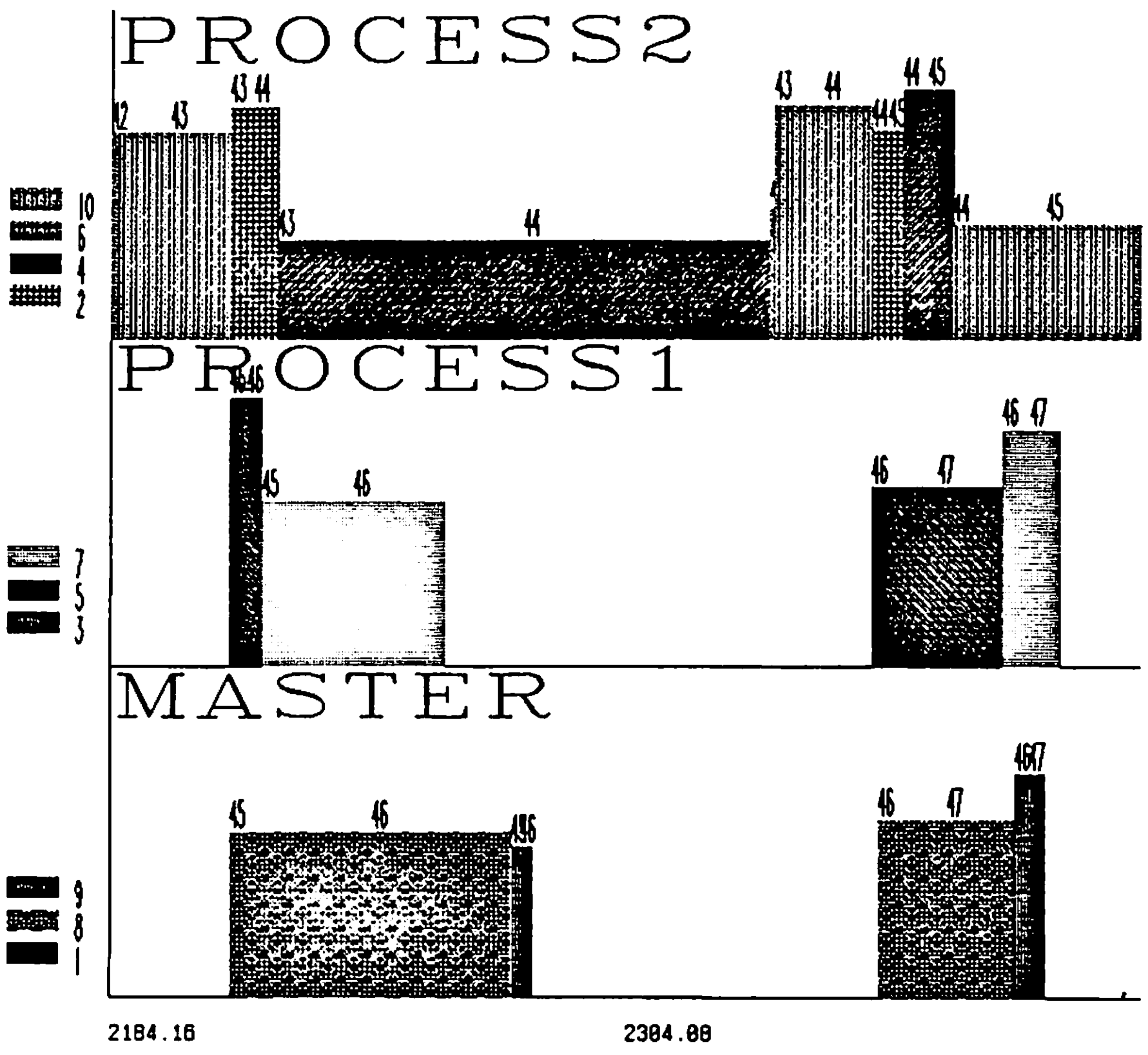


Figure 6: Computational Time Graph.

A graphical depiction of a small portion of a simulated run. The X-axis represents time (in seconds). Each block represents a node execution (to the left of each process is the correspondence between nodes and bitmap patterns). The height of a block corresponds to the CPU utilization during the execution (height of the block divided by the height of the y-axis for the process gives CPU utilization). Above each block are the simulated start and end times of the node execution. Because the maximum amount of simulated time that nodes are allowed to be out of synchronization in this particular simulation was 2, the nodes on the MASTER process and PROCESS_1 must wait at times 45 and 46 for PROCESS_2 to bring its nodes up to times 43 and 44 respectively. The MASTER and PROCESS_1 are thus idle for much of this 2 minute interval.

Such gains are not without cost. For example, the processes in the simulation must have some knowledge as to the activities in the other processes in order to decide that the load is unbalanced. Hence, there is the overhead required in monitoring for a poorly balanced situation. Furthermore, once such a situation is detected, some decision must be made as to how the nodes should be redistributed so that a better balancing is achieved. Because moving a node from a process on one machine to a process on another would involve the transfer of very large amounts of data between processes (during which memory acts as the bottleneck due to the accessing and storing of the data), the processes should try to find a way to optimize the load balancing while minimizing the number of nodes transferred. Such a calculation in itself might be very time consuming and prone to error since the future real-time requirements of a node cannot be accurately predicted, Finally, while nodes are being transferred, resources that could be used in processing for the simulation are instead being expended in order to achieve better load balancing. Therefore, dynamic load balancing is very expensive, but if increased concurrency results from a better balance, the gains from this more effective use of the processors might offset the costs in achieving it.

In determining whether dynamic load balancing is appropriate to the parallel simulation of the VMT, a number of factors must be considered. The most important concern deals with the rate at which a node's processing needs change. If a node changes processing needs very rapidly, then the load balancing may also fluxuate rapidly. In such a circumstance, the improvements achieved by dynamic load balancing will be short-lived, and the load balancing procedures might be invoked frequently. The overhead of the dynamic load balancing would result in a heavy cost for only a small amount of gain. Furthermore, an eventual goal of the VMT is to provide a mechanism by which the organizational responsibilities of nodes can change as the problem solving in the network progresses, so that a node that has exhausted its local processing needs may be given additional responsibilities by the simulated network. Although other factors such as responsiveness and redundancy will affect the assignment of responsibilities, the need for dynamic load balancing will be obviated to the extent that the processing requirements are balanced dynamically among the nodes within the simulation.

Currently, there is insufficient evidence indicating that the implementation of a dynamic load balancing mechanism in the parallel simulation would result in any gains. Furthermore, when considering the tremendous amount of data associated with a node that would have to be transferred if a node were moved from one process to another, intuition dictates that such a mechanism would be useful only if rarely invoked. The current implementation environment is therefore not suitable for dynamic load balancing. As alternative clustering hardware in which all disk I/O occurs on a high speed bus and file server is obtained, movement of information between machines may be sufficiently fast to justify dynamic load balancing.

5.1.2 Static Load Balancing.

Currently, all of the runs of the parallel simulation are based on the static assignment of nodes to processes established at the outset of the run. Unlike a dynamic load balancing situation where a poor initial assignment would be modified as the run progresses, the static load balancing situation means that the run continues with the initial assignment no matter how poorly it balances the load. It is therefore important that the assignment of nodes to processes achieve as good a load balancing over time as possible.

In order to make such an assignment, the processing requirements for each node must be estimated, and the activities of the each node as the problem solving progresses must be predicted. Making such predictions and estimations is a difficult problem. An optimal assignment could only be guaranteed if all of the future processing needs of the nodes were known — but such information can only be obtained by running the simulation. In the quote preceding this chapter, Alice finds that she cannot divide up the cake until it is already divided. Similarly, we cannot make a completely accurate decision as to how to divide up the nodes until the system has already been run — which would require that the nodes be divided already!

It is possible that this strategy might be used in determining eventual node to process assignments. Typically, a set of experiments on which the VMT is run are comprised of one environment upon which very small modifications might be made. For the first such experiment, nodes might be allocated arbitrarily, and a sub-optimal run might be executed. Once this run is complete, it may be used to decide the processing needs of each node, and a better, perhaps optimal, node to process allocation would be developed and used for all of the remaining experiments in that set.

Rather than make an arbitrary initial assignment, however, it is possible that simply by inspecting the particular environment being simulated, one could determine a good approximation to the optimal node to process assignment. The assignment would be based on a prediction of the problem solving activities of the nodes (which provide an rough indication of the real-time computational needs of the nodes). By distributing nodes to processes such that the estimated processing done by the nodes is relatively balanced over time, a better allocation can be obtained.

It is, therefore, very important to understand the characteristics of a particular environment being simulated in order to determine a good node to process allocation. For this reason, a brief study of the environments upon which the experimental results are based will now be made.

5.2 The Simulated Environments

The parallel simulation presented in this thesis has been used on a large number of environments, ranging from two to twenty-five nodes. In all cases, the real time required by the simulation was smaller than that required by the same environment run on a single process. The amount of improvement has been based predominantly on the size of the environment. For example, small environments (less than five nodes) are easily run on a single process, and do not provide enough nodes per process to allow for good CPU utilization when run in parallel. On the other hand, very large environments (over twenty nodes) require more memory than can currently be provided by a single machine, and can therefore *only* be run by the parallel simulation.

In order to gain some quantitative results on the improvements possible by the parallel simulation, environments were chosen such that they would be large enough to allow for interesting node to process allocations, while small enough so that they could be run on a single process (and push the memory to its limits). These environments were based on a ten sensor configuration as portrayed in figure 7. Over the eighteen discrete sensed intervals, data for both the "true" track extending from (6,2) to (40,36), and the "ghost" track from (4,4) to (38,38) are received. Note that the ghost track is moderately sensed throughout, while the true track has interleaved sections of strongly and weakly sensed data. The presence of the ghost data serves to *distract* the problem solving activities as sections of the ghost track are created and merged with sections of the true track.

5.2.1 Environment E1.

The first environment (E1) consists of ten nodes, the sensors being assigned to the nodes on a one-to-one basis. Node *i* will therefore refer to the node associated with sensor *i*. In this environment, nodes with overlapping sensors can communicate, and each node attempts to form a track hypothesis that encompasses the entire true track.

The solution generation in this *heterarchical* environment can be broken into three phases: First, the nodes with strongly sensed true data (2,4,7,9) create track hypotheses which they pass to their neighbors. Second, the nodes with weak true data (5,8,10) join the highly rated received tracks through the weak section, then pass these longer tracks to their neighbors. Third, node 8, having generated one of these longer tracks and receiving both of the others, merges these pieces together to generate the solution track.

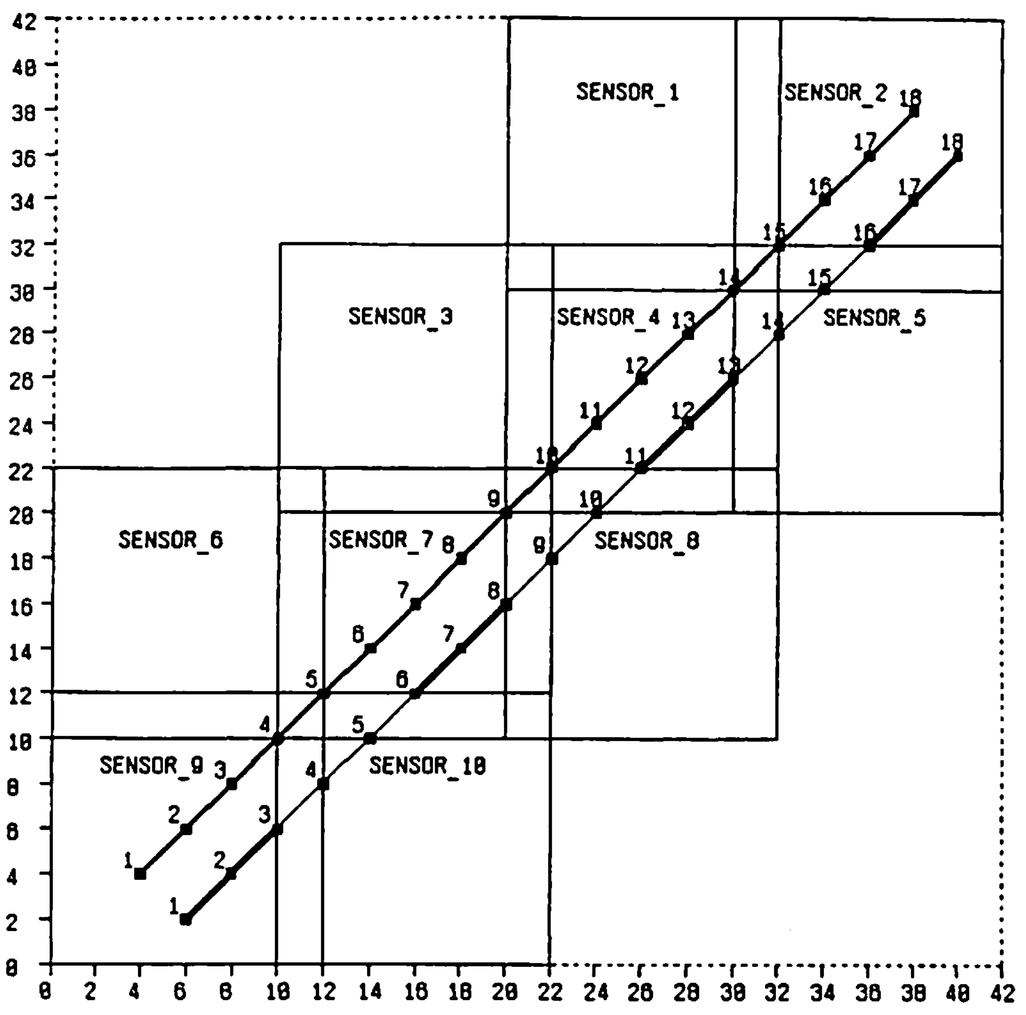


Figure 7: The Ten Sensor Environment.

The eighteen discrete intervals at which sensing occurs are marked, and the line thickness for the track indicates the strength of the sensed data at that point.

From this high level perspective on the nodes' activities, one can surmise that nodes 1, 3, and 6 will have little processing to do once they have processed their small amount of sensor data, while nodes 5, 8, and 10 will be busiest later in the run. It would be anticipated that the remaining nodes would be relatively busy throughout the run. When assigning nodes to processes, one should keep in mind what nodes will require less computational resources as the run progresses and which will require more.

5.2.2 Environment E2.

The second environment (E2) is composed of eleven nodes. Ten of these are assigned to the sensors as in E1, but the last node has no sensor responsibility. The nodes are organized *hierarchically*; the ten sensing nodes send information only to node 11, which in turn is responsible for integrating the received data into a solution track.

Solution generation in E2 can be broken into two phases: First, the ten sensing nodes generate track hypotheses spanning their sensor areas and pass these to node 11. Second, node 11 combines these pieces into a single solution track.

In this environment, it is apparent that node 11 will become significantly busier as the run progresses, while the other nodes would require processing in proportion to the amount of sensor data they have. It is interesting to note that both E2 and E1 require the same amount of simulated time to generate the solution. Although E2 incurs fewer delays due to transmission of intermediate results, node 11 must integrate data from the entire sensor network. By requiring a single node to combine all of the pieces into a solution, much potential concurrency *in simulated time* is lost. Also, in E1, the reception of highly rated true hypotheses by nodes 5, 8, and 10 stimulate activity on the weaker true data sooner than if these hypotheses were not received. Thus, the richer interaction provided by the heterarchical environment can help to focus the attention of the individual nodes on the true data earlier.

5.3 Experimental Results

In this section, results from the experiments performed on environments E1 and E2 will be presented. The statistics shown were gathered during the problem solving part of the run and do not reflect the relatively small overhead of establishing and deleting the distributed environment (activities which typically require 2-3 minutes of real time per process). At the time that the experiments were performed, the machines used were fully dedicated to the simulation to provide as consistent an environment as possible.

5.3.1 Experiments on Environment E1.

The results for a number of experiments on environment E1 are presented in table 1. Experiments 1.1 and 1.2 were each run on a single process and represent a basis for comparison for the distributed simulator. Notice that, as previously mentioned, the doubling of memory drastically improves the real-time requirements and CPU utilization. In addition, the smaller amount of paging behavior results in reducing the amount of CPU time needed. Finally, note that in experiment 1.2 the CPU utilization approaches one, implying that increasing the memory further will not improve the run significantly.

In experiments 1.3 and 1.4, we can see that the effects of distributing the run over three machines with small amounts of memory can drastically reduce the real-time needs compared to the single machine case (1.1). The bulk of the improvement can be attributed to decreasing the memory requirements per machine.

Experiment	Process_1		Process_2		Process_3		Real	Cpu	Cpu	Concurrency	Speed-up	
	Nodes	M	Nodes	M	Nodes	M	Time	Time	Utilization		1.1	1.2
1.1	1-10	2	-	-	-	-	358	119	.33	-	-	-
1.2	1-10	4	-	-	-	-	98	97	.99	-	3.66	-
1.3	1,8,9	2	2,4,6,10	2	3,5,7	2	56	84	.68	.46	6.33	1.73
1.4	1,2,5	2	3,4,7,8	2	6,9,10	2	62	84	.72	.31	5.73	1.56
1.5	1,3,5,8,9	2	2,4,6,7,10	2	-	-	84	91	.59	.83	4.28	1.17
1.6	1-5	2	6-10	2	-	-	84	90	.63	.69	4.26	1.16
1.7	1,3,5,8,9	4	2,4,6,7,10	4	-	-	50	80	.98	.64	7.16	1.96
1.8	1-5	4	6-10	4	-	-	49	79	.98	.67	7.36	2.01
1.9	1,3,5,6,8,10	4	2,4,7,9	4	-	-	46	79	.98	.75	7.72	2.11
1.10	1,2,3,6,9	4	4,5,7,8,10	4	-	-	54	81	.98	.52	6.57	1.80
1.11	1,8,9	2	2,4,6,10	4	3,5,7	2	36	78	.85	.67	9.94	2.72

Abbreviations:

M	Megabytes of memory
Real Time	Elapsed time in minutes
CPU Time	Computation time in minutes
CPU Utilization	(CPU time) / (Real Time – Idle time)
Concurrency	(Σ Elapsed time all processes are busy) / (Real Time)
Speedup	(Real Time for single process experiment) / (Real Time of current experiment)

Table 1: Experiment Set 1.

However, concurrency considerations do play a minor role, as can be seen by comparing 1.3 and 1.4. Both distribute nodes to processes in equal numbers, but 1.3 runs faster. Because the sensed data is incorporated over time, those nodes that receive data earlier will have processing to do earlier. In 1.4, the nodes assigned to a particular process are all from one region of the sensed area, while in 1.3 nodes are more evenly distributed. Hence, in 1.3 each process will have busy nodes throughout the sensing time and more concurrency results.

In experiments 1.5 through 1.8, two distributions are considered both on machines with small amounts of memory and on machines with large amounts. Once again, real-time improvements can be attributed to both reduced memory requirements and increased concurrency. It is interesting to note that both of the distributions resulted in nearly equal rate improvements even though they had very different node assignment strategies. In 1.6 and 1.8, the allocation strategy groups locally adjacent nodes together, a strategy that caused experiment 1.4 to run slower than 1.3. Although this clustering of nodes does indeed reduce concurrency early in the run, the fact that the clustering reduces the number of nodes that can receive messages from nodes on other processes means that there are fewer potential points where blocking can occur (recall that an unblocked node cannot block any nodes on the same process). Therefore, a reduction in blocking results in an increase in concurrency.

Experiments 1.9 and 1.10 illustrate the fact that assigning equal numbers of nodes to the processes might not always be the best strategy. In environment E1, nodes 1, 3, and 6 receive only a small amount of sensor data, and process this quickly. These nodes lie idle for half of the run, and assigning them to a single process (experiment 1.10) can result in reducing the concurrency later in the run. However, if one assigns them to a process and gives that process additional responsibility (for example, nodes 5, 8, and 10 which are more busy later in the run), a better balance can occur (experiment 1.9). Because memory limitations had little effect in these experiments, they are a good indication as to how rate can be improved by increased concurrency.

The node to process allocation problem becomes more difficult when machines with different capabilities are used together. Some estimation must be made as to the relative rates at which the machines can run the VMT simulator. Although this problem will not be deeply considered in the experiments, experiment 1.11 does indicate how replacing one machine with a more powerful machine can affect the rate. Here, we note that the same allocation used in experiment 1.3 can be improved dramatically if the machine that has the largest number of nodes is more powerful (has more memory). Where before this machine was the bottleneck, now more concurrency can result.

5.3.2 Experiments on Environment E2.

Environment E2 presents more interesting capabilities than environment E1 due to its hierarchical organization. Because all nodes could receive information from their neighbors in E1, there was a limit as to how far any node in the simulation could get ahead of the furthest behind node. However, in E2, nodes one through ten only send information, and so, will never be blocked. Hence, these *run-away* nodes will always be executable as long as they have some work to do.

This has the potential of causing problems. Because the nodes are always executable, it is possible that they might simulate activities beyond the simulated time at which the solution is generated. Although such activities will have no effect on the manner in which the solution is generated, they do require that the techniques for merging output from the VMT processes be more intelligent so as to edit out this unnecessary information.

Furthermore, although run-away nodes with advanced simulated times will not progress as long as there are unblocked nodes at earlier times, the question of whether the generation of large amounts of extraneous results could hamper the effectiveness of the simulator is raised. In order to test for this, provisions were made so that a user could supply an optional value that would specify the maximum difference between a node's simulated time and the minimum time of the nodes in the node network. In this manner, the potential run-away nodes could be constrained.

Experiments 2.1 and 2.2 (table 2) provide a basis for comparison in the E2 simulation. Once again, notice that increased memory results in decreased real-time needs. Comparing these values to experiments 1.1 and 1.2, we note that E2 requires less processing even though the solution is found at the same simulated time. These differences can be attributed to the fact that KSs that work on tracks require more real time as the tracks get longer. In E1, each node may be working on long sections of the track, while in E2 only node 11 will be doing so. Since E2 has fewer nodes running these long KSs, the run time will be shorter.

Experiments 2.3 and 2.4 summarize the effects of distributing E2 over three machines with small amounts of memory. Rate improvements can be attributed both to reduction in thrashing as well as to concurrency. Note that the concurrency

Experiment	Process_1		Process_2		Process_3		Real	Cpu	Cpu	Concurrency	Speed-up	
	Nodes	M	Nodes	M	Nodes	M	Time	Time	Utilization		1.1	1.2
2.1	1-11	2	-	-	-	-	135	72	.53	-	-	-
2.2	1-11	4	-	-	-	-	64	63	.99	-	2.12	-
2.3	1,8,9,11	2	2,4,6,10	2	3,5,7	2	37	83	.74	.98	3.62	1.71
2.4	1-4	2	5-8	2	9-11	2	33	82	.84	.97	4.08	1.92
2.5	1,2,3,6,9,11	2	4,5,7,8,10	2	-	-	46	72	.78	.99	2.95	1.39
2.6	1,2,3,6,9	2	4,5,7,8,10,11	2	-	-	59	71	.73	.65	2.28	1.08
2.7	1,2,3,6,9,11	2	4,5,7,8,10	2	-	-	54	62	.74	.54	2.49	1.17
2.8	1,2,3,6,9	2	4,5,7,8,10,11	2	-	-	57	62	.76	.43	2.39	1.13
2.9	1,3,5,8,9,11	4	2,4,6,7,10	4	-	-	34	65	.96	.99	3.98	1.88
2.10	1-5	4	6-11	4	-	-	38	72	.97	.99	3.61	1.70
2.11	1,3,5,8,9,11	4	2,4,6,7,10	4	-	-	36	68	.97	.66	3.76	1.77
2.12	1-5	4	6-11	4	-	-	42	68	.97	.40	3.21	1.51

Abbreviations:

M	Megabytes of memory
Real Time	Elapsed time in minutes
CPU Time	Computation time in minutes
CPU Utilization	(CPU time) / (Real Time – Idle time)
Concurrency	(Σ Elapsed time all processes are busy) / (Real Time)
Speedup	(Real Time for single process experiment) / (Real Time of current experiment)

Table 2: Experiment Set 2.

is very high because the run-away nodes do not allow the process to become idle. Furthermore, comparing experiment 2.3 with experiment 1.3, notice that by arbitrarily adding the eleventh node to one process, a well balanced distribution can become less well balanced. Experiment 2.4, on the other hand, is an entirely new assignment and performs better.

In experiments 2.5 and 2.6, we use the poorly performing allocation from experiment 1.10 and see if we can improve it by adding the eleventh node. By comparing 2.5 and 2.6, it is apparent that in 2.5 the addition of the eleventh node improved the concurrency, while in 2.6 it made it even worse. These same assignments were run again but this time the amount of time by which nodes could get ahead was limited. The results are shown in experiments 2.7 and 2.8. It is interesting to note that 2.8 actually performed better than 2.6 even though there is less concurrency. This can be attributed to the decreased demands on the memory by 2.8 because it does not create unnecessary information. Finally, note that, on the average, the processes in 2.5 and 2.6 are doing about ten minutes of unnecessary computation.

Experiments 2.9 and 2.10 serve to test the distributor on two machines with large amounts of memory. Experiment 2.9 performs better than 2.10, and it is interesting to note that the same node assignment schemes (without node eleven) provided opposite results in experiments 1.7 and 1.8. As a point of comparison, results for experiments 2.11 and 2.12 are shown. These are the same as 2.9 and 2.10 except that the amount of time nodes could get apart was constrained. Due to decreased concurrency, both cases performed worse, but one can see that the concurrency for 2.11 is indeed higher than that for 2.12.

5.3.3 Conclusions from the Experiments.

A number of conclusions can be based on these experiments. First, it is quite apparent that the simulation requires a large amount of memory, and when insufficient memory is available the real time needed for a simulation is drastically increased. In such cases, the distributed simulator can allow larger experiments to be performed because it can divide the memory requirements among machines so that each is within an acceptable range.

Second, the problem of assigning nodes to processes in order to maximize the concurrency is complex. In order to solve it, one must have knowledge as to the role of the node in generating the problem solution, the times and locations for sensor data, and a grasp of the communication strategies involved since these affect the degree to which nodes can get ahead of each other.

Third, and most importantly, to the question of whether the distributed simulator can significantly reduce the real-time requirements for running a simulation, the answer is a definite yes. In the case where memory is limited, the distributed simulator can run a simulation in less than a quarter of the time by just distributing it over two machines (experiments 1.5 and 1.6) while distributing it over three machines results in even better performance (experiments 1.3 and 1.4). In the case where memory is not the limiting factor, the concurrency achieved in the distributed simulator still results in impressive rate increases. Depending on the degree to which the load is balanced, real-time requirements can be nearly halved by dividing the simulation over two machines (experiment 2.9, for example).

First, it was necessary to provide some lower level facilities that would enable communication between concurrent processes on remote machines. This led to the development of buffer processes, which not only satisfy this need, but also provide mechanisms that will insure message integrity across the net, prevent deadlock due to full message buffers, and aid in the initialization and termination of the parallel simulation.

With the facilities in place for creating the processes and communication links needed in the parallel simulation, the next concern involved the changes needed in the VMT that would make the internal mechanisms more flexible so that they could treat with only a subset of the nodes. Provisions were made to eliminate reliance of shared memory for communication between nodes, and facilities to allow for user interaction with the distributed simulation were created.

Once the simulator was capable of being distributed, the next issue was to ensure that the cooperating processes be sufficiently synchronized such that the simulated events would still occur in the proper order. This was achieved in two different ways, one involving centralized control and the other distributing the control. In order to implement the latter, the scheduling of node executions on a single process evolved so as to both improve the synchronization mechanism and to increase the flexibility in the simulation itself.

Finally, with the parallel simulation implemented, the question of assigning tasks to processes was investigated. It was found that some rough predictions on the future activities of nodes could provide a basis for allocating nodes to processes so as improve the parallelism. Experimental results were presented that confirmed that the parallel simulation does indeed provide significant amounts of rate increase.

The experiments also suggested some important considerations in choosing a node to process allocation.

6.2 Future Research Opportunities

Just as the King finds that he cannot manage his large pencil (in the quote preceding this chapter), so too do researchers often find it difficult to manage large systems. Although an overall sense of direction is maintained, the different influences of past and present workers can have such surprising effects that it is sometimes difficult to predict the details of the evolving system, particularly if one also considers the rapid rate at which changes to the available hardware occur.

The implementation outlined in this thesis was constructed such that changes in hardware or software can be incorporated with a minimal amount of extra effort. For example, changes in the language of implementation may make interrupting mechanisms feasible, so that alternative communication strategies might be employed. The implementation of these new strategies should be relatively straight-forward.

As the available facilities expand, it is hoped that experiments on ten or more VAXs will be possible. It is also hoped that tools may be developed that will automatically generate a good node to process assignment based on some aspects of a particular environment. Furthermore, dynamic load balancing might become more feasible as alternative hardware is obtained. With faster I/O capabilities, nodes or even processes may be moved between machines. By increasing the amount of primary memory per machine, it may become possible to assign numerous processes to each machine, each process responsible for a single node. In such a scenario, dynamic load balancing will consist of the movement of processes among machines.

6.3 Significance of this Research

The principal motivation for the research presented in this paper was to improve the real-time rate of the VMT. As indicated by the experimental results, this objective was met. Therefore, a very significant aspect of this research is that it will allow larger and more interesting distributed problem solving networks to be studied.

In and of itself, the research is significant because it represents one of the few actual implementations of a large-scale distributed simulation, and perhaps the only distributed simulation of a distributed problem solving network. Although it introduces no new concepts to the theory of distributed simulations, this research is significant in that it extracts the pertinent ideas from a number of sources and integrates them to achieve effective results.

In converting systems so as to run in a distributed environment, some researchers might alter their systems significantly. If the modified systems are not run in a distributed environment, they could behave much worse than the original systems. Fortunately, in the simulation presented in this thesis, this was not the case. The changes that allow explicit message passing and variable numbers of nodes per process introduce negligible overhead. The more sophisticated scheduling mechanism has also not had a noticeable effect on the simulation rate. The single process rate of the simulation has therefore remained essentially the same, while distributing the simulation results in significant real-time rate gains.

Therefore, the research substantiates the initial claim that the simulation of a distributed problem solving network has an inherent propensity for parallelism that can be exploited to improve the real-time rate. Although this should not come as much of a surprise, it is hoped that the success of this work might stimulate other researchers to implement parallel simulations of other types of distributed problem solving networks. As has been mentioned throughout this thesis, the implementation is closely related to the particular problem solving network of the VMT. It is hoped that the details presented in this thesis can provide a framework for subsequent distributed simulations, and it would be of interest to see how the distributed simulation of a different problem solving network would differ from that presented here.

APPENDIX A

AN EXAMPLE DISTRIBUTION FILE

This appendix contains a commented copy of a distribution file for a ten node run (corresponding to experiment 1.5).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;                      DISTRIBUTION FILE                             ;;;;
;;;;                                                                    ;;;;
;;;;                       EXPERIMENT 1.5                               ;;;;
;;;;  Distributing the ten node environment among two machines in such  ;;;;
;;;;  a way as to distribute them both spacially as well as by their    ;;;;
;;;;  input data.                                                       ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;   NODE TO PROCESS ASSIGNMENT
;;;;         VMT process named MASTER gets nodes 1,3,5,8, and 9
;;;;         VMT process named PROCESS1 gets nodes 2,4,6,7, and 10

(("MASTER" 1 3 5 8 9)("PROCESS1" 2 4 6 7 10))

;;;;   PROCESS TO MACHINE ASSIGNMENT
;;;;         VAX6 gets VMT process named MASTER
;;;;         VAX7 gets VMT process named PROCESS1

((6 "MASTER")(7 "PROCESS1"))

;;;;   INSTRUCTIONS FOR PROCESSES
;;;;         MASTER will receive instructions directly from user
;;;;         PROCESS1 is instructed to
;;;;              1) set the system-trace variable to the desired attributes
;;;;              2) route the trace of the vmt simulator acting on the
;;;;                 specified environment to the given file
;;;;              3) evaluate the file which contains routines to output the
;;;;                 concurrency statistics for the given run.

(("MASTER" nil)
 ("PROCESS1" (progn (setq *system-trace '(node-executions matched-solutions
                                          ks-invocations transmitted-hyps))
                    (route-out "[disehd]e67a1.tra"
                               (vmt "[disehd.newvmt]ae10a020.env" nil))
                    (eval-file "[disehd.newvmt]idlestate.evl"))))
```

APPENDIX B

MERGED TRACE FILES

This appendix contains excerpts from the trace files of a distributed run. The environment run consisted of four nodes divided among two machines. The first two excerpts correspond to the traces from the individual processes, and the third is the merger of these two traces. Finally, an excerpt from a trace of the same environment run on a single process is provided for comparison. For a description of trace files, see [CORK83].

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Excerpt from trace generated by MASTER process which had nodes 1 and 4.
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 1 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- INVOKED KSI -----------> ksi:01:0016 21 s:gl:vl 39 (g:01:0010) (h:01:0020
                           h:01:0021 h:01:0022) (244)
- CREATED HYP -----------> h:01:0035 vl ((5 (12 12))) 2 (683)
- INSTANTIATED KSI ------> ksi:01:0022 fb:vl:vt (g:01:0025) (h:01:0035) <341 683> (408)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- RECEIVED HYP ----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
  INSTANTIATED KSI ------> ksi:04:0032 hyp-receive:vt nil nil <-10000 3973> (3973)
  INVOKED KSI -----------> ksi:04:0032 38 hyp-receive:vt 38 nil nil (3973)
- CREATED HYP -----------> h:04:0066 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
- INVOKED KSI -----------> ksi:04:0017 11 s:sl:gl 39 (g:04:0013) (h:04:0021
                           h:04:0022) (1467)
- CREATED HYP -----------> h:04:0067 gl ((5 (12 12))) 1 (3680)
- CREATED HYP -----------> h:04:0068 gl ((5 (12 12))) 2 (6519)
- CREATED HYP -----------> h:04:0069 gl ((5 (12 12))) 3 (3680)
- INSTANTIATED KSI ------> ksi:04:0033 s:gl:vl (g:04:0025) (h:04:0049
                           h:04:0050 h:04:0051 h:04:0067 h:04:0068 h:04:0069)
                           <794 9286> (2492)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 1 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
- INVOKED KSI -----------> ksi:01:0022 41 fb:vl:vt 41 (g:01:0025) (h:01:0035) (408)
- CREATED HYP -----------> h:01:0036 vt ((4 (10 10)) (5 (12 12))) 2 (683)
- INSTANTIATED KSI ------> ksi:01:0023 s:vt:pt (g:01:0029) (h:01:0036) <683 683> (682)
- INSTANTIATED KSI ------> ksi:01:0024 hyp-send:vt nil (h:01:0036) <-10000 683> (683)
- INVOKED KSI -----------> ksi:01:0024 43 hyp-send:vt 43 nil (h:01:0036) (683)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
- INVOKED KSI -----------> ksi:04:0025 23 s:gl:vl 41 (g:04:0023) (h:04:0043
                           h:04:0044 h:04:0045) (1429)
- CREATED HYP -----------> h:04:0070 vl ((4 (10 10))) 1 (3973)
- INSTANTIATED KSI ------> ksi:04:0034 fb:vl:vt (g:04:0031) (h:04:0070) <1986 6830> (2954)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 1 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
- INVOKED KSI -----------> ksi:01:0023 43 s:vt:pt 43 (g:01:0029 g:01:0030)
                           (h:01:0036) (736)
- CREATED HYP -----------> h:01:0037 pt ((4 (10 10)) (5 (12 12))) 2 (683)
- CREATED HYP -----------> h:01:0038 pt ((4 (13 7)) (5 (15 9))) 3 (341)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
- INVOKED KSI -----------> ksi:04:0027 25 s:gl:vl 43 (g:04:0025) (h:04:0049
                           h:04:0050 h:04:0051) (1429)
- CREATED HYP -----------> h:04:0071 vl ((5 (12 12))) 1 (3973)
- INSTANTIATED KSI ------> ksi:04:0035 ff:vl:vt (g:04:0034) (h:04:0071) <1986 6830> (2954)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Excerpt from trace generated by PROCESS1 which had nodes 2 and 3.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 2 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
  INSTANTIATED KSI ------> ksi:02:0046 hyp-receive:vt nil nil <-10000 3973> (3973)
  INVOKED KSI -----------> ksi:02:0046 38 hyp-receive:vt 38 nil nil (3973)
- CREATED HYP -----------> h:02:0103 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
* INVOKED KSI -----------> ksi:02:0043 35 s:gl:vl 39 (g:02:0028) (h:02:0046
                           h:02:0047 h:02:0048 h:02:0094 h:02:0095 h:02:0096) (2637)
* CREATED HYP -----------> h:02:0104 vl ((6 (16 12))) 1 (9688)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 3 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
  INSTANTIATED KSI ------> ksi:03:0046 hyp-receive:vt nil nil <-10000 3973> (3973)
  INVOKED KSI -----------> ksi:03:0046 38 hyp-receive:vt 38 nil nil (3973)
- CREATED HYP -----------> h:03:0103 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
* INVOKED KSI -----------> ksi:03:0043 35 s:gl:vl 39 (g:03:0028) (h:03:0046
                           h:03:0047 h:03:0048 h:03:0094 h:03:0095 h:03:0096) ( 2637)
* CREATED HYP -----------> h:03:0104 vl ((1 (6 2))) 1 (9688)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 2 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
* INVOKED KSI -----------> ksi:02:0044 37 s:gl:vl 41 (g:02:0030) (h:02:0052
                           h:02:0053 h:02:0054 h:02:0097 h:02:0098 h:02:0099) (2637)
* CREATED HYP -----------> h:02:0105 vl ((7 (18 14))) 1 (9688)
* INSTANTIATED KSI ------> ksi:02:0047 fb:vl:vt (g:02:0046) (h:02:0105) <4844 9688> (5812)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 3 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
* INVOKED KSI -----------> ksi:03:0044 37 s:gl:vl 41 (g:03:0030) (h:03:0052
                           h:03:0053 h:03:0054 h:03:0097 h:03:0098 h:03:0099) (2637)
* CREATED HYP -----------> h:03:0105 vl ((2 (8 4))) 1 (9688)
* INSTANTIATED KSI ------> ksi:03:0047 fb:vl:vt (g:03:0045) (h:03:0105) <4844 9688> (5812)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 2 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
* INVOKED KSI -----------> ksi:02:0045 39 s:gl:vl 43 (g:02:0032) (h:02:0058
                           h:02:0059 h:02:0060 h:02:0100 h:02:0101 h:02:0102) (2637)
* CREATED HYP -----------> h:02:0106 vl ((8 (20 16))) 1 (9688)
* INSTANTIATED KSI ------> ksi:02:0048 fb:vl:vt (g:02:0048) (h:02:0106) <4844 9688> (5812)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 3 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
* INVOKED KSI -----------> ksi:03:0045 39 s:gl:vl 43 (g:03:0032) (h:03:0058
                           h:03:0059 h:03:0060 h:03:0100 h:03:0101 h:03:0102) (2637)
* CREATED HYP -----------> h:03:0106 vl ((3 (10 6))) 1 (9688)
* INSTANTIATED KSI ------> ksi:03:0048 fb:vl:vt (g:03:0047) (h:03:0106) <4844 9688> (5812)
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Excerpt from trace formed by merging traces of MASTER and PROCESS1.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
   Executing Node 1 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
 - INVOKED KSI ------------> ksi:01:0016 21 s:gl:vl 39 (g:01:0010) (h:01:0020
                             h:01:0021 h:01:0022) (244)
 - CREATED HYP ------------> h:01:0035 vl ((5 (12 12))) 2 (683)
 - INSTANTIATED KSI -------> ksi:01:0022 fb:vl:vt (g:01:0025) (h:01:0035) <341 683> (408)
   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
   Executing Node 2 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
 - RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
   INSTANTIATED KSI -------> ksi:02:0046 hyp-receive:vt nil nil <-10000 3973> (3973)
   INVOKED KSI ------------> ksi:02:0046 38 hyp-receive:vt 38 nil nil (3973)
 - CREATED HYP ------------> h:02:0103 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
 * INVOKED KSI ------------> ksi:02:0043 35 s:gl:vl 39 (g:02:0028) (h:02:0046
                             h:02:0047 h:02:0048 h:02:0094 h:02:0095 h:02:0096) (2637)
 * CREATED HYP ------------> h:02:0104 vl ((6 (16 12))) 1 (9688)
   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
   Executing Node 3 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
 - RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
   INSTANTIATED KSI -------> ksi:03:0046 hyp-receive:vt nil nil <-10000 3973> (3973)
   INVOKED KSI ------------> ksi:03:0046 38 hyp-receive:vt 38 nil nil (3973)
 - CREATED HYP ------------> h:03:0103 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
 * INVOKED KSI ------------> ksi:03:0043 35 s:gl:vl 39 (g:03:0028) (h:03:0046
                             h:03:0047 h:03:0048 h:03:0094 h:03:0095 h:03:0096) (2637)
 * CREATED HYP ------------> h:03:0104 vl ((1 (6 2))) 1 (9688)
   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
   Executing Node 4 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
 - RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
   INSTANTIATED KSI -------> ksi:04:0032 hyp-receive:vt nil nil <-10000 3973> (3973)
   INVOKED KSI ------------> ksi:04:0032 38 hyp-receive:vt 38 nil nil (3973)
 - CREATED HYP ------------> h:04:0066 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
 - INVOKED KSI ------------> ksi:04:0017 11 s:sl:gl 39 (g:04:0013) (h:04:0021
                             h:04:0022) (1467)
 - CREATED HYP ------------> h:04:0067 gl ((5 (12 12))) 1 (3680)
 - CREATED HYP ------------> h:04:0068 gl ((5 (12 12))) 2 (6519)
 - CREATED HYP ------------> h:04:0069 gl ((5 (12 12))) 3 (3680)
 - INSTANTIATED KSI -------> ksi:04:0033 s:gl:vl (g:04:0025) (h:04:0049
                             h:04:0050 h:04:0051 h:04:0067 h:04:0068 h:04:0069)
                             <794 9286> (2492)
   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
   Executing Node 1 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
 - INVOKED KSI ------------> ksi:01:0022 41 fb:vl:vt 41 (g:01:0025) (h:01:0035) (408)
 - CREATED HYP ------------> h:01:0036 vt ((4 (10 10)) (5 (12 12))) 2 (683)
 - INSTANTIATED KSI -------> ksi:01:0023 s:vt:pt (g:01:0029) (h:01:0036) <683 683> (682)
 - INSTANTIATED KSI -------> ksi:01:0024 hyp-send:vt nil (h:01:0036) <-10000 683> (683)
 - INVOKED KSI ------------> ksi:01:0024 43 hyp-send:vt 43 nil (h:01:0036) (683)
   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
   Executing Node 2 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
 * INVOKED KSI ------------> ksi:02:0044 37 s:gl:vl 41 (g:02:0030) (h:02:0052
                             h:02:0053 h:02:0054 h:02:0097 h:02:0098 h:02:0099) (2637)
 * CREATED HYP ------------> h:02:0105 vl ((7 (18 14))) 1 (9688)
 * INSTANTIATED KSI -------> ksi:02:0047 fb:vl:vt (g:02:0046) (h:02:0105) <4844 9688> (5812)
   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
   Executing Node 3 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
 * INVOKED KSI ------------> ksi:03:0044 37 s:gl:vl 41 (g:03:0030) (h:03:0052
                             h:03:0053 h:03:0054 h:03:0097 h:03:0098 h:03:0099) (2637)
 * CREATED HYP ------------> h:03:0105 vl ((2 (8 4))) 1 (9688)
 * INSTANTIATED KSI -------> ksi:03:0047 fb:vl:vt (g:03:0045) (h:03:0105) <4844 9688> (5812)
```

```
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
- INVOKED KSI -----------> ksi:04:0025 23 s:gl:vl 41 (g:04:0023) (h:04:0043
                           h:04:0044 h:04:0045) [1429]
- CREATED HYP -----------> h:04:0070 vl ((4 (10 10))) 1 [3973]
- INSTANTIATED KSI ------> ksi:04:0034 fb:vl:vt (g:04:0031) (h:04:0070) <1986 6830> [2954]
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 1 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
- INVOKED KSI -----------> ksi:01:0023 43 s:vt:pt 43 (g:01:0029 g:01:0030)
                           (h:01:0036) [736]
- CREATED HYP -----------> h:01:0037 pt ((4 (10 10)) (5 (12 12))) 2 [683]
- CREATED HYP -----------> h:01:0038 pt ((4 (13 7)) (5 (15 9))) 3 [341]
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 2 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
* INVOKED KSI -----------> ksi:02:0045 39 s:gl:vl 43 (g:02:0032) (h:02:0058
                           h:02:0059 h:02:0060 h:02:0100 h:02:0101 h:02:0102) [2637]
* CREATED HYP -----------> h:02:0106 vl ((8 (20 16))) 1 [9688]
* INSTANTIATED KSI ------> ksi:02:0048 fb:vl:vt (g:02:0048) (h:02:0106) <4844 9688> [5812]
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 3 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
* INVOKED KSI -----------> ksi:03:0045 39 s:gl:vl 43 (g:03:0032) (h:03:0058
                           h:03:0059 h:03:0060 h:03:0100 h:03:0101 h:03:0102) [2637]
* CREATED HYP -----------> h:03:0106 vl ((3 (10 6))) 1 [9688]
* INSTANTIATED KSI ------> ksi:03:0048 fb:vl:vt (g:03:0047) (h:03:0106) <4844 9688> [5812]
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
- INVOKED KSI -----------> ksi:04:0027 25 s:gl:vl 43 (g:04:0025) (h:04:0049
                           h:04:0050 h:04:0051) [1429]
- CREATED HYP -----------> h:04:0071 vl ((5 (12 12))) 1 [3973]
- INSTANTIATED KSI ------> ksi:04:0035 ff:vl:vt (g:04:0034) (h:04:0071) <1986 6830> [2954]
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Excerpt from the trace generated by the non-distributed four node run.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 1 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- INVOKED KSI ------------> ksi:01:0016 21 s:gl:vl 39 (g:01:0010) (h:01:0020
                            h:01:0021 h:01:0022) (244)
- CREATED HYP ------------> h:01:0035 vl ((5 (12 12))) 2 (683)
- INSTANTIATED KSI -------> ksi:01:0022 fb:vl:vt (g:01:0025) (h:01:0035) <341 683> (408)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 2 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
  INSTANTIATED KSI -------> ksi:02:0046 hyp-receive:vt nil nil <-10000 3973> (3973)
  INVOKED KSI ------------> ksi:02:0046 38 hyp-receive:vt 38 nil nil (3973)
- CREATED HYP ------------> h:02:0103 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
* INVOKED KSI ------------> ksi:02:0043 35 s:gl:vl 39 (g:02:0028) (h:02:0046
                            h:02:0047 h:02:0048 h:02:0094 h:02:0095 h:02:0096) (2637)
* CREATED HYP ------------> h:02:0104 vl ((6 (16 12))) 1 (9688)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 3 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
  INSTANTIATED KSI -------> ksi:03:0046 hyp-receive:vt nil nil <-10000 3973> (3973)
  INVOKED KSI ------------> ksi:03:0046 38 hyp-receive:vt 38 nil nil (3973)
- CREATED HYP ------------> h:03:0103 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
* INVOKED KSI ------------> ksi:03:0043 35 s:gl:vl 39 (g:03:0028) (h:03:0046
                            h:03:0047 h:03:0048 h:03:0094 h:03:0095 h:03:0096) (2637)
* CREATED HYP ------------> h:03:0104 vl ((1 (6 2))) 1 (9688)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 19 ------ Time Frame 8 ------ Node Time 39
- RECEIVED HYP -----------> 1 38 h:01:0031 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
  INSTANTIATED KSI -------> ksi:04:0032 hyp-receive:vt nil nil <-10000 3973> (3973)
  INVOKED KSI ------------> ksi:04:0032 38 hyp-receive:vt 38 nil nil (3973)
- CREATED HYP ------------> h:04:0066 vt ((4 (10 10)) (5 (12 12))) 1 (3973)
- INVOKED KSI ------------> ksi:04:0017 11 s:sl:gl 39 (g:04:0013) (h:04:0021
                            h:04:0022) (1467)
- CREATED HYP ------------> h:04:0067 gl ((5 (12 12))) 1 (3680)
- CREATED HYP ------------> h:04:0068 gl ((5 (12 12))) 2 (6519)
- CREATED HYP ------------> h:04:0069 gl ((5 (12 12))) 3 (3680)
- INSTANTIATED KSI -------> ksi:04:0033 s:gl:vl (g:04:0025) (h:04:0049
                            h:04:0050 h:04:0051 h:04:0067 h:04:0068 h:04:0069)
                            <794 9286> (2492)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 1 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
- INVOKED KSI ------------> ksi:01:0022 41 fb:vl:vt 41 (g:01:0025) (h:01:0035) (408)
- CREATED HYP ------------> h:01:0036 vt ((4 (10 10)) (5 (12 12))) 2 (683)
- INSTANTIATED KSI -------> ksi:01:0023 s:vt:pt (g:01:0029) (h:01:0036) <683 683> (682)
- INSTANTIATED KSI -------> ksi:01:0024 hyp-send:vt nil (h:01:0036) <-10000 683> (683)
- INVOKED KSI ------------> ksi:01:0024 43 hyp-send:vt 43 nil (h:01:0036) (683)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 2 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
* INVOKED KSI ------------> ksi:02:0044 37 s:gl:vl 41 (g:02:0030) (h:02:0052
                            h:02:0053 h:02:0054 h:02:0097 h:02:0098 h:02:0099) (2637)
* CREATED HYP ------------> h:02:0105 vl ((7 (18 14))) 1 (9688)
* INSTANTIATED KSI -------> ksi:02:0047 fb:vl:vt (g:02:0046) (h:02:0105) <4844 9688> (5812)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 3 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
* INVOKED KSI ------------> ksi:03:0044 37 s:gl:vl 41 (g:03:0030) (h:03:0052
                            h:03:0053 h:03:0054 h:03:0097 h:03:0098 h:03:0099) (2637)
* CREATED HYP ------------> h:03:0105 vl ((2 (8 4))) 1 (9688)
* INSTANTIATED KSI -------> ksi:03:0047 fb:vl:vt (g:03:0045) (h:03:0105) <4844 9688> (5812)
```

```
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 20 ------ Time Frame 8 ------ Node Time 41
- INVOKED KSI -----------> ksi:04:0025 23 s:gl:vl 41 (g:04:0023) (h:04:0043
                           h:04:0044 h:04:0045) (1429)
- CREATED HYP -----------> h:04:0070 vl ((4 (10 10))) 1 (3973)
- INSTANTIATED KSI ------> ksi:04:0034 fb:vl:vt (g:04:0031) (h:04:0070) <1986 6830> (2954)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 1 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
- INVOKED KSI -----------> ksi:01:0023 43 s:vt:pt 43 (g:01:0029 g:01:0030)
                           (h:01:0036) (736)
- CREATED HYP -----------> h:01:0037 pt ((4 (10 10)) (5 (12 12))) 2 (683)
- CREATED HYP -----------> h:01:0038 pt ((4 (13 7)) (5 (15 9))) 3 (341)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 2 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
* INVOKED KSI -----------> ksi:02:0045 39 s:gl:vl 43 (g:02:0032) (h:02:0058
                           h:02:0059 h:02:0060 h:02:0100 h:02:0101 h:02:0102) (2637)
* CREATED HYP -----------> h:02:0106 vl ((8 (20 16))) 1 (9688)
* INSTANTIATED KSI ------> ksi:02:0048 fb:vl:vt (g:02:0048) (h:02:0106) <4844 9688> (5812)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 3 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
* INVOKED KSI -----------> ksi:03:0045 39 s:gl:vl 43 (g:03:0032) (h:03:0058
                           h:03:0059 h:03:0060 h:03:0100 h:03:0101 h:03:0102) (2637)
* CREATED HYP -----------> h:03:0106 vl ((3 (10 6))) 1 (9688)
* INSTANTIATED KSI ------> ksi:03:0048 fb:vl:vt (g:03:0047) (h:03:0106) <4844 9688> (5812)
  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  Executing Node 4 ------ Inv Ksis 21 ------ Time Frame 8 ------ Node Time 43
- INVOKED KSI -----------> ksi:04:0027 25 s:gl:vl 43 (g:04:0025) (h:04:0049
                           h:04:0050 h:04:0051) (1429)
- CREATED HYP -----------> h:04:0071 vl ((5 (12 12))) 1 (3973)
- INSTANTIATED KSI ------> ksi:04:0035 ff:vl:vt (g:04:0034) (h:04:0071) <1986 6830> (2954)
```

APPENDIX C

SAMPLE LOG FILE

The following is an excerpt from a log file generated by a four node run. Annotations are included within braces and are not part of the generated output.

```
{
 This is an excerpt of the log file generated by the buffer process on the
 machine that did not contain the MASTER process for the four node run
 distributed over two machines (MASTER had nodes 1 and 4, PROCESS1 had nodes
 2 and 3). }

transfer = F forward = F wait = F  { these flags indicate:
                                      transfer - are we in transfer mode?
                                      forward - are we trying to forward
                                                 messages?
                                      wait - are we in a read-wait state? }
The message is  { This is the message as read from the buffer process's
                   mailbox.  It is a hyp/goal message. }
%PROCESS1%%MASTER%$(hyp/goal (2 50 (4) hyp (gh0003 gh0011 gh0019 gh0027 gh0035
gh0058 gh0059 gh0060) (1 2 3 4) nil ((3 (10 6)) (4 (10 10))) 1 6830 (9688 3973)
 h:04:0072 nil vt))

Store   { Storing the message }
positions is         17   positiond is          9     {parsing the header}
Source is MASTER        Destination is PROCESS1
Message is                                            {and extracting the
                                                              message}
(hyp/goal (2 50 (4) hyp (gh0003 gh0011 gh0019 gh0027 gh0035 gh0058 gh0059 gh006
0) (1 2 3 4) nil ((3 (10 6)) (4 (10 10))) 1 6830 (9688 3973) h:04:0072 nil vt))


transfer = F forward = F wait = F
The message is                     {reception of end-of-message message}
%PROCESS1%%MASTER%$(endofmessages)


Store
positions is         17   positiond is          9
Source is MASTER        Destination is PROCESS1
Message is
(endofmessages)


transfer = F forward = F wait = F
The message is                              { reception of another
                                                   hyp/goal message}
%PROCESS1%%MASTER%$(hyp/goal (3 50 (4) hyp (gh0003 gh0011 gh0019 gh0027 gh0035
gh0058 gh0059 gh0060) (1 2 3 4) nil ((3 (10 6)) (4 (10 10))) 1 6830 (9688 3973)
 h:04:0072 nil vt))

Store
positions is         17   positiond is          9   { and storing it }
Source is MASTER        Destination is PROCESS1
Message is
(hyp/goal (3 50 (4) hyp (gh0003 gh0011 gh0019 gh0027 gh0035 gh0058 gh0059 gh006
0) (1 2 3 4) nil ((3 (10 6)) (4 (10 10))) 1 6830 (9688 3973) h:04:0072 nil vt))
```

```
transfer =  F forward =  F wait =  F     { and end-of-message message }
The message is
%PROCESS1%%MASTER%$(endofmessages)


Store
positions is            17   positiond is             9
Source is MASTER           Destination is PROCESS1
Message is
(endofmessages)

transfer =  F forward =  F wait =  F
The message is                            { Reception of an update message
                                            stating that node 4 is active and
                                            is at time 47. Note that update
                                            messages do not need end-of-message
                                            messages because they are
                                            guaranteed not to be divided. }
%PROCESS1%%MASTER%$(update (4 active 47 47))


Store
positions is            17   positiond is             9

Source is MASTER           Destination is PROCESS1
Message is
(update (4 active 47 47))


transfer =  F forward =  F wait =  F
The message is                            { Reception of a control message
                                            which asks buffer process to
                                            forward the messages. }

$fp$$


This is a control message
Forward enabled, numsg is          3     { This is the number of messages }
Forwarded message is (update (4 active 47 47))   { first forward updates }
Forwarded message is                           { Then hyp/goal messages }
(hyp/goal (2 50 (4) hyp (gh0003 gh0011 gh0019 gh0027 gh0035 gh0058 gh0059 gh006
0) (1 2 3 4) nil ((3 (10 6)) (4 (10 10))) 1 6830 (9688 3973) h:04:0072 nil vt))
Forwarded message is
(hyp/goal (3 50 (4) hyp (gh0003 gh0011 gh0019 gh0027 gh0035 gh0058 gh0059 gh006
0) (1 2 3 4) nil ((3 (10 6)) (4 (10 10))) 1 6830 (9688 3973) h:04:0072 nil vt))

                                          { And then back to reading mailbox }
transfer =  F forward =  F wait =  F
The message is
%PROCESS1%%MASTER%$(hyp/goal (2 52 (4) hyp (gh0006 gh0014 gh0022 gh0030 gh0038
gh0063 gh0064 gh0065) (1 2 3 4) nil ((5 (12 12)) (6 (16 12))) 1 6830 (3973 9688
) h:04:0074 nil vt))

Store
positions is            17   positiond is             9
Source is MASTER           Destination is PROCESS1
Message is
(hyp/goal (2 52 (4) hyp (gh0006 gh0014 gh0022 gh0030 gh0038 gh0063 gh0064 gh006
5) (1 2 3 4) nil ((5 (12 12)) (6 (16 12))) 1 6830 (3973 9688) h:04:0074 nil vt)
)
                                                  { etc. }
```

References.

BRYA79 Randal E. Bryant.
Simulation on a distributed system.
Proceedings of the First International Conference on Distributed Computing Systems, pages 544-552, October 1979.

CHAN78 K. M. Chandy and J. Misra.
A nontrivial example of concurrent processing: Distributed simulation
Proceedings of COMPSAC '78, pages 822-826, IEEE 1978.

CORK83 Daniel D. Corkill.
A Framework for organizational self-design in distributed problem solving networks.
PhD Thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, February 1983.

ERMA80 Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy.
The Hearsay-II speech understanding system: Integrating knowledge to resolve uncertainty.
Computing Surveys 12(2):213-253, June 1980.

JEFF82 David Jefferson and Henry Sowizral.
Fast concurrent simulation using the time warp mechanism, Part I: Local Control.
The Rand Corporation, N-1906-AF, 1982.

LESS81 Victor R. Lesser and Daniel D. Corkill.
Functionally accurate, cooperative distributed systems.
IEEE Transactions on Systems, Man, and Cybernetics SMC-11(1):81-96, January 1981.

LESS83 Victor R. Lesser and Daniel D. Corkill.
The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks.
AI Magazine, 4(3):15-33, Fall 1983.

PEAC79 J. Kent Peacock, J. W. Wong, and Eric G. Manning.
Distributed simulation using a network of processors.
Computer Networks 3(1):44-56, February 1979.

PEAC80 J. Kent Peacock, Eric Manning, and J. W. Wong.
Synchronization of distributed simulation using broadcast algorithms.
Computer Networks 4(1):3-10, February 1980.