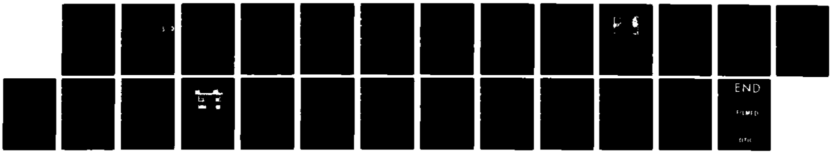
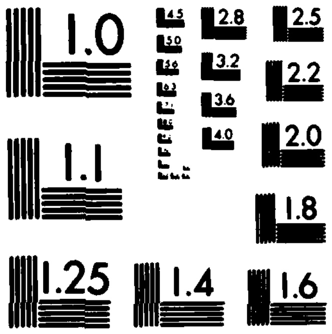
AD-A149 065 DISTRIBUTING A DISTRIBUTED PROBLEM SOLVING NETWORK SIMULATOR(U) MASSACHUSETTS UNIV AMHERST DEPT OF COMPUTER AND INFORMATION SCIENCE E H DURFEE ET AL. 1/1

UNCLASSIFIED MAY 84 N00014-79-C-0439 F/G 9/2 NL




1.0
4.5
2.8
2.5
5.0
5.6
3.2
2.2
6.3
3.6
7.1
8.0
4.0
2.0
1.1
1.8
1.25
1.4
1.6

Distributing a Distributed Problem Solving Network Simulator

Edmund H. Durfee, Daniel D. Corkill, and Victor R. Lesser
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

May, 1984

Abstract

We describe our experience in improving the real-time performance of a particular large and complex simulator through distributed processing. Our goal was to both reduce the real time required and to incr ase 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. We present data showing that the additional CPU power and the combined physical memory available in the network contribute to significant real-time speedup. Experience with a two and three machine network indicates that where there was no memory contention in a single process simulator, we obtain a speedup proportional to the number of processes. Where there was memory contention in the single process simulator, the speed up is much more dramatic. We also detail the capabilities that were added to the conventional network communication structure to implement, debug, and interact with the distributed simulator.

1. Introduction

One of the most intensive types of computational tasks is simulation [JEFF82]. Some software simulations may require hours or even days of computer time. When simulations are used to experiment with the design of complex systems, the *real time* required for each simulation significantly limits the number of alternative designs that can be explored. A simulator that finishes in minutes will be used quite differently from one that requires many hours. In this paper we describe our experience in improving the real-time performance of a particular large and complex simulator through distributed processing.

This work was supported in part by the National Science Foundation under Grant MCS78-04212, and in part by the Office of Naval Research under Grant N0014-79-C-0439.

Our 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 a DECNet/ETHERNET-II communication structure.

Our distributed simulator itself models a distributed system, and although the simulated distributed system and the distributed simulator impinge on one another, it is important to keep them clearly separate. We have implemented the distributed simulator—not the simulated distributed system.

Our simulator is called the Vehicle Monitoring Testbed (VMT) [LESS83], and its characteristics strongly influenced our approach and success. The VMT is used for empirically evaluating different strategies for organising distributed problem solving networks. Distributed problem solving networks are a special class of distributed processing systems in which semi-autonomous and loosely-coupled processing *nodes* cooperatively interact to solve a *single* problem where nodes are capable of making sophisticated local control decisions about what tasks they are to perform and what information they should communicate. Each task requires a relatively large amount of computation and memory to simulate. Contrastingly, a relatively small amount of internode communication needs to be simulated due to the loosely-coupled nature of distributed problem solving networks.

The distribution of the VMT simulator is interesting from a number of perspectives. First, there is little practical experience with the design and performance of distributed simulation systems [JEFF82], especially with the issues of how to build such a system onto the top of a conventional operating system (such as VAX/VMS) which has primitive network facilities. There is also little experience with the tools needed to debug and interact with such a distributed simulation system. Second, implementing a simulator of a network of loosely-coupled nodes (where the simulation of each node requires a significant amount of computation and memory) on a loosely-coupled distributed hardware system presents design options that are infeasible in more conventional simulations (where nodes or objects are more tightly coupled or where activity can be cheaply simulated). Finally, reducing the real time required by the VMT simulator comes from increasing the physical memory available to simulation by having more processors as well as from concurrency in the distributed simulation. A simulation of a small node system of five to ten nodes on a single-user VAX 11/750 with two megabytes of memory takes about seven hours of real time and two hours of CPU usage. The five hours of idle time stems from *thrashing* due to limited physical memory. By exploiting both additional processor cycles and memories available through the network, we are able to significantly reduce the real time required for a particular simulation experiment. Additionally, the use of multiple processors allow us to reasonably simulate distributed problem solving networks containing 50 to 100 nodes.

The remainder of the paper is divided into five sections. The first section describes the basic VMT simulator. The implementation of the distributed simulator is inextricably linked with the simulation of the node network itself. Therefore, the implementation of the distributed simulator requires an understanding of certain aspects of the VMT. The second section details how the simulator was distributed and discusses the process decomposition of the simulator and the coordination strategy among these processes (needed to produce results that are identical to those of the single process VMT simulator). The third section overviews facilities that were developed to facilitate interaction and debugging with the ongoing simulation. The fourth section presents data indicating the speed up achieved through distribution and the effect of different static load balancing strategies. The last section summarizes the paper and discusses future research plans.

2. The Distributed Vehicle Monitoring Network Simulator

The Vehicle Monitoring Testbed is a simulator for a cooperative distributed problem solving network [LESS83]. 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. 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.

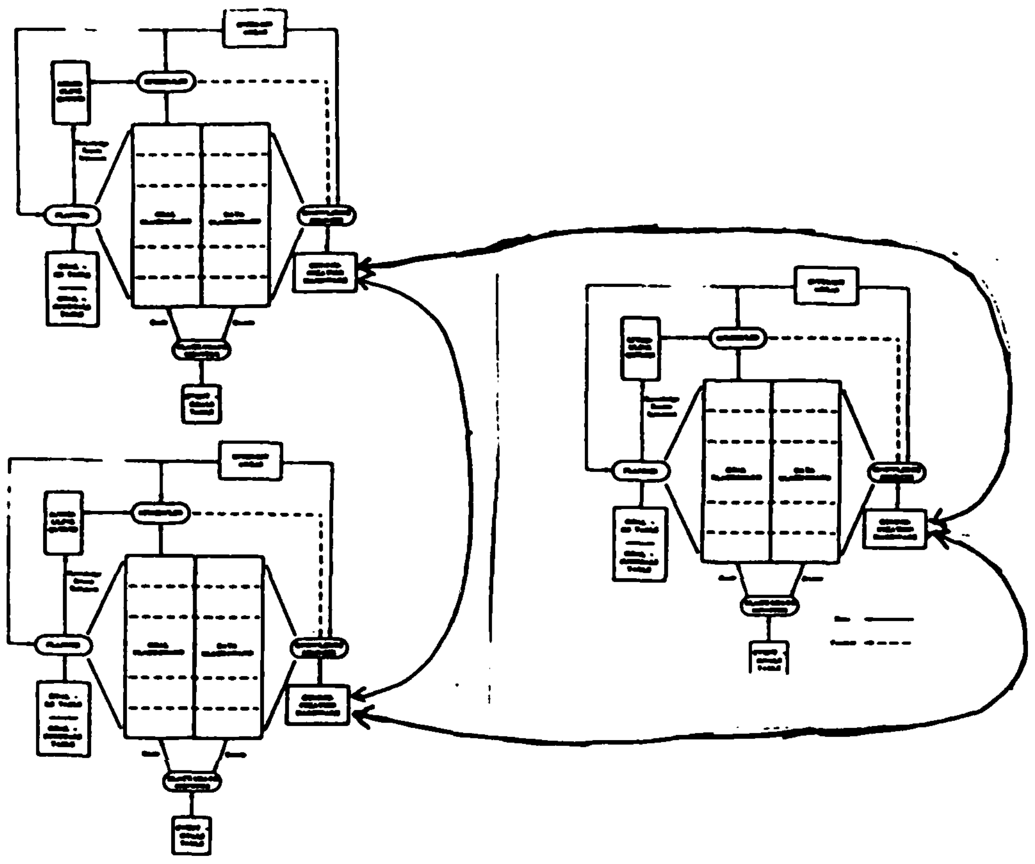
Each of the problem solving nodes has the architecture first developed in the Hearsay-II speech understanding system [ERMA80], with extensions to this architecture that enable the nodes to have more sophisticated control and to give the nodes the ability to communicate data and control information (Figure 1) [CORK82].

2.1 The Architecture of Each Simulated Node

Each simulated node is assumed to have three processors and a set of shared memory *blackboards* Figure 2. The majority of the computation occurs on the node's *local processor*. This processor is responsible for performing all of the node's basic problem solving and controls the other two processors. The *transmission processor* has a queue containing the messages to be sent to other nodes. The *reception 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 and executing tasks called *knowledge source* (KS) executions. The local processor executes *local problem solving* KSs, the transmission processor *send* KSs, and the reception processor *receive* KSs. Each of the processors has a priority-ordered queue of pending KS executions waiting to be executed. Each processor also has access to the blackboards, although the activities of the transmission and reception processors are controlled by the local processor. In this way, local processing and the transmission and reception of messages can all be done concurrently.

KS executions are *non-interruptable*—once started an execution continues to completion. In addition, messages received during a local problem solving KS execution are not seen by that KS.



The problem solving components of three nodes are shown along with the communication links among them.

Figure 1: Node Problem Solving Architecture.

2.2 Node Executions

In order to simulate the concurrent processing of a number of nodes in a single process it is necessary to interleave the nodes' activities. Each of these interleaved activities is referred to as a *node execution* and typically consists of some number of communication KS executions and a single local KS execution.

The simulator must not only keep track of the simulated time of each node (defined as the time of the local problem solving processor of the node), but also the simulated times of the transmission and reception processors. The transmission and reception processors may be at somewhat different clock times from the simulated node time, although all of the times are related, since a node cannot be allowed to proceed too far ahead of its reception processor lest it proceed beyond the time that it should incorporate some received data

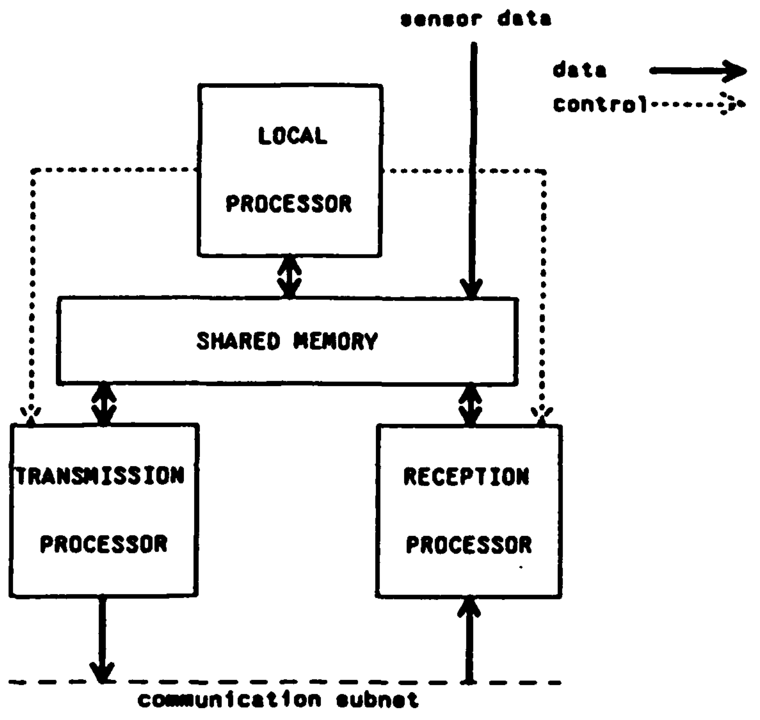


Figure 2: Simulated Node Architecture.

into its blackboards.

Since KS executions are non-interruptable, a node execution should not be started unless it can also be completed. To guarantee this, all messages that could possibly be incorporated into the blackboard up until the start time of the next local problem solving KS of the node execution must be received. Associated with each message is a *timestamp* value [CHAN78] indicating the time at the receiving processor that the message is to be received. This timestamp value is based on the simulated transmission time of the sending node and the simulated channel delay between the sending and receiving node.

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 pending KS execution queue from which the local processor will choose its next KS execution. Next, the local KS execution with the highest priority is extracted from the queue, and its simulated runtime 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 KS execution. Finally, the local KS is executed, and the local time of the node is advanced to the end time of this KS execution. Note that it is important that transmission KSs be executed before the local KS, since the changes on the blackboard due to the local KS should not affect the transmission KSs. Transmission KSs do not alter the blackboards, so their executions will have no effect on the local KS.

It is likely that there will not always be sufficient communication KS executions 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 accelerated to exactly the initial local time [BRYA79], 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 KS if the sending queue becomes empty.

On the other hand, if the local KS execution queue becomes empty, the runtime of the "local KS execution" for this node execution is sero. 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 details of node scheduling in the distributed simulator are described by Durfee [DURF84].

3. Distributing The Simulator

There are two major difficulties in the parallel implementation of a simulation that will be addressed:

- How the tasks in the simulation should be divided among processes and machines so as to maximize concurrency and thus minimise run time.
- How the processes 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 synchronisation are highly dependent on the simulation that is to be implemented. For example, instead of avoiding synchronisation errors by forcing processes to wait for one another, a detection and recovery mechanism could be used [JEFF82]. In such a strategy, detection of a synchronisation error requires that a process recognise 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 can require much more memory, and if the rate of errors is high, then there can 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.

Before we discuss the details of the simulator, it is important to make clear the difference between the simulated distributed problem solving network and the distributed implementation of that simulator. To avoid confusion, we adhere to the following terminology (Figure 3):

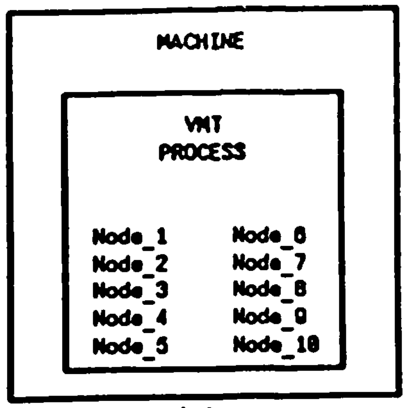
node	A *simulated* semi-autonomous problem solving system.
node network	A *simulated* distributed network of communicating, semi-autonomous problem solving nodes.
process	An instance of the VMT simulator.
machine	A *physical* computer capable of supporting one or more processes.
machine network	A *physical*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

3.1 A Cooperating Network of VMT Simulators

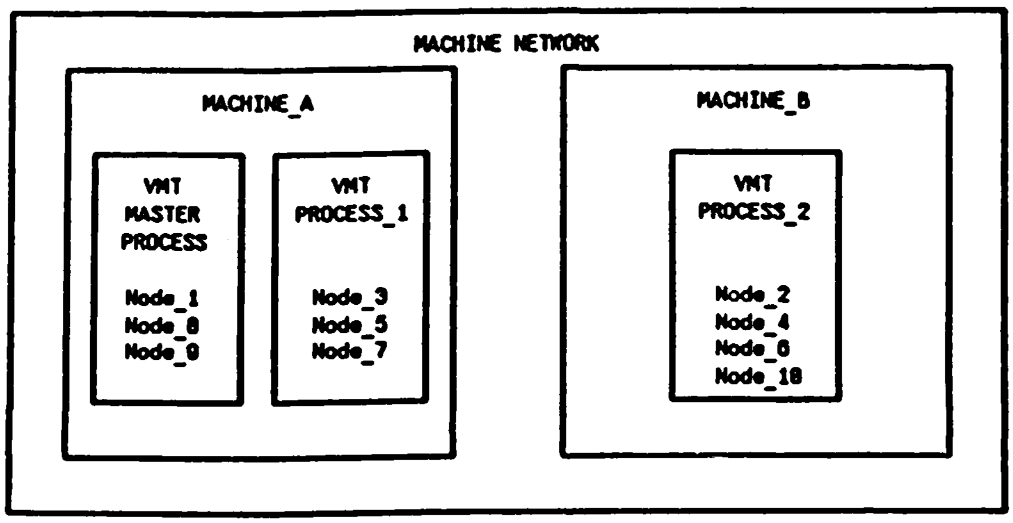
Our approach in distributing the VMT simulator is relatively straightforward and closely mirrors the cooperative semi-autonomous process structure that the VMT is simulating. The distributed simulator consists of a number of slightly modified single-process VMT simulators replicated in a number of processes. Each of these VMT processes is responsible for the simulation of one or more problem solving nodes (Figure 3). These processes are then assigned to machines. In our current implementation, this machine assignment is static. Given the memory intensive nature of the simulation and the constant memory overhead associated with a VMT process, there is never an advantage in placing more than one VMT process on a machine. The limited physical memory available on our machines favors the allocation of multiple nodes to a process rather than multiple processes to a machine. Similarly, in our DECNet/ETHERNET-II environment, there is an extreme transfer cost in moving a process to to another machine due to the size of the process image. In a more closely-coupled machine network where the bandwidth between machines is much greater (and where there is sufficient physical memory), process movement among machines may be practical and dynamic load balancing a possibility. In such a situation, assigning one simulated node to each process is probably worthwhile. Such an allocation would provide maximum flexibility in balancing the processing activity during a simulation.

3.2 Establishing the Process Environment for the Distributed Simulation

The distributed simulation was implemented using DECNet on a set of VAX 11/750 computers running VMS and interconnected by a local area network (ethernet). Interprocess communication is achieved through the use of mailboxes. Associated with each VMT process is a mailbox that will contain the incoming messages to that process. In order to communicate with another process, a process need merely put a message into the other process's mailbox. The loosely-coupled nature of the desired distributed simulator required that the interprocess communication be asynchronous. Unfortunately, general



(a)



(b)

In part (a) a ten node simulation is shown as it is run as a single process. Part (b) shows how the same simulation experiment might be distributed.

Figure 3: Node, Process, and Machine Allocation.

asynchronous interprocess communication between the VMT processes was not directly supported by DECNet. Therefore, we were forced to build an extra communication layer to support the distributed simulator.

The key to the extra communication layer was the use of a *buffer process* on each machine. Each buffer process is responsible for receiving all messages destined for the VMT processes on its machine and for forwarding them to the appropriate mailboxes of its machine's VMT processes. Each buffer process contains a message buffer for each of its VMT processes. These buffers accumulate messages until the particular VMT process requests them. Buffer processes receive messages of two types: messages to be forwarded to VMT processes and instructions to the buffer process itself.

Initiating a VMT simulation on the distributed network starts with the user invoking

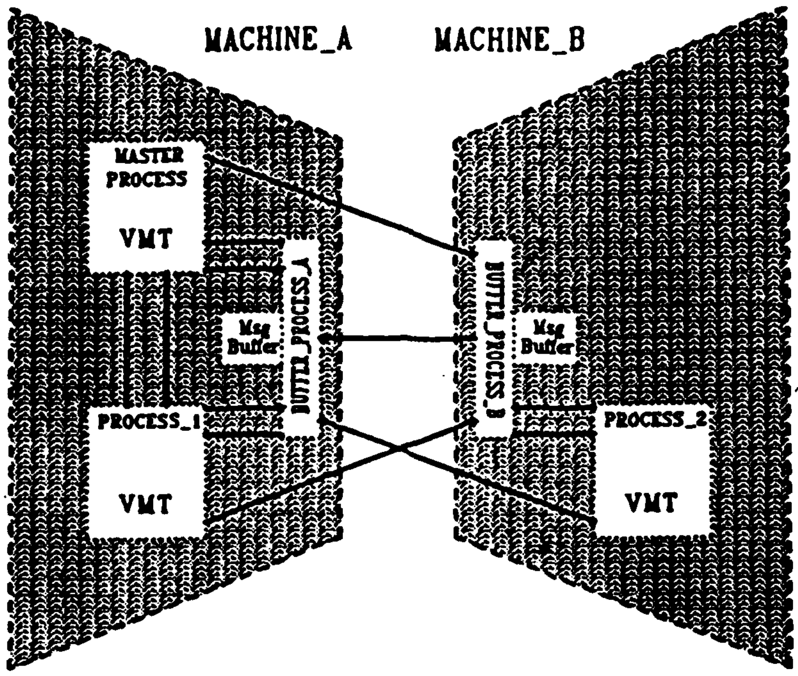


Figure 4: VMT Processes and Buffer Processes.

a VMT process on a machine. This process, which becomes the process responsible for the run, is known as the *master* VMT process. Its first act is to read the distributed simulation information from a previously prepared file called a *distribution file*. This file informs the master of what other VMT processes to create, what machine to create each on, and what nodes are assigned to each VMT process.

The next step involves creating the buffer processes on each machine, starting with the machine on which the master resides. Once all buffer processes have been created, the master informs each the buffer processes as to what VMT processes to create on its machine. The master through the buffer processes inform each VMT process of the communication paths needed to communicate with simulated nodes executing in other VMT processes. 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 VMT processes itself. The resulting process structure for the example node/process/machine allocation of Figure 3 is shown in Figure 4.

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. The master can ask the buffer process to delete itself.

In addition to these activities, the buffer processes also facilitate breaking long messages into packets which are reassembled by the buffer process and the detection of deadlock due to insufficient message buffer capacity. Details of these activities are described by Durfee [DURF84].

3.3 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. 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.

Therefore, 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.

3.4 VMT Process Synchronisation

The distribution of the VMT simulator 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. Certainly, 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 distributed simulator must have some sort of synchronisation 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.

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. Hence, knowledge of the current times of any nodes that can send it a message are 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 synchronisation without potential non-determinism.

In a distributed simulation, 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. The scheduling strategy is modified such that *the node furthest behind* *that is unblocked is the next node to be executed.* Hence, 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.

There is a difficulty in the treatment of idle nodes. 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. However, the next known transmission or reception time is also inappropriate, because the node could receive a message causing this next event time estimate to be changed to an earlier time.

As a solution to this predicament, the concept of *global time* is introduced into the distributed simulation [PEAC80]. The global time is the minimum of all the node's next event times, as perceived by a particular process. 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. Hence, it is 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.

3.5 Node Update Messages

Note that the scheduling of actions in the distributed 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 the VMT system, a single node execution usually encompasses a large amount of computation and memory usage and typically requires a large amount of time relative to the amount of time required for machine network communication. 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 requires several seconds.

When the relative times are considered, it seems unlikely 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.

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. Certainly, 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 it 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.

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 [PEAC80]. It should be noted that a similar conclusion might not be reached when trying to determine a strategy for another distributed simulation. 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.

3.6 Blocked Processes and Termination

If all nodes of a process are 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. Therefore, *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 in a very simple manner. The process simply waits for a message to enter and processes the message. If the message allows a node to proceed, then tl no longer blocked. If the process is still blocked, it repeats this procedure u blocked. Hence, processes other than the master are purely dependent upon st an external source.

On the other hand, the master process must be capable of initiating e case of some exception. To this end, when the master process enters a bloc sets a timer. The master then waits for a message to enter its mailbox m other process. If a message arrives before the timer expires, then the master p message and proceeds either to an unblocked state or repeats its blocked stat However, if the timer expires before a message arrives, then the master for warning the user that an exception has occurred.

Therefore, any hardware or software error that causes a process to cra tually cause the master to issue an error message (since there will be no upd from that process, eventually all other processes will have to wait for it). I itself crashes, this will be immediately apparent to the user. Hence, if the ru abnormally, the user will be warned of that fact and the master will allow cont to the user.

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

If the master becomes blocked, and as far as it knows, all nodes hav event at infinity, then it begins a test for quiescence. The master sends to a request for them to determine whether they perceive the system as quies any respond negatively, then the master treats the situation simply as bloc timer, and goes into its blocked mode. However, if all processes agree with master waits a small amount of time, and sends a second round of quiescent t In this way, if some message were in transit during the first quiescence check arrived before the second, and any changes to the system resulting can be re processes still believe that the system is quiescent, then the master sends all quiescence message, which allows the processes to end the run, reporting that could not be found.

Finally, in the case where the solution is generated at some node, th terminates normally by having the solution broadcast by that node to each p node is executed until it reaches the time at which the broadcast solution w ulated to arrive if transmitted to all nodes. When all nodes of a process h this time, the process terminates and informs the master of its completion. mote VMT processes have completed and informed the master process, the simulation is complete.

4. Debugging Tools

One of the more challenging aspects of implementing a distributed 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 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.

4.1 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. Hence, the log file is a useful tool for analysing the activities of the remote task after the task has completed.

In our implementation, each buffer process is a remote task and has 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.

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.

4.2 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 distributed 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 synchronisation 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. Instead, 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.

4.3 User Interaction

The interface between the VMT and the underlying communication mechanisms 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 master. In addition, a user can interrupt a remote VMT process and instruct it to enter a debugging mode in which the user can interact with the remote process as if it were a local process.

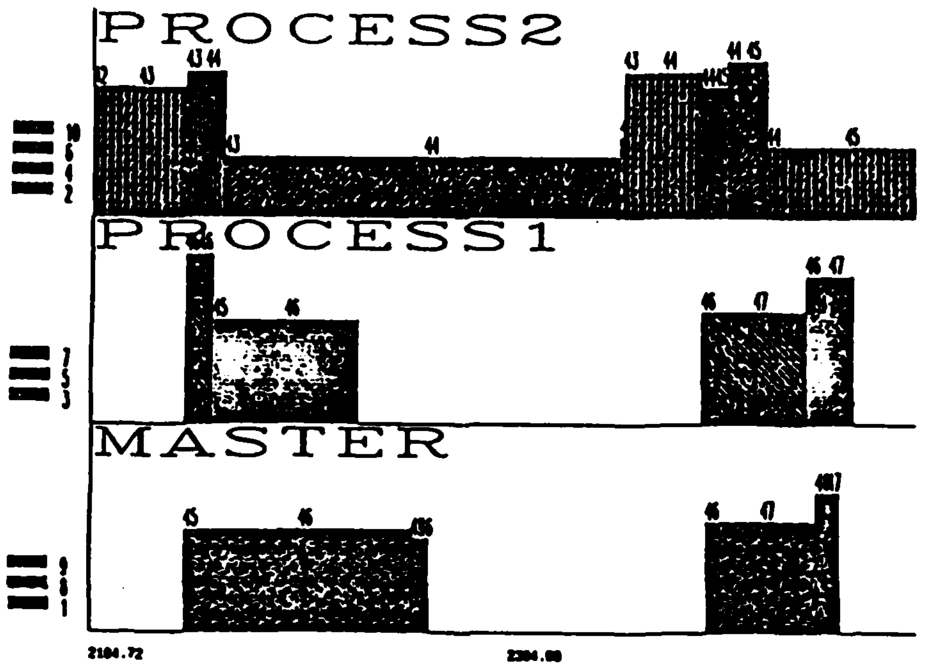
Therefore, interactive testing and debugging of the distributed simulation was achieved. Although it has been found that general debugging of the simulation is still best carried out on a single process, for debugging the problems specific to the distributed simulation, the methods above have together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

5. Experiments with the Distributed Simulator

The success of the distributed simulator must be based on two criteria. First, the simulator must fall within the constraints previously outlined; in particular, the simulation results must be independent of whether the simulator is itself distributed. Second, the distributed simulator must be judged based on the degree to which it can reduce the amount of real time needed to perform a simulation.

The implementation of the synchronisation and scheduling strategies insures that the simulated events occur in their proper order. Through the use of some simple programs, the separate outputs from each of the processes participating in a particular simulation run can be merged together, and the result is indistinguishable from the output generated by a non-distributed run. Therefore, the first criterion is met.

The real time requirements of the simulation are reduced by balancing its memory and computational needs among the machines. As previously mentioned, a major advantage of creating a distributed simulator for the VMT is that the simulated activities are already divided among the nodes. Hence, in order to balance the simulation's needs, one must assign the nodes to VMT processes so as to divide the memory and computational needs in the best possible way.



A graphical depiction of a small portion of a simulation run. The X-axis represents time (in seconds). Each block represents a node execution and the height of a block corresponds to the CPU utilization during that execution. 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.

Figure 5: Computational Time Graph.

In terms of balancing memory requirements, the memory needs of the nodes are roughly equivalent since they each have similar sets of 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 minimise the amount of time processes lie idle due to having all of their nodes blocked (Figure 5). 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

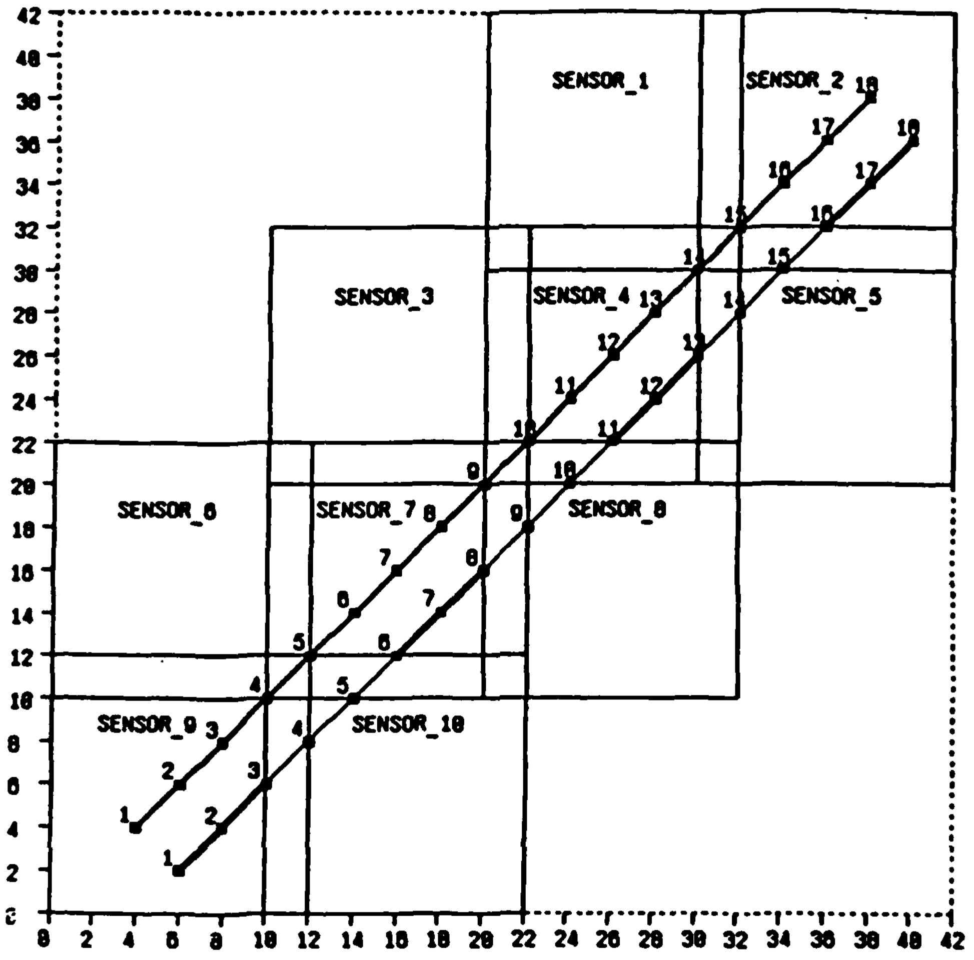


Figure 6: The Ten Sensor Configuration.

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. Therefore, concurrency of simulated events does not imply real time concurrency, although these concepts are intertwined because of the dependence of real time processing to the synchronisation of simulated time. Nevertheless, knowledge as to activities of nodes over time can suggest possible node to process assignments, and for this reason it is important to understand the characteristics of the environment being simulated.

5.1 The Simulated Environments

The experiments were based on a ten sensor configuration as portrayed in Figure 6. 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.

Two simulated environments have been based upon the ten sensor configuration. The first (E1) consists of ten nodes, the sensors being assigned to the nodes on a one-to-one basis. 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 to generate the solution track.

From such a high level perspective, 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.

The second environment (E2) consists 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 organised *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 can then 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.

Two important observations should be made about these environments. The first is that the existence of ghost data *distracts* the problem solving activities as sections of the ghost track are created and merged with sections of the true track. The second observation is that an equivalent amount of simulated time is needed to generate the solution in each of these environments. 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 focus the attention of the individual nodes on the true data earlier.

5.2 Experimental Results

In this section, we will present results from the experiments performed on environments E1 and E2. 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).

Environment E1

Experiment	Process_1		Process_2		Process_3		Real	Cpu	Cpu	Concurrency	Speed-up	
	Nodes	M	Nodes	M	Nodes	M	Time	Time	Utilisation		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	.66	.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	1.58
1.5	1,3,5,8,9	2	2,4,6,7,10	2	-	-	84	91	.59	.53	4.28	1.17
1.6	1-5	2	6-10	2	-	-	84	90	.63	.49	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

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.

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 utilisation. 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-utilisation 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 (Experiment 1.1). The bulk of the improvement can be attributed to decreasing the memory requirements per machine. 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 Experiments 1.6 and 1.8, the allocation strategy groups locally adjacent nodes together, a strategy that caused Experiment 1.4 to run slower than Experiment 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.

Finally, 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 busier 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.

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 Environment 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 provide a basis for comparison in the Environment E2 simulation (Table 2). 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

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	.96	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	.84	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	3.96	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	58	.97	.66	3.76	1.77
2.12	1-5	4	6-11	4	-	-	42	58	.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 1: Experiment Set 2.

reduction in thrashing as well as to concurrency. Note that the concurrency 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.

Overall Conclusions

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).

The simulator has been used on a number of other environments and produces similar results. When considering larger node networks, it appears that the most important factor is providing sufficient memory either by having more machines or by increasing the amount of memory per machine. The former case is preferable in that more concurrency will also result, although at a cost of increased overhead as more messages are required to synchronize the processes. The maximum size that a machine network can attain before the cost of the overhead will outweigh the gains in concurrency are not known, but the indications are that it will have to be significantly larger than any now available to us.

6. Summary and Future Research

In this paper we have presented the implications of distributing a simulator of a loosely-coupled distributed problem solving network on a network of VAX 11/750s with DECNet/ETHERNET-II communication. This simulator requires a large amount of computation and memory to simulate each node. We found a straightforward approach to the decomposition of the simulator without optimized message communication among the simulator processes to be quite effective for this type of simulation. We presented data showing that the additional CPU power and the combined physical memory available in the network contribute to significant real-time speedup. Experience with a two and three machine network indicates that where there was no memory contention in a single process simulator, we obtain a speedup proportional to the number of machines. Where there was memory contention in the single process simulator, the speed up is much more dramatic due to the increased physical memory available in the network. We also detailed the capabilities that had to be added to the conventional network communication structure to implement, debug, and interact with the distributed simulator.

In the future we plan to extend these results to networks containing as many as ten VAX 11/750s. As we obtain alternative clustering hardware in which all disk I/O occurs on a high speed bus and file server, movement of processes between machines will be sufficiently fast to justify dynamic load balancing. We also hope to develop analysis tools to determine from the simulation input data a static node-to-process allocation that minimizes the amount of real time required for the simulation.

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.

CORK82
Daniel D. Corkill, Victor R. Lesser, and Eva Hudlicka.
Unifying data-directed and goal-directed control: An example and experiments.
In *Proceedings of the Second National Conference on Artificial Intelligence*, pages 143–147, August 1982.

DURF84
Edmund H. Durfee.
Masters Thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Massachusetts. In preparation.

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.

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.

PEAC80
J. K. Peacock, E. G. Manning, and J. W. Wong.
Synchronisation of distributed simulation using broadcast algorithms.
Computer Networks 4(1):3–10, February 1980.

u
t
n
s