May 1990 NASA NCA-2-385

Center for Reliable and High Performance Computing
Coordinated Science Laboratory
College of Engineering

P-25

FINAL REPORT FOR NASA GRANT NCA-2-385

DSIM: A Distributed Simulator

KUMAR K. GOSWAMI AND RAVISHANKAR K. IYER

University of Illinois at Urbana-Champaign

DSIM: A Distributed Simulator

Final Report for NASA Grant NCA-2-385

Kumar K. Goswami and Ravishankar K. Iyer [1]
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, Illinois 61801 USA

August 7, 1990

[1] Funds for the support of this study have been allocated by the NASA-Ames Research Center, Moffett Field, California, under Interchange No. NASA AMES NCA 2-385.

Abstract

Discrete event-driven simulation makes it possible to model a computer system in detail. However, such simulation models can require a significant time to execute. This is especially true when modeling large parallel or distributed systems containing many processors and a complex communication network. One solution is to distribute the simulation over several processors. If enough parallelism is achieved, large simulation models can be efficiently executed. This study proposes a distributed simulator called DSIM which can run on various architectures. A simulated test environment is used to verify and characterize the performance of DSIM. The results of the experiments indicate that speedup is application-dependent and, in DSIM's case, is also dependent on how the simulation model is distributed among the processors. Furthermore, the experiments reveal that the communication overhead of ethernet-based distributed systems makes it difficult to achieve reasonable speedup unless the simulation model is computation bound.

Index Terms–Event-driven simulation, distributed simulation, parallelism, virtual time, performance, communication overhead.

1. Introduction

Two common techniques for modeling systems are queueing theory and discrete event simulation. Queueing theory can be effective for simple analysis but it is limited because of the restrictive assumptions that have to be made to keep the model tractable. Discrete event simulation allows one to model a system more accurately and in greater detail. However, a highly detailed simulation model can require significant computation time. This is especially true if the system being simulated is a large parallel or distributed system where each processor and the communication network need to be modeled accurately. One solution is to distribute the simulation over several computers. If enough parallelism can be achieved, linear speedup is possible and large simulation models can be executed in less time.

The aim of our project is to investigate a general purpose distributed simulator with the following features: it can run on various architectures; the user can develop simulations without regard to the underlying architecture; and the user can write simulation programs without making special allowances because the program is run on a distributed simulator. To test our ideas and develop a synchronization algorithm we developed a simulation environment in which we simulated DSIM, our prototype distributed simulator. The simulation environment made it possible to test DSIM on a multiprocessor architecture and a distributed architecture which is similar to the Sun system.

The focus of this report is on the details of the synchronization mechanism and how it is affected by the two different architectures. The results of our experiments suggest that speedup is application dependent. In the case of DSIM it is also dependent on how the various processes are assigned among the distributed DSIM elements. Furthermore, the experiments reveal that the communication overhead of ethernet-based distributed systems make it difficult to attain reasonable speedup.

The following section briefly discusses the two most common distributed simulation approaches and their advantages and drawbacks. Section 3 describes the DSIM synchronization mechanism in detail. Section 4 elaborates on the test-bed environment and the experiments that were undertaken. The results of the experiments are discussed in section 5.

2. Background

A physical system consisting of physical processes that operate autonomously except to interact with other processes in the system can be simulated by logical processes that mimic the actions of their corresponding physical processes. The interactions between the physical processes can be modeled by messages sent between the corresponding logical processes.

To simulate a physical system, an event-driven simulator requires a variable *clock* that records the time up to which the physical system has been simulated and a structure called the *event list.* The event list maintains the set of all messages and the future time at which they are to be transmitted. At each step, the simulator selects the message with the smallest time stamp, deletes it from the event list and transmits it to its destination. This simulates the corresponding interaction between two physical processes in the system. Each time a message is transmitted it may result in other messages being added or deleted from the event list. It also causes the clock to be advanced to the time at which the message was sent.

The inherently sequential nature of the event list makes it very difficult to distribute the simulation across several machines. At each cycle of the simulation only one item is removed from the list, its effect is simulated and the list is updated. In order to distribute a discrete event simulator across several machines, the event list needs to be discarded or a method has to be developed to distribute segments of the list. This will allow many machines to simultaneously process the events in the event list. The difficulty lies in how to synchronize the machines with

each other to execute the events in the correct order.

For example, if logical process 2 *(lp_2)*, cannot send a message until it receives a message from logical process 1 *(lp_1)*, and if different machines are simulating the two logical processes, then the machines have to communicate with one another to determine that *lp_2* must receive a message before sending one. If, however, *lp_2's* message transmission is not dependent on *lp_1's* message transmission times, then they can be transmitted in any order or transmitted simultaneously. Herein lies the key to an effective synchronizing scheme. The idea is to find a way to synchronize cheaply and execute the independent events simultaneously.

There are two well known approaches to distributed simulation: the optimistic approach and the conservative approach. Time Warp, developed by Jefferson and Sowizral [Jefferson 85], is an example of an optimistic approach. The idea behind Time Warp is to execute events without synchronizing. When dependent events are executed out of sequence, a rollback scheme is used to go back in time and perform the events in the correct sequence; hence the name Time Warp. The advantage of Time Warp is that the user does not have to be concerned with synchronization or saving the state in case of rollback. All of this is done by Time Warp. The drawback is that a significant amount of state information has to be saved. The cost of recording and maintaining this state information and performing rollbacks is not clear. The Chandy-Misra [Misra 86] algorithm is an example of a conservative scheme. Here no events are executed until the correct sequence is determined. As a result a notable amount of time is spent synchronizing to determine the next event to execute. The Chandy-Misra algorithm has not been incorporated into a distributed simulation package like Time Warp. Therefore, the user intending to use it must embed the algorithm in his simulation. Another disadvantage is that the algorithm can cause deadlock and so a suitable deadlock detection or deadlock avoidance scheme needs to be run simultaneously with the simulation.

3. DSIM Simulator

The organization of DSIM and its synchronization mechanism is discussed in detail in the following two subsections. The synchronization method is a conservative approach and in that sense is similar to that used by Chandy and Misra. However, the structural setup of DSIM is different from the Chandy-Misra method. This has impacted the synchronization method and is the reason why synchronization is transparent to the user.

3.1 The Organization of DSIM

Figure 1 shows the organization of DSIM for a single communication channel based system. Each element of DSIM consists of a sequential simulator that runs a part of the user simulation model and a synchronization mechanism. The sequential simulator will be a slightly modified version of the process-based CSIM simulator [Schwetman 86]. The compatibility will make it possible to port existing CSIM simulation programs onto DSIM with only minor changes. This will require that a heuristic to distribute the logical processes of the simulation program to the various DSIM elements be developed.

In the Chandy-Misra method the application simulated is depicted by a control-flow graph as shown in Figure 2. Here, the nodes are the logical processes that simulate the corresponding physical processes and the arcs between the nodes depict the interaction between the processes. In order for the synchronization to work, each logical process must be aware of the number of arcs that enter and exit its node. For this reason, synchronization is not transparent to the user writing the logical processes.

In DSIM, the nodes are not logical processes written by the user. Rather, they are DSIM elements. Furthermore, the connections among the nodes are not based on the interactions in the application; the nodes are completely connected. The logical processes written by a user are

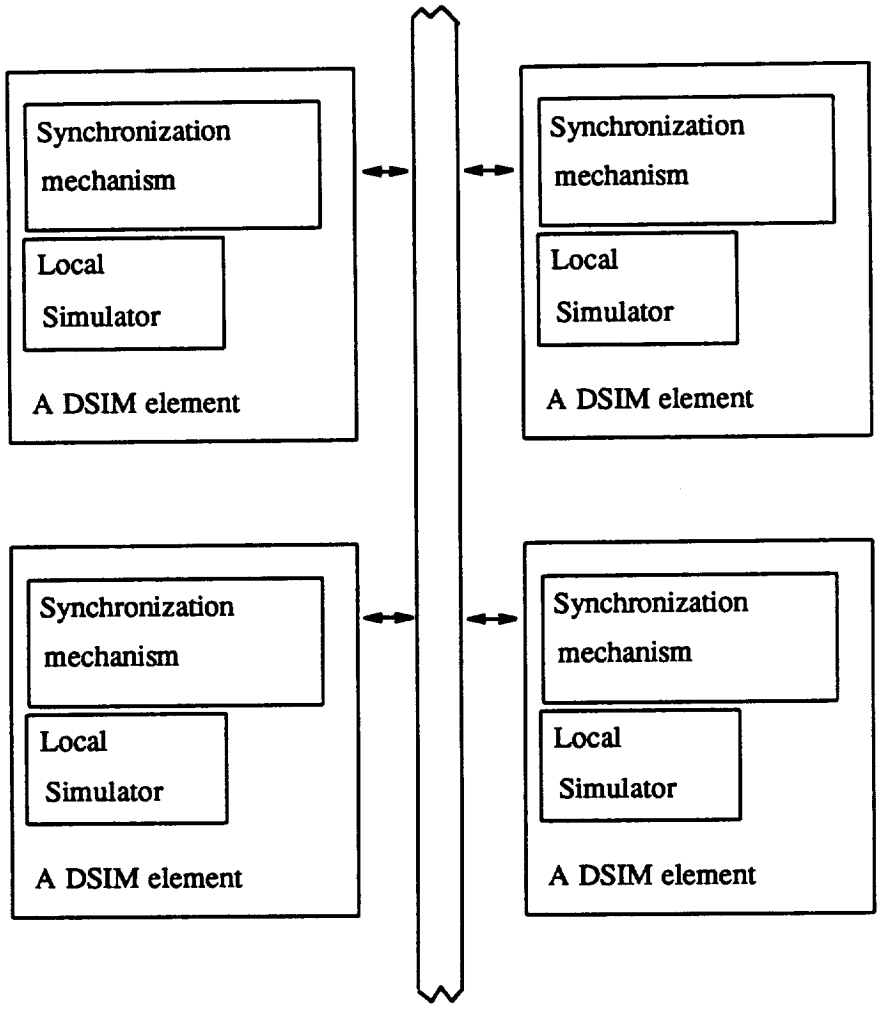

Figure 1. DSIM in a single bus-based architecture.

distributed to these DSIM elements and simulated. The disadvantage of this approach is that simple applications with linear control flow structures must still run in the framework of a completely connected graph. However, there are many advantages.

Any synchronization that is required is performed automatically and transparently by the DSIM elements. Knowing the exact control flow graph (a complete connection) and knowing that each node is a known element (a DSIM element and not a user implemented logical process) it is possible to avoid deadlocks faced by the Chandy-Misra approach and it is possible to

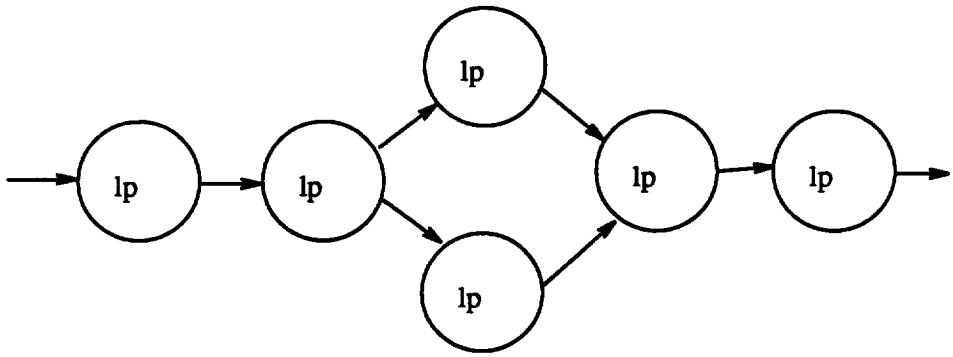

Figure 2. Control flow graph of an application.

detect deadlocks in the application. Furthermore, since each DSIM element contains a sequential simulator, inter- element simulation is fast and efficient. If the interaction among the logical processes in different DSIM elements is low, linear speedup is possible. Figure 3 shows the control flow of an application with feedback loops and nodes with several input arcs. The Chandy-Misra approach has been shown to perform poorly for such applications [Reed 88]. However, if DSIM is used, the application can be distributed in a manner that removes the feedback loop and converts the flow diagram into a linear one. With this arrangement the local simulator can simulate the otherwise complex interactions without any synchronization. Finally, distributed simulation has not been very successful because the computation time has been small relative to the time spent synchronizing. By collapsing large applications into a few DSIM elements you can increase the computation granularity and decrease the need for synchronization. This can lead to better performance.

3.2 The Synchronization Mechanism

The state of each DSIM element is defined by the following items:

1. The local virtual time (Lvt).

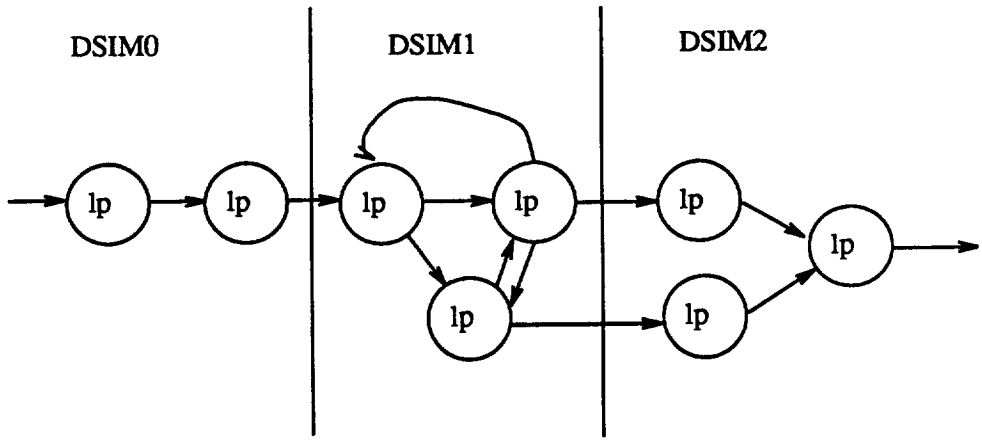

Figure 3. Example application and its distribution to DSIM elements.

2. The next anticipated local virtual time (Next_lvt).
3. The time at which the earliest message was transmitted and for which an acknowledgment has not been received (Early_msg).
4 The global virtual time (Gvt).
5 The local event list.

Lvt is the time up to which the DSIM element has simulated the physical system. The time of the next event is greater than or less than Lvt. Next_lvt is the next anticipated local virtual time. It is equal to the time stamp of the next anticipated event on the event list. The next event is 'anticipated' because it is always possible to receive a message from a logical process in another DSIM element that can modify the event list thereby pushing the current next event further down into the local event list.

The Early_msg variable keeps track of the earliest message that was sent by DSIM but which has not been acknowledged. Each time a DSIM element sends a message[1] it places the message's identification and the time at which it was sent in a local list. The DSIM element receiving the message places it in its event list and sends an acknowledgment back to the sender. The sending DSIM element then deletes the entry from its local list after receiving the acknowledgment. The Early_msg variable is set equal to the message in the list with the

smallest time stamp. The need for Early_msg will be made clear later. The Gvt is the time up to which it is safe to execute events in the event list without synchronizing.

The synchronization mechanism described in the next few paragraphs makes the following assumptions:

1. If message J is transmitted at time t_j then $t_j \geq t_{j-1}$ where t_{j-1} is the time at which message $J-1$ was transmitted.
2. Messages are received in the order they are sent.
3. No messages are lost. This is a fault free communication network.
4. The broadcast message used to poll the simulators is indivisible. In other words, no other message can be sent until all the simulators have received the broadcast message.

All these assumptions can be readily implemented in a multiprocessor and a single communication channel based system.

The basic algorithm followed by each DSIM element is to execute all the events in its event list whose time stamp is less than or equal to Gvt. Once done with these events, the DSIM element polls all the other elements in order to compute a new Gvt value. It does this by broadcasting a 'State Inquire' message. DSIM elements receiving this message respond by forwarding their current state to the requesting element. When all the responses have been received by the requesting element, it uses the state information to compute the new Gvt as follows:

1. Compute the minimum Lvt among all the other simulators. If this value is greater than the time stamp of the first event on the local event list set Gvt equal to the minimum Lvt. Otherwise go to step 2.
2. Compute the minimum Next_lvt value and the minimum Early_msg value. Here, include this element's values when making the comparison.

[1]By messages we mean messages sent by a logical process in one DSIM element to a logical process in another DSIM element; that is, the messages that simulate the interactions in the physical system. System messages sent by the DSIM elements to synchronize and maintain a global time are not included.

2a. If the minimum Next_lvt is less than the minimum Early_msg and it's greater than the current Gvt, set Gvt equal to the minimum Next_lvt.

2b. Otherwise, if the minimum Early_msg value is less than the minimum Next_lvt and it's greater than the current Gvt, set Gvt equal to the minimum Early_msg.

2c. If none of the above are true, the Gvt cannot be changed.

If Gvt is now greater than the time stamp of the first event on the event list, the synchronization process is complete. The new Gvt value is broadcast to the other simulators and simulation is resumed. Otherwise, the simulator sleeps for a specified period of time, broadcasts another 'State Inquire' message and repeats the cycle. If while the simulator is sleeping, it receives a new Gvt value from another simulator or a message that alters the event list, it is awakened. Note that even when the simulator is sleeping, the synchronization mechanism is always active receiving messages and responding to requests.

The Early_msg variable, described earlier, is needed to account for messages that are in transit. The following example demonstrates the need for the Early_msg variable. Suppose that DSIM0 broadcasts a 'Status Inquire' message to DSIM1 and DSIM2. Also suppose that a logical process in DSIM1 has sent a message to a process in DSIM2 at time 11. (Whether the broadcast message sent by DSIM0 is received by DSIM1 before or after DSIM1 has sent its message to DSIM2 is irrelevant.) If the Early_msg is not included in the state information, the responses DSIM0 will receive is shown in Figure 4. With this information, DSIM0 will erroneously compute the new Gvt to be 15 because without the Early_msg field it has no way of knowing that there is a message in transit with a smaller time stamp. This message can potentially trigger events that could cause a process in DSIM0 to receive a message with a time stamp less than 15. This would be an error because DSIM0 will have already increased its Gvt to 15 and could potentially have simulated events up to time 15.

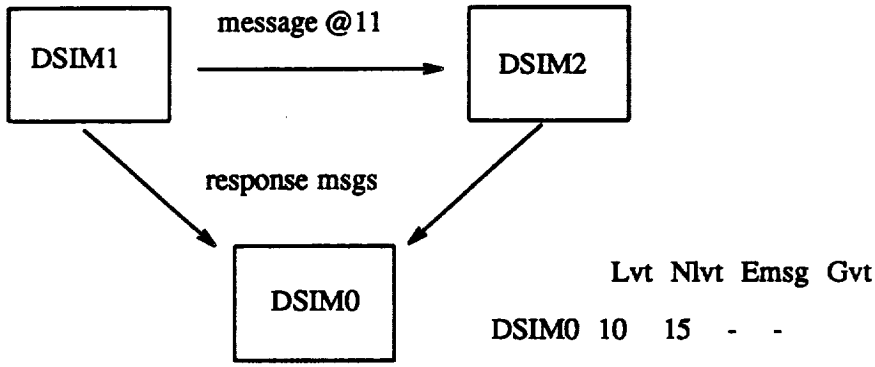

Figure 4. An example

State Information DSIM0 Receives				
Case	Sim	Lvt	Nlvt	Emsg
Before DSIM1 sent msg	DSIM0	10	15	-1
	DSIM1	10	11	-1
	DSIM2	10	15	-1
After DSIM1 sent msg	DSIM0	10	15	-1
	DSIM1	11	15	11
	DSIM2	10	15	-1
After DSIM1 rcv ACK	DSIM0	10	15	-1
	DSIM1	11	15	-1
	DSIM2	10	11	-1

The assumption that the broadcast message be indivisible is necessary for the synchronization mechanism to function correctly. For the example in the previous paragraph, now assume that the Early_msg field is used and that the broadcast operation is indivisible. There are three possible responses that DSIM0 can receive based on when it broadcasts the 'State Inquire' message. These responses are shown in the table above. To simplify the discussion, the Gvt values are not shown. In all three cases the Gvt computed by DSIM0 (using the algorithm described earlier) will be Gvt = 11. Now assume that the broadcast operation is not indivisible. The following sequence demonstrates the consequence.

1. DSIM2's state is Lvt = 10, Next_lvt = 15 and Early_msg = Null (Lvt:10 Nlvt:15 Emsg:Null).
2. DSIM0 polls DSIM2 and receives its state (Lvt:10 Nlvt:15 Emsg:Null).
3. DSIM2 receives a message from DSIM1 with a time stamp of 11. This changes DSIM2's state to (Lvt:10 Nlvt:11 Emsg:Null).
4. DSIM2 sends an acknowledgment back to DSIM1.
5. DSIM1 receives the acknowledgment and changes its state from (Lvt:11 Nlvt:15 Emsg:11) to (Lvt:11 Nlvt:15 Emsg:Null).
6. DSIM0 polls DSIM1 and receives its state (Lvt:11 Nlvt:15 Emsg:Null).
7. DSIM0's database now contains the following states (DSIM2 Lvt:10 Nlvt:15 Emsg:Null) and (DSIM1 Lvt:11 Nlvt:15 Emsg:Null). Note that neither state reflects the fact that an unexecuted event at time 11 exists.
8. If DSIM0 uses this information to compute a new Gvt it will set Gvt to 15. This is, of course, wrong.

Note that in this scenario messages were sent between DSIM1 and DSIM2 before *both* of them received the 'Status Inquire' message. This scenario demonstrates the need for the fourth assumption. It also reveals why piggybacked state information cannot be used to facilitate the synchronization process. Early in the development stage we felt that we could piggyback state information on top of the messages that were being transmitted between the DSIM elements. Then when a DSIM element needed to compute a new Gvt value it could do so based on the information it received through piggybacking. This would save the time needed to poll all the DSIM elements and receive their responses. Unfortunately, the state definitions received via piggybacking violates the fourth assumption and can lead to erroneous calculations of Gvt.

4. Experimental Test-Bed

A test-bed environment was built to study the characteristics of the DSIM synchronization mechanism on two architectures: the multiprocessor and the single communication channel based distributed system. The test-bed consists of the simulated architectures and a simple version of DSIM. This version fully implements the synchronization algorithm described in section 3. However, the local simulator in each DSIM element is not process-based but is rather an

event-driven simulator. Furthermore, the distribution of the logical processes to the DSIM elements is, at this stage, performed manually. These short cuts were taken so that we could study the synchronization mechanism before focusing on other aspects of the system.

The simulated architectures and the event-driven simulator in each DSIM element were written in C++ and were run on CSIM++. CSIM++ is an object-oriented, process-based simulator. It is essentially a C++ version of CSIM [Schwetman 86] in which the facilities, tables and mailboxes are objects. CSIM is a process-based simulation language written in C. The main advantage of CSIM++ is that it allows the user to write the application in C++. In cases where the application is large and complex the development phase and especially the debugging is more manageable if the application is written in C++.

In the simulated multiprocessor model, we assumed the time to access a lock was 70 microseconds and the time to write a byte into shared memory was 5 microseconds. These are slightly larger than the figures extracted from a Sequent Balance manual. For the simulated single communication channel distributed system we assumed the overhead of sending a message was one millisecond. The cost of transferring a byte, including the time to read and write it, was assumed to be ten microseconds.

A simple tandem queue application was written to test DSIM. A tandem queue is like a pipeline in which each process does a bit of processing and forwards the item to the next process. For the Chandy-Misra approach, where the distributed simulation algorithm is embedded in the application, the knowledge of the control flow can be used to simplify the synchronization scheme and achieve better speedup. In fact, the Chandy-Misra approach achieves its best speedup figures when simulating tandem queues [Reed 88]. However, in our approach the elements of DSIM are oblivious to the type of application being run and are not privy to control flow information. All applications are run in a completely connected graph framework. The

control flow of the application is not as relevant to the speedup achieved by DSIM. What is relevant and what has a significant bearing on the speedup obtained is the manner in which the logical processes of the simulation program are distributed among the DSIM elements. To test this we used two different configurations of the tandem queue application. Given N processors and M DSIM elements, where N is a multiple of M, the first N/M processors were assigned to DSIM0, the second to DSIM1 and so on. This is configuration one. For the second configuration, the processors were dealt out to the DSIM elements as a deck of cards is dealt to a set of players. Both configurations of the tandem queue application were executed to determine the effect they had on the speedup achieved.

Another factor affecting the performance of the distributed simulator is the computation granularity versus the amount of synchronization. If most of the events simulated require a lot of computation time relative to the synchronization time, distributing the simulation will result in increased speedup. To test the effect of computation granularity on the speedup obtained, the time each server in the tandem queue application spent processing each message was varied.

Tandem queue sizes with 8, 16, 32 and 48 servers were used in the experiments. Each server processed 50 messages. The simulations were executed with 1, 4 and 8 DSIM elements. The speedup figures reported in section 5 are based on the time it took DSIM with one element to complete the simulation. A one-element DSIM simulator takes about the same amount of time as the sequential CSIM simulator because it does not incur any synchronization overhead.

5. Experiment Results

Configuration one of the tandem queue application was executed on DSIM with 4 elements with a simulated multiprocessor architecture. Figure 5 shows the speedup obtained. The graph shows two trends. First, the speedup obtained increases as the number of servers in the

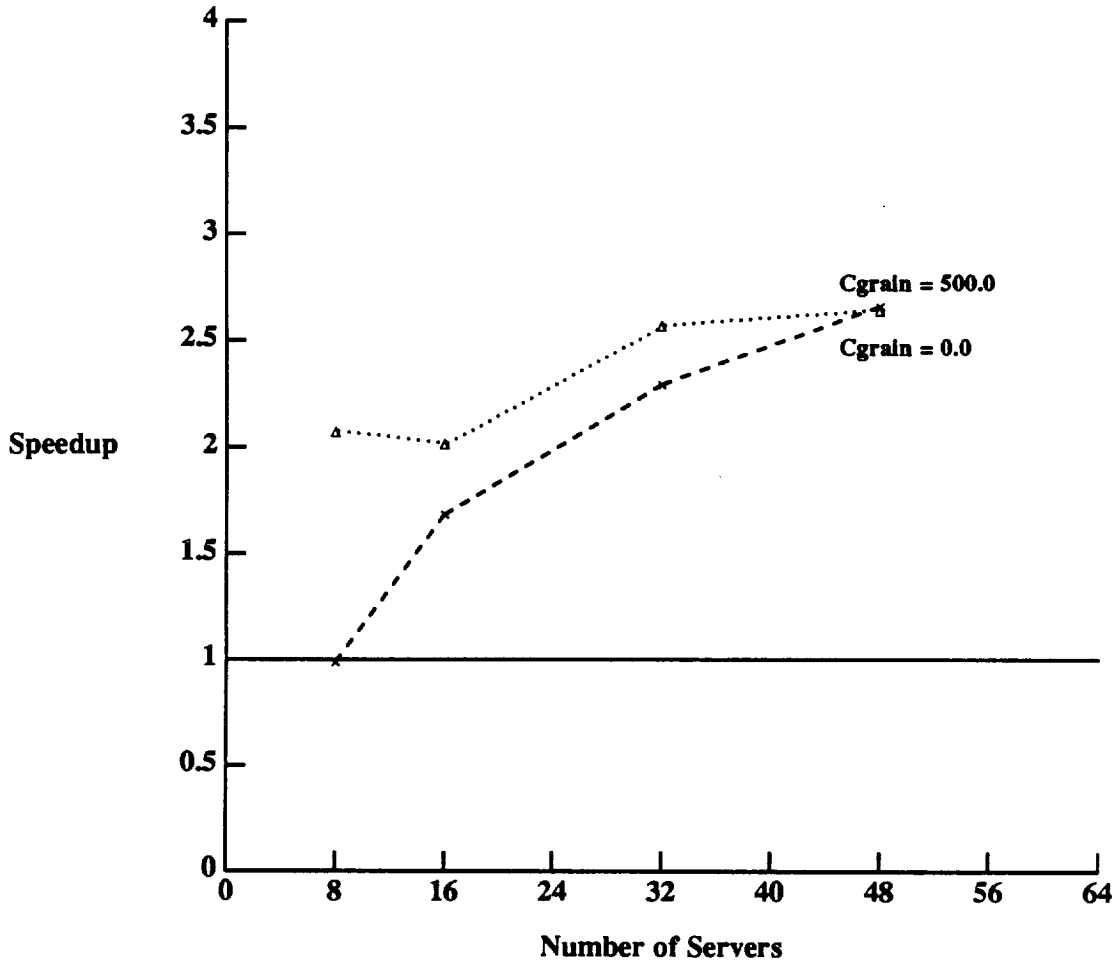

Figure 5. Configuration 1 run on a four- element DSIM (Multiprocessor).

tandem queue application is increased. The reason is that as the number of servers increases, the event list in the sequential simulator increases. Since a simple O(n) sorted queue is used to implement the event list the time to update the event list increases substantially as the number of servers is increased from 8 to 48. Meanwhile, in the distributed simulator each DSIM element is only responsible for one fourth of the servers and so their event lists are not as large. Furthermore, having more servers increases the amount of processing and extends the time between successive synchronizations.

The other trend depicted in the figure is that the speedup improves as the computation granularity is increased. Configuration one was executed with computation granularity of 0 and 500 microseconds. Increasing the granularity magnifies the load placed on the sequential simulator but in the distributed simulator it extends the time gap between synchronizations.

Figure 6 shows the results of a similar experiment conducted with an eight element DSIM simulator. Again more speedup is obtained as the amount of work to be done (the number of servers and the computation performed for each message) is increased.

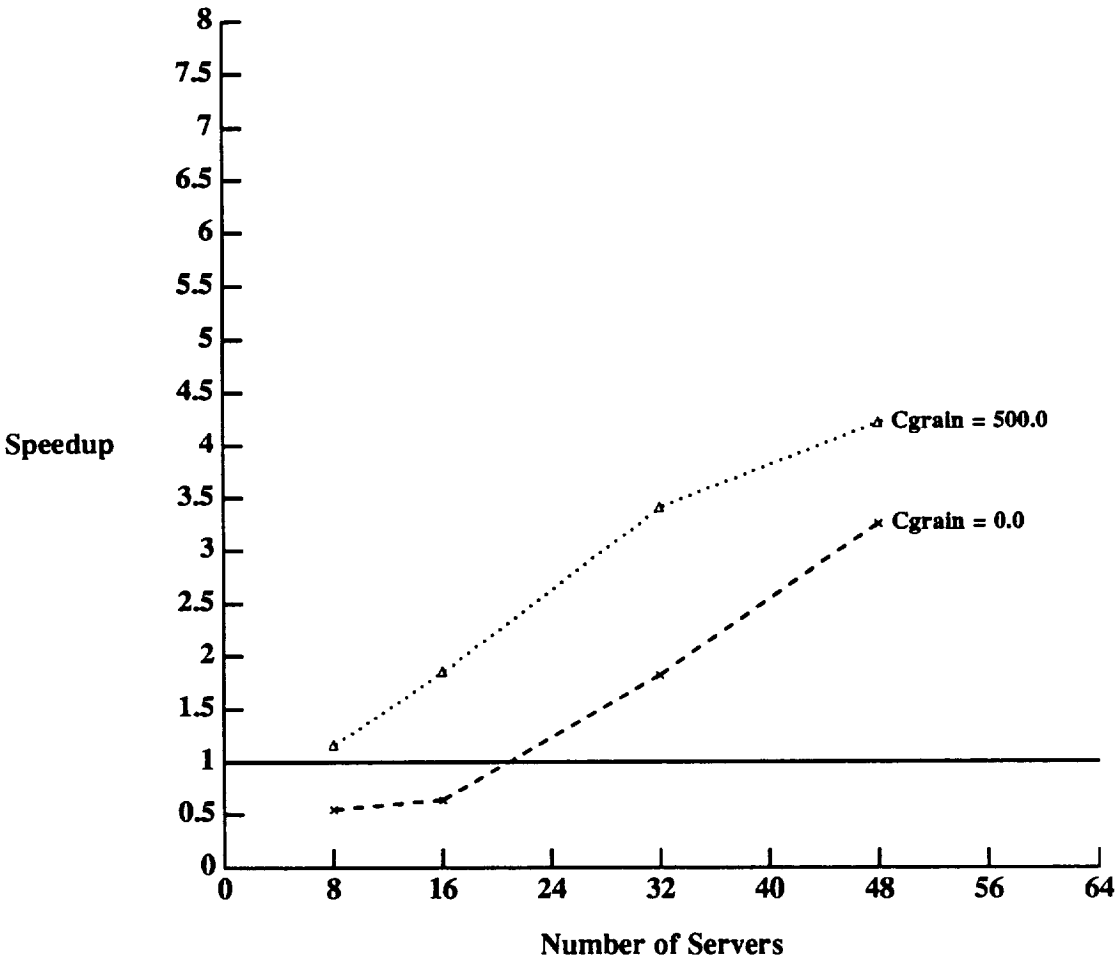

Figure 6. Configuration 1 run on an eight-element DSIM (multiprocessor).

Figure 7 shows the percent of the maximum speedup the four-element and the eight-element simulators achieved with a 500 microsecond computation granularity. For this particular application size and granularity the four-element simulator is more efficient. However, the eight-element simulator system achieves the larger absolute speedup.

We ran configuration two on DSIM with four elements and found that in all cases it performed worse than the sequential simulator (Fig. 8). Configuration two is a worst case situation because it forces the queue servers in each DSIM element to work in a lock step fashion. As a result there is very little simultaneous processing and most of the time is spent synchronizing.

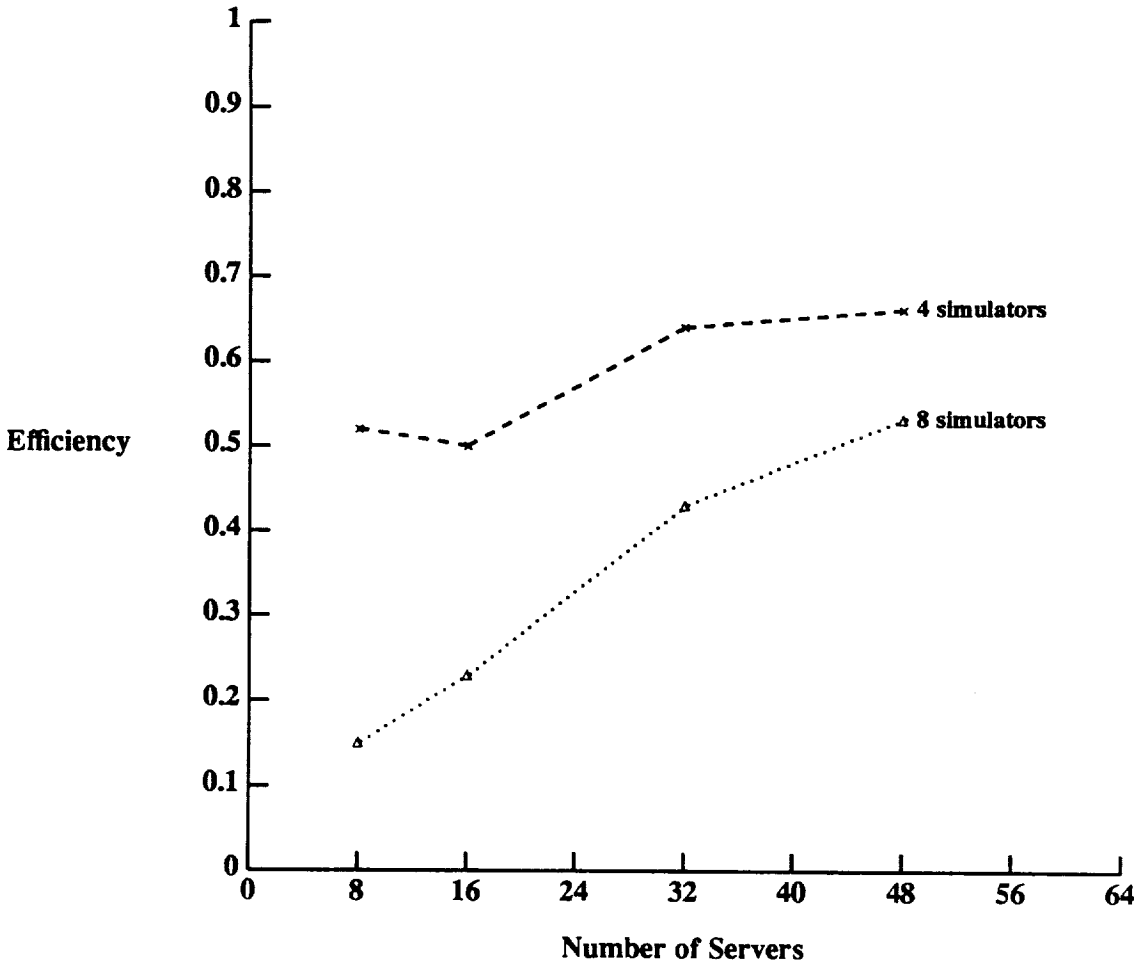

Figure 7. Percent of Speedup per DSIM element.

With this configuration each element of DSIM spent approximately 64% of its time synchronizing as opposed to 12% for configuration one. This demonstrates that judicious distribution of the logical processes of a simulation is crucial to the performance achieved.

Finally, we executed configuration one with a four-element DSIM simulator running on the simulated single communication channel system. Figure 9 is a graph of the 'speedup' obtained for different computation granularities. The cost of synchronizing on an ethernet type communication channel is at least 10 orders of magnitude greater than when a shared memory multiprocessor is used. As a result, little or no speedup is possible unless the granularity of the

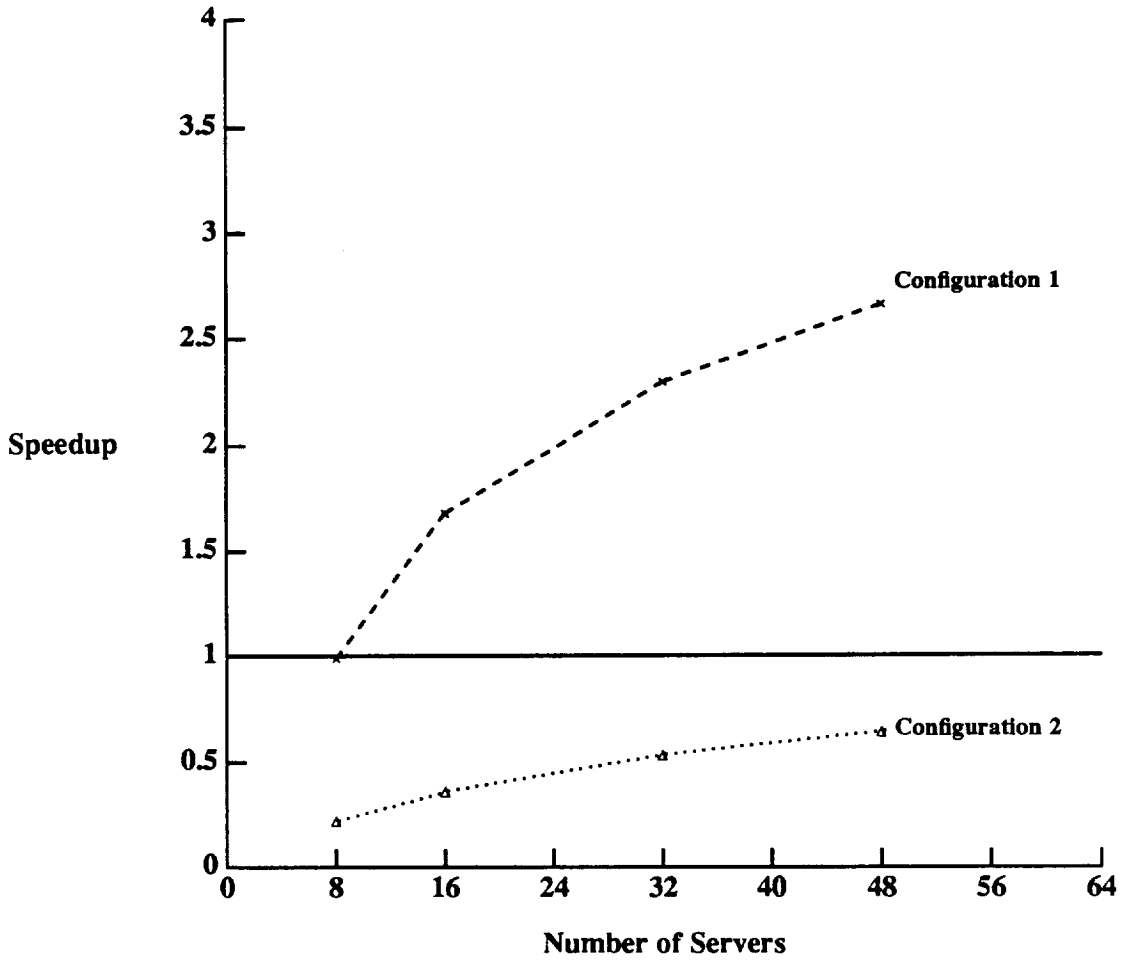

Figure 8. Configuration 1 versus Configuration 2.

computation per event is very large.

6. Conclusion

We developed a test-bed environment to determine the characteristics of our distributed simulator, DSIM. The results of the experiments suggest that the performance of DSIM is linked to: 1) the type of application being run; 2) its computation granularity; 3) how it's distributed among the DSIM elements; and 4) what type of a system DSIM is run on.

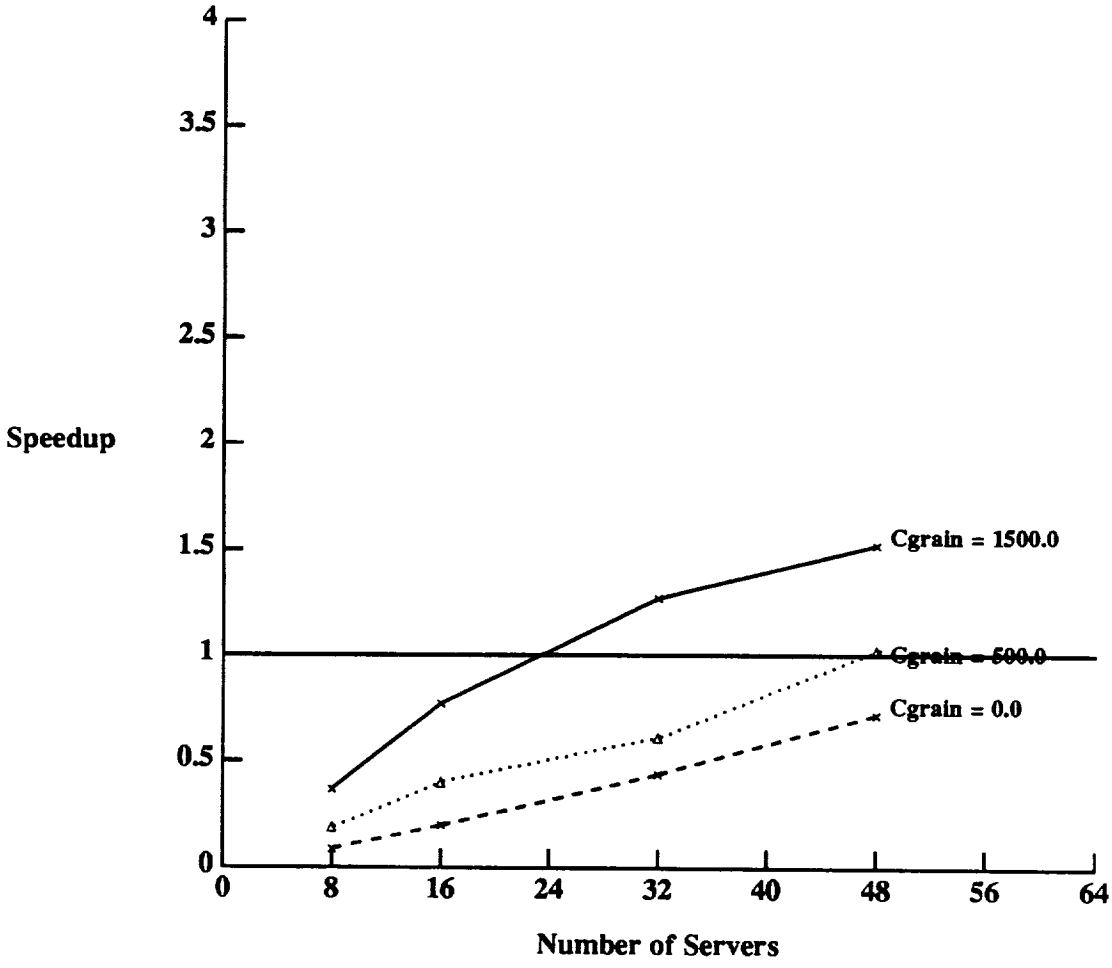

Figure 9. Configuration 1 run on a four-element DSIM (single-bus).

Not all applications are suitable for distributed simulation. Simulations that require significant computation time have something to gain. Other applications may actually take longer to complete on a distributed simulator. One major advantage of DSIM is that an application can be broken up and distributed over several DSIM elements. Each element uses a sequential simulator to execute the logical processes assigned to it and it synchronizes only if an external event will affect the local event list. This circumstance increases the computation granularity and extends the time gap between synchronizations. Furthermore, it can reduce a complex control flow graph of the application into a simpler graph that is more amenable to distributed simulation.

The experiments show that a shared memory multiprocessor is much more suitable for distributed simulation than a single communication channel-based distributed system because the cost of sending and receiving messages in the distributed system is an order of magnitude more. We do not recommend using distributed systems, like the Sun system, for distributed simulation unless the computation to synchronization ratio is very large.

For our future research we plan to extend our study and look at more realistic applications and see how effective a judicious distribution is at achieving higher performance. Work by the authors and others have shown that load-balancing centralized heuristics perform better in small- to medium-sized systems [Goswami 89]. We plan to develop a centralized synchronization mechanism. Here, only one DSIM element will be responsible for computing Gvt. The centralized approach may help reduce the message traffic and the time each element spends synchronizing. We also plan to compare the effectiveness of synchronizing periodically as opposed to demand-driven synchronization.

7. Acknowledgments

The authors would like to thank John Peterson and Jerry Yan for their support and their help. A special thanks goes to Robert Dimpsey, In-Hwan Lee and Linda Lin for many useful discussions and a critique of this report. This work was performed under a NASA consortium agreement grant, NCA-2-385.

8. References

[Bezivin 83]
J. Bezivin and H. Imbert, "Adapting a Simulation Language to a Distributed Environment," *Proceedings of the 3rd International Conference on Distributed Computing Systems*. IEEE, New York, pp. 596-603.

[Christopher 83]
T. Christopher, M. Evens, R. R. Gargeya, and T. Leonhardt, "Structure of a Distributed Simulation System," *Proceedings of the 3rd International Conference on Distributed Computing* Systems." IEEE, New York, pp. 584-589.

[Devarakonda 89]
M. Devarakonda and R. K. Iyer, "Predictability of Process Resource Usage: A Measurement-Based Study of UNIX," *IEEE Trans. on Software Engineering*, Vol. 15, No. 12, December 1989.

[Goswami 89]
K. Goswami, R. Iyer and M. Devarakonda, "Load Sharing Based on Task Resource Prediction," *Proc. 22nd Hawaii International Conf. on System Sciences*, Kona, Hawaii, January 1989.

[Jefferson 85]
D. Jefferson and H. Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism," *Proceedings of the SCS Distributed Simulation Conference*, San Diego, January, 1985.

[Jefferson 1985]
D. R. Jefferson, "Virtual Time," *ACM Trans. Programming Language System*, Vol. 7, No. 3, pp. 404-425, July, 1985.

[Kaubisch 76]
W. H. Kaubisch, R. H. Perrott and C. A. R. Hoare, "Quasiparallel Programming," *Software-Practice and Experience*, Vol. 6, pp. 341-356, 1976.

[Misra 86]
Jayadev Misra, "Distributed Discrete-Event Simulation," *Computing Surveys*, Vol. 18, No. 1, March 1986.

[Peacock 79]
J. K. Peacock, J.W. Wong and E. G. Manning, "Distributed Simulation Using a Network of Processors," *Computer Networks*, Vol. 3, No. 1, pp. 44-56, Feb. 1979.

[Reed 88]
D. Reed, A. Malony and B. McCredie, "Parallel Discrete Event Simulation Using Shared Memory," *Trans. on Software Engineering*, Vol. 14, No. 4, April, 1988.

[Reynolds 82]
P. Reynolds, "A Shared Resource Algorithm for Distributed Simulation," *Proceedings of the 9th International Symposium on Computer Architecture*, IEEE, New York, pp. 259-266, 1982.

[Schwetman 86]
H. Schwetman, "CSIM: A C-BASED, Process-Oriented Simulation Language," *Proceedings Winter Simulation Conference*, 1986.