Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

USING A SHOT CLOCK TO DESIGN AN EFFICIENT PARALLEL DISTRIBUTED SIMULATION

John T. Douglass and Brian A. Malloy

Department of Computer Science
Clemson University
Clemson, SC 29634

ABSTRACT

Parallel discrete event simulation represents an important area of research because large simulations are prohibitively expensive to execute on a sequential machine. Though simulation programs seem to have substantial amounts of parallelism, workable techniques to efficiently parallelize these programs have been elusive. In this paper, we show that by increasing the granularity of computation that each processor performs, the degradation in performance due to expensive communication can be mitigated. We simulate a network traffic flow problem using PVM to construct our parallel system. To further reduce the cost of communication, we use a novel windowing technique called a shot clock. The initial value of a shot clock is a lookahead value and expiration of a shot clock triggers actions which dictate the granularity of the messages passed in the system. The finest grained message is a single car and the coarsest grained message is a queue of cars. We achieve a speed up of 6 using PVM on a network of 16 workstations. Our experiments show that we can achieve better speed up using a system with faster communication.

1 INTRODUCTION

Parallel discrete event simulation (PDES) refers to the execution of a single simulation program on a parallel computer. PDES is an important area of research because large simulations are prohibitively expensive on a sequential machine (Righter 1989). Though simulation programs appear to have substantial amounts of parallelism, workable techniques to efficiently parallelize simulation programs have been elusive. The main bottleneck in the parallel simulation is the need for frequent communication among the processors due to the maintenance of global system time and the data dependencies that exist among the various events in the system.

In conservative simulations, a processor does not execute an event e at simulation time t_e until all messages with time stamp less than t_e have been processed. This sequencing of events is known as the *local causality constraint* and adherence to this constraint guarantees that the execution of event e is correct. Since the sequencing constraints that dictate the relative order in which events must be executed are highly data dependent, successful conservative algorithms must take advantage of *lookahead*, the ability to predict what will or will not happen in the simulated future.

The problem with the conservative approach is that without a large lookahead value, very little parallelism in the simulation program can be exploited. The degree of lookahead becomes crucial if the parallel computer exacts a high cost for communication. A software package that allows the user to construct a parallel machine using a network of workstations is *Parallel Virtual Machine* (PVM). PVM provides software support for a message passing system composed of a network of heterogeneous workstations. The advantages of PVM are that it is readily available, easy to use and allows the user to construct a parallel machine from existing sequential hardware. The disadvantage of PVM is that since communication between the processors of the virtual machine occurs across the local area network it is expensive. Therefore, unless the granularity of parallelism is coarse, any gain achieved from executing the program in parallel may be completely eroded.

In this paper, we present the design and implementation of a parallel distributed simulation for a traffic flow problem. The parallel computer is constructed using PVM on a network of Sun workstations. We investigate the effects of increasing the granularity of computation on each processor by assigning increasing number of traffic lights per processor. Also, since communication is expensive with PVM, we investigate the effects of reducing the frequency of communication by increasing the granularity of the messages. To increase the grain of the messages, we pack more cars in each message using a novel windowing technique that we call a *shot clock*. The shot clock is initialized to the size of the lookahead in the simulation; expiration of the shot clock triggers actions which dictate the message granularity, where the finest grain is a single car in the traffic flow problem and the coarsest grain is a queue of cars.

Our results show that in a system where communication cost is high, such as a virtual machine using the local area network for communication, the effects of this high cost can be mitigated by increasing the amount of computation that each processor performs. We show further that increasing the granularity of messages and reducing the frequency of communication can have a positive influence on the speed up achieved in the parallel system. We were able to achieve a speed up of 6 on a parallel system of 16 workstations connected using PVM. This speed up is a lower bound on the potential speed up using our shot clock technique since other workstations were active during the experiments and since we recorded wall time rather than processor time. Our experiments indicate further that, using our notion of a shot clock, we can achieve better speed up using a system with faster communication.

The next section of this paper presents background and reviews work related to conservative parallel distributed simulation. Section 3 presents our approach using a shot clock and section 4 presents the PVM implementation. Section 5 illustrates our results while section 6 draws conclusions and presents ideas for future work.

2 BACKGROUND

The protocols used to design and implement parallel simulation programs fall broadly into two categories: conservative and optimistic. We begin this section with a brief overview of these two protocols. We then present the highlights of PVM, the software package that enables our construction of a parallel machine using a network of workstations.

2.1 Protocols for Parallel Simulation

The protocols currently used to parallelize a simulation program fall into two general categories: conservative and optimistic. Excellent surveys of these approaches can be found in Fujimoto (1990) and Righter (1989).

In conservative simulations, a processor does not execute an event e at simulation time t_e until all messages with time stamp less than t_e have been processed. This sequencing of events is known as the *local causality constraint* and adherence to this constraint guarantees that the execution of event e is correct. There are numerous conservative algorithms that differ primarily in the manner in which they communicate this guarantee. Some approaches execute synchronously (see Nicol 1993, Lubachevsky 1989a) while other approaches are asynchronous (e.g., Nicol 1988). In optimistic simulations, a processor may execute events in any order and violations of the local causality constraint are corrected by *rolling back* the processor to a state where the constraint holds. The optimistic approach, such as Time Warp (Jefferson 1985), has shown to produce substantial speed up due to parallelism (Madisetti and Hardaker 1992, Fujimoto 1990, Baezner, Rohs and Jones 1992, Unger et al. 1990). An excellent variation of the Time Warp approach can be found in Madisetti, Walrand, and Messerchmitt (1988).

There are disadvantages to each protocol. For the conservative approach, the local causality constraint dictates the relative order in which events must be executed. Each processor maintains a *local system time* whose value is the lowest time stamped message of any communicating processor. However, since the events in a simulation are highly data dependent, there is little opportunity to exploit parallelism because the frequency of communication needed to maintain the constraint enforces a virtual lock step execution. Successful conservative algorithms must take advantage of *lookahead*, or the ability to predict what will or will not happen in the simulated future. Thus, if a processor, p_i, knows that it will not receive a message from another processor until time t_i, then all pending messages with time stamp less than t_i can be processed.

A disadvantage of the optimistic approach is the overhead incurred due to the maintenance of all necessary information to perform a roll back. This state saving information must be maintained even if a roll back is never performed. Also, a roll back on one processor may incite a roll back on another processor, leading to cascading roll backs. An excessive number of roll backs may erode the gains accrued by parallelization of the simulation program.

2.2 PVM

PVM (Parallel Virtual Machine) , Geist et al. (1993), is a software package[1] that provides support for the

[1] available through anonymous ftp from netlib2.ornl.gov in the pvm3 directory

construction of a parallel computer using a network of workstations. PVM supports a message passing communication paradigm that can accommodate more than 25 platforms, ranging from a Cray/YMP to an 80386 personal computer running the Unix operating system. Messages may be passed between any of the machines supported; data conversions, for platforms which use different data representations, are transparent to the user.

There are two communication protocols supported by PVM, allowing the programmer to choose between *dynamic TCP sockets* or *UDP communication through the PVM daemon*. TCP refers to Transmission Control Protocol which provides connection oriented communication across a network. UDP refers to User Datagram Protocol, this protocol is a connectionless transport protocol. The default communication protocol is UDP communication. In UDP communication, user processes make PVM library calls. The PVM daemon receives this information and mediates the communication. In the dynamic TCP socket communication, the user makes the same library calls as in the UDP approach, however on the first communication between two processes, a TCP socket is established by daemons running on each of the machines. Once established, this socket is used for all subsequent communication between the two processes so that the daemons are not involved in the mediation. The initial communication using the dynamic TCP sockets is significantly more expensive than subsequent TCP communication or any UDP communication because of the overhead incurred to establish the sockets. However, subsequent TCP communication is far less expensive than UDP communication; thus, if communication occurs many times over the course of a program than TCP socket communication is significantly more efficient than UDP communication. The results of our experimentation demonstrate the savings of repeated TCP communication.

The cost of the communication in PVM, regardless of the protocol used, is high. Since PVM is running on a network of machines all contending for the use of the network, the time needed to pass a message is many times that of messages being passed in a dedicated multiprocessor. Reducing communication in programs running in PVM is therefore a crucial consideration.

3 THREE MODELS THAT USE A SHOT CLOCK

Our approach to parallelizing the traffic flow problem is influenced by PVM, the software package that we utilize to construct our parallel machine. PVM is readily available and easy to use but exacts a high cost for communication, especially if other workstations are vying for access to the connecting network. Thus, the **three factors** listed below focus on reducing communication cost and guided the design of our simulation program:

1. Increasing the amount of computation per processor will help to offset the cost of communication in a system where communication is expensive.
2. Reducing the frequency of communication will positively influence speedup.
3. Reduction of null messages will positively influence speedup.

A major goal of this work is to measure the effects of the above factors on a parallel simulation given that the target multiprocessor exacts a high cost for communication. A second goal is to obtain as much speedup as possible by parallelization of the simulation program and to investigate the scalability of the parallelization using PVM. Our approach is based on a *windowing protocol* (Nicol 1993, Chandy and Sherman 1989, Lubachevsky 1988) where scalability results have been proven for this approach (Nicol 1993, Lubachevsky 1989b, Lubachevsky, Shwartz, and Weiss 1989). In a windowing protocol, processor p_i at simulated time t_i, is guaranteed not to receive a message with timestamp less than some future time, say $t_i + f$. In this case, processor p_i is said to have a lookahead of size f. In our work, we refer to our windowing protocol as a *shot clock* where the initial value of the shot clock is the lookahead value. If current simulation time is t_i and the lookahead value is f, then the shot clock is said to *expire* at simulated time $t_i + f$. In our model, expiration of the shot clock triggers actions which address factors two and three listed above. Our model is explained in the following sections.

3.1 Overview of the Approach

In addition to **establishing** the three factors listed previously, this work investigates the **degree of influence** the above factors might have on the ability of a parallel simulation to achieve good speed up. For factor one, we increase the computation on each processor to mitigate the effects of expensive communication by designing and implementing a *parameterized traffic flow network*. The parameter to the network constructor is the size of the local grid. The advantage of the parameterized network is that the

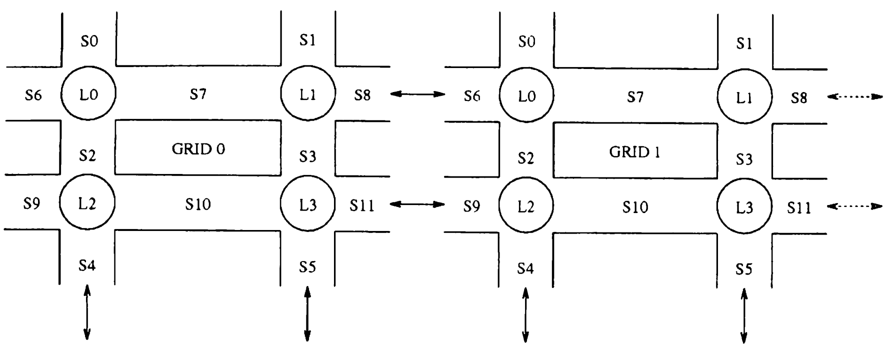

Figure 1: Two sample grids of size 2 by 2. Each grid is mapped onto a processor; in this example, GRID0 is mapped to processor p_0 and GRID1 is mapped to processor p_1.

flexibility permits easy scale to an increasing number of lights per processor, therefore an increase in the computation performed by each processor.

Figure 1 illustrates a sample traffic network composed of two 2 by 2 grids, labeled GRID0 and GRID1. Each grid is mapped onto a processor; in this example, GRID0 is mapped to processor p_0 and GRID1 is mapped to processor p_1. GRID0 contains four traffic lights labeled L0, L1, L2 and L3, and twelve street segments labeled S0 through S11. In the implementation, the lights are represented by a three valued flag indicating that the light is red, green or yellow and the street segments are represented by queues of cars. Street segments serve as both sources and sinks since traffic can flow in both directions. For example, the north end of street segment S0 serves as both a source where cars generated at random intervals are permitted to travel south on the segment, and as a sink where cars traveling north on street segment S0 are consumed when they reach the end of the segment. Street segments have a fixed size and for some segments, cars leaving the street must be passed to other processors for further travel. For example, cars traveling east on street segment S8 of GRID0 must be passed to processor p_1 when they reach the end of the street; these cars correspond to messages in the parallel model. Thus, a message must be passed to processor p_1 when a car traveling east reaches the end of street segments S8 and S11; similarly cars traveling south on street segments S4 and S5 must be passed to another processor when they reach the end of the street. If the simulation program is executed on four processors, then cars traveling east on street segments S8 and S11 are consumed when they reach the end of the street; if more than four processors are used then cars must be passed as messages when they reach the end of street segments S8 and S11.

A drawback of our parameterized network is that the regularity imposed by the parameterization precludes replication of an actual traffic network; streets in our network are the same length, have traffic in both directions, do not permit one car to pass another and there are no freeways. The number of lights per processor and the number of processors are restricted to be perfect squares. Work is underway to investigate the effects on the parallel model when these limitations are removed. The parameterized traffic flow network is used in all three models presented in this paper.

To investigate the effects of factors two and three listed previously, we designed three models of simulation. These models differ by the composition of the messages that each processor passes and by the actions that a processor perform when a shot clock expires.

3.2 Model One: send one car in each message

In the first model, a processor p_i maintains a shot clock for each boundary segment which can have inter-processor communication. For example, if GRID0 and GRID1 are assigned to processors p_0 and p_1 respectively, then processor p_0 must maintain a shot clock entry for street segments s8 and S11 since these segments neighbor processor p_1. A *street segment* is a segment of road connecting two lights; for example, street segment s7, shown in Figure 1, connects lights L0 and L1.

Algorithm *SimulateTraffic* shown in Figure 2, overviews the simulation process which runs on each processor. The input to the algorithm is the *MaxSimTime*, which is specified in a common startup file which all processes may access. The simulation continues until the local clock exceeds

```
algorithm   SimulateTraffic
input       $MaxSimTime$
output      Simulation of this local grid

begin SimulateTraffic
    $LocalTime = 0$
    while $LocalTime <= MaxSimTime$ loop
        while more null messages in the receive buffer loop
            process the null message
        end while
        while more car messages in receive buffer loop
            process the car message
        end while
        if proceed does not violate causality constraint
            then
            update lights
            process segments by
                1. gen cars for source
                2. consume cars for sink
                3. pass cars to neighboring processors
                4. move cars to adjacent segments
            increment local time
        end if
    end while
end
```

Figure 2: General algorithm executed by each processor to simulate a traffic network

MaxSimTime.

The first of the two inner **while** loops processes null messages. A null message consists of two parts: a destination segment and a time stamp. The timestamp in this case specifies how much the receiver's local clock is allowed to proceed (the window for this segment) by guaranteeing that no subsequent messages (car or null) with a smaller timestamp will be received for that segment.

The second of the inner **while** loops processes car messages. In this instance, a car message consists of a destination segment and the car structure. The processing of the car message consists of unpacking the message and inserting the car onto the proper queue of the destination segment. The shot clock entry for that segment is then updated to reflect the receipt of the car.

After the processing of the messages from other processors is complete the local causality constraint is verified. If the local clock is less than the timestamp of the last message received on all boundary segments then the simulation is allowed to proceed. If the simulation can proceed, each of the segments in the local grid is processed sequentially. For segments which are boundaries of the global system, cars are either generated based upon a user specified probability, or consumed. Cars are passed on segments which are internal to the local grid, and cars are passed between processors.

```
algorithm   PassBetween
input       $SegNo$, $ProcNum$
output      Cars/null messages passed to neighbor processor

begin PassBetween
    determine destination ($SegNo$, $ProcNum$)
    if the queue for this segment is not empty
        for each car whose queue time has expired loop
            pack destination segment
            pack car structure
            send message to proper processor
        end for
    end if
    if the shot clock has expired and no cars were sent then
        pack destination
        pack time stamp
        send null message to proper processor
    end if
end
```

Figure 3: Algorithm used in Model One to pass cars from one processor to another processor

The part of this processing which is of interest here is the passing of cars to neighboring processors. Algorithm *PassBetween*, shown in Figure 3, illustrates how cars are passed in Model One. In this model we check to see if there are cars on this segment's queue (assume in the outgoing directions). For each car whose time on this queue has expired, i.e. the cars which have traveled the segment's length, an individual message is packed and sent. If no cars were sent and the shot clock has expired then a null message is sent. The null message specifies that no further communication will come from this segment until time t, where t is the local time of the sender's grid plus time remaining until the next car will be passed.

In Model One, cars which are passing between processors are individually packed and sent in messages. Furthermore, if there are cars on the queue which are not ready to be sent, i.e. their time on the queue has not expired but the shot clock has run down, a null message is sent to the appropriate processor with a timestamp which is the arrival time of the car at the front of the queue. Our experiments revealed that Model One produced no appreciable speedup and, in many instances including the large cases, was demonstratably slower than the sequential approach!

Using PVM on the ethernet to perform message passing, communication delay is significant. The lack of performance gain for Model One can be attributed to the scheme of passing cars in individual messages while also passing what might be unnecessary null messages; this, together with the relatively high communication delay of PVM produced little or no speed up. Thus, we now present coarser grained approaches in Models Two and Three, in order to reduce the cost of communication.

3.3 Model Two: send the queue in each message

The second approach incorporates two improvements. The first improvement in Model Two is to reduce the number of null messages. The second improvement for Model Two is the packing of the car messages. Once cars arrive on the sending queue, there is no need for the car to wait until it's time on the queue has expired. The car can be passed immediately as long as the timestamp in the message is appropriately adjusted to make it appear as if the car had been sent at the proper time. Further, there is no need to send each car in an individual message. Any car which is available can be packed together into one message and sent.

3.4 Model Three: send the queue only when the corresponding shot clock expires

The average number of cars being sent per message in Model Two was 2.1. The overall number of messages being passed was still exceedingly high, translating into a pvm process which controls the communication still grabbing nearly 15% of the CPU time. In Model Three, we applied the shot clock to the car messages so that no car message is sent until the shot clock has expired. In contrast, the second approach passed cars as soon as they entered the queue on the sending side, whether or not the shot clock had expired.

4 THE IMPLEMENTATION

We implemented the three models presented in the previous section but, to provide a basis for comparison, we first implemented *Sequential Model*, a sequential version of the traffic flow problem. In the Sequential Model, we maintain an array of data structures and local grid information is kept in this structure. This implementation was then extended in several ways to obtain the three models.

In the Sequential Model, when a car travels from one street segment on a local grid to a street segment on another local grid, the car is simply removed from one queue and inserted into another queue. To implement the parallel version for the models, travel from a street segment on one local grid to a street segment on a different local grid requires a message to be passed from one processor to another. Therefore, using PVM, information about the car being passed

```
algorithm   ConstructLocalGrid
input       $LocalGridSize$
output      Initialize data structures storing local grid layout

begin ConstructLocalGrid
    $EastWestVal = LocalGridSize$
    $i = 0$
    while $i < LocalGridSize^2$ loop
        for $j = 0$ to $LocalGridSize - 1$ loop
            $val = i + j$
            LightNumber = $val$
            Segment North of Light is numbered $val$
            Segment South of Light is numbered $val+$
              $LocalGridSize$
            Segment West of Light is numbered $val+$
              $LocalGridSize^2 + EastWestVal$
            Segment East of Light is numbered $val+$
              $LocalGridSize^2 + EastWestVal + 1$
        end for
        $EastWestVal = EastWestVal + 1$
        $i = i + LocalGridSize$
    end while
end
```

Figure 4: Algorithm to construct a local grid.

was packed into a message and sent to the appropriate processor. Similarly, the receiving processor needed to unpack the information, and insert the car into the proper street segment. This code is added to the main loop of the simulation program.

When comparing the performance of the sequential and various parallel approaches physical wall time was used rather than processor time. There are two main reasons for this. First, since communication through PVM is implemented through a daemon process, the overall processor time for the simulation would need to take into account the processor time of the daemon. There is no easy and convenient way to do this. Secondly, when running an application the most important time is how long it takes to see the result; this is the wall time. The experiments in this paper were run on a network of SUN SLC workstations.

5 PERFORMANCE

Our experiments investigate the effects of the three factors listed in section 3. The first factor focuses on increasing computation to offset communication. Table 1 illustrates the results of our experiments, using Model Three, to determine the effects of increasing the computation performed by each processor to mitigate the effects of expensive communication. Recall that with Model Three, the entire queue is packed into a message and a message is sent only when the shot clock expires. The first row in the table is a heading indicating that, for both the Sequential Model (using one workstation) and for Model Three (using 16 workstations), our simulation program contained first 64 lights, then 144 until finally the last column shows the results for the program containing 576 lights. The second row of the table shows the execution time in minutes for the Sequential Model and the third row of the table shows the execution time in minutes for Model Three. The fourth row shows the speed up[2] achieved for Model Three using 16 workstations over the Sequential Model using a single workstation. For example, comparing the results in the first column, the Sequential Model required 10.1 minutes to execute the simulation program containing 16 lights, while Model Three (the parallel version) required 3.2 minutes to execute the parallel program containing 16 lights; this is a speed up of 3.19.

This fourth row of Table 1 verifies our hypothesis that for a parallel system with expensive communication between processors, increasing the amount of computation on each processor can offset the high cost for communication. In the case of the simulation program containing 16 lights, a speed up of 3.19 was achieved in the parallel version. However, in the case of the simulation program containing 576 lights, we were able to achieve a speed up of 5.7 in the parallel version. Furthermore, we were able to achieve increasingly better speed up as the simulation programs contain more lights, and therefore more computation per processor.

Tables 2, 3 and 4 illustrate a comparison of the three models described in section 3. Also, we compare the execution of Model Three using both UDP and TCP communication. Section 3 describes factors 2 and 3, which claim that reducing the frequency of communication and reducing null messages can positively influence speed up. By comparing the three models, we verify factors 2 and 3.

[2]Speed up is the ratio of the execution time for the sequential program as compared to the execution time for the parallel program.

No. of Lights	64	144	256	400	576
Seq Model	19.5	36.2	57.9	75.9	128.9
Model Three	5.8	9.9	12.9	16.3	22.6
Speed-up	3.33	3.65	4.46	4.65	5.70

Table 1: **Execution time (in minutes) for the Sequential Model (1 processor) and for Model Three (16 processors), with increasing number of lights used in the simulation program**

No. of Lights	64	144	256	400	576
Model One	38.9	53.6	76.7	89.8	108.6
Model Two	18.7	32.3	39.3	48.3	59.6
Model ThreeU	10.2	18.6	27.0	34.1	34.5
Model ThreeT	5.8	9.9	12.9	16.3	22.6

Table 2: **Comparison of execution time (in minutes) on 16 processors for each of the models; also, we compare Model Three using both UDP and TCP communication**

	Null	Msgs	cars/msg
Model One	10241	606382	1.0
Model Two	1495	290958	2.1
Model ThreeU	1462	126556	4.8
Model ThreeT	1476	126527	4.8

Table 3: **Comparison of total messages sent using each model for a simulation program containing 576 lights when executed on 16 processors**

Table 2 illustrates five different executions for each of the models using PVM on a network of 16 workstations. The first row of the table illustrates the number of lights contained in each of the simulation programs, row two illustrates the time in minutes for execution using Model One, and row three illustrates the time in minutes for execution using Model Two. Rows four and five illustrate the time in minutes for execution using Model Three with the model in row four using UDP communication and the model in row five using TCP communication.

Our results indicate that using a shot clock may produce better speed up for a multiprocessor system where communication is not as expensive as in the PVM system. Table 3 shows that the average num-

No. of Lights	64	144	256	400	576
Model One	0.50	0.67	0.75	0.86	1.19
Model Two	1.04	1.12	1.47	1.57	2.16
Model ThreeU	1.91	1.94	2.14	2.23	3.74
Model ThreeT	3.33	3.65	4.46	4.65	5.70

Table 4: **Comparison of speedup on 16 processors for the models; also, we compare Model Three using both UDP and TCP communication**

ber of messages sent for Model Three using UDP and TCP communication in PVM was the same. However, Table 4 shows that Model Three using TCP communication performed better than Model Three using UDP communication. Since TCP communication requires less overhead than UDP communication, our shot clock technique may perform better in a system that exacts a lower cost for communication.

6 CONCLUDING REMARKS

In this paper, we presented the design and implementation of a distributed parallel simulation of a traffic network problem. We constructed our parallel computer using PVM on a network of Sun workstations. Since communication cost using PVM is high, we investigated the effects of increasing the granularity of computation performed by each processor in the parallel execution of the simulation program. We also investigated the effects of increasing the granularity of messages while decreasing their frequency.

Our results show that increasing the amount of computation that each processor performs can offset the high cost of communication. We use a shot clock to control the frequency and grain of the messages passed in our parallel system. Our results show that by packing more cars in each message and decreasing the frequency of the messages, we can achieve better speed up. Our results show further that in a system with faster communication, the shot clock can achieve even better results.

AUTHOR BIOGRAPHIES

BRIAN A. MALLOY is an Assistant Professor in the department of Computer Science at Clemson University. He received an M.S. and Ph.D. from the University of Pittsburgh in 1984 and 1991 respectively. His research interests include compilation techniques for parallelism and language design techniques for simulation.

JOHN T. DOUGLASS is a graduate student in the department of Computer Science at Clemson University. He received an M.S. from Clemson University in 1994. His research interests include program analysis, and language design techniques for parallelism.

REFERENCES

Baezner, D., C. Rohs, and H. Jones. 1992. U. S. Army MODSIM on Jade's Time Warp. *Proceedings of the 1992 Winter Simulation Conference* 665–671.

Chandy, K. and R. Sherman. 1989. The Conditional Event Approach to Distributed Simulation. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation* 93–99.

Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM* 33:31–53.

Geist, A., A. Beguilin, J. Dongarra, et. al. 1993. PVM 3 User's Guide and Reference Manual. *Oak Ridge National Laboratory* ORNL/TM-12-87.

Jefferson, D. R. 1985. Virtual Time. *Transactions on Programming Languages and Systems* July: 404–425.

Lubachevsky, B. 1988. Bounded Lag Distributed Discrete Event Simulation. *Proceedings of the 1988 SCS Multiconference on Distributed Simulation* 183–191.

Lubachevsky, B. 1989a. Efficient Distributed Event-driven Simulations of Multiple Loop Networks. *Communications of the ACM* January:111–123.

Lubachevsky, B. 1989b. Scalability of the Bounded Lag Distributed Event Simulation. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation* 100–105.

Lubachevsky, B., A. Shwartz, and A. Weiss. 1989. Rollback Sometimes Works...If Filtered. *Proceedings of the 1989 Winter Simulation Conference* 630–639.

Madisetti, V., and D. Hardaker. 1992. Synchronization Mechanisms for Distributed Event-Driven Computation/. *ACM Transactions on Modeling and Computer Simulation* January: 12–51.

Madisetti, V., J. Walrand, and D. Messerschmitt. 1988. WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems. *Proceedings of the 1988 Winter Simulation Conference* 296–305.

Nicol, D. M. 1988. Parallel Discrete-event Simulation of FCFS Stochastic Queueing. *Proceedings of ACM Sigplan PPEALS 1988* 124–137.

Nicol, D. M. 1993. The Cost of Conservative Synchronization in Parallel Discrete Event Simulation. *JACM* April.

Righter, R., and J. C. Walrand. 1989. Distributed Simulation of Discrete Event Systems. *Proceedings of the IEE* 77:99–113.

Unger B. W., J. Cleary, A. Dewar, and Z. Xiao. 1990. A Multi-Lingual Optimistic Distributed Simulator. *Transactions of the Society for Computer Simulation* June: 121–152.