Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

EXPLOITING LOOKAHEAD IN SYNCHRONOUS PARALLEL SIMULATION

Gregory D. Peterson
Roger D. Chamberlain

Computer and Communications Research Center
Washington University
St. Louis, Missouri 63130, U.S.A.

ABSTRACT

The performance impact of event simultaneity and barrier synchronization on globally-clocked, discrete-event simulations is quantified using both a performance model and empirical results. We investigate the use of both conservative and optimistic lookahead to improve event simultaneity. The use of conservative lookahead is shown to be clearly beneficial, significantly improving overall simulation performance. Optimistic lookahead, however, is more problematic. Empirical results show that the overhead needed to support optimistic lookahead defeats the performance gains due to increased simultaneity.

1 INTRODUCTION

Most recent work on parallel, discrete-event simulation has concentrated on time synchronization protocols for asynchronous distributed simulation algorithms (Fujimoto 1990, Jefferson 1985, Misra 1986). However, there are classes of problems for which the conceptually simpler synchronous simulation algorithm has adequate performance, eliminating the need to use the more complex asynchronous protocols. In addition, the performance of synchronous simulations is more predictable, avoiding the sometimes erratic behavior of other techniques (Chamberlain and Franklin 1991). (Lin and Lazowska(1991) coined the term "S phenomenon" to describe the observation that speedup curves for an optimistic asynchronous algorithm often have several local minima and maxima. This observation was made over a large set of different simulation applications (Jefferson et al. 1987, Wieland et al. 1989, Ebling et al. 1989, Hontalas et al. 1989)). This paper explores the performance impact of event simultaneity and barrier synchronization on synchronous simulation algorithms and investigates the use of both conservative and optimistic lookahead to improve simulator performance.

In globally-clocked distributed simulation algorithms, each processor is constrained to process events at the same point in simulated time. Upon completion of the event processing for the current simulated time, the processors reach a barrier synchronization, where they determine the next point in simulated time at which there are events to process. This iteration (processing of events followed by barrier synchronization) is repeated throughout the course of the simulation.

The global clock algorithm has the advantage of being straightforward to understand, implement, and debug; however, the algorithm can reduce the amount of parallelism that is exploited by the simulator. This occurs because the global clock forces the serialization of events with distinct timestamps, even if there is no causal relationship between the events. Due to this constraint, processors that have completed their work for the current simulated time must remain idle, contributing to load imbalance.

This paper explores the performance impact of modifications to the global clock algorithm that diminish both the cost of barrier synchronization operations and the load imbalance effects of the strict global clock algorithm. The algorithm modifications retain the iterative nature of the original global clock algorithm, but relax the constraint that only events with timestamp equal to the current simulated time be processed during the same iteration. This class of algorithms is described as synchronous algorithms by Fujimoto (1990).

The execution of events timestamped later than the current simulated time is referred to as lookahead. Lookahead comes in two forms: conservative and optimistic. In conservative lookahead, an event is executed only if it can be shown to not violate the causality of the simulation (i.e., if it is *safe*). Previous work by Nicol (1988) and Fujimoto (1989) has explored the use of conservative lookahead in the context of locally-clocked algorithms, show-

ing clear performance benefits when lookahead is exploited. Lubachevsky (1988) uses this technique in his "bounded lag" algorithm. Optimistic lookahead is a form of speculative computation. Events are executed assuming it is safe to do so. If some later event (with an earlier timestamp) invalidates the actions associated with a previously executed event, the simulator rolls back its state information and executes the two events in the proper (time-ordered) sequence. Mehl (1991) proposed this technique in the context of a conservative asynchronous algorithm but did not report on its performance. Here, we are interested in understanding and quantifying the performance impact of both conservative and optimistic lookahead on synchronous algorithms.

The paper is organized as follows. The next section presents a model of synchronous simulation algorithm performance on an MIMD hypercube machine. The following section uses the both the model and simulation execution experiments to quantify the impact of event simultaneity on simulator performance (We describe two events as simultaneous if they are both executed in the same iteration). This includes the effects of both load imbalance and barrier synchronization operations. Section 4 describes the performance impact of using conservative lookahead, and Section 5 presents the performance impact of optimistic lookahead. The final section presents conclusions and directions for future work. Event simultaneity is the key to adequate performance with synchronous simulators. As is the case with asynchronous algorithms, conservative lookahead is quite effective at improving the simultaneity of simulation events, thereby improving simulator performance. The benefits of optimistic lookahead are less clear, with the implementation overhead defeating the performance gains in empirical measurements.

2 MODEL DEVELOPMENT

In this section, a performance model is presented to quantify the effects of various parameters. This allows us to understand the relative importance of the model components and attempt to improve performance. The model is similar to others developed for synchronous iterative computations (Dubois and Briggs 1982, Madala and Sinclair 1991). In a synchronous simulator, each processor is executing the same program with local data pertaining to a portion of the system. One can model the runtime of the simulation over B busy ticks (simulation time points when there is an event) on P processors (Peterson 1992, Peterson and Chamberlain 1993) as:

$$R_P = B[t_{cpu} + t_{comm} + t_{sync} + t_{overhead}]$$

In this equation, t_{cpu} is the time spent processing events for the most heavily loaded processor. Similarly, t_{comm} is the time spent processing event messages. The t_{sync} term corresponds to the amount of time required to perform a barrier synchronization using a complete exchange algorithm. Finally, the simulator has overhead that is incurred each iteration for calculating the next simulation time, cleaning data structures, and for initializing variables. The time spent on these activities is represented by $t_{overhead}$.

Each iteration, a barrier synchronization is performed using the complete exchange algorithm (Chamberlain and Franklin 1990). The runtime of the complete exchange algorithm is logarithmic with respect to the number of processors in the hypercube. Assuming that the time to synchronize two neighbors is t_{cx}, the time for synchronizing each iteration can be expressed as:

$$t_{sync} = t_{cx} \log_2 P$$

We next consider the role of event processing. In the serial case, the time spent processing events can be expressed as the product of the number of events and the average time spent processing an event. Using the same reasoning, the time spent processing events in the parallel case is equal to the time spent processing events divided by the number of processors, assuming that all nodes have an equal portion of the events to process. Unfortunately, this assumption is not true in general because some processors will have more events to process than others. Hence, a factor β is defined as the ratio of the number of events at the most heavily loaded processor to the number at a processor with an average load. This results in the time spent processing events for the most heavily loaded processor:

$$t_{cpu} = \frac{\beta E t_e}{P}$$

E is the mean number of events that are processed each iteration. Each event takes t_e time to be processed. P is the number of processors, and β is the imbalance factor. Note that the ideal value of β is 1 which corresponds to a perfectly balanced system. As the load imbalance worsens, β increases.

The cost for communications is the amount of time spent processing event messages and ensuring that all messages have been received through the use of acknowledgments. Each event message that is received is processed by inserting an event into the local event

queue for the appropriate simulation time. The reception of these messages is followed by sending an acknowledgment message to the processor where the event message originated. The amount of time required to process an individual message is denoted by t_{procm}. The final acknowledgment sent will then take t_{msg} to arrive at its destination. The order in which these actions are completed indicates that the time spent with communications can be expressed as:

$$t_{comm} = M t_{procm} + t_{msg}$$

Each iteration, one of the processors receives the largest number of event messages. The number of messages that arrive at this processor is denoted by M.

Table 1 shows the parameters of the model and their definitions. More information is available on the development and verification of the model (Peterson 1992, Peterson and Chamberlain 1993). The next section explores the impact of event simultaneity on simulator performance. These results are explained in terms of the runtime expressions above.

Table 1: Parameters for Performance Model

R_P	runtime with P processors
P	number of processors
B	number of busy simulation time points
β	load imbalance factor
E	number of events per iteration
M	max # event msgs at proc per iteration
t_e	time to process an event
t_{procm}	time to process an event message
t_{msg}	time to send a message
$t_{overhead}$	time to complete overhead tasks
t_{cx}	time to perform pairwise exchange
r	simulation time resolution

3 SIMULTANEITY

The performance of synchronous parallel simulation is greatly impacted by the amount of work that must be performed by each processor in an iteration. In order for a parallel simulator to have significant speedup over a serial simulator, the amount of parallelism that can be exploited must be large enough that the time spent processing events dominates the overhead (message processing time, barrier synchronization, etc.). Increasing the amount of work that can be performed in parallel will reduce the impact of the system overhead.

There are a number of factors that can impact the ratio of event processing to overhead, including event simultaneity and number of required barrier synchronizations. As the number of events to be processed each iteration increases, the event processing to overhead ratio also increases. Here, we are increasing the event simultaneity, or number of simultaneous events processed each iteration. In the model, this corresponds to an increase in E, the mean number of events per iteration. Clearly this is a function of the time synchronization algorithm. Using a strict global clock algorithm, the events at each iteration are limited to those events equal in simulated time. But if this strict interpretation of global clocking is relaxed, additional events can be simulated within the same iteration, thereby increasing the event simultaneity.

The second factor is the number of required barrier synchronizations. If the number of required synchronizations is decreased, the event processing to overhead ratio is helped. This is modeled by a decrease in the value of B. Again, the strict global clock algorithm provides little flexibility in affecting this value, but synchronous algorithms that exploit lookahead can decrease the number of iterations required to perform the same target simulation.

To quantify the performance impact of both event simultaneity and required iterations, we consider a simulation with varying time resolution. The target simulation is a queueing network simulation with 256 queues per processor (the problem size is scaled with the number of processors), each server has an exponentially distributed service discipline with mean 10 and minimum service time of 1, and the network has a bidirectional ring topology. When the simulation is executed using different time resolutions, the total number of events to process (BE) is constant. However, at a finer time resolution the events per iteration, E, goes down and the number of iterations B goes up. Both of these factors contribute to the finer time resolution having a performance penalty.

For the simulated queueing network, E can be expressed in terms of the average utilization of each server, U, the number of servers in the queueing network, Q, the mean service time at a server, S, and the time resolution factor, r (Peterson 1992).

$$E = \frac{r\,UQ}{S}$$

The queueing network above was simulated at time resolutions (r) of both 1 and 0.1 on an nCUBE/7. At $r = 1$, $E = 350$ and $B = 10,000$ when run on 16 processors. When $r = 0.1$, $E = 35$ and $B = 100,000$,

a significant decrease in events per iteration and corresponding increase in number of iterations. Figure 1 shows the resulting parallel runtimes for the two different time resolutions. Clearly the simultaneity has a significant impact on the simulation runtime, with the fine time resolution not performing nearly as well as the coarse time resolution simulation.

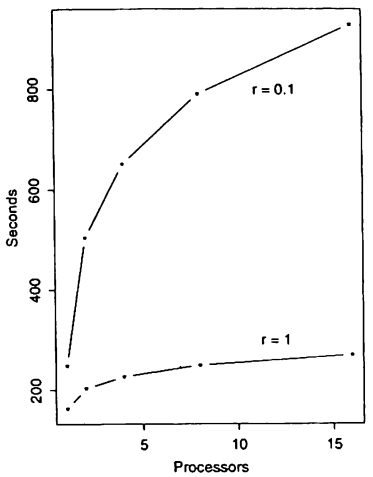

Figure 1: Simulation Runtime

The decreased performance at the fine time resolution corresponds to the increased value of B [$t_{comm} + t_{sync} + t_{overhead}$] in the performance model. Given the strong performance impact of event simultaneity (quantified by E) and number of synchronizations (quantified by B), time synchronization algorithms that can increase E and decrease B (holding the product BE constant) can potentially improve overall simulator performance. This is the goal of lookahead techniques. The next two sections explore the performance of conservative and optimistic lookahead algorithms.

4 CONSERVATIVE LOOKAHEAD

The purpose for a time synchronization algorithm is to ensure the causality of the system so that events are processed in the proper order. In the queueing networks simulated, the queues follow a First-Come-First-Served (FCFS) discipline and there are no preemptions. In addition, there is a single job class with each job requiring a non-zero service time. Because of these properties, it is possible to process certain events that occur at a later point in simulated time while ensuring that causality is maintained.

While a job is being serviced at a station, the server will remain busy until the job is completed. The time of any job completion is required to be at least the minimum service time after the current simulated time (the events correspond to the arrival of a job at a station at a given simulated time). Hence, there can be no incoming event messages with timestamp less than the minimum service time beyond the current simulated time. This implies that all events scheduled in the period from the current time until the minimum service time into the future can be processed while guaranteeing that causality is maintained. For example, if the minimum service time of the simulation is one and the time resolution is one, the processors are restricted to processing events for the current simulation time or one time step into the future. Note that jobs cannot be differentiated, so that the order of job arrivals at the same simulated time is not important.

This property of parallel simulation (the ability to process future events while maintaining causality) is referred to in the literature as conservative lookahead (Fujimoto 1990). Nicol (1988) has described a number of methods for determining lookahead in queueing network simulations. These methods include minimum service times and precomputed service times, among others. Fujimoto (1989) has explored the performance impact of conservative lookahead on locally-clocked time synchronization algorithms, characterizing the relationship in terms of a "lookahead ratio." We are interested in characterizing the performance improvement due to conservative lookahead in synchronous simulation. Lubachevsky (1988) described his "bounded lag" algorithm for synchronous simulation, this is essentially a form of conservative lookahead.

Given the potential to process events in the future, it is straightforward to exploit additional parallelism by processing such events. Conservative lookahead results in the ability to process more events each iteration (increasing E), thus potentially reducing the number of barrier synchronizations that must be performed (decreasing B). Note that the total number of events to be processed will not change, but more can be processed each iteration. This increases the event processing to overhead ratio, resulting in improved performance.

The amount of lookahead to be exploited places bounds on the performance improvement to be expected. With a lookahead of L, the number of synchronizations can be decreased by a factor of at most $(L + 1)$. For example, when simulating a queueing network with minimum service time and time resolution of 1, the lookahead is 1, implying the maximum decrease in the number of iterations is a factor of two. This occurs when all the events for the next time

point can be processed with the current time point for all iterations. With a larger minimum service time (or finer time resolution), more events could be processed each iteration, potentially decreasing the number of required iterations further.

While the number of iterations is reduced by using conservative lookahead, E, the number of events processed each iteration, increases. This reduces the impact of overhead on the simulator performance. Hence, the simulator is more efficient, resulting in better speedups. To quantify the performance of conservative lookahead, empirical data was collected. Two simulations of a torus network (256 stations per processor) were executed, one using conservative lookahead and the other using the strict global clock algorithm. Here, the minimum service time equals the time resolution ($r = 1$). As can be seen in Figure 2, the conservative lookahead algorithm performs better than the global clock algorithm. As expected, the number of iterations required using conservative lookahead is slightly more than half the global clock algorithm, $B = 5145$ for conservative lookahead versus $B = 10,000$ for the global clock algorithm. In addition, the number of events processed each iteration was nearly doubled, $E = 675$ for conservative lookahead versus $E = 347$ for the global clock algorithm. Because the simulator is already very successful in exploiting the potential parallelism of the system using the global clock algorithm, the use of conservative lookahead does not produce a dramatic improvement in performance. Speedup does improve, however, from 9.5 to 11.0 at 16 processors.

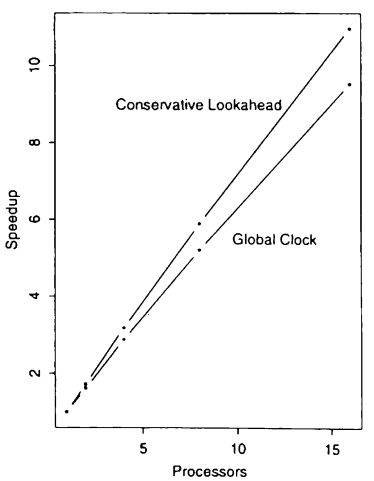

Figure 2: Conservative Lookahead Speedup, r = 1

If the time resolution is increased to 0.1, the event simultaneity decreases and the required number of iterations increases, resulting in a serious degradation in the performance of the global clock algorithm. By using conservative lookahead, this effect can be greatly diminished. In Figure 3, the torus network simulation is repeated with a time resolution $r = 0.1$. The performance of the simulator using conservative lookahead is vastly superior to the global clock algorithm, with the speedup of the conservative lookahead case actually exceeding the speedup with a resolution of 1. The simultaneity using conservative lookahead, $E = 371$ is much greater than using global clock, $E = 35$. Also, the number of iterations with conservative lookahead, $B = 9344$, is significantly lower than with the global clock algorithm, $B = 100,000$.

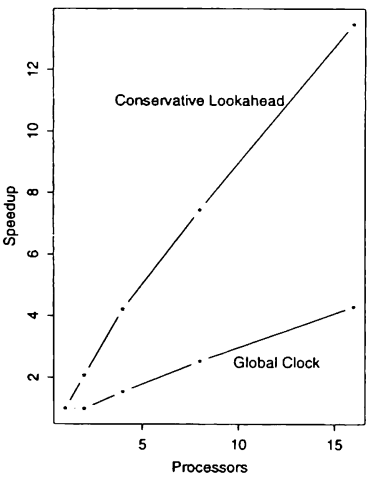

Figure 3: Conservative Lookahead Speedup, r = 0.1

Clearly, the conservative lookahead algorithm can have a significant positive impact on the runtime of synchronous simulations. One contributing factor to this performance improvement is the lack of significant additional overhead needed to implement the conservative lookahead decision. Deciding whether or not an event can be safely executed is not appreciably more time consuming than in the global clock case. The next section explores the performance of optimistic lookahead.

5 OPTIMISTIC LOOKAHEAD

In the conservative lookahead algorithm, the amount of lookahead is restricted to events that can be shown to not violate causality. This restriction limits the number of events that can be processed each iteration. In this section, we attempt to further increase the event simultaneity through the use of optimistic lookahead.

In order to allow the processors to proceed relatively unhindered while maintaining causality, we utilize the technique known as *speculative computing* (Burton 1985). In speculative computing, work is completed before it is known that it is needed or correct. If the work is not needed, it is discarded. By performing speculative computations when the processor would otherwise be idle, there is the potential to improve the performance of the simulator (Mehl 1991). We refer to this technique as optimistic lookahead.

Each iteration, there is a barrier synchronization. Until the last processor reaches the barrier synchronization, the other processors must remain idle. Instead of sitting idle, optimistic lookahead allows the processors to execute future events. We define a static window size that is used to restrict the amount of optimistic lookahead that is performed. No processor can process events timestamped greater than the window length beyond the current simulation time. In order to prevent violations of causality, optimistic lookahead computations are not accepted until it can be ensured that they are correct. The results of optimistic lookahead are saved until their correctness can be guaranteed, at which time they are accepted and take effect. If an event is processed that invalidates the optimistic lookahead computations, the optimistic lookahead results are corrected to compensate for the earlier event.

Each iteration, the optimistic lookahead results computed in earlier iterations are checked to see if they can be accepted. If so, the statistics are updated as needed and the job sent to the appropriate output upon service completion (this may require an event message). The simulation algorithm using optimistic lookahead cannot have a violation of causality because the optimistic lookahead computations are not accepted until they can be guaranteed to be correct.

The performance of the optimistic lookahead algorithm is investigated via the queueing network simulation of a torus network with 1024 stations per processor. The results of these experiments are shown in Figure 4. Note that the window size is five for these simulations.

As one can see, the performance of the optimistic lookahead simulation is not very good. The amount of time spent on optimistic event computations is larger than the time spent processing events at the current simulation time. Because each speculatively processed event requires one or more tests to see if it can be accepted, the execution time per speculative event is larger (i.e., t_e has grown). Measurements indicate that the majority of the events are processed during the optimistic lookahead processing phases. The additional overhead in supporting the optimistic protocol overwhelms any performance gains for this implementation.

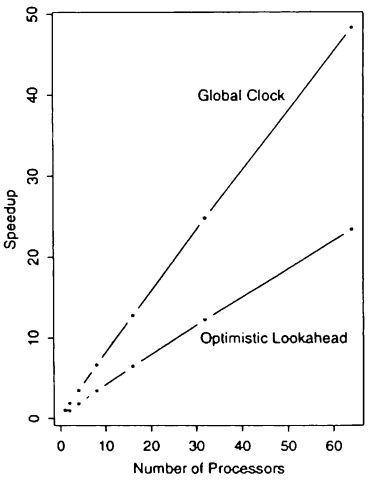

Figure 4: Optimistic Lookahead Speedup

6 CONCLUSIONS

The use of synchronous algorithms for simulation is attractive because of the relative ease in understanding, implementing, debugging, and verifying correct operation. Developing a performance model enables us to identify and reason about the contributors to the execution time of the algorithm. Using both the model and empirical results, we find that both event simultaneity and the number of required barrier synchronizations have a strong impact on performance.

Increasing event simultaneity and decreasing synchronizations both have positive performance implications. Lookahead techniques, both conservative and optimistic, attempt to exploit this fact by executing future events within an iteration, increasing simultaneity and decreasing the number of required synchronizations.

In order to gather empirical performance data, queueing networks were simulated using both conservative and optimistic lookahead techniques. Conservative lookahead showed clear performance improvement, especially for simulations with a fine resolution time clock. Optimistic lookahead, however, was more problematic. The increased overhead associated with implementing the event evaluations defeated the performance benefits due to increased simultaneity and decreased synchronization.

The empirical results presented here are limited

to the simulation of FCFS queueing networks. The model has also been used in the context of logic simulation (Chamberlain and Franklin 1990) using a global clock time synchronization algorithm. Future work relates the application of the model to more general discrete-event simulation problems, and the use of the model to predict performance across a wider range of applications (can the concept of lookahead be generalized to the class of synchronous, iterative algorithms). We also plan to investigate whether alternative implementations of the optimistic lookahead technique will provide better performance.

ACKNOWLEDGMENTS

This work was partially supported by the Department of Education and National Science Foundation Grant No. MIP–9309658.

REFERENCES

Burton, F.W. 1985. Speculative Computation, Parallelism, and Functional Programming. *IEEE Transactions on Computers* C-34:1190–1193.

Chamberlain, R.D. and M.A. Franklin. 1991. Analysis of Parallel Mixed-Mode Simulation Algorithms. *Proceedings of the Fifth International Parallel Processing Symposium.*

Chamberlain, R.D. and M.A. Franklin. 1990. Hierarchical Discrete-Event Simulation on Hypercube Architectures. *IEEE Micro* 10:10–20.

Dubois, M. and F.A. Briggs. 1982. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Transactions on Software Engineering* SE-8(4):419–431.

Ebling, M. et al. 1989. An Ant Foraging Model Implemented on the Time Warp Operating System. *Proceedings of the SCS Multiconference on Distributed Simulation.*

Fujimoto, R.M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM* 33:30–53.

Fujimoto, R.M. 1989. Performance Measurements of Distributed Simulation Strategies. *Transactions of Society for Computer Simulation* 6:89–132.

Hontalas, P. et al. 1989. Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part 1: Asynchronous Behavior and Sectoring). *Proceedings of the SCS Multiconference on Distributed Simulation.*

Jefferson, D. 1985. Virtual Time. *ACM Trans. on Programming Languages and Systems* 7:404–425.

Jefferson, D. et al. 1987. Distributed Simulation and the Time Warp Operating System. *Proc. 11th ACM Symposium on Operating Systems Principles.*

Lin, Y.B. and E.D. Lazowska. 1991. Processor Scheduling for Time Warp Parallel Simulation. *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation.*

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

Madala, S. and J.B. Sinclair. 1991. Performance of synchronous parallel algorithms with regular structures. *IEEE Transactions on Parallel and Distributed Systems* 2:105–116.

Mehl, H. 1991. Speed-Up of Conservative Distributed Discrete-Event Simulation Methods by Speculative Computing. *Proc. of the SCS Multiconference on Advances in Parallel and Distributed Simulation.*

Misra, J. 1986. Distributed Discrete-Event Simulation. *Computing Surveys* 18:39–65.

Nicol, D.M. 1988. High Performance Parallelized Discrete-Event Simulation of Stochastic Queueing Networks. *Proceedings of 1988 Winter Simulation Conference.*

Peterson, G.D. 1992. Qnet: A Globally-Clocked Discrete-Event Queueing Network Simulator. MS Thesis, Department of Electrical Engineering, Washington University, St. Louis, Missouri.

Peterson, G.D. and R.D. Chamberlain. 1993. Performance of Globally-Clocked Parallel Simulation. *International Conference on Parallel Processing.*

Wieland, F. et al. 1989. Distributed Combat Simulation and Time Warp: The Model and Its Performance. *Proceedings of the SCS Multiconference on Distributed Simulation.*

AUTHOR BIOGRAPHIES

GREGORY D. PETERSON received his BSCS and BSEE in 1990, the MSCS and MSEE in 1992, and is currently a doctoral student in electrical engineering at Washington University in St. Louis, Missouri. He was commissioned a Second Lieutenant in the U.S. Air Force in 1990. His research interests include simulation, parallel computation. load balancing, and performance analysis. He is a member of Eta Kappa Nu, Tau Beta Pi, ACM, and IEEE.

ROGER D. CHAMBERLAIN is an Assistant Professor of Electrical Engineering at Washington University in St. Louis, Missouri. He conducts research and teaches in the areas of computer architecture, simulation, and parallel computation. He received the BSCS and BSEE in 1983, the MSCS in 1985, and the DSc computer science degree in 1989, all from Washington University. He is a member of IEEE, ACM, and Eta Kappa Nu.