Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

SIMULTANEOUS PARALLEL SIMULATIONS OF CONTINUOUS TIME MARKOV CHAINS AT MULTIPLE PARAMETER SETTINGS

Philip Heidelberger

IBM Thomas J. Watson Research Center, Hawthorne
P.O. Box 704
Yorktown Heights, New York 10598

David M. Nicol

Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185

ABSTRACT

An earlier paper of ours described how uniformization can be used as the basis of a parallel algorithm for simulating a continuous time Markov chain (CTMC) at a single, fixed set of input parameters. In this paper, we show how the algorithm can be extended so as to simultaneously simulate the CTMC at multiple input parameter settings. The effectiveness of this approach is explored in experiments performed on a 16 node Intel iPSC/2 Hypercube.

1 INTRODUCTION

Consider a queueing network that can be modeled as CTMC. The input to such a model consists of a set of parameters, e.g., the arrival rate, service rates, and routing probabilities. In Heidelberger and Nicol (1991), we described a conservative parallel algorithm (called "PUCS" for Parallel Uniformized Continuous time Simulation) for simulating a CTMC at a single, fixed set of input parameters. In that algorithm, which is based on the technique of uniformization (see, e.g., Gross and Miller (1984)), a relatively short "presimulation" is performed to identify those times, and only those times, at which the simulation algorithm requires processor pairs to synchronize. This presimulation, in effect, builds a communications schedule for the parallel simulation and can be viewed as a "lookahead" calculation (see Nicol (1988) or Fujimoto (1990)). Given the communications schedule, the simulation is carried out in such a way as to produce statistically valid sample paths, and an appointments protocol is used to synchronize processors.

In practice, it is frequently of interest to simulate a model at more than one parameter setting, e.g., at several different arrival rates. In this paper, we describe how the "standard clock" technique (see Vakili (1992)) for simultaneously simulating CTMCs at multiple parameter settings can be combined with PUCS. In this approach, the presimulation is performed in such a way so as to construct the same communications schedule for all parameter settings. Thus synchronization messages for the simulations at the different parameter settings can be piggybacked on top of one another, thereby amortizing the cost of interprocessor communications across the multiple runs. In addition, some statistical savings might be obtained by inducing correlation between estimates from the multiple runs. We call this approach to multiple parameter simulations "Multi-PUCS." A different approach that combines parallel processing and standard clocks is described in Vakili and Lau (1991).

However, the effectiveness of Multi-PUCS is unclear when compared to alternative strategies of assigning processors to parameter settings. For example, suppose there are 16 processors and we wish to simulate the model at two input parameter settings, θ_1 and θ_2. One alternative strategy, that we call the "Consecutive" strategy, is to first run a PUCS simulation using all 16 processors at parameter θ_1, followed by a run of a PUCS simulation using all 16 processors at parameter θ_2.

In this paper, we will consider the relative merits of the Multi-PUCS and Consecutive strategies.

Note that a third potentially attractive strategy, that we call the "Simultaneous" strategy, is to assign 8 processors to a PUCS simulation at parameter θ_1, and a different 8 processors to a PUCS simulation at parameter θ_2. These two PUCS runs are then executed simultaneously. However, the Simultaneous strategy will not be considered in this paper because the way in which our PUCS implementation schedules clusters of queues makes meaningful timing comparisons between 8 processors and 16 processors difficult. (This point will be discussed further in Section 4.)

The rest of the paper is organized as follows. An overview of the basic PUCS algorithm is given in Section 2. The extension of PUCS to multiple parameter settings is described in Section 3. Details of a queueing network simulator that was implemented on a 16 node Intel iPSC/2 Hypercube are described in Section 4. Experiments exploring the effectiveness of Multi-PUCS are described in Section 5. Finally, the results are summarized in Section 6.

2 PUCS SUMMARY

In this section, we outline the basic PUCS algorithm; see Heidelberger and Nicol (1991) for more details. Let θ denote a set of input parameters that serve to define a CTMC $\mathbf{X}_\theta = \{\mathbf{X}_\theta(t), t \geq 0\}$ with a generator matrix $\mathbf{Q}(\theta)$. In a queueing network context, θ might represent the arrival rates, service rates and routing probabilities. We assume that there are P processors on which a simulation of $\mathbf{X}_\theta$ is to be executed. We assume that the state space of $\mathbf{X}_\theta$ can be represented by P-dimensional integer valued vectors $\mathrm{s} = (s_1, \ldots, s_P)$. In the queueing context, we think of assigning queues to processors in which case s_i is an encoding of the states of all the queues assigned to processor i. There are two types of transitions: internal and external. An internal transition is one in which only one component of the state vector changes. For example, if queues 1 and 2 are assigned to processor i, then a job departing queue 1 and entering queue 2 is considered to be an internal transition (to processor i). An external transition is one in which two (or more) components of the state vector change. For example, if queue 3 is now assigned to processor j, then a job departing queue 1 and entering queue 3 is considered to be an (i,j) external transition. Similarly, a job departing queue 3 and entering queue 1 is considered to be a (j,i) external transition.

We assume that $\mathbf{Q}(\theta)$ has the following structure. This structure will make parallel simulation of the CTMC (potentially) effective. If s and s$'$ differ only in component i, then $Q(\mathrm{s}, \mathrm{s}'; \theta) = q_i(s_i, s_i'; \theta)$. If s and s$'$ differ only in components i and j, then $Q(\mathrm{s}, \mathrm{s}'; \theta) = q_{ij}(s_i, s_j, s_i', s_j'; \theta)$, or $Q(\mathrm{s}, \mathrm{s}'; \theta) = q_{ji}(s_i, s_j, s_i', s_j'; \theta)$, depending upon whether we view the transition from s to s$'$ as being an (i,j) or a (j,i) external transition. For simplicity, we assume that at most two components of the state vector can change on any transition.

Let $\lambda(\mathrm{s}; \theta) = -Q(\mathrm{s}, \mathrm{s}; \theta)$ denote the total transition rate out of state s. From the above assumptions, $\lambda(\mathrm{s}; \theta)$ can be written as

$$\lambda(\mathrm{s}; \theta) = \sum_{1 \leq i \leq P} \lambda_i(s_i; \theta) + \sum_{1 \leq i \neq j \leq P} \lambda_{ij}(s_i, s_j; \theta) \quad (1)$$

where

$$\lambda_i(s_i; \theta) = \sum_{s_i' \neq s_i} q_i(s_i, s_i'; \theta) \quad (2)$$

is the total internal transition rate on processor i and

$$\lambda_{ij}(s_i, s_j; \theta) = \sum_{(s_i', s_j') \neq (s_i, s_j)} q_{ij}(s_i, s_j, s_i', s_j'; \theta) \quad (3)$$

is the total rate of (i,j) external transitions.

We assume that the external transitions can be uniformized, i.e., there exist finite constants λ_{ij} such that

$$|\lambda_{ij}(s_i, s_j; \theta)| \leq \lambda_{ij} \quad (4)$$

for all states s.

Equation (1) allows us to view the occurrence of events as being the superposition of internal event processes and external event processes. By Equation (4), the external event processes can be uniformized (with rates λ_{ij}).

Given this framework, PUCS works as follows. In parallel, a presimulation is performed to generate Poisson processes $\{N_{ij}(t)\}$ with rates λ_{ij} $(i \neq j)$. The events in $\{N_{ij}(t)\}$ represent the potential times of (i,j) external transitions. In a communications phase, each processor is made aware of all the potential external event times that affect it. Specifically, processor i receives the times of all the events in $\{N_{ij}(t)\}$ for some j and all the events in $\{N_{ji}(t)\}$ for some j. Processor i then sorts these event times to produce a list $\{(T_i(n), C_i(n)), n \geq 0\}$ of its external events where $T_i(n)$ is the time of the n-th event and $C_i(n)$ is the type of the n-th event, i.e. $C_i(n) = (i,j)$ or (j,i) for some j.

Next processors simulate their events. Suppose that the state of processor i is s_i, that the last event on processor i occurred at time t_i, and that $T_i(n_i)$ is the time of the next (potential) external event. An exponential holding time E_i with mean $1/\lambda_i(s_i; \theta)$ is generated. If $t_i + E_i < T_i(n_i)$, then the next event to occur on processor i is an internal event. In this case, processor i chooses its next state s_i' with probability $q_i(s_i, s_i'; \theta)/\lambda_i(s_i; \theta)$ and updates it's clock to time $t_i + E_i$. If $t_i + E_i \geq T_i(n_i)$, then the next event to occur on processor i is an external event. Suppose that $C_i(n) = (i,j)$. Then processors i and j synchronize and select their next states (s_i', s_j') with probability $q_{ij}(s_i, s_j, s_i', s_j'; \theta)/\lambda_{ij}$. Note that this transition is a pseudo transition (i.e., the state does not change) with probability 1 - $\lambda_{ij}(s_i, s_j; \theta)/\lambda_{ij}$. Both processors then advance their clocks to time $T_i(n_i)$. If $C_i(n) = (j,i)$, the transition is handled similarly.

In queueing applications, an (i,j) transition typically involves processor i sending a time-stamped message to processor j indicating that either the transition is a pseudo transition, or that a job is departing some queue on processor i and is destined for some queue on processor j. Processor i is then free to proceed with its simulation. In conservative simulations, processor j waits until the message from processor i is received.

3 MULTIPLE PARAMETER PUCS

Suppose now that we wish to simultaneously generate the sample paths of two processes $\mathbf{X}_{\theta_1}$ and $\mathbf{X}_{\theta_2}$ corresponding to two sets of input parameters θ_1 and θ_2. Suppose that the uniformization bounds $\{\lambda_{ij}\}$ of Equation (4) hold for both θ_1 and θ_2. Then the same presimulation is valid, and can be used, for both simulations. Thus the same communications schedule $\{(T_i(n), C_i(n)), n \geq 0\}$ can be used for both simulations. By doing so, the time to presimulate and communicate the global event schedules is shared among the two parameter settings.

In addition, by appropriately coding the simulator, the synchronization messages can also be shared, or piggybacked. For example, suppose processor i has advanced to time $T_i(n)$ in both the θ_1 and θ_2 simulations (and has processed the external event at that time). Then processor i can simulate all the internal events at parameter setting θ_1 until time $T_i(n+1)$ and then simulate all the internal events at parameter setting θ_2 until time $T_i(n+1)$. Now suppose that $C_i(n+1) = (i,j)$, i.e., the external event at time $T_i(n+1)$ is of type (i,j). Then processor i can send a single message to processor j describing the specific external events for both the θ_1 and θ_2 simulations. Similarly, if $C_i(n+1) = (j,i)$, i.e., then processor i would wait until receiving a message from processor j describing the external events for both the θ_1 and θ_2 simulations. In this way, all synchronization messages are shared among the two simulations. On the other hand, suppose the θ_1 simulation proceeds faster than the θ_2 simulation. Then, the θ_1 simulation is slowed down by tying its execution to the θ_2 simulation.

4 IMPLEMENTATION DETAILS

We applied the Multi-PUCS method to a queueing network simulation, implemented on a sixteen processor Intel iPSC/2 distributed memory multiprocessor. The queueing network studied has 512 "central server clusters" (see Buzen (1973)): each cluster is comprised of a central server model having one CPU server, which feeds twenty I/O servers. There are a total of 10752 queues in this network. Each cluster is identical; this model can be described by three input parameters: $\theta = (\mu_C, \mu_I, p_c)$ where μ_C is the service rate at each CPU, μ_I is the service rate at each I/O device, and p_c is the probability that a job goes to the CPU in the same cluster after leaving an I/O device. The destination for a job leaving a cluster is chosen uniformly at random from among all other clusters; the job joins the target cluster's CPU queue. The network is closed; each cluster initially has ten jobs, distributed randomly among the cluster's queues. The clusters were distributed evenly among the sixteen processors. Observe that the model is fully connected, in the sense that the model may require communication between any pair of processors.

The implementation supports multiple parameter settings simply by adding another index—the parameter index—to state-dependent data structures that appear in a one-parameter simulation. One simulates additional parameter settings by iterating over this index, as needed. Further details concerning this iteration are described later.

The communication lists are created in a mapping-independent fashion, as follows. We generate the communication lists as though each cluster were to be simulated on its own processor. Each cluster therefore generates a schedule of its interactions with every other cluster. Knowledge of the cluster-to-processor mapping then allows the disparate cluster schedules to be sent to the appropriate processors. A processor's complete communication schedule is created by receiving all such schedules (including ones describing the interactions between co-resident clusters) and sorting them by time-stamp. Observe then that interactions between co-resident clusters is logically treated as though the clusters are on separate processors. However, *implementation* of the interaction between co-resident clusters is quite efficient, being aware of, and taking advantage of the fact the clusters reside on the same processor. One might argue that treating clusters in a mapping independent fashion requires unnecessary generation of communication schedules between co-resident clusters. The benefits of a mapping independent implementation (relative ease of debugging, and direct comparison of the same sample path implemented under differing strategies) temper this objection. Furthermore, our earlier studies (Heidelberger and Nicol, 1991) suggest that the cost of generating communication schedules is comparatively small. Our implementation does not cause any more inter-processor communication than would one that minimizes the length of communication schedules.

The Multi-PUCS algorithm is applied using the processor's aggregate communication schedule, as follows. The time of the next "communication event" is retrieved from the schedule, say with time time-stamp t_0. The internal work up to time t_0 is then performed, which consists of the internal simulation of all the processor's clusters, at all the parameter settings, up to time t_0. While this is a simple and intuitive mechanism, an important point to remember is that some clusters may be unnecessarily blocked. Consider: *all* clusters may wait for an off-processor communication at time t_0, even though only one cluster (the one from whose schedule t_0 is derived) is logically required to synchronize. A more sophisticated scheduler might be used to allow the asynchronous advancement of co-resident clusters.

Our implementation attempts to keep the random number streams associated with the different parameter settings in synchronization. This is accomplished by the use of functionally dedicated per-cluster-per-setting streams, synchronized whenever the cluster engages in a communication event (real or pseudo). For example, every parameter setting for every cluster has its own stream for selecting holding times in the current internal state. Whenever a communication event for the cluster is encountered, the separate streams for the different settings for the cluster are synchronized by having their seeds set to a common value obtained from an orthogonal stream which is dedicated to this purpose. The random number streams used to select queues, route jobs, and so on were synchronized to achieve the property that the j^{th} random number used in a functional stream for a given cluster at a given setting following the cluster's last communication event is the same for every setting in that cluster, for $j = 1, 2, \ldots$. Note that different settings may call upon these synchronized streams a different number of times between a cluster's communication events: depending on the cluster state—a cluster under one setting may execute more internal events before "the next" communication time t_0 than it does under a different setting.

5 EXPERIMENTAL RESULTS

Let $T_P(\theta_i)$ be the time it takes to run the model (for a given amount of simulated time) on P processors at parameter setting θ_i, and let $T_P(\theta_1, \theta_2)$ be the time it takes to run Multi-PUCS (for the same amount of simulated time) on P processors at parameter settings θ_1 and θ_2. The completion time of the Consecutive strategy is $T_P(\theta_1) + T_P(\theta_2)$ so the Consecutive to Multi-PUCS execution time ratio is

$$\frac{T_P(\theta_1) + T_P(\theta_2)}{T_P(\theta_1, \theta_2)}. \tag{5}$$

We ran experiments to compute these ratios for $P = 16$ and a variety of parameter settings on the Intel iPSC/2. Each simulation was run for 500 seconds of simulated time, and involved at least three million events. In addition, at least three replications of each run were made and averaged; very stable estimates of run times were thus obtained.

We ran three sets of experiments: a first set with $p_c = 0.99$, a second set with $p_c = 0.90$, and a third set with $p_c = 0.75$. As p_c decreases, a greater fraction of events become external events and thus a greater amount of interprocessor synchronization is required. The effect of decreasing p_c is thus to make the single parameter version of PUCS less efficient. For example, when $\mu_C = 20$ and $\mu_I = 1.0$, then approximately 1.5% of the events are external when $p_c = 0.99$, 15% of the events are external when $p_c = 0.90$, and 37% of the events are external when $p_c = 0.75$.

For each value of p_c, we fixed the CPU (I/O) rate and varied the I/O (CPU) rate by both 10% and 50%. When the I/O rate is fixed, the best possible external uniformization bounds $\{\lambda_{ij}\}$ are obtained (and used) by assuming all I/O service centers are busy. However, when the CPU rate is fixed, the Multi-PUCS external uniformization bounds $\{\lambda_{ij}\}$ are obtained by assuming all I/O service centers are busy serving at the faster of the two I/O rates. This results in additional external events in the Multi-PUCS simulation of the system with the lower I/O rate. In the Consecutive strategy, the best possible external uniformization bounds are used.

The issue of performance is paramount in parallel computation. One useful measure of performance is *processor utilization*, or the average fraction of time a processor spends performing "useful" simulation work. The time spent in list generation, communication and synchronization is not viewed as "useful" (although it is necessary), in the sense that an optimized serial application does not spend time in such activities. Speedup can be estimated by the product of processor utilization and the number of processors. One can often measure utilization in situations where it is inconvenient (or impossible) to measure true "speedup" (ratio of optimized serial execution time to parallel execution time).

The processor utilizations we measured are insensitive to changes in μ_C and μ_I in the ranges considered by our experiments (the execution times *were* sensitive to these parameters, but relative performance was not). For single-parameter runs using $p_c = 0.99$ we observed 82% efficiencies; for $p_c = 0.9$ we observed 55% efficiencies, and for $p_c = 0.75$ we observed 45% efficiencies.

The results of these experiments are displayed in

Table 1: Consecutive/Multi-PUCS Timing Ratios

(μ_C, μ_I)	$p_c = 0.99$	$p_c = 0.90$	$p_c = 0.75$
$\theta_1 = (20, 1.0)$ $\theta_2 = (22, 1.0)$	1.06	1.13	1.17
$\theta_1 = (20, 1.0)$ $\theta_2 = (30, 1.0)$	1.05	1.13	1.17
$\theta_1 = (20, 1.0)$ $\theta_2 = (20, 1.1)$	1.04	1.10	1.13
$\theta_1 = (20, 1.0)$ $\theta_2 = (20, 1.5)$	1.02	1.02	1.02

Table 1 which lists the Consecutive to Multi-PUCS timing ratios as defined in Equation (5). These ratios should be interpreted in view of the single-parameter utilizations just mentioned. Multi-PUCS simply attempts to amortize the cost of list generation, and communication, over more parameter settings. We can bound the degree to which such amortization improves performance very simply. Suppose that a one-parameter setting model achieves a processor utilization of ρ, and an execution time of X. We may write $X = E + O$ where E is the time spent executing useful work, and O is the time spent in overhead activities. Thus $\rho = E/X$; or equivalently, $E = \rho X$ and $O = (1 - \rho)X$. Consider two one-parameter runs, and assume they have the same execution time, X. The time required by the Consecutive strategy is $2X$, the time required by Multi-PUCS is no less than $2E + O = 2\rho X + (1 - \rho)X$. The ratio of Consecutive performance to Multi-PUCS performance is no greater than $2/(1 + \rho)$. Thus we cannot hope for ratios larger than 1.09, 1.29, and 1.38 for the $p_c = 0.99$, $p_c = 0.9$, and $p_c = 0.75$ models, respectively.

Reading across rows in Table 1, we see that Multi-PUCS becomes more efficient relative to the Consecutive strategy as p_c decreases. As p_c decreases, a greater fraction of the events are external and are therefore shared by both parameter settings in Multi-PUCS. For a fixed value of p_c, Multi-PUCS is more efficient when the CPU rate is varied and the I/O rate is fixed than when the I/O rate is varied and the CPU rate is fixed. When the I/O rate varies, Multi-PUCS has to simulate at an unnecessarily high external uniformization rate for one of the parameter settings, a cost which dominates the overhead. Furthermore, the relative efficiency of Multi-PUCS decreases as the discrepancy between the I/O rates increases.

These figures (and intuition) suggest that relative performance will improve as the number of settings increases without inducing further uniformization. This was confirmed by another $p_c = 0.75$ experiment, where we simulated four models with fixed $\mu_I = 20$ but varied $\mu_C \in [20, 30]$. In this case the relative ratio of Consecutive to Multi-PUCS was 1.28. Thus, when the cost of communication is high, one can hope to achieve significant performance gains with Multi-PUCS provided the external uniformization does not induce additional communication overhead.

As mentioned earlier, another potential benefit of the Multi-PUCS approach (or other common random number techniques) is the induction of correlation between estimates from the multiple runs. We obtained moderate positive correlation. For example, the estimated correlation coefficients for the mean CPU queue length estimates corresponding to the runs in Table 1 with $p_c = 0.99$ ranged from between 0.44 to 0.48.

6 SUMMARY AND CONCLUSIONS

In this paper we described Multi-PUCS, an approach based on uniformization for simultaneously running parallel simulations of CTMCs at multiple parameter settings. In Multi-PUCS, interprocessor communications messages are shared among the multiple simulations. The efficiency of Multi-PUCS, relative to another multiple parameter simulation approach, the Consecutive strategy, was studied empirically through simulations of a large queueing network on a 16 node Intel iPSC/2. Generally speaking, if the parameter being varied is such that the external uniformization rates are unaffected, then Multi-PUCS becomes (relatively) more efficient as the amount of interprocessor communications increases. However, the efficiency gains over the Consecutive strategy were fairly modest when combining two parameter settings. Better performance can be achieved when more parameter settings are included. In addition, moderate positive correlation was induced using this approach.

In order to obtain greater efficiency, more of the execution code would have to be shared among the multiple simulations. However, this might prove difficult in practice. For example, an efficient PUCS implementation uses dynamic tree-like data structures for selecting the next internal event. Effectively sharing the code and data structures to simultaneously search multiple such trees (one for each parameter setting) appears complicated.

While it would have been desirable to compare Multi-PUCS to the Simultaneous strategy, a meaningful comparison requires a more sophisticated scheduler. We discovered this following a comparison of the Simultaneous and Consecutive strategies, which showed Consecutive to be clearly faster. At first this seemed counter-intuitive, since Simultaneous ought to enjoy a better computation/communication ratio (with high p_c). Further investigation showed that processors under the Simultaneous strategy suffer far more "blocking time" — time waiting for a synchronization message — than they do under the Consecutive strategy. We conjecture that this is due to our simple scheduling strategy, which blocks all clusters on a processor waiting for the cluster which is farthest behind to receive a synchronization message. Under the Simultaneous strategy a processor holds twice as many clusters as under the Consecutive strategy, thereby worsening the problem. This phenomenon might be avoided using a more sophisticated scheduler. With such a scheduler, we believe that the absolute efficiency of both PUCS and Multi-PUCS would increase, although the effect on the Consecutive to Multi-PUCS execution time ratio is, at this point, unknown.

REFERENCES

Buzen, J.P. 1973. Computational algorithms for closed queueing networks with exponential servers," *Commun. ACM* **16**, No. 9: 527-531.

Fujimoto, R.M. 1990. Parallel discrete event simulation. *Commun. ACM* **33**, No. 10: 31-53.

Gross, D. and D.R. Miller. 1984. The randomization technique as a modeling tool and solution procedure for transient Markov processes. *Operations Research* **32**: 343-361.

Heidelberger, P. and D.M. Nicol. 1991. Conservative parallel simulation of continuous time Markov chains using uniformization. IBM Research Report RC16780, Yorktown Heights, New York.

Nicol, D.M. 1988. Parallel discrete-event simulation of FCFS stochastic queueing networks. In: *Proceedings of the ACM/SIGPLAN PPEALS 1988. Parallel Programming: Experiences with Applications, Languages and Systems*, 124-137. ACM Press.

Vakili, P. 1992. Using a standard clock technique for efficient simulation. To appear in: *Operations Research Letters.*

Vakili, P. and E. Lau. 1991. Massively parallel simulation and optimization of queueing networks. In: *Computer Science and Statistics: Proceedings of the 23rd Symposium on the Interface.*

AUTHOR BIOGRAPHIES

PHILIP HEIDELBERGER has been a Research Staff Member at the IBM Thomas J. Watson Research Center since receiving a Ph.D. in Operations Research from Stanford University in 1978. He is an Area Editor of the ACM's *Transactions on Modeling and Computer Simulation* and is currently serving as the Program Co-Chairman of the ACM Sigmetrics/Performance '92 Conference. He was the Program Chairman of the 1989 Winter Simulation Conference.

DAVID M. NICOL received an B.A. in Mathematics from Carleton College in 1979, worked as a programmer analyst with the Control Data Corporation from 1979 to 1982, and received a Ph.D. in Computer Science from the University of Virginia in 1985. He spent the subsequent two years at the Institute for Computer Applications in Science and Engineering at the NASA Langley Research Center, and then joined the faculty of the College of William and Mary, where is currently an Assistant Professor. He is an associate editor for the ACM's *Transactions on Modeling and Computer Simulation* and for the *ORSA Journal on Computing*, and has served as the 1989 Program Chairman and the 1990 General Chairman of the Workshop on Parallel and Distributed Simulation (PADS). His interests are in parallel simulation, performance analysis, and algorithms for mapping parallel workload.