A Comparative Study of Parallel Algorithms for Simulating Continuous Time Markov Chains

DAVID M. NICOL
The College of William and Mary
and
PHILIP HEIDELBERGER
IBM T. J. Watson Research Center

This article describes methods for simulating continuous time Markov chain models, using parallel architectures. The basis of our method is the technique of uniformization; within this framework there are a number of options concerning optimism and aggregation. We describe four different variations, paying particular attention to an adaptive method that optimistically assumes upper bounds on the *rate* at which one processor affects another in simulation time, and recovers from violations of this assumption using global checkpoints. We describe our experiences with these methods on a variety of Intel multiprocessor architectures, including the Touchstone Delta, where excellent speedups of up to 220 using 256 processors are observed

Categories and Subject Descriptors: I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Markov chains, simulation

Portions of this paper are reprinted with permission from "Parallel Algorithms for Simulating Continuous Time Markov Chains," in *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, and from "Parallel Simulation of Markovian Queueing Networks Using Uniformization," *in Proceedings of the 1993 ACM SIGMETRICS Conference*. D. M. Nicol's work was initiated while he was a visiting scientist at the IBM T. J. Watson Research Center. This work was also supported in part by NSF grants ASC 8819373 and CCR-9201195, and NASA grants NAG-1-1060 and NAG-1-995. This research was also partially supported by NASA contract NAS1-19480 while the author was on sabbatical at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton Virginia 23681. P. Heidelberger's research was partially supported by NASA contract NAS1-19480 while the author was on sabbatical at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681.
Authors' addresses: D. M. Nicol, Department of Computer Science, The College of William and Mary, Williamsburg, VA 23185; P. Heidelberger, IBM T. J. Watson Research Center, Hawthorne, P.O. Box 704, Yorktown Heights, NY 10598.

1. INTRODUCTION

Discrete-event simulation is an invaluable tool for the design and analysis of complex systems such as factories, transportation networks, computer systems, and communication networks. Large scale simulations require a long time to execute, and because of this many researchers are interested in parallelizing their execution. One of the key issues is synchronization between processors, as the synchronization demands are highly variable, depending dynamically on the simulation model's state. Comprehensive surveys on the topic are found in Fujimoto [1990], Righter and Walrand [1989], and Nicol and Fujimoto [1994].

Parallel simulation is difficult because synchronization between processors is very dynamic and often unpredictable. Each processor maintains its own simulation clock, and we require that the end result of the simulation be consistent with a scenario in which every processor executes its time-stamped events in monotone increasing order. The problem is that execution of an event on one processor can cause an event on another processor. Synchronization protocols are necessary to either ensure that local clocks increase monotonically, or ensure that the results are the same as if they had. A spectrum of approaches to parallel simulation have been considered, including parallel replications, functional decomposition, synchronous time-stepped methods, conservative methods, optimistic methods, fixed-point methods, and the application of parallel prefix allgorithms.

In conservative parallel simulations, a *logical process* (LP) does not execute an event e at simulation time t until it is sure that the effects of all events occurring at earlier simulation times $s < t$ that can possible affect e are known. Thus execution of the event e is guaranteed to be correct. In an optimistic method (e.g., Time Warp [Jefferson 1985]) an LP *may* execute e before it is certain that e is correct. If the LP later discovers it should have executed an event e' at time $s < t$, it must roll back, recover its state at time s, and undo the effects of all its simulation activity at times greater than s. This recovery is made possible if the LP frequently saves its state. State-saving is usually cited as the leading source of overhead; the potential problem is severe enough that efforts to provide hardware support have been proposed [Fujimoto et al. 1992].

Optimism can also be employed by periodically checkpointing the entire simulation state, simulating optimistically between checkpoints, and noting any errors that occur along the way due to optimism. Should any errors occur between checkpoints n and $n + 1$, the first such can be identified, and all processors resimulate their submodels in such a way that the same sample path is taken up the point of the error, but then the error is avoided. This sort of approach is promising when state-saving costs are high, and errors are very rare; it has proven to be successful in simulating "colliding pucks" [Lubachevsky 1990] and is the approach we take with one of our methods.

In a series of previous papers[1] we have investigated the idea of using *uniformization* as the basis for synchronization in parallel discrete-event simulation of continuous time Markov chains (CTMCs). CTMC models are important, appearing frequently in the study of computer and communication systems. Uniformization exploits the mathematical structure of these models, making it possible to precompute instants in simulation time where logical processes ought to synchronize. The decision whether an LP actually influences another at one of these instants is left until run-time. Conceptually, a simulation is performed in several phases. In the first phase the simulation model is partitioned into LPs, which are mapped to processors. All simulation activity associated with an LP is assumed to be performed by its assigned processor. In the second phase one randomly generates synchronization points; in the third phase one simulates a mathematically correct sample path through those points. We call the general method PUCS, for *Parallel Uniformized Continuous-Time Simulation*. This approach generalizes Lubachevsky's algorithm [1987] for simulating cellular arrays.

We have developed four different variations of PUCS that differ in their treatment of LP aggregation, communication management, use of optimism, and generation of communication schedules. Each of these methods has strengths and weaknesses that are revealed by problem characteristics. The objectives and contributions of this article are:

(1) To give an overview of uniformization-based synchronization.
(2) To describe an adaptive algorithm that dynamically adjusts the rate at which LPs synchronize.
(3) To empirically examine these different methods on a variety of Intel multiprocessors, including the iPSC/2 [Rattner 1985], the Intel Touchstone Delta [Lillevik 1991], and the Intel Paragon [Intel Corp. 1993]. These studies show that the adaptive algorithm is the most robust of our implementations. It is capable of providing good performance over a wider range of problems than any of our other algorithms.

A description of the adaptive method and a preliminary empirical comparison previously appeared only in conference proceedings papers [Nicol and Heidelberger 1993b, 1993c].

Not all CTMCs are suitable for parallel simulation using our methods. A key requirement is that one be able to partition the CTMC into loosely synchronous interacting subchains. Such partitioning follows intuitively when the CTMC has a basis in a physical domain, because partitioning the domain often has the desired effect. Nevertheless, the issue of defining suitable LPs automatically is one that we have not yet addressed.

The remainder of the article is organized as follows: Section 2 gives an overview of direct Markovian simulation, uniformization, and different ways

[1] See Heidelberger and Nicol, and Nicol and Heidelberger [1993a, 1993b, 1993c].

we have exploited uniformization for parallel simulation. Section 3 develops our adaptive method and presents related performance data. Section 4 presents and analyzes our experimental results, and Section 5 gives our conclusions.

2. UNIFORMIZATION-BASED SYNCHRONIZATION

In this section we describe the basic notions of direct Markovian simulation, and uniformization. More rigorous and complete mathematical details can be found in [Heidelberger and Nicol 1993]. Following the descriptions we illustrate them concretely with an example.

Let us first review some basic elements of the theory of CTMCs. Readers unfamiliar with CTMCs are encouraged to consult Ross [1983] for a more complete and exact introduction to the topic. A CTMC is a stochastic process $\{X(t)\}$, where $X(t)$ is the *state* of the CTMC at time t. For the purposes of general description, $X(t)$ is taken to be nonnegative integer; in practice it is often more natural to describe $X(t)$ as a vector of integers, for example, the vector of queue lengths in a network, a vector of token markings in a stochastic Petri net, or a vector of cell states in an Ising spin model. Upon entering a state $\mathbf{N}$ at time t, the CTMC remains in that state for a random period of time called the *holding time*, which is an exponential distribution with state-dependent rate $\lambda(\mathbf{N})$. This is also called the *transition rate* out of state $\mathbf{N}$. At the end of the holding time, the CTMC randomly changes state, jumping to some state $\mathbf{N}'$. It is convenient to think of this jump as choosing a winner among all possible jumps, in the following way. While in state $\mathbf{N}$ the chain is attempting to make a transition to every other state, simultaneously. It is as though there are a large number of stochastic processes—one for each state distinct from $\mathbf{N}'$—that are all concurrently active. The transition rate for the process attempting to jump to $\mathbf{N}'$ is some $Q_{\mathbf{NN}'}$; note that $\lambda(\mathbf{N}) = \sum_{\{\mathbf{N}' \neq \mathbf{N}\}} \mathbf{Q}_{\mathbf{NN}'}$. Each of these processes has an exponentially distributed holding time; the rate of $\mathbf{N}'$'s holding time is just $Q_{\mathbf{NN}'}$. We may imagine that each of these holding times are randomly sampled at the time $\{X(t)\}$ enters $\mathbf{N}$. Now the time and nature of $\{X(t)\}$'s transition out of $\mathbf{N}$ are defined by the process whose next transition time is least among all possibilities. Thus, the probability that the exponential associated with a given state $\mathbf{N}'$ is least among its peers is just $P_{\mathbf{NN}'} = Q_{\mathbf{NN}'}/\lambda(\mathbf{N})$; $P_{\mathbf{NN}'}$ is the probability transition matrix of the embedded discrete time Markov chain associated with the CTMC.

Observe that we can also interpret a transition in terms of $\{X(t)\}$ simultaneously attempting jumps in one of a number of *sets* of transitions. For example, we might partition transitions into two sets A and B, and interpret the transition as a competition among all transitions in A and all transitions in B. This interpretation will be particularly useful in a parallel simulation setting when A is the set of transitions that affects only one processor's state and B is the set of transitions that affects the states of multiple processors.

A direct simulation of a CTMC involves sampling holding times, and choosing transitions, as follows. Upon entering state **N**, one advances time by sampling an exponential with rate $\lambda(\mathbf{N})$, essentially simulating the duration of time the CTMC remains in state **N**. To choose a transition it is not necessary to choose the least of a large number of exponentials. It suffices to construct the transition distribution by computing the rates $Q_{\mathbf{NN}'}$, and then sampling from the distribution $\{P_{\mathbf{NN}'} = Q_{\mathbf{NN}'}/\lambda(\mathbf{N})\}$.

Uniformization of a CTMC is a mathematical device (originally used to simplify numerical solution [Gross and Miller 1984] designed so that every holding time is drawn from the same distribution. The basic idea is to find a *uniformization rate* $\lambda_{\max}$ such that for every state **N**, $\lambda(\mathbf{N}) \leq \lambda_{max}$. All holding times are sampled from the exponential distribution with rate $\lambda_{\max}$. However, to make the uniformized chain stochastically identical to the original chain, we introduce transitions back to the same state. In the uniformized chain, the probability of making a transition from **N** to $\mathbf{N}'$ ($\neq \mathbf{N}$) is $Q_{\mathbf{NN}'}/\lambda_{\max}$. The probability of making a transition back to **N** is $1 - \lambda(\mathbf{N})/\lambda_{\max}$. Transitions of the latter form are known as *pseudo transitions*, as they do not affect the state of the Markov chain. The mathematical basis for uniformization is simply that a geometrically distributed sum (with mean $1/p$) of i.i.d. exponential random variables (with mean $1/\mu$) is itself an exponential, with mean $1/(p\mu)$. Whenever the original chain is in state **N**, its holding time is exponential with rate $\lambda(\mathbf{N})$. Now suppose the uniformized chain (at rate $\lambda_{\max}$) enters state **N**; the number of pseudo transitions that occur before actually leaving **N** is geometrically distributed with mean $\lambda_{\max}/\lambda(\mathbf{N})$, and the distribution of time spent in **N** before leaving is that of a geometrically distributed sum of exponentials, each with mean $1/\lambda_{\max}$. The effective distribution of time the uniformized chain spends in state **N** is exponential with mean $1/\lambda(\mathbf{N})$, just as in the original chain.

For the case of parallelized simulation we assume that the model is partitioned into submodels, with one submodel assigned to each processor (although some of our methods assume multiple distinct submodels for each processor). We also assume that the model has the characteristic that a state change in one such submodel may instantaneously affect the state in at most one other submodel (this too may be relaxed to include more submodels, but it is important that the set of affected submodels be small). Finally, we assume that the model can be analyzed to determine, for every pair of processors P_i and P_j, an upper bound λ_{ij} on the rate at which P_i's submodel makes transitions affecting P_j's submodel. For instance, in a queueing network λ_{ij} would bound the rate at which P_i routes jobs to P_j. P_i and P_j then agree ahead of time to synchronize as though P_i's submodel continuously affects P_j's at rate λ_{ij} in a Poisson stream, and presample the simulation instants at which these synchronizations occur. During the simulation run P_i randomly decides at each such prechosen synchronization point whether to actually make a transition that affects P_j, in effect "thinning" (see Lewis and Shedler [1979]) the Poisson process. The probability of making such a transition depends on the state of P_i's submodel at that instant. P_i must send a message to P_j indicating either that the threatened transition occurred, or

that the communication event is a pseudo. Using uniformization in this way permits processors to identify all synchronization times, in advance of actually running the simulation. A variety of synchronization protocols that exploits this knowledge can then be devised.

Experiments have shown that uniformization-based methods work well when the model can be partitioned so that workload is balanced, and there is an adequate number of events processed between successive synchronization events. Performance degradation is observed, however, when λ_{ij} is much larger than the actual average rate at which P_i makes transitions affecting P_j. In such cases most of the communications are pseudos—which do not occur in an optimized serial simulation—and hence are overheads that lower speedup. This problem is inescapable for all uniformization-based methods that require that each λ_{ij} be a true upper bound. The *adaptive* method highlighted in the article reduces overheads by dynamically altering λ_{ij} in an effort to find lower rates that effectively serve as upper bounds, even if there is a small change of violating the bound. Our experiments show that the method may raise poor performance to acceptable levels, in some cases more than doubling the execution rate. We also observe significant speedups; in one experiment a speedup of over 220 was obtained on 256 nodes of the Intel Touchstone Delta.

We first use an example of a distributed computing system to explain uniformization, and then illustrate its use in parallel simulation of Markovian queueing networks. This example is also used as a test model for our experiments. Finally we identify model characteristics that may lead to excessive pseudo events.

2.1 Example

The model consists of a number P of computing clusters. The model is a closed queueing network with $J \times P$ jobs. Each cluster is a central server model (see Buzen [1973]) with a single CPU and K I/O devices. All queueing disciplines are FCFS (an assumption that is not necessary for uniformization to work). The service times at the CPU are assumed to be exponentially distributed with rate μ_c. When a job leaves the CPU it goes to one of the I/O devices, which is selected uniformly. The service times at the I/O devices are assumed to have a hyperexponential distribution with two phases; with probability p_f a fast phase is chosen that has rate μ_f, and with probability $1 - p_f$ a slow phase is chosen that has rate μ_s ($\mu_f \geq \mu_s$). When a job leaves an I/O device, it returns to the CPU in the cluster with probability p_c; otherwise it selects another cluster according to a uniform distribution and enters the CPU queue in the selected cluster. The state of such a system can be described by a multidimensional vector $\mathbf{N} = (\mathbf{N}_P, \ldots, \mathbf{N}_P)$ where $\mathbf{N}_i$ describes the state of cluster i. Specifically, let $\mathbf{N}_i = (n_{i0}, n_{i1}, \ldots, n_{iK}, I_{i1}, \ldots, I_{iK})$ where n_{i0} is the total number of jobs at the CPU, n_{ij} is the total number of jobs at I/O device j, and I_{ij} is an indicator variable indicating whether the jth I/O device is serving in the fast or slow phase. Let $M_i(f)$ (alternatively, $M_i(s)$) denote the total number of cluster i I/O devices serving in the fast

(slow) phase and let $M_i(c) = 1$ if the CPU in cluster i is busy. The total rate of events in cluster i is then simply given by

$$\lambda_i(\mathbf{N}_i) = \mu_c M_i(c) + \mu_f M_i(f) + \mu_s M_i(s). \tag{1}$$

The system remains in a given state $\mathbf{N}$ for an exponentially distributed period of time with rate $\lambda(\mathbf{N}) = \sum_{i=1}^{P} \lambda_i(\mathbf{N}_i)$. After the holding time, the chain makes one of several possible transitions, chosen randomly. A transition due to a cluster i CPU completion is chosen with probability $\mu_c M_i(c)/\lambda(\mathbf{N})$. An I/O completion from a fast phase I/O server on cluster i is chosen with probability $\mu_f M_i(f)/\lambda(\mathbf{N})$, and an I/O completion from a slow phase I/O server on cluster i is chosen with probability $\mu_s M_i(s)/\lambda(\mathbf{N})$. Following simulation of the chosen transition and its effect on $\mathbf{N}$, a new holding time is chosen based on the new state, and the simulation process continues. Note that this way of generating sample paths is quite different from the usual event list approach used in most discrete-event simulations.

Uniformization provides a different way to simulate this chain. The maximum possible rate for any state is $\lambda_{\max} = P\mu_c + PK\mu_f$ which is obtained by assuming that all servers are busy serving in their fastest phases. In the uniformized chain all transitions occur at rate $\lambda_{\max}$. In the uniformized chain, the transition is a cluster i CPU departure with probability $M_i(c)\mu_c/\lambda_{\max}$, a cluster i fast phase I/O completion with probability $\mu_f M_i(f)/\mu_{\max}$, a cluster i slow phase I/O completion with probability $\mu_s M_i(s)/\lambda_{\max}$, and a pseudo-transition back to the same state with probability $1 - \lambda(\mathbf{N})/\lambda_{\max}$.

2.2 Application to Parallel Simulation

We now apply uniformization to the parallel simulation of the model just described. We assume that queue m has a finite number of servers S_m, and that a job's routing probability is independent of both the identity of the job's specific server in the queue and its service rate. More complex situations can be handled by uniformization; once the basic idea is understood the means to such extensions are readily apparent.

Suppose that the queueing network has been partitioned among processors. Let A_i denote the set of queues assigned to processor P_i. Let $\overline{\mu}_m$ be the maximum service rate a job receives at queue m and let p_{mk} be the probability that a job departing queue m is routed to queue k.

The behavior of the submodel assigned to a processor P_i (which we treat as a single LP) can be viewed as the merging (superposition) of a number of different event streams. An "internal" stream is comprised of entirely on-processor events, namely, service completions whose jobs are routed to other queues assigned to P_i. For every P_j with queues receiving jobs from P_i $(j \neq i)$ there is also a stream of "external" events, which we may uniformize with rate

$$\lambda_{ij} = \sum_{m \in A_i} S_m \overline{\mu}_m \left(\sum_{k \in A_j} p_{mk} \right). \tag{2}$$

The maximum rate defined by Equation (2) is obtained by assuming that every server is busy in its fastest phase. For the example system of clustered central server models, Equation (2) becomes $\lambda_{ij} = K \times \mu_f \times (1 - p_c)/(P - 1)$, which reflects the rate at which P_i sends jobs to P_j provided all I/O devices are busy in the fast phase. (The total rate of off-processor events is simply $K \times \mu_f \times (1 - p_c)$.)

The parallel simulation algorithm basically views the simulation as the superposition of internal and external processes. Internal processes are simulated directly, whereas external processes are simulated using uniformization. A parallelized simulation can be done in two phases. In the first, each processor P_i randomly samples transition times for Poisson streams $\{N_{ij}(t)\}$ with rates λ_{ij} $(i \neq j)$. The events in $\{N_{ij}(t)\}$ represent the potential times that jobs are sent from P_i to P_j. Next, in a communications phase, each processor is made aware of all the potential external event times that affect it. Specifically, P_i receives all the events in $\{N_{ji}(t)\}$ for all P_j. (Alternatively, by appropriate synchronization of random number generator seeds, P_i and P_j can both compute the times in $\{N_{ji}(t)\}$ thereby avoiding the communications phase.) Each P_i then merges 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 nth event and $C_i(n)$ is the type of the nth event, that is, $C_i(n) = (i,j)$ or (j,i) for some j, reflecting a $P_i \rightarrow P_j$ or $P_j \rightarrow P_i$ message, respectively.

Uniformization gives the ability to completely precompute a set of times guaranteed to include the transitions at which jobs actually move between processors. This ability, which is called "lookahead" in the parallel simulation literature, greatly simplifies the interprocessor synchronization problem. Of course, it remains still for the simulation to determine which of these transitions are real.

As we simulate in parallel, each processor will execute independently of the others, except for synchronization at the prearranged instants in time. For example, suppose that the state of P_i's submodel is $\mathbf{N}_i$, that the last event on P_i occurred at time t_i, and that $T_i(n_i)$ is the time of the next (potential) external event. Let $\lambda_{ii}(\mathbf{N}_i)$ denote the total internal (i.e., on-processor) event rate on processor P_i. (For the example model, $\lambda_{ii}(\mathbf{N}_i)$ is the rate of CPU completions plus I/O completions that are routed back to the CPU on the same cluster; $\lambda_{ii}(\mathbf{N})_i) = M_i(c)\mu_c + (M_i(f)\mu_f + M_i(s)\mu_s)p_c$.) Note that the total rate of events initiated on processor P_i is given by $\lambda_i(\mathbf{N}_i) = \sum_{j=1}^{P} \lambda_{ij}(\mathbf{N}_i)$.

An exponential holding time R_i with mean $1/\lambda_{ii}(\mathbf{N}_i)$ is generated. If $t_i + E_i < T_i(n_i)$, then the next event to occur on P_i is an internal event. In this case, among all possible internal transitions, P_i chooses one with probability proportional to its transition rate, simulates, and updates its clock to time $t_i + E_i$. (For instance, in the example model, an internal fast phase I/O completion is selected with probability $M_i(f)\mu_f p_c/\lambda_{ii}(\mathbf{N}_i)$. After that, one of the I/O queues serving in the fast phase is randomly selected as the queue on which the completion occurs.)

If $t_i + E_i \geq T_i(n_i)$, then the next event to occur on P_i is an external event. Suppose then that $C_i(n) = (i,j)$ and let $\lambda_{ij}(\mathbf{N}_i)$ denote the current actual rate of transitions from P_i to P_j. (For the example model, $\lambda_{ij}(\mathbf{N}_i) = (M_i(f)\mu_f +$

$M_i(s)\mu_s)(1 - p_c)/(P - 1)$.) P_i decides whether the transition is pseudo or real by selecting a real transition with probability $\lambda_{ij}(\mathbf{N}_i)/\lambda_{ij}$ and a pseudo transition with probability $1 - \lambda_{ij}(\mathbf{N}_i)/\lambda_{ij}$. In the case of a real transition, P_i selects the job whose service completes according to a probability proportional to the rate at which that job is departing for a processor P_j queue. In either case, P_i sends a message to P_j specifying the job transfer (including the identity of the target queue), or a pseudo transition and continues. Alternatively, if $C_i(n) = (j,i)$, then P_i waits for a message from P_j, simulating the specified arrival (or pseudo event) depending on the message contents. Following simulation of $C_i(n)$, P_i advances its clock to time $T_i(n_i)$. A new holding time for the internal process is selected, and the procedure continues.

We have explored a number of logical and implementation issues for uniformization-based synchronization. As we report on the performance of each, we first briefly cover their salient points.

2.3 Conservative Aggregated PUCS

CA-PUCS (identified simply as PUCS in Heidelberger and Nicol, and Nicol and Heidelberger [1993a, 1993b]) was one of the first methods we developed. In implementation it is almost identical to the description given in the last section. It has the additional characteristics that the entire submodel assigned to a processor is considered to be one LP, and that syunchronization lists are generated and simulated on a window-by-window basis. The latter feature is needed for the simple reason that computers' memories can retain only a finite number of external transition descriptions, and very long runs will require very long transition lists.

The rationale for aggregating all coassigned workload into one LP is two-fold. First, a one-LP-per-processor implementation is much easier to develop than one that allows multiple LPs. The architecture used in our studies—the Intel family of multiprocessors—supports interprocessor communication via explicit sends and receives. Receives may be either asynchronous (post a receive and periodically check on whether the anticipated message arrived yet) or synchronous (block until the anticipated message arrives). Furthermore, the Intel iPSC/860 and Touchstone Delta operating system, NX, support only one process per processor. Any multitasking—such as switching between LPs—has to be done at the application layer. By aggregating all of a processor's workload into one LP we avoid scheduling issues; furthermore, there is no need to buffer incoming communication at the application layer. When the processor expects message m at time t from processor j, it simply does a synchronous receive, and blocks until that message materializes. One cannot use synchronous receives if switching between LPs is necessary. Secondly, massive aggregation avoids *internal* pseudo events that may occur when multiple LPs are assigned to one processor. The problem here is that if uniformization is applied at the LP level, then two LPs on the same processor synchronize with each other just as though they were assigned to separate processors. We surely can develop the code so that the communication between coresident LPs is cheap, but we cannot

easily avoid the overhead of generating, communicating, and synchronizing upon a pseudo event. An important rationale for massive aggregation is to eliminate the possibility of internal uniformization.

2.3.1 *Conservative Partitioned PUCS.* The other side of the aggregation issue is that massive aggregation can cause artificial blocking. Events on a processor under CA-PUCS are executed in increasing monotonic order. If any piece of a processor's submodel needs a message at time t and if that message is not yet present, the entire processor blocks. However, it may be that another piece of the submodel is free to continue past time t. To block at time t is to cheat oneself of some potential parallelism.

CP-PUCS (identified as PUCSThreads in Nicol and Heidelberger [1993a]) allows multiple LPs per processor, and also strives to reduce the communication overhead of list generation. The principal features of the method are as follows.

—**LP independence:** A processor may manage any number of distinct LPs. In addition, by appropriate assignment of random number generator seeds, the sample path that is executed can be made independent of the way in which LPs are assigned to processors.

—**Scheduling:** At any time, each LP is classified as being *ready* or *blocked*, depending on whether it is free to execute or its waiting for an incoming message. Scheduling consists of selecting the ready LP with least timestamp, performing a communication (either a send or a receive) and simulating until it reaches its next communication instant. If an LP blocks waiting for a message, a description for that message is stored in a binary search tree. Between LP activations we probe for any newly received messages, accepting all such and storing them in the application space. As each new message is processed we examine the search tree to see if some LP is blocked on this message. If so, the LP is unblocked and placed on the list of ready LPs.

—**List Generation:** Every pair of LPs i and j maintain a synchronized random number generator. This means that LP i can compute for itself the same transition times that j computes for the LP j to LP i external stream. Although each LP now executes more work by duplicating the generation of external stream transition times, we avoid having to communicate and merge the lists. There is an additional advantage in that no window is needed now to limit the memory usage of external transition times. We simply generate the "next" transition time for a stream when it is needed.

Somewhat to our surprise, our previous empirical studies found no real benefit of CP-PUCS over CA-PUCS. Those studies examined situations in which the deleterious effect of internal pseudos was the dominant bottleneck to achieving good performance, and thus the benefit of avoiding them outweighed the benefit of more parallelism. However, data in the present article show that this is not always the case and there are situations in which CP-PUCS outperforms CA-PUCS. We comment more on this in Section 4.

2.4 Optimistic PUCS

Opt-PUCS (identified in Nicol and Heidelberger [1993a] as OptAll) endows CP-PUCS with optimism. This comes into play when an LP reaches an incoming communication instant, and the message it is to receive is not yet present. The LP can optimistically assume that the message will report a pseudo-transition, and hence there is no need to wait for it. When the message does finally arrive, if the receiving LP's guess was correct, then there is no need to roll back. This is an application of the idea of "lazy reevaluation" explored first in West [1988]. Otherwise, as with standard optimistic algorithms such as Time Warp [Jefferson 1985], the receiving LP is rolled back to the time of the late message.

PUCS' general framework makes possible some unique optimizations.

State Certainty: In a general-purpose optimistic environment, one can never be certain whether the next event processed will end up being committed, or will be discarded as a result of rollback. In Opt-PUCS an LP can sometimes know that its state is **sure**, that it will not be rolled back past its present point. The key to this determination is that we know all instants in simulation time where messages may arrive. If LP i knows it will not receive any message between times s and t, and it knows that its present state is **sure** (all LPs are initially **sure**), then its state remains **sure** while processing all internal events up to time t. Furthermore, if LP j sends the message at t and was also **sure** at the time the message was sent, then the message may be received and LP i remains **sure**. However, if either LP j was **unsure** at time t, or if the LP i decides to optimistically bypass that communication, then LP i becomes **unsure**. In Nicol and Heidelberger [1993a] we show how every LP can maintain a *Least Sure Time* (LST) that describes the last instant in simulation time when the LP was **sure**. By simply appending **sure/unsure** tags to messages and analyzing these, every LP's LST advances without extra calculation. Because we may release any state saved at a time less then the LST, the LST calculation gives us the benefits of the usual GVT calculation, without the additional overhead of actually performing a GVT calculation.

State-Saving: Optimistic simulations generally save state prior to every event, because as far as the LP knows, the simulation can in theory be rolled back to any point in simulation time ahead of the last known GVT. Within the PUCS framework, a rollback can occur only at some communication instant, hence there is no advantage to saving state before an internal event. The only time state must be saved is at a communication instant, and then only if the receiving LP is either **unsure** or becomes **unsure** by either receiving an **unsure** message or by optimistically bypassing it.

Scheduling: Our ability to ascertain whether an LP's state is **sure** permits smarter scheduling than is usually possible under Time Warp because we may give highest priority to an LP with some work to do that we know is **sure**, and cannot be rolled back. In fact, our studies in Nicol and Heidelberger [1993a] found that a very effective scheduling strategy is one that is averse to state-saving, as follows. An LP's execution slice is delimited at

either end by an external communication (either incoming or outgoing); the execution slice begins by performing a communication, then all internal work up to (but not including) the next communication is performed. Whether we perform a state-save at the initial communication depends on the present **sure/unsure** state of the LP, whether the communication is outgoing or incoming, and whether a communication is present or **unsure**. We define the following scheduling classes, listed in decreasing order of priority.

(1) **sure** LPs that will not save state because the first communication is either an incoming message from a **sure** LP, or is an outgoing message.
(2) **unsure** LPs whose first communication is either an incoming message from a **sure** LP, or is an outgoing message. The LP need not save its state.
(3) **sure** LPs that must save state on the first communication, because that communication (necessarily incoming) is either not yet present, or was sent by an **unsure** LP.
(4) **unsure** LPs that must save state on the first communication, because that communication (necessarily incoming) is either not yet present, or was sent by an **unsure** LP.

One of our aspirations for Opt-PUCS was that it would reduce the cost of pseudo transitions. Although pseudos would still appear logically in the external event streams, the hope was that not having to communicate them from **unsure** LPs would lead to some savings. Our initial experiments showed that this intuition held true, provided that the fraction of pseudo events was very high. For lesser fractions of pseudos, the overheads of optimism largely concealed the benefits of optimism. This observation is also borne out in the data we present in this article. One should also bear in mind that the version we study here is highly optimized. Our previous study suggested that its performance is as large as a factor of 2 better than standard Time-Warp style algorithms.

2.5 The Problem of Pseudos

Parallel simulation based on uniformization yields excellent speedups when communication costs are the minimum possible (i.e., when the percentage of pseudos is low), and the overall computation/communication ratio is favorable. However, our experiments bear out the intuition that performance can suffer when the percentage of pseudos is high. This happens when the external streams' actual transition rates are much lower than their uniformization rates in situations where communication dominates performance. For queueing networks, situations where this can occur include the following.

—*Large variations in service rates:* Consider the example model with hyperexponential service times in which fast I/O service (at rate μ_f) is given with low probability (p_f) and slow service (at rate μ_s) is given with high probability. Even though fast service is rare, a strict uniformization bound must assume that the job in service is fast. If $\mu_f \gg \mu_s$, then the strict uniformization bound will be much too high. Another example in which

there may be a large variation in the service rates is when there are multiple priority levels that receive service at different rates.

—*Low utilizations:* Uniformization bounds generally need to assume that all servers are always busy. If instead a server is almost always idle, then the uniformization bound will not be tight.

—*State-dependent routing:* Imagine a queue that routes a job as a function of its queue length L. If some processor receives a job from this queue only from rarely achieved values of L, then the actual transition rate for such transfers is very low, whereas the uniformized rate must assume that all jobs are sent to that processor.

It is not difficult to construct scenarios where the uniformization rate is orders of magnitude larger than the actual transition rate. Hyperexponentials that choose rate 1000 with probability 0.001 and choose rate 1 otherwise are one example; a 100-server queue with utilizations less than 1% is another. Such situations induce uniformized communication costs that are orders of magnitude larger than the ones induced by actual job transfer.

Overly high uniformized rates have also been observed in a parallel simulation of a wireless network [Greenberg et al. 1994] when extended to include handoffs. Here the rate at which one transmitting tower a hands calls off to another tower b (due to movement) depends on a's instantaneous load, which depends on b's load, which depends in turn on a's load! Although we could uniformize handoff events assuming each tower has its maximum load, this uniformized rate is much much higher than the normal handoff rate.

As the problem of excessive pseudos is common and the effects may degrade performance severely, we have investigated the approach of uniformizing based on much lower *estimated* bounds, which may turn out to be incorrect. Our hope is that we can observe the simulation to determine what maximal transition rates appear to be, and uniformize at rates only slightly greater than these. This idea is explored more fully in the next section.

3. ADAPTIVE UNIFORMIZATION

In this section we present our algorithm for adaptive uniformization (which we call Adaptive PUCS, or APUCS). Before describing the algorithm, we note that the uniformization approach is valid even if one thins a nonhomogenous Poisson process (NHPP) with nonconstant rate (see Lewis and Shedler [1979] and Shanthikumar [1986]). More specifically, samples from NHPP $\{N_\lambda(t)\}$ with rate $\lambda(t)$ can be generated by thinning a NHPP $\{N_\beta(t)\}$ with rate $\beta(t)(\lambda(t) \leq \beta(t)$ for all $t)$, that is, an event at time T in $N_\beta(t)$ is accepted as an event in $\{N_\lambda(t)\}$ with probability $\lambda(T)/\beta(T)$. (The case that $\beta(t)$ is a constant corresponds to uniformization.) The adaptive uniformization algorithm that we describe next uses this idea of thinning a NHPP.

3.1 Algorithm

The basic idea behind adaptive uniformization is to estimate upper bounds λ_{ij} on the off-processor event rates, and adapt them as a function of the observed behavior of the external event streams. We permit the estimates to

be incorrect, but continuously monitor the actual $P_i \rightarrow P_j$ transition rates and so detect whether the actual transition rate ever exceeds the presumed bound, an occurrence called a *rate fault*. A rate fault causes resimulation, and appropriate reuniformization to avoid a repeat of that fault.

We hope and expect that rate faults are rare, and so desire that the scheme for dealing with them be as simple and inexpensive as possible. We adopt the strategy of checkpointing the simulation state periodically in simulation time, for example, every 250 simulation time units. The period between two checkpoints is called a *window*. Processors synchronize globally at the beginning of a window; then each processor P_i chooses a new uniformization rate $\hat{\lambda}_{ij}$ for every processor P_j whose submodel it may affect. The processors generate and exchange all communication events for the next window's worth of simulation time, and proceed as before. Given $\hat{\lambda}_{ij}$, we adopt a conservative parallel simulation policy within a window, that is, processors wait at all incoming external event times until that incoming event arrives. However, the overall algorithm is optimistic in the sense that we are optimistically assuming that rate faults will not occur. If the end of the window is reached without a rate fault on any processor, then the simulation of that window is valid, and the next window is simulated. However, if a rate fault occurs, the simulation is rolled back to the beginning of the window (which is the time of the last checkpoint), and *every processor* resimulates the window, regardless of whether it experienced a rate fault. Contrast this approach with Time Warp, where only processors that experience temporal faults resimulate. The disadvantage of our approach as compared to Time Warp is the very high cost of resimulation; the advantage of our approach as compared to Time Warp is the very low cost of state-saving—only once per window.

Suppose at least one rate fault occurs on some stream during the course of simulating one window, and suppose that t is the time of the earliest fault. The simulation state past time t is potentially invalid. In order to correct the error without biasing the sample paths, we must resimulate the window so that every processor follows precisely the same sample path as before up to time t. Just prior to t, the faulting streams (several may simultaneously fault) must be uniformized at new rates, at least as large as the actual rates that caused the fault. We support this correction mechanism with the notion of a *uniformization schedule*, maintained for every stream. The schedule specifies uniformization levels, and simulation times at which they take effect. For instance, the schedule might specify that the stream be uniformized at rate 10 for the first 50 time units, then be raised to 17 for 150 units, and finally be raised to 21 for the remainder of the window. When the processor generates communication events for the steam, it samples with one holding time distribution for the first 50 time units, then samples from another for the next 150 time units, and so on. In effect, a NHPP with piece-wise constant rates is used for uniformization of external events.

Each rate fault on a stream causes a new entry in its uniformization list, as follows. A parallel global or-reduction on processors' "rate fault occurred" flag detects whether any steams faulted during the window. If a fault is detected, we use a global min-reduction to determine the time, say t, of the earliest

ones to occur. The processor holding the stream(s) faulting at t is thus able to recognize that its first fault was first in the system, and adds new entries to the streams' uniformization schedule. An entry specifies time t, and a uniformization level at least as large as the actual transition rate that caused the fault. Those streams' communication events are regenerated and sent to their target processors. All processors recover their checkpointed states (which include random seeds so that the same behavior is generated up to t), and attempt to simulate the window again.

The preceding scheme makes no special effort to detect and react to a rate fault quickly. We experimented with schemes that did, and found that performance did not improve. All our experiments suggest that rate faults are so costly (even when detected more quickly), that we should strive to avoid them. All fast fault detection mechanisms have their own additional costs suffered when looking for a possible fault. It is better not to suffer those costs, and better yet not to rate fault.

The issue of choosing a stream's initial uniformization rate for each window is important. Rate faults may result from choosing too low a value; excessive communication may result from choosing too high a value. The fundamental requirement is that we find a uniformization rate that is at least as large as the maximum transition rate that will be achieved by the stream during the next window. Because we cannot know the maximum ahead of time, the problem is to estimate an upper bound on it.

We have investigated a number of schemes involving a base rate λ_b and a multiplier β. The base rate estimates the anticipated maximum rate as a function of observed behavior, and the multiplier is "insurance," as the uniformization rate chosen is $\lambda_b \beta$. The schemes vary in their definition and evolution of $\lambda_b \beta$. Among the methods we considered, an intuitively simple two-phase scheme achieved the best performance, sometimes markedly. Recall that at any time s, $\lambda_{ij}(\mathbf{N}_i(s))$ gives the transition rate of jobs from P_i to P_j, where $\mathbf{N}_i(s)$ is the state of P_i's submodel at time s. Define

$$\lambda_{ij}^{\max}(t) = \max_{0 \le s \le t} \left\{ \lambda_{ij}(\mathbf{N}_i(s)) \right\}. \tag{3}$$

In the first phase, for a window beginning at time t, we use the maximum rate seen so far, $\lambda_{ij}^{\max}(t)$, as the base rate for the next window. Similarly, if a rate fault occurs at time $u \ge t, \lambda_{ij}^{\max}(u)$ is the base rate for the rest of the window. The second phase begins at the first time t' such that if N is the number of $\{N_{ij}(t)\}$ transitions made by time t', then $\lambda_{ij}^{\max}(s)$ remains unchanged throughout the 0.01 Nth to N external transitions. During the second phase we fix $\lambda_b = \lambda_{ij}^{\max}(t')$, with a provision to increase it in the presence of too frequent (e.g., 5%) additional rate faults. Of course, we never permit the uniformization rate to be set higher than its maximum possible rate. (An implicit assumption is that large rates are rare and that the model is not initialized in such a way as to artificially introduce a large rate at the beginning of the simulation.)

The selection of the parameter β is an important issue, which we address experimentally (we have also addressed it theoretically in Nicol and Heidel-

berger [1993b]). Our intuition is that in many models $\lambda_{ij}^{\max}(t)$ will grow slowly after an initial warm-up period, and therefore picking a modest value of β will both protect against rate faults and keep the level of interprocessor communication reasonable. In our experiments, we have observed that $\beta = 2$ has yielded consistently good results. Although it might seem that the uniformization level produced by such a scheme is high, some inflation is unavoidable. In the subsection to follow we examine this issue, identifying conditions under which adaptive uniformization can be expected to achieve good performance while nonadaptive uniformation does not. We also give some insight as to why $\beta = 2$ can be expected to work well in many situations.

3.2 Empirical Studies of APUCS

In this subsection we present experimental results obtained from simulations of the system with clusters of central server models as described in Section 2.1 on both nodes of the Intel Paragon and (up to) 256 nodes of the Intel Touchstone Delta machine. Models of this structure were also used for experimentation in Heidelberger and Nicol [1993] and Nicol and Heidelberger [1993a]. The model has a few simple parameters that can be varied to study the effect on speedup. For example, if each cluster is assigned to its own processor (as is done here), then p_c can be used to control the computation to communication ratio; increasing p_c means that more on-processor events are executed between each off-processor (communications) event.

For the purposes of studying adaptive uniformization, we performed the following types of simple experiments, both of which led to poor PUCS performance.

(1) The service distributions are phase-type and there is a large discrepancy between the phase rates.
Keeping all other parameters fixed, increasing μ_f has the effect of increasing the maximum possible off-processor uniformization rate. As described before, if p_f is small but μ_f is large, then most (busy) I/O devices will be in the slow phase and the typical instantaneous off-processor rate is much less than this maximum possible rate.

(2) The model consists of a large number of servers, most of which are idle. With all other parameters fixed, increasing the number of I/O devices K has the effect of increasing the number of idle servers which again leads to a large difference between the typical instantaneous off-processor rate and the maximum possible rate.

In all experiments, the model parameters were set so that good parallel performance was achievable if enough external pseudo event communications could be eliminated. Speedups were calculated relative to an efficient serial simulator similar to the one described in Heidelberger and Nicol [1993] and Nicol and Heidelberger [1993a]. The models were run for a sufficiently long time that stable estimates of the event processing rate (the number of real events executed per unit of real time) were obtained. For these experiments,

we used a window of size $T = 250$; we comment more on the choice of window size later.

Although simple, the model we study presents a challenge to any performance-oriented study, especially of a conservative synchronization algorithm. There is virtually no locality; every cluster communicates with every other cluster—there are approximately P^2 distinct communication paths to manage when using P processors. In addition, every time a communication occurs there is a $(P - 1)/P$ chance that the communication is between different processors. Furthermore, the model uniformization is consistent with a queueing policy where newly arriving fast jobs preempt slow jobs, a situation that is known to cause problems for conservative simulations. The *only* benign assumption made is that $p_c = 0.99$, an assumption needed to ensure a sufficient computation/communication ratio. With $p_c = 0.99$, there is a healthy computation/ communication ratio proportional to 200 (an average of 100 visits to the CPU and some I/O device before exiting the cluster)—but only in an "optimal" parallel simulation whose only communication costs are those of moving jobs. The actual ratio will be degraded from this level by uniformization. Because of the relatively high cost of message-passing, any application running on machines such as the Paragon and Delta must have a respectable computation/communication ratio to achieve respectable speedups. Finally, the maximal processor size $P = 256$ is significantly larger than that used in most studies, and may be as large as any previous study using MIMD processors. The fact that we do achieve significant performance over optimized serial execution on a difficult problem proves the validity of our methods.

In the experiments, our primary metric of interest is the *event execution rate*, which measures the rate at which **useful** events are executed (per second). We specifically exclude from this rate pseudo events and, for Opt-PUCS, optimistically executed events that are later rolled back. The rates we present are from single runs; this is justified, as in our experience there is very little variation (perhaps 1%) in these execution rates between runs of the same model.

We ran a number of experiments to determine a good value for the uniformization multiplier β. We considered two models with parameter settings chosen so as to observe the two different kinds of behavior previously described. In parameter Set I, $p_c = 0.99$, $u_c = 1000$, $J = 10000$, $K = 20$, $\mu_s = 1$, $p_f = 0.01$, and $\mu_f = 128$, corresponding to Situation 1. In parameter Set II, $p_c = 0.99$, $\mu_c = 10$, $J = 50$, $K = 1024$, and $\mu_s = \mu_f = 1$, corresponding to Situation 2.

Table 1 gives both the event execution rate and a measure of the number of rate faults as a function of β. The rate fault activity is measured by the percentage of extra window simulated compared to the number of windows (assuming no rate faults) that need to be simulated to complete the run. These experiments were run on 16 nodes of the Intel Paragon, with a window size of $T = 250$. On this machine, our best sequential simulator runs at about 12,000 $\pm$ 4%events/second for both Sets I and II, depending on the particu-

Table I. Event Processing Rates and Rate Fault Activity as a Function of β on 16 Paragon Nodes

	Set I: Fast Jobs		Set II: Idle I/O Devices	
β	*% Extra Windows*	*Events/Second*	*% Extra Windows*	*Events/Second*
1.00	33	76,591	18	115,411
1.25	33	70,081	0	144,416
1.50	38	55,427	0	141,573
1.75	12	57,973	0	138,597
2.00	0	88,987	0	135,793
2.25	0	86,191	0	133,791
2.50	0	80,890	0	130,737
2.75	0	77,800	0	129,858
3.00	0	74,130	0	126,816

lar parameter settings. (We implemented several sequential simulators; see Section 4 for a further discussion.)

For these experiments, low values of β results in a significant number of rate faults. (It is worth noting that, if β is too low, we can expect the percentage of windows resimulated to increase significantly as the number of processors increases, inasmuch as only one rate fault in one processor is required to cause resimulation.) Once rate faults are eliminated, the processing rate falls rather slowly as β increases. Observations on data not presented here show that increasing the window size amplifies the cost of choosing β too small. However, if β, is chosen appropriately so as to eliminate most rate faults, then the performance is little affected by the window size. For example, with a window size of 1000 and $\beta = 1.25$, the Set I execution rate is about 45,000 events/second; with $\beta = 2.00$ no rate faults occur and the execution rate is about 80,000 events/second. These observations (and the analysis in Nicol and Heidelberger [1993b]) suggest that β should be set rather conservatively. Setting $\beta = 2$ permits up to double the number of occurrences of whatever phenomenon defined the base rate without incurring a rate fault. For example, in parameter Set I, the base rate corresponds to one (or two) fast jobs in service; $\beta = 2$ permits two (or four) simultaneous fast jobs without a rate fault. Similarly, in parameter Set II, $\beta = 2$ permits double the maximum number of busy servers that are likely to occur in a window without a rate fault.

Based on these results, the remaining experiments used $\beta = 2$ for both phases. Nevertheless, although this policy performed well in all the experiments we conducted, there is still a danger that our $\beta = 2$ selection is application dependent. If this approach were used in production runs, either pilot runs or a different policy that tried to optimize β might be required.

The next experiment was also performed on 16 nodes of the Intel Paragon (with 16 clusters). We fixed $p_c = 0.99$, $\mu_c = 1000$, $J = 10000$, $K = 20$, $\mu_s = 1$, $p_f = 0.01$, and increased μ_f from 2 to 1024. With these settings, the I/O devices are busy nearly all the time in the slow phase. In this experiment, a window of size $T = 250$ contains (on average) approximately 50 outgoing real external transitions per processor. Figure 1 shows the speedup as a function

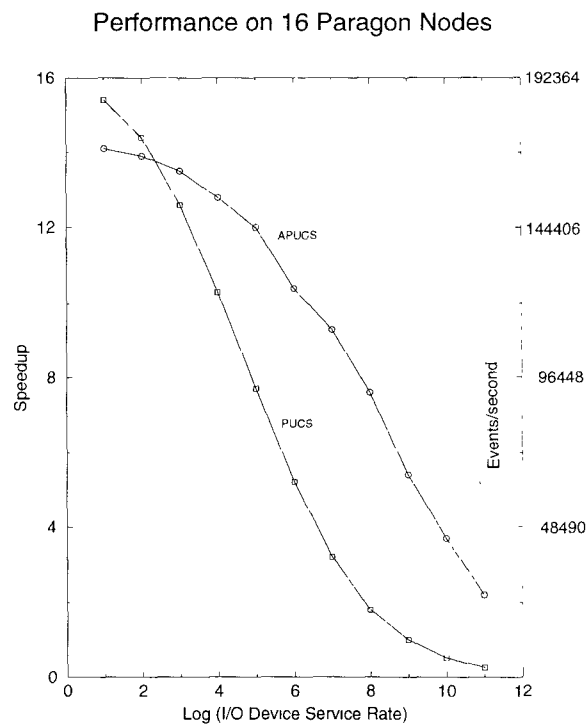

Fig. 1. Speedups on 16 Intel Paragon nodes as a function of μ_f.

of μ_f for both PUCS and APUCS. As expected, the PUCS performance drops rapidly as μ_f increases due to excessive pseudo event synchronizations. The performance of APUCS is less sensitive to μ_f; efficiencies greater than 0.5 (speedups greater than 8) are maintained even when μ_f is two orders of magnitude larger than μ_s. At a three orders of magnitude difference (μ_f = 1024), the APUCS speedup is 2.2 whereas the PUCS speedup (slowdown) is only 0.3. Earlier experiments on a 16-node Intel iPSC/2 hypercube showed that the APUCS speedup drops off more slowly; the speedups at $\mu_f = 16$ and $\mu_f = 1024$ were appoximately 14 and 6, respectively. This difference in speedup reflects the different balance between computation and communications speeds on the machines. The 386-based processor on the hypercube is relatively slow compared to its communications network; our sequential simulator runs at only about 800 events/second on the hypercube. On the other hand, the Paragon has a (relatively) faster 50 MHz processor and high bandwidth interconnection network, although the software overhead to send and receive messages is fairly high.

The third experiment (also performed on 16 Paragon nodes) studied the effect of increasing the number of idle servers. We fixed $p_c = 0.99$, $\mu_c = 10$, $J = 50$, $\mu_s = \mu_f = 1$, and increased K from 32 to 2048. With these settings, the fraction of idle I/O devices increases as K increases. Figure 2 shows the speedup as a function of K for both PUCS and APUCs. Again, the PUCS

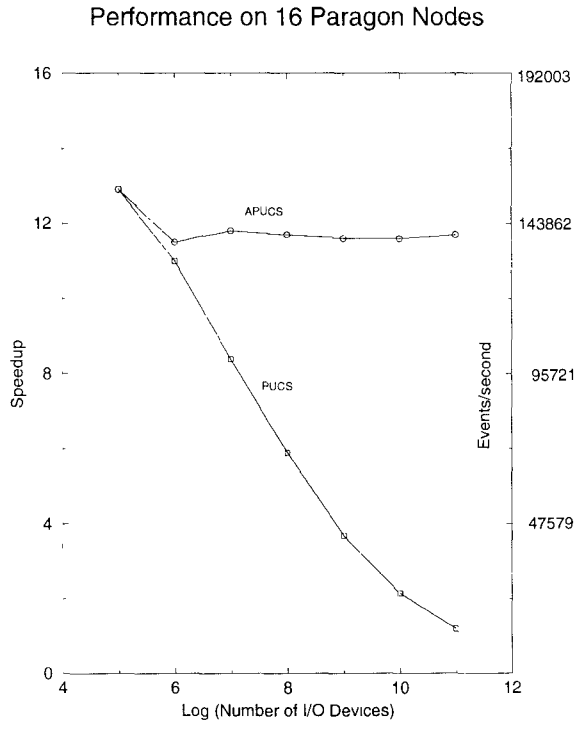

Fig. 2. Speedups on 16 Intel Paragon nodes as a function of the number of I/O devices in a cluster.

performance drops as K increases; however, the APUCS performance is little affected by K with all speedups in excess of 11. In this experiment, the maximum observed off-processor rate is closer to the average off-processor rate than it was in the previous experiment. Thus APUCS is more efficient in this case.

The next experiment was designed to test how well APUCS scales for large models and a large number of processors. This experiment was run on the Intel Touchstone Delta machine with from 32 to 256 processors. The number of clusters was set equal to the number of processors, so that the model size scaled proportionally to the number of processors. We fixed $p_c = 0.99$, $\mu_c = 1000$, $J = 10000$, $p_f = 0.01$, $\mu_s = 1$, $\mu_f = 64$, and $K = 20$. Again, with these settings, most I/O devices are busy in the slow phase. Figure 3 shows the speedup as a function of the number of processors for both PUCS and APUCS. Both PUCS and APUCS performance increase as the number of processors increases. The APUCS speedup stays at about 1.9 times the PUCS speedup throughout these runs. For 256 processors, the APUCS speedup is 155.9, whereas the PUCS speedup is 75.5 corresponding the efficiencies of 0.61 and 0.29. With 256 processors, APUCS processed about 973,000 (real) events/second while PUCS processes about 471,000 events/second.

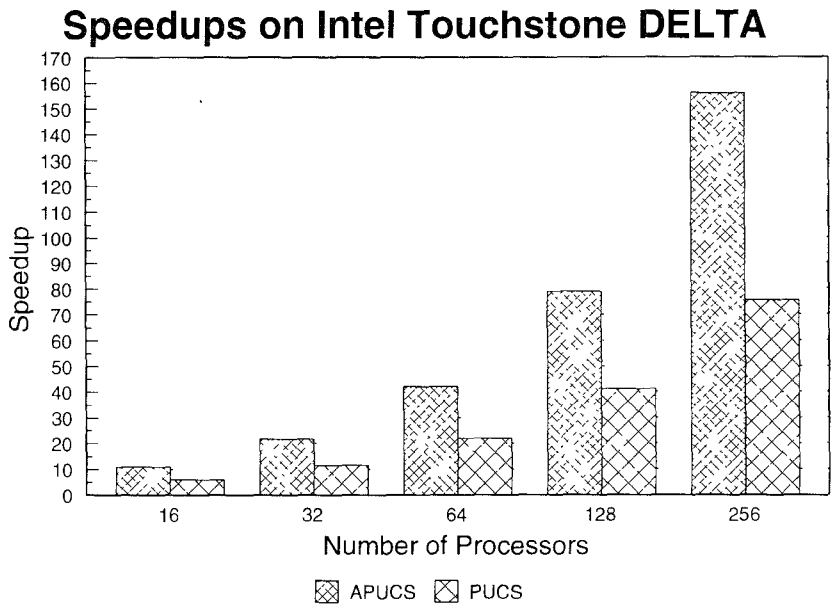

Fig. 3. Speedups on the Intel Touchstone Delta

As mentioned in Section 2 we implemented two versions of ordinary PUCS; one that aggregates all coresident workload into a single LP, and another that permits coresident submodels to be handled separately. The same distinction could, and has, been applied to APUCS. Experiments conducted on the Intel Touchstone Delta showed no advantage to the "partitioned" case (although some experiments conducted on the Intel iPSC/2 do show an advantage). In the remainder we speak only of the "aggregated" APUCS algorithm as presented, with the understanding that an alternative partitioned form is feasible.

4. EXPERIMENTS COMPARING ALTERNATIVE ALGORITHMS

In this section we present the results of experiments performed on the Intel Touchstone Delta multiprocessor [Lillevik 1991], using 16, 64, and 256 processors, as well as results using 16, 32, and 64 nodes of the Intel Paragon. The purpose of these experiments is to compare the performance achieved by the various algorithms described in Section 2.

We continue to study the fully connected network of central server queueing clusters with a hyperexponential I/O service time distribution as previously described. For these experiments we fix the number of I/O servers in each cluster at 20 and fix the probability of a fast job (p_f) to be 0.01. Our study fixed the number of central server clusters at 256. This selection gives us a moderately large simulation model, and also enables us to examine the effects of managing many LPs (up to 16) on a processor. Finally, we set the CPU service rate to 20, and the slow I/O job rate of 1. This ensures that in

steady-state the distribution of jobs will be more or less uniform among all queues.

The parameters we vary are as follows.

—*Number of jobs*: We examine *lightly loaded* scenarios, where there are 10 jobs per cluster (about 0.5 jobs/queue), and *heavily loaded* scenarios where there are 1000 jobs per cluster (about 50 jobs/queue).

—μ_f: We examine a fast job rate of 1 (so there is no distinction between fast and slow jobs), and a fast job rate of 8. The latter selection, coupled with $p_f = 0.01$, induces moderately high rates of uniformization relative to actual stream transition rates.

—*Number of processors*: We study our models on 4×4, 8×8, and 16×16 submeshes of the Delta.

Every experiment was run long enough so that every processor executed approximately 0.5 million events.

Before analyzing the results of our experiments, we address the issue of "speedup." Speedup is intended to measure the user's benefit of running the parallel algorithm. For this reason, one ought to compare parallel performance to that of an optimized serial algorithm. Some difficulties arise, however, when the serial algorithm which is optimal changes as the problem parameters of interest change. To illustrate the point, Table II gives serial execution rates (on the Delta) as a function of problem characteristics, for an optimized serial direct Markovian simulation, and CP-PUCS run on one processor. Although PUCS on one processor is faster by almost 20% on one set of parameters, it is slower by 33% on another. By comparison, the optimized serial algorithm varies by only a few percent over these problems. A user is far more likely to choose a serial algorithm that is consistently good over one whose performance varies so widely.

Table III presents the results of our experiments. Simulation is executed on 16, 64, and 256 processors of the Intel Delta. Without resorting to a definition of speedup, we *can* say that on the heavily loaded problem with $\mu_f = 1$ using 256 processors, CP-PUCS is 260 times faster than the particular serial simulator we used, and is 221 times faster than its own one-processor implementation (and 14 times faster than its 16-processor implementation). In either case, it is clear that a very substantial improvement over serial execution is being achieved. As an additional point of comparison, we measured the execution rate of the commercial queueing network simulator RESQ [Gordon et al. 1991], executing on an IBM 3090 mainframe. The model simulated by RESQ is actually substantially smaller than this one, having only 16 clusters (due to memory constraints). The RESQ execution rate is only 1,781 events/second compared to PUCS execution rates in excess of 1,500,000 events/second. Of course, one must take into account that RESQ is an industrial quality simulator able to handle a wide range of problems, whereas the PUCS code is handcrafted and optimized, with a much more restrictive domain. Nevertheless, this comparison illustrates parallel simulation's tremendous potential for accelerating solution times.

Table II. Execution Rates (events/sec) of Optimized Serial Algorithm and PUCS Running on One Intel Delta Processor

(load, μ_f)	*Optimized Serial Algorithm*	*PUCS on One Processor*
(light, 1)	6211	7014
(heavy, 1)	6563	7706
(light, 8)	6219	4166
(heavy, 8)	6554	6469

Table III. Execution Rates (events/second) of Fully Connected Model of 256 Central Server Clusters With $p_c = 0.99$, $p_f = 0.01$

	16 Processors		64 Processors		256 Processors	
	light	*heavy*	*light*	*heavy*	*light*	*heavy*
			Fast Job Rate = 1			
CA-PUCS	80,032	102,504	301,765	411,362	985,327	1,575,146
CP-PUCS	109,585	122,186	378,329	393,418	1,043,609	1,709,567
Opt-PUCS	103,707	121,609	343,510	353,873	874,617	855,737
APUCS	79,711	102,329	311,168	403,380	989,351	1,567,038
			Fast Job Rate = 8			
CA-PUCS	53,339	76,580	181,785	299,660	668,323	1,147,282
CP-PUCS	58,753	90,708	202,205	311,920	457,252	934,120
Opt-PUCS	57,314	89,642	167,382	328,711	445,880	802,732
APUCS	74,580	90,857	258,018	352,763	851,204	1,304,502

Fast job service rate is varied between 1 and 8; average number of jobs per cluster is varied from 10 (light) to 1000 (heavy). Simulation is executed on 16, 64, and 256 processors of the Intel Delta.

We also ran this model on 16, 32, and 64 nodes of the Paragon. For the heavily loaded case with $\mu_f = 1$, the APUCS execution rates were approximately 168,000, 315,000, and 615,000 events/second, respectively, whereas with $\mu_f = 8$, the APUCS execution rates were approximately 154,000, 304,000, and 590,000 events/second, respectively.

We next analyze the Delta data with an eye towards addressing the issues of aggregation, communication costs, optimism, and adaptiveness.

4.1 CP-PUCS Versus CA-PUCS

Our earlier studies of CA-PUCS and CP-PUCS (on an Intel iPCS/2) indicated that the CP-PUCS overheads of managing multiple LPs and of internal pseudos between on-processors clusters outweighed the advantages of increased opportunity for parallelism and avoidance of synchronous appointment generation. Yet the data in the present study shows that this is not always true. Consider Table IV which gives the ratio of CP-PUCS rates to CA-PUCS rates, as a function of problem characteristics and architecture size.

The overall trend is for CP-PUCS to outperform CA-PUCS, but there are still instances where the reverse is true.

Table IV. Ratio of CP-PUCS/CA-PUCS Execution Rates

(*load*, μ_f)	16 *Processors*	64 *Processors*	256 *Processors*
(light, 1)	1.37	1.25	1.05
(heavy, 1)	1.19	0.95	1.08
(light, 8)	1.10	1.11	0.68
(heavy, 8)	1.18	1.35	0.81

CP-PUCS and CA-PUCS differ both with respect to aggregation, and with respect to message handling. As such, it is difficult to separate the influences of aggregation and communication costs. Furthermore, the communication costs will depend on the underlying architecture, as well as the operating system. There are at least four factors to take into consideration, which sometimes interact in a complex manner.

—An LP's execution time-slice is delimited by communication instants. When $\mu_f = 8$ the uniformization rate is eight times larger, so that there are eight times as many communication instants per unit time. An LP's execution time-slice is much shorter, so that the overhead of switching between LPs is suffered eight times as often.

—In the lightly loaded experiments (and those where $\mu_f = 8$), most communications report pseudo events. Thus, when CA-PUCS blocks, it usually waits for a communication that does not affect its state. There is thus no useful purpose gained by blocking, other than the assurance of logical correctness. CP-PUCS is better able to find and execute useful work, when such work exists.

—As we increase the number of processors we decrease the number of clusters on a processor. This increasingly limits CP-PUCS' ability to find useful work that CA-PUCS cannot find. Of course, at 265 processors, both CP-PUCS and CA-PUCS each have one cluster per processor, and thus behave identically with respect to synchronization.

—CA-PUCS has a global step where synchronization appointments are generated and exchanged. Its performance will thus be affected by the efficiency with which an all-to-all exchange can be performed, and by the frequency of this exchange. CP-PUCS has no corresponding cost.

Let us examine performance with these factors in mind. On these experiments CP-PUCS tends to perform better. Apparently, on this model, the scheduling and appointment generation advantages outweigh CA-PUCS advantages. The difference between the two tends to diminish as the number of processors increases, which is consistent with the fact that (i) the CP-PUCS scheduling advantage gets smaller as a processor has fewer and fewer clusters, and (ii) in a CA-PUCS appointments exchange, essentially the same communication workload is spread over more network hardware, reducing the frequency of collisions and blocking. Thus, as the number of processors increases, the CA-PUCS advantage diminishes and the CP-PUCS disadvan-

tage diminishes. However, there are clearly other factors at work, as the performance differences change neither smoothly nor monotonically as the number of processors increases.

Our earlier comparison of CP-PUCS and CA-PUCS found CA-PUCS to be clearly superior. One explanation is that the models studied are different in an important way. The earlier model appends 10 "local clusters" of queues to every central server queue. In those studies, $p_c = 0.0$, and a job leaving an I/O device can be routed either to another central server cluster (with probability p_{cc}) or to one of its local clusters. Upon leaving the local cluster the job returns to the same central server. This model provides another way of boosting the computation/communication ratio, because a local cluster is always mapped to the same processor as its parent central server cluster. Our previous study varied the probability p_{cc} of routing a job from one central server to another one, on a different processor. As p_{cc} increases, CP-PUCS performance drops faster than that of CA-PUCS, because CP-PUCS suffers increasingly from internal pseudo transitions between a central server and its local clusters. The present set of experiments is somewhat kinder to CP-PUCS, as the level of interaction between coresident LPs is much lower. It seems then that the level of internal uniformization is the deciding factor between CA-PUCS and CP-PUCS. This implies that close attention must be paid when partitioning a simulation model into LPs for PUCS, perhaps deciding which style of synchronization to use as a function of uniformization rates.

4.2 Whither Optimism?

These experiments offer clear insight into the potential of exploiting optimism in PUCS, because the only substantive difference between CP-PUCS and Opt-PUCS is the optimistic processing. Towards this end, Table V computes the ratio of CP-PUCS to Opt-PUCS execution rates.

The first thing we notice is that CP-PUCS tends to do a little better than Opt-PUCS. Next we notice that the degree to which CP-PUCS does better tends to increase as the number of processors increases. Indeed, for all practical purposes, the performance on 16 processors is identical; yet, at 256 processors, on one case CP-PUCS was nearly twice as fast as Opt-PUCS.

Explanations for this behavior are found by looking at the cost suffered by executing optimistically, primarily event re-execution and state-saving. Table VI computes the ratio of the number of total events (excluding pseudos) executed to the number of events (excluding pseudos) committed. One can also view this as the average number of times a nonpseudo event is executed. The table also computes the average number of state-saves per committed nonpseudo event.

One thing clearly shown is that, in this example, the cost of saving the state of one central server cluster (about 3,000 bytes) is usually amortized over many events. Its effect on performance must be negligible. Any significant differences between CP-PUCS and Opt-PUCS are related to the cost of

Table V. Ratio of CP-PUCS/Opt-PUCS Execution Rates

(load, μ_f)	16 *Processors*	64 *Processors*	256 *Processors*
(light, 1)	1.05	1.10	1.19
(heavy, 1)	1.00	1.10	1.99
(light, 8)	1.02	1.20	1.02
(heavy, 8)	1.01	0.90	1.16

Table VI. Overheads Associated With Opt-PUCS

	Total/Committed Events			Average State Saves/Event		
(load, μ_f)	16 *Processors*	64 *Processors*	256 *Processors*	16 *Processors*	64 *Processors*	256 *Processors*
(light, 1)	1.11	1.19	1.68	0.008	0.010	0.027
(heavy, 1)	1.03	1.40	2.10	0.001	0.002	0.007
(light, 8)	1.01	1.06	1.34	0.060	0.100	0.017
(heavy, 8)	1.01	1.04	1.27	0.004	0.015	0.041

rolling back and re-executing events. Indeed, there is a direct correlation between high event execution ratios and significant gaps between CP-PUCS and Opt-PUCS.

Because re-execution costs define the difference between CP-PUCS and Opt-PUCS, it is simple to explain why the gap between them increases as the number of processors increases. On only 16 processors, many LPs are assigned to the same processor, and thus Opt-PUCS has a good chance of being able to schedule a **sure** cluster. However, for a large number of processors there are relatively few LPs on a processor. Without a large number of LPs, a processor quickly executes its **sure** workload and is left to forge ahead optimistically. Apparently its optimism is frequently misplaced, and significant fractions of events end up being resimulated. This effect is somewhat lessened when there are many pseudo events, as in such cases the optimistic assumption that the event is a pseudo event is, in fact, correct.

4.3 Adaptivity

Pseudo events are the largest source of performance degradation in all versions of PUCS. Many CTMC models have characteristics that cause the best upper bound on an external event stream's transition rate to be very far from the stream's average transition rate. In our experiments fast jobs appear infrequently, and one almost never sees more than three simultaneous fast jobs in a central server cluster. Yet the uniformization bound must be based on the assumption that all servers are busy with fast jobs.

Table VII illustrates the sensitivity of each method to increased uniformization, by computing the ratio of its execution rate using $\mu_f = 1$ to its rate using $\mu_f = 8$. These data show clearly that APUCS is more tolerant of increased uniformization than are the other methods. Similar observations held in our previous study of APUCS that varied μ_f more widely, up to

Table VII. Ratio of $\mu_f = 1$ to $\mu_f = 8$ Execution Rates

	Light Load			Heavy Load		
Algorithm	16 *Processors*	64 *Processors*	256 *Processors*	16 *Processors*	64 *Processors*	256 *Processors*
CA-PUCS	1.50	1.66	1.47	1.34	1.37	1.37
CP-PUCS	1.86	1.87	2.28	1.34	1.25	1.83
Opt-PUCS	1.80	2.05	1.96	1.35	1.07	1.06
APUCS	1.07	1.20	1.16	1.12	1.14	1.20

$\mu_f = 1024$. Even at levels of $\mu_f = 256$, APUCS gives a respectable performance whereas CA-PUCS performance has thoroughly degenerated. We believe that any standardized version of PUCS must include adaptivity if it is to work on a wide range of problems.

5. CONCLUSIONS

This paper looked at the problem of parallelizing the simulation of continuous time Markov chains. We showed how the notion of uniformization can be applied so that the simulation can be conducted by essentially precomputing an inter-LP synchronization schedule, and then simulating a mathematically correct sample path through that schedule. This basic method is called PUCS. We described four different PUCS variations, and examined performance on a parameterized model designed to illustrate their respective strengths and weaknesses. The experiments were conducted on a variety of Intel multiprocessors, including the Intel Touchstone Delta using 16, 64, and 256 processors.

The results of these experiments, taken in conjunction with others previously conducted, imply that an optimized PUCS algorithm ought to incorporate conservative synchronization and adaptive uniformization rates. On the issue of aggregation, although partitioned PUCS sometimes runs faster than the aggregated PUCS, it rarely runs faster than the adaptive, aggregated PUCS. Thus, if the workload is well balanced, there seems to be little advantage in patitioning the submodel assigned to a processor into multiple LPs. The performance we observe can often be quite good, depending on the problem characteristics. However, PUCS performance *is* inescapably dependent on the number of pseudo events, and every effort must be made to reduce these.

Although our experiments prove the promise of PUCS, some important issues remain open. In particular, we have not addressed automated partitioning, automated load balancing, or the effect one has on the other. There are several interesting fronts of future research. The class of models for which a PUCS-style approach is valid is broader than just CTMCs (e.g., strictly internal processes need not be Markovian and the exponential distribution assumption can be relaxed for external events provided the hazard rate of the distribution is bounded). An interesting problem is to incorporate and take advantage of uniformization-based synchronization within other

synchronization protocols for supporting models in which only some of the external processes can be uniformized. The requirement for such a protocol would arise for a model in which some service time distributions are Coxian phase type, whereas others are constant, discrete, or uniform. Another interesting problem is making the results of uniformization-based synchronization practically available to a simulation modeler, in the context of a general-purpose simulation tool. We are dealing with both these issues in the context of a tool we call the Utilitarian Parallel Simulator (U.P.S.). U.P.S. extends parallel processing to an existing serial simulator, CSIM [Schwetman 1986], through libraries used in conjunction with CSIM models. A preliminary report appears in Nicol and Heidelberger [1995]. We hope to address partitioning and load-balancing problems in U.P.S. as well.

ACKNOWLEDGMENTS

This research was performed in part using the Intel Touchstone Delta System operated by Cal. Tech. On behalf of the Concurrent Supercomputing Consortium. Access to this facility and to the Intel Paragon was provided by the NASA Langley Research Center.

REFERENCES

BUZEN, J. P. 1973. Computational algorithms for closed queueing networks with exponential servers. *Commun. ACM 16*, 9 (Sept.), 527–531.

FUJIMOTO, R. M. 1990. Parallel discrete event simulation. *Commun. ACM 33*, 10, 31–53.

FUJIMOTO, R. M., TSAI, J.-J., AND GOPALAKRISHNAN, G. C. 1992. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Trans. Comput. 41*, 1 (Jan.), 68–82.

GORDON, K. J., GORDON, R. F., KUROSE J. F., AND MACNAIR, E. A. 1991. An extensible visual environment for construction and analysis of hierarchically-structured models of resource contention systems. *Manage. Sci. 37*, 6 (June), 714–732.

GREENBERG, A., LUBACHEVSKY, B., NICOL, D., AND WRIGHT, P. 1994. Efficient massively parallel simulation of dynamic channel assignment schemes for wireless cellular communications. In *Proceedings of the 1994 Workshop on Parallel and Distributed Simulaton* (Edinborough, Scotland), 187–194.

GROSS, D., AND MILLER, D. R. 1984. The randomization technique as a modeling tool and solution procedure for transient Markov processes. *Oper. Res. 32*, 2 (March–April), 343–361.

HEIDELBERGER, P. AND NICOL, D. M. 1993. Conservative parallel simulation of continuous time Markov chains using uniformization. *IEEE Trans. Parallel Distrib. Syst. 4*, 8 906–921.

INTEL CORP. 1993. *Paragon User's Guide*, Order Number 312489-002. Oct.

JEFFERSON, D. R. 1985. Virtual Time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July), 404–425.

LEWIS, P. A. W. AND SHEDLER, G. S. 1979. Simulation of nonhomogeneous Poisson processes by thinning. *Naval Res. Logistics Q. 26*, 3 (Sept.) 403–413.

LILLEVIK, S. L. 1991. The Touchstone 30 gigaflop DELTA prototype. In *Proceedings of the 1991 Distributed Memory Computer Conference* (April), IEEE Press, Piscataway, NJ, 671–677.

LUBACHEVSKY, B. D. 1990. Simulating colliding rigid disks in parallel using bounded lag without Time Warp. *Distri. Simul. 1990, 22*, 2. (Simulation Series), (Jan.), 194–204.

LUBACHEVSKY, B. D. 1987. Efficient parallel simulations of asynchronous cellular arrays. *Complex Sys. 1*, 1099–1123.

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

NICOL, D. M. AND FUJIMOTO, R. 1994. Parallel simulation today. *Ann. Oper. Res.* (Dec.), 249–286.

NICOL, D. M. AND HEIDELBERGER P. 1995. On extending parallelism to serial simulators. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation (PADS95)* (Lake Placid, NY, June 14–16), IEEE Computer Society Press, Los Alamitos, CA, 60–67.

NICOL, D. M. AND HEIDELBERGER, P. 1993a. Optimistic parallel simulation of continuous time Markov chains using uniformization. *J Parallel Distrib. Comput. 18*, 4 (Aug.), 395–410.

NICOL, D. M. AND HEIDELBERGER, P. 1993b. Parallel simulation of Markovian queueing *networks*. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, ACM Press, New York, 135–145.

NICOL, D. M. AND HEIDLEBERGER, P. 1993c. Parallel algorithms for simulating continuous time Markov chains. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS93)*, IEEE Computer Society Press, Los Alamitos, CA, 11–18.

RATTNER, J. 1985. Concurrent processing: a new direction in scientific computing In *AFIPS Conference Proceedings, National Computer Conference*, vol. 54, 157–166.

RIGHTER, R. AND WALRAND, J. V. 1989. Distributed simulation of discrete event systems. *Proc. IEEE 77*, 1 (Jan.), 99–113.

SCHWETMAN H. 1986. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, 387–396.

ROSS, S. 1983. *Stochastic Processes*. John Wiley and Sons, New York.

SHANTHIKUMAR, J. G. 1986. Uniformization and hybrid simulation/analytic models of renewal processes *Oper. Res. 34*, 4 (July–Aug.), 573–580.

WEST D. 1988. Lazy rollback and lazy reevaluation. Univ. of Calgary, M.S. Thesis, Jan.

Received May 1994; revised June 1995; accepted July 1995.