Parallel Discrete-Event Simulation Of FCFS Stochastic Queueing Networks

David M. Nicol*
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185

Abstract

Physical systems are inherently parallel; intuition suggests that simulations of these systems may be amenable to parallel execution. The parallel execution of a discrete-event simulation requires careful synchronization of processes in order to ensure the execution's correctness; this synchronization can degrade performance. Largely negative results were recently reported in a study which used a well-known synchronization method on queueing network simulations. In this paper we discuss a synchronization method, *appointments*, which has proven itself to be effective on simulations of FCFS queueing networks. The key concept behind appointments is the provision of *lookahead*. Lookahead is a prediction on a processor's future behavior, based on an analysis of the processor's simulation state. We show how lookahead can be computed for FCFS queueing network simulations, give performance data that demonstrates the method's effectiveness under moderate to heavy loads, and discuss performance trade-offs between the quality of lookahead, and the cost of computing lookahead.

*This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-18107 while the author consulted at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1988 ACM 0-89791-276-4/0007/0124 $1.50

1 Introduction

Physical systems are inherently parallel; intuition suggests that simulations of these systems may be amenable to parallel execution. The parallel execution of a *discrete-event simulation* [2] requires careful synchronization of processes in order to ensure the execution's correctness. A number of synchronization methods have been proposed; some have been studied empirically. With few exceptions the evidence is that overhead inherent in these methods prevents any significant performance benefit from parallel execution.

Queueing network simulations provide a stress test for parallel discrete-event simulation because so little computation is associated with each event. Parallel queueing network simulations are also interesting from a historical point of view, as much of the early work in this field implicitly uses a queueing network model for the simulation. The seminal work in parallel simulation by Chandy and Misra[1] identified the concept of *lookahead* as being sufficient to avoid logical deadlock between processors. Lookahead is the ability of a process to predict (possibly minutely) those aspects of its future behavior which affect the synchronization requirements of other processes. Implementations of the Chandy/Misra algorithms invariably create a lookahead ability by requiring that each job receive a minimum service time ϵ. Knowledge that a future job requires at least ϵ service allows a processor to predict that a job which arrives immediately will not depart for at least ϵ time. Because most probability distributions of interest are not bounded from below, implementations must choose ϵ to be very small. Performance studies[6,14] have strongly suggested that this poor lookahead ability leads to dismal performance due to extremely high synchronization overhead.

In [11] we proposed that more extensive lookahead be calculated by analyzing a process's simulation state, and showed how this could be accomplished in both queueing network simulations, and logic network simulations. In [12] we examined the effect that increased lookahead has on overall performance. More recently Fujimoto re-examined the Chandy/Misra algorithms and focused on increasing lookahead ability by increasing ϵ. His results are more encouraging, but poor performance is still observed when the ratio of mean service time to ϵ is high (say, 10). Lubachevsky[9] also uses lookahead which is computable under the "bounded-lag" and minimum service time assumptions. While he does not report any empirical results, one can expect his scheme to suffer from similar failings as the Chandy/Misra algorithms as bounded-lags are not present in queueing networks, and reliance on minimum service times has already been shown to yield poor performance.

The purpose of this paper is to point out the feasibility of using detailed simulation-specific information to compute lookahead in stochastic FCFS queueing network models. Unlike past treatments of parallel queueing network simulations this lookahead does not rely upon a minimum service time. We discuss the trade-offs between lookahead quality and the cost of computing it, and use a parallel implementation of our method to show that under moderate to heavy loads a protocol based on lookahead can yield good performance on simulation models that have defeated other protocols. Fujimoto has independently performed a similar study[1].

Every processor in a parallel discrete-event simulation maintains its own simulation clock, and its own event list. A simple example clearly illustrates the need for synchronization. Figure 1 depicts the simulation of a three queue network on three processors. Queue Q_1 sends a job to queue Q_2 with a time-stamp of 10. The first event in Q_2's event list is the one which accepts this job. Simulation correctness is ensured if, within every processor, the simulation time order of evaluated events is monotonically increasing. To ensure this monotonicity Q_2 does not process the first event in its event list until it is certain that some other event with a smaller time-stamp will never be inserted into the event list. Such an event might occur, for example, if at time 1 an external arrival appears at Q_3, is given 2 units of service, and then is routed to Q_2. The role of a conservative synchronization protocol is to coordinate Q_1, Q_2, and Q_3 so every processor evaluates events monotonically in simulation time, so that a processor evaluates the first event in its list as soon as it is safe to do so, and so that system deadlock is avoided (or detected/corrected). It should be mentioned that *optimistic* synchronization is currently a topic of active study[7]; under an optimistic protocol Q_2 would process the first event in its list with the expectation that no job with a smaller time-stamp will appear. If one does appear, then corrective measures must be taken. The protocol discussed in this paper is conservative.

Discrete-event simulation synchronization protocols are typically described in terms of message passing behavior between *logical processes (LP)*'s, or the subsystems modeled by processors. Associated with each LP is a set of *readers* and a set of *writers*. LP_i is a writer for LP_j if it is possible for the processing of an event in LP_i to cause a "message" to be sent to LP_j, who in turn modifies the event list in LP_j. In this case LP_j is a *reader* for LP_i. It is useful to distinguish between "content" and "protocol" messages. As the titles suggest, a content message directly concerns the simulation and its state while a protocol message concerns only the implementation of the synchronization protocol. In the example above a content message from Q_1's processor to Q_2's processor causes the insertion of the event in Q_2's processor's event list. At some point Q_3 might send a protocol message to Q_2 promising that it will send no jobs with a time-stamp less than 15 (although we have not yet identified how Q_3 can provide such a promise), thereby allowing Q_2's processor to evaluate the arrival at time 10. Protocol messages may themselves be time-stamped.

In previous protocols [13,1,15] a protocol message from LP_i to LP_j with a time-stamp of t provides a promise that LP_i's next message to LP_j (a message which may cause modification of LP_j's event list) will have a time-stamp no greater than t. The established protocols vary in their details, but all share a distinctive characteristic: the protocol mechanism is largely independent of the system being simulated. This generality is attractive, but requires that an *LP's* decision to send a protocol message with a time-stamp of t is based solely on the time-stamps of protocol and content messages that the LP has received. To ensure the protocol's generality information about the simulation state, or how an LP responds to a content message is not used. As a consequence many

[1]Private communication from Richard Fujimoto.

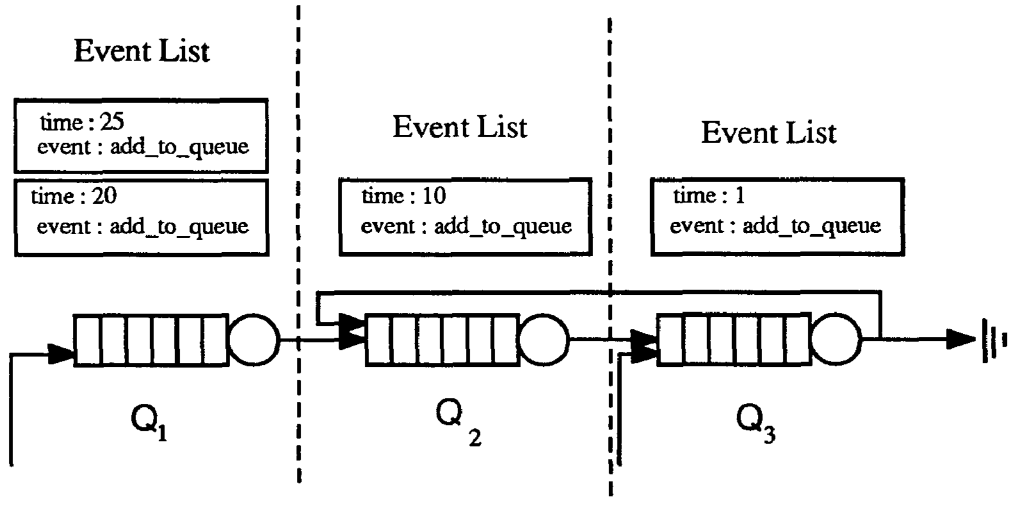


Figure 1: Distributed Simulation of Three Queues

protocol messages must be exchanged, as each protocol message allows the simulation to precede only incrementally. Studies of the "Null Message" method have shown that the ratio of protocol messages to content messages is very high. Reed's recent empirical study of this method on queueing network simulations shows it to be of limited utility[14].

The role of a protocol message from LP_i to LP_j is to provide a lower bound on the simulation time at which LP_i may next affect LP_j's event list. The quality of this bound depends on LP_i's ability to predict its future behavior. In the quest for generality, the previous synchronization protocols fail to take advantage of knowledge about the simulated system. A better bound on future behavior can be obtained by analyzing the LP's simulation state to find lookahead. The section to follow outlines a synchronization protocol that relies upon the computation of simulation-specific lookahead.

2 The Appointment Protocol

Before discussing means of identifying lookahead we will introduce the synchronization protocol that uses it. A small number of definitions must first be given. LP_i's simulation clock is denoted C_i; LP_i's event list is denoted E_i, and is assumed to be ordered by increasing time-stamps. e_i denotes the event at the head of E_i, and f_i denotes its time-stamp. We assume the usual relationship between C_i and E_i—just prior to processing the event e_i with time-stamp f_i, C_i is advanced to f_i.

A serial simulation repeatedly executes a three-step cycle: advance the simulation clock to the time-stamp f_i of the first event in the event list e_i, process e_i (which may alter the event list, but will never add events with time-stamps less than f_i), and remove the event just processed. LP_i in a parallel simulation must not process e_i until it is certain that none of LP_i's writers will cause an earlier event than e_i to be inserted into E_i. The mechanism we use to prevent LP_i from processing an event "too early" is the *appointment*. Every one of LP_i's writers provides LP_i with an appointment time beyond which LP_i will not advance its clock without further permission. An appointment that LP_i gives LP_j is denoted A_{ij}; we denote the set of all appointments given to LP_j by $\{W_j\}$. Only an LP's writers must supply it with appointment times.

Figure 2 gives high level pseudo-code describing the appointment protocol. We have left unspecified other necessary mechanisms, e.g. asynchronous message-passing routines to update appointment values and modify the event list. For clarity we have also left unspecified direct optimizations which ensure that a new appointment is not requested before the last such request was satisfied. Like all conservative synchronization protocols, this one prevents the processing of an event if there is any chance that an event with a smaller time-stamp will be inserted into the event

Definitions

C_i	Value of LP_i's simulation clock
E_i	LP_i's event list
e_i	First event on E_i
f_i	Time-stamp of first event on E_i
A_{ki}	Appointment provided by writer LP_k to reader LP_i
$\min\{W_i\}$	Minimum over all appointments given to LP_i by its writers

```
Loop {
        If ( $f_i \leq \min\{W_i\}$ )
                { $C_i = f_i$;
                  Process event $e_i$;
                  Remove $e_i$ from $E_i$;
                }
        Else
                { For every writer $LP_k$
                        If ( $A_{ki} < e_i$ )
                                Request a new appointment from $LP_k$;
                  For every reader $LP_j$
                        If ( $LP_j$ has requested a new appointment )
                                Compute and send a new appointment $A_{ij}$;
                }
  } Forever
```

Figure 2: Appointment Synchronization Pseudo-code

list.

The ability of this protocol to reduce synchronization overhead to acceptable levels clearly depends on the ability to provide lookahead. A queueing network often has structure which allows a queue Q_A to periodically provide upper bounds on the times at which it will route jobs to other queues. The aggregation of these bounds form the basis of an appointment. The sections to follow show how various degrees of lookahead can be computed in queueing network simulations.

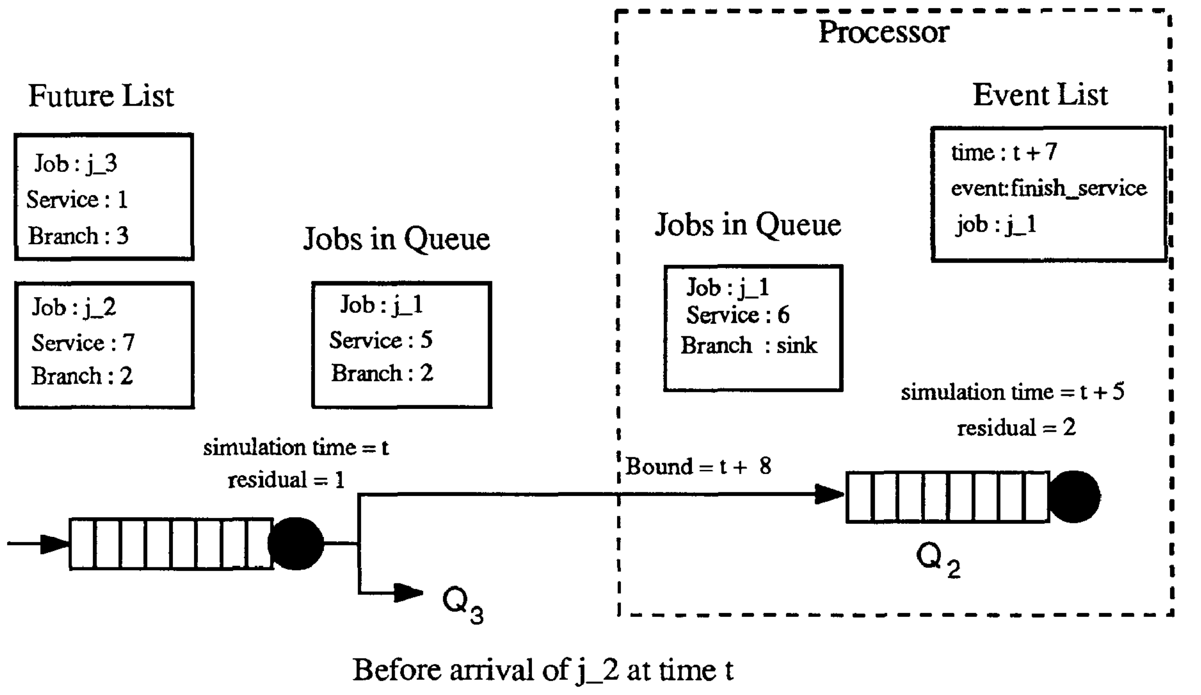
3 Lookahead in FCFS Queueing Networks

Lookahead is easily computed in a stochastic simulation of a network of FCFS queues. The simulation is distributed by assigning queues to processors. Depending on the size of the queueing network, a processor may be assigned several queues. A processor is responsible for simulating the queuing activity of each of its queues, and for maintaining all statistical information collected about the queues' behavior. An LP then consists of the possibly fragmented subnetwork assigned to a processor. It is important to note that past treatments of parallel queueing simulations have treated each queue individually as an LP; this invariably leads to high overhead because synchronization costs are suffered on a per-LP basis.

A typical simulation of a queue requires three event handlers: `AddToQueue`, `BeginService`, and `FinishService`. The random service time of a job entering service is traditionally sampled by `BeginService`, and the destination of the completed job is traditionally chosen by `FinishService`. A serial simulation gains nothing by choosing the service time and branching destination any sooner than required. For the purposes of computing lookahead there is much to be gained by choosing them earlier. Our ability to do so depends in large part on the model assumptions. In the simplest but most common type of stochastic simulation the service time of every job at a queue is drawn from a common distribution and the branching destination is chosen from a common distribution. Note that these quantities could be drawn at any time—it can be advantageous to select a job's service time and branching destination *before* the job arrives. For example, if at time t queue Q_A has no jobs enqueued for Q_B but it is known that the next job which branches to Q_B has service time s, then Q_A will send no jobs to Q_B before time $e_A(t) + s$, where $e_A(t)$ is the time at which Q_A will next be empty if no further arrivals occur: t plus the sum of service times of all jobs in queue at time t. $e_A(t) + s$ is a sharp bound if the next job arrives prior to time $e_A(t)$, and has Q_B chosen as its branching destination.

The observation above led us to an organization which associates with every queue a *future list* of jobs which have not yet arrived. A job's service time and branching destination are determined when it joins the future list. The future list is kept large enough so that it contains a job for every possible branching destination. When the event handler `AddToQueue` is called at simulation time t to simulate a job arrival at Q_A, the first job in Q_A's future list is removed and is used to represent the arrival. If that job branches to Q_B, and its removal empties the future list of jobs which branch to Q_B, then additional jobs are appended to the future list in a manner which preserves the statistical integrity of the simulation—jobs with randomly selected service times and branching destinations are appended to the future list until a job with destination Q_B is added. Note also that once a job J_B arrives at Q_A its arrival time at the next queue Q_B is already determined; consequently the processor holding Q_B may be immediately informed of J_B's arrival there. This is advantageous when Q_A and Q_B reside in different processors, as it may allow Q_B to simulate J_B's arrival ahead (in real time) of its simulated departure from Q_A. After computing J_B's arrival time at Q_B, we compute a lower bound on the time of Q_A's next, as yet unseen job to Q_B, called J_{Next}. A description of J_{Next} is found in Q_A's future list. Because Q_A is FCFS, we know that J_{Next} cannot depart at least until all jobs current enqueued at Q_A receive service, at time $e_A(t)$. Furthermore, J_{Next} does not receive service until every job ahead of it in the future list receives service. Letting S be the sum of service times of all jobs ahead of and including J_{Next} in the future list, $e_A(t) + S$ is then a lower bound on the time that Q_A will next route a job to Q_B. This bound is cheaply computed, and is passed to Q_B's processor along with the message reporting the arrival of J_B. Figure 3 illustrates these points, and a possible transformation of a queue and its future list upon the simulated arrival of a job. Figure 4 outlines the roles played by the the event handlers in this scheme.

It is apparent from the description above that lookahead information is continually being computed



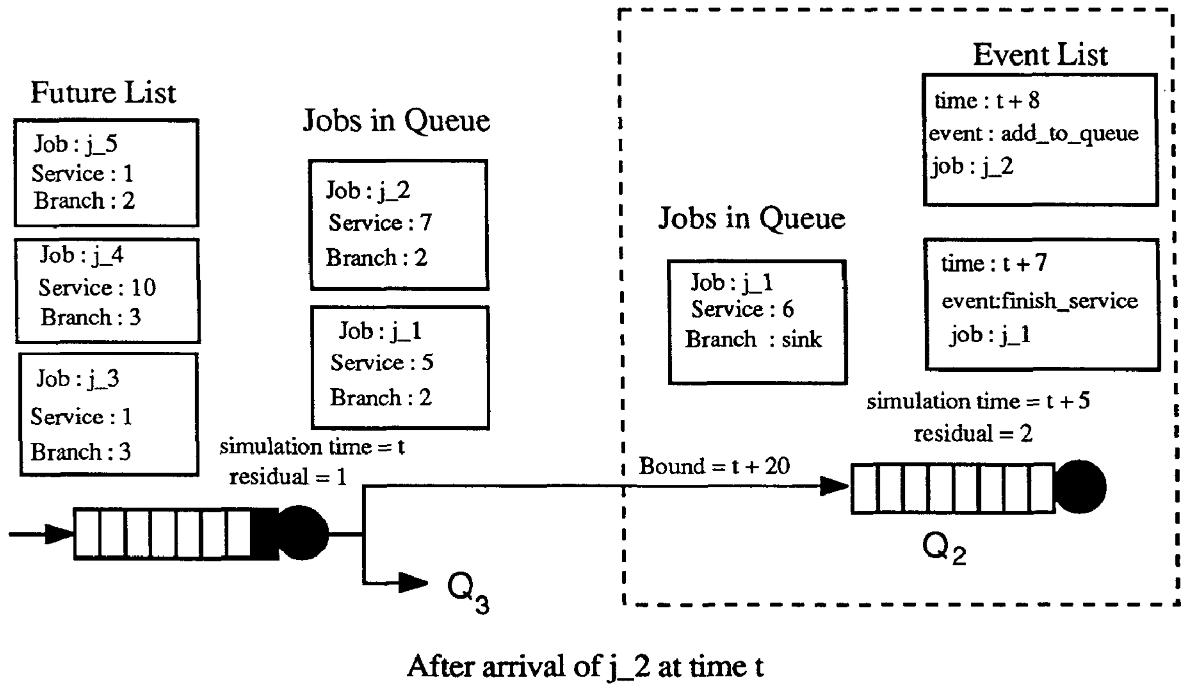


Figure 3: Transformation of queue and future list after a job arrival

```
AddToQueue ($Q$ , $t$)
    {   Remove first job $J_f$ from front of $Q$'s future list;
        Append $J_f$ to end of $Q$'s real queue;
        If ( No jobs in future list that branch to $J_f$.Branch )
            Repeat {
                Create job $J$;
                Randomly chose service time $J$.Service;
                Randomly chose branch $J$.Branch;
                Append $J$ to end of $Q$'s future list;
                } Until ($J$.Branch $=$ $J_f$.Branch)
        $J_B$ $=$ first job in future list such that $J_B$.Branch $=$ $J_f$.Branch;
        $S$ $=$ Sum of future list service times through $J_B$;
        $Apt = S + e_A(t)$;
        Notify queue $J_f$.Branch of arrival at time $e_A(t)$;
        Notify queue $J_f$.Branch of appointment at time $Apt$;
        Add BeginService event to event list at time $e_A(t) - J_f$.Service;
    }

BeginService ($Q$, $t$)
    {   $J$ $=$ First job in $Q$'s real queue;
        Compute desired statistics;
        Add FinishService event to event list at time $t +$ $J$.Service;
    }


FinishService ($Q$, $t$)
    {   Remove first job in $Q$'s real queue;
        Compute desired statistics;
    }
```

Figure 4: Queue Mangement Routines

and exchanged between queues. Observe however that a bound b provided by Q_A for Q_B can become "stale"—the bound is predicated on the assumption that the next job from Q_A is routed as soon as possible; it is possible for the simulation clock in Q_B's processor to advance up to b without another job being sent from Q_A to Q_B. In the absence of further jobs from Q_A, and in the absence of active measures by Q_A's processor to compute a new bound on the time of the next job from Q_A to Q_B, the appointment time a provided by Q_A's processor to Q_B's processor cannot exceed b. As Q_B's processor advances in simulation time it may find that the first event in its list has a time-stamp larger than a. In this case a new appointment is requested from Q_A's processor. It is eventually incumbent upon Q_A's processor to satisfy this request by computing a new appointment.

LP_i can construct an appointment for LP_j in several different ways. Two of the simplest ways are described below.

1. LP_i scans all of the latest bounds its queues have already provided to queues in LP_j. The appointment value is the least of these.

2. LP_i scans its event list to find the first future job arrival to any one of its queues. It compares this time to the minimum appointment given to it by a writer LP, and denotes the minimum of these two values by m; this quantity is a lower bound on the time at which a job next arrives at any queue in the LP. Then for every every pair of queues Q_A and Q_B such that Q_A lives in LP_i and Q_B in LP_j we compute a new bound. The new bound is computed on the assumption that the next job to arrive at Q_A (not necessarily with branching destination Q_B) arrives at simulation time m. Letting J_{Next} be the first job in Q_A's future list with destination Q_B and letting S be the sum of service times of jobs ahead and including J_{Next} in the the future list, we compute the appointment value $\max\{m, e_A(C_i)\} + S$. This appointment reflects the possibility of an arrival precisely at time m—the max term computes the earliest time at which the job represented by that arrival begins service in the queue. Among all such bounds computed for all queues, the minimum is the new appointment.

The first of these methods is the cheapest to compute, but will not not produce a usable appointment if any of the old bounds are stale. Furthermore, LP_j can compute this value for itself whenever it desires. The second method uses more information (the value m) and so may produce better bounds at the cost of some additional computation. It is important to note though that even if some bounds are improved, the appointment improves only if the minimum bound is improved upon. It is also important to note that for any given queue, recomputing a bound with an increased value of m will not improve the bound if m is less than the next known time that the queue could be empty. The key idea behind using Q_A's future list to compute an appointment is to find a lower bound on the time at which the next job arrives at Q_A. The second scheme is quite pessimistic when computing this bound. It is possible that value defining m is associated with a queue far removed from Q_A, and that a much better bound on the next arrival at Q_A is possible. We have implemented a method which analyzes the full simulation state in an LP in order to determine for each Q_A the best possible bound t_A on the time of its next arrival. Then for every writer/reader pair $Q_A \rightarrow Q_B$ a bound is constructed just like the one above, with t_A taking the place of m in the calculation. The minimum bound so calculated is the new appointment. A description and analysis of this method follows.

We first freeze all incoming bounds to LP_i's queues by making copies of their current values; this eliminates any further effect that other processors can have on the forthcoming algorithm. Every queue which reads from an off-processor queue has its *min_apt* value set to the minimum of its frozen off-processor bounds. Next, we scan the event list for job arrival events. Associated with each such event is a target queue; the arrival time is used to update the queue's *min_apt* value if that value either exceeds the job arrival time, or is in the initial state. Following these initialization steps, every queue's *min_apt* value is either null, or is equal to the minimum time at which a job might arrive either from off-processor, or from the event list. The problem now is to analyze the effects of job arrivals at those minimum times. This analysis is performed by essentially *simulating* the effects of job arrivals. For every queue with some value in its *min_apt* field we place in a *shadow event list* a *shadow event* which denotes a job arrival at time *min_apt*. The shadow-time-stamps of shadow-events taken off of the shadow-event list will be monotonically increasing. Proceeding with the shadow-simulation, we remove the minimum time shadow-event from the shadow-event list. If the specified queue has already been "touched" by the shadow-simulation we simply discard the shadow-event. Otherwise we consider the effects of a job arrival at the specified queue (say Q_A), at the shadow-event time. This is accomplished by computing a bound for each of Q_A's readers, based on the assumption that a job arrives at the shadow-arrival time. Shadow-events describing these arrivals are inserted into the shadow-event list, the queue is marked as having been touched by the shadow-simulation, and a count of "touched" queues is incremented. We are finished if this count equals the number of unfixed queues. Because the shadow-simulation simulates propagation of jobs through the network at the earliest possible times, the shadow-time associated with the first touch of a queue by the shadow-simulation is a lower bound on the time of the next true job arrival at the queue. Once the shadow-simulation has finished it is a simple matter to compute new bounds for queues in other processors by using the shadow-job arrival times.

The complexity of this method is $O(E \log E)$, where E is the number of inter-queue connections in the LP. This follows because any given inter-queue link will have a simulated shadow-arrival scheduled to cross it at most once (because a queue is touched at most once), and priority lists such as heaps exact a logarithmic cost for each access. This complexity does not consider the cost of initializing the priority heap. Initialization requires that we determine each queue's minimum incoming off-processor bound. Letting $\hat{E}$ denote the number of links from off-processor queues, this is achieved in $O(\hat{E})$ time. We must also determine for each queue whether there is a future job arrival in the event list. It is possible to link events in a such a way that the first arrival event for any given queue is accessible in constant time. This endows the initialization phase with an $O(n)$ complexity, where n is the number of queues on the LP. The $O(E \log E)$ cost thus dominates. It is appropriate to point out that this method is similar in spirit to that discussed in [5]. Due to differences in the models and applications, Groselj and Tropper's algorithm has a slightly smaller complexity—$O(n \log n + E)$.

Yet another approach to computing lookahead is quite general, and does not employ the inter-queue bounds at all; instead, it analyzes each processor's event list. Imagine momentarily that all processors are temporarily inhibited from modifying their event lists. Let $t_{\min}$ be the minimum time stamp among all job arrival events on the event lists. Then clearly any appointment value $a < t_{\min}$ between any two processors can be increased to $t_{\min}$. This type of lookahead is equivalent to that proposed by Lubachevsky [9]; however, the "bounded lag" and "minimal propagation" delays his method depends on are usually zero in general stochastic queueing networks. To have positively-bounded lags we would have to ascribe some time delay to a job passing from one server to another; to have minimal propagation delays we would have to impose minimal service times on each server. Lubachevsky's method calls for global synchronizations between processors so that $t_{\min}$ can be found, and events which can be performed concurrently be identified. Our overall approach is asynchronous, and we prefer to avoid global synchronizations if possible. A lower bound on $t_{\min}$ can be constructed asynchronously under the assumption that messages between LP_i and LP_j are received in the order that they are sent. Let $Tone1$ and $Tone2$ be two arrays such that $Tone1_i$ contains a snapshot of of LP_i's minimum job arrival event at some real time s_{1i}, $Tone2_i$ contains a snapshot of of LP_i's minimum job arrival event at some real time s_{2i}, and $s_{1i} < s_{2j}$ for any i and j. It can be shown that the minimum value in the $Tone1$ array is a lower bound on *any* future job arrival event time, and is consequently a lower bound on any interprocessor appointment. The $Tone$ arrays are easily maintained by appending minimal future job arrival times onto messages exchanged between processors. This method is even easier to implement on a shared-memory machine—if the event lists are stored in common memory, one processor can be solely dedicated to the task of collecting $Tone$ values and updating stale appointments.

We have implemented the second, third and fourth of these methods. The following section discusses their observed performance.

4 Performance Results

We have implemented a parallel discrete-event queueing network simulation on NASA Langley's Flex/32 [10] multiprocessor. The Flex/32 is a bus-oriented shared-memory architecture which supports both local and global memory. Our implementation takes advantage of the global memory—each processor's event list is in global memory, and one processor may insert an event into another's list. Mutual exclusion is enforced using low-level primitives such as spinlocks. Data structures describing the bounds between queues and the appointments between processors are also organized in the global memory.

The synchronization method employed to ensure simulation correctness is only one of a host of performance issues that must be addressed by a parallel simulator. In order to study the effectiveness of the synchronization method largely in isolation from other factors (such as load balancing), we haved studied simple, very homogeneous queueing networks which arise in the design of inter-processor communication networks: rings, meshes, hypercubes, and multistage routing networks. We assume that every server in a network has the same service time distribution, and the same homogeneous branching probabilities. The studies we describe here concern closed networks of 256 nodes (except for 384 nodes in the multistage case) simulated using sixteen processors. Queue i is assigned to processor i mod n, where n is the number of processors.

Speedup is the time required to solve the problem on a serial implementation divided by the time required by a parallel implementation. It is easy to use the parallel code on one processor as the serial version—the *algorithmic speedup* so calculated measures the method's efficiency as a function of the number of processors used. It does *not* however measure the end-user's benefit from parallelism. This benefit can only be measured by comparing the performance of an optimized serial version with the parallel version. Our performance measurements are based on this latter measurement of speedup; the optimized serial version was created from the parallel version by removing all code related to mutual exclusion and synchronization, and by removing all computations related to the future queue. A comparison between the optimized serial version and the parallel version on one processor tells us something about the cost of a processor's internal overhead of doing parallel processing (e.g., calls to synchronization routines); it also gives us an upper bound on the speedups we can expect. Each of our performance graphs is marked with this upper bound to better reflect how efficient the program is relative to its inescapable internal overhead.

The data structures and algorithms used to manage the event-list have a critical effect on performance. In the interests of rapid-prototyping we first implemented the event list as a naive, doubly-linked list. Under moderate loads we achieved a speedup of 24 using 8 processors! This anomaly is simply explained by realizing that the serial version is sub-optimal (see [8] for a performance study of various list-management algorithms); anomalies of this type have been observed in other contexts [4]. We subsequently implemented a simple, but more efficient list management algorithm by associating an ordered queue of events with each individual queue, and then use a combining tree to identify the event list with smallest minimal event.

The statistics collected by our program are minimal: for each queue we maintain a 128 element histogram of job waiting times. Updating the histogram requires only a binary search to select a bin, and an increment.

The ring topology allows a queue to send jobs to either a left or right neighbor; the mesh topology connects North, South, East, and West neighbors, and wraps around the edges to create a torus. The hypercube topology is the usual one; the multistage network consists of six stages, each of which has sixty-four queues, and which feed forward to the next stage using the Butterfly interconnection pattern. The last stage feeds the first stage.

All of our experiments employ sixteen processors. In one set of experiments we assume that the service time is exponential with mean $\mu = 1.0$; another set of experiments treats the service time as the constant 1.0. Because these networks are closed, the simulation load is varied by adjusting the number of jobs placed into the system. Because of homogeneity the load can be described simply by ν, the average number of jobs in queue at a server. For every topology and service distribution we varied ν within the set $\{1, 2, 4, 6, 8, 16\}$. For each set of parameters we simulated the network ten times, starting from an initial configuration where each queue has exactly ν jobs in queue. The simulation was terminated after all processors had advanced to simulation time 100. Larger termination times would be desirable if we were interested in accurate queueing network statistics; however, the timings on experiments with larger termination times scaled directly, required much more CPU time, and were subsequently dropped. The execution time measurements exclude the I/0 time required to initially load the problem, but include all other I/0 required during the course of a run. Our performance curves plot intervals to represent speedup. The intention is to both show what sort of speedups can be expected, and what variation there is in the speedup estimates. It is unreasonable to measure true speedup by inducing precisely the same branching and service time behavior in the serial and parallel versions. Instead, for each set of experimental

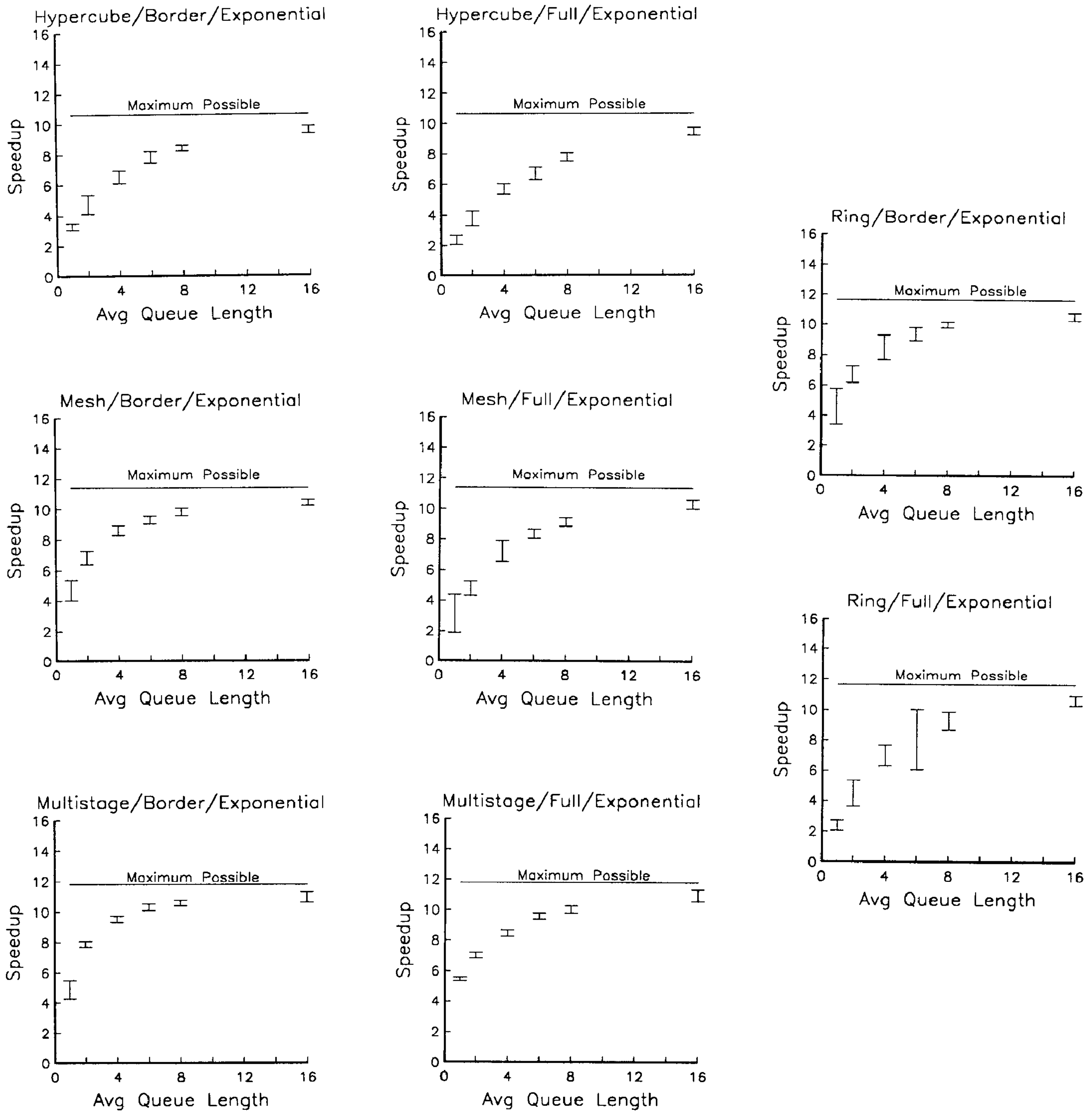


Figure 5: Speedup Curves, 16 Processors, Exponential Service Times

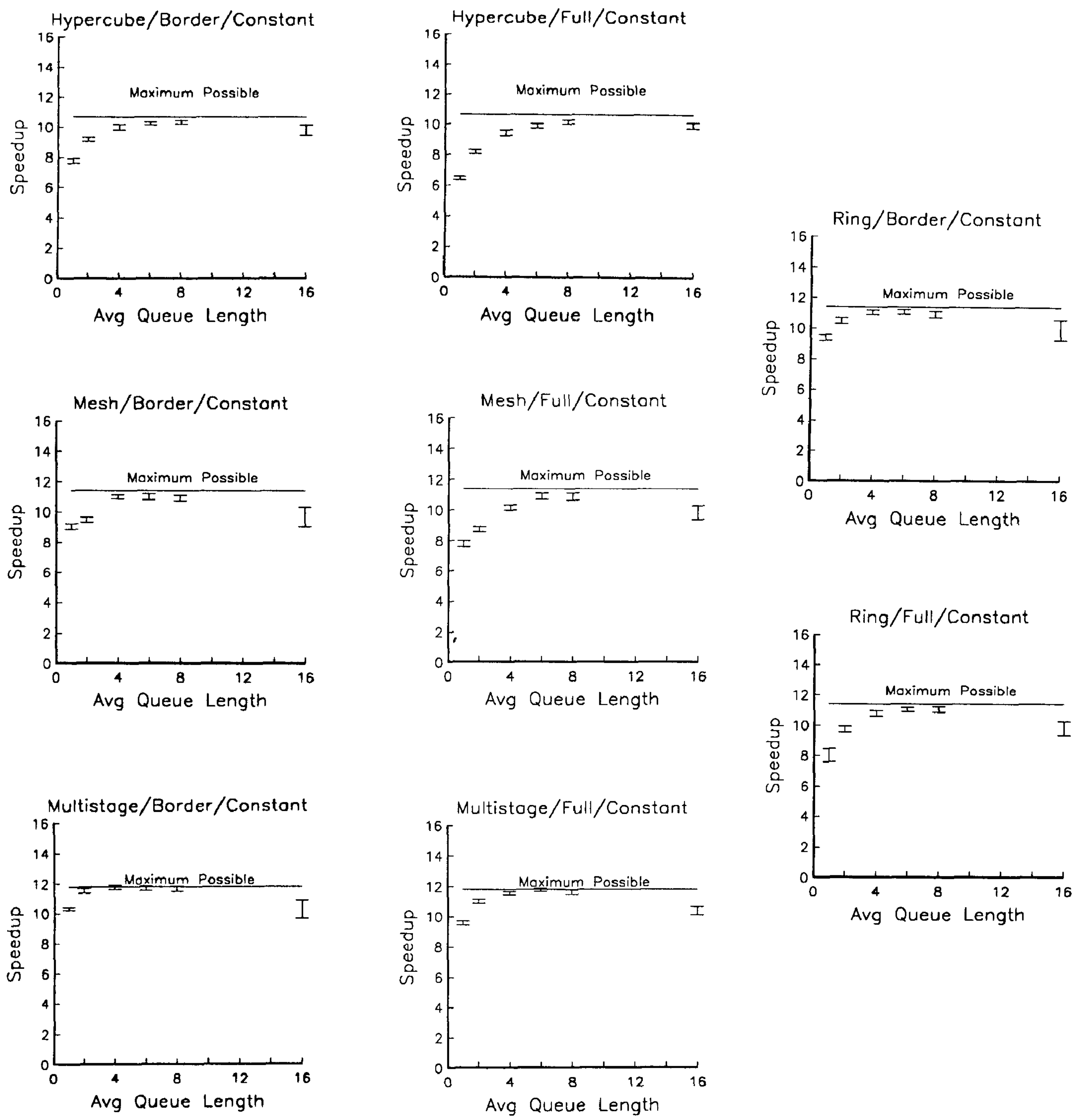


Figure 6: Speedup Curves, 16 Processors, Constant Service Times

parameters we measured the mean μ_p and standard deviation σ_p of ten parallel runs, and the mean μ_s and sample deviation σ_s of ten serial runs. Then we plot an interval containing a high speedup estimate, $(\mu_s + \sigma_s)/(\mu_p - \sigma_p)$, and a low speedup estimate, $(\mu_s - \sigma_s)/(\mu_p + \sigma_p)$.

Figures 5 and 6 presents the speedup intervals. Each graph's title has the form "Topology/Lookahead type/Distribution"; the lookahead type is *Full* or *Border*, depending on whether the lookahead calculation analyzed the full *LP* state or simply computed bounds at the queues which feed off-processor queues. A number of observations stand out. Ordered roughly by importance, they are:

1. Under moderate to heavy simulation loads every graph approaches its optimal level (a speedup which tends to be close to eleven). These experiments show that good speedups are sometimes possible in these types of simulations. If the simulation load is low the proportion of useful work to lookahead computation has to diminish, yielding poor speedups.

2. The service time variation has a strong effect on speedup. Under high variation very small lookahead values are possible, meaning that lookahead is computed more often, thereby incurring increased overhead. This is in agreement with Fujimoto's experiments[3].

3. Network topology strongly affects performance under low loads. Hypercubes have a richer interconnection structure, which causes increased uncertainty in future behavior (meaning that lookahead bounds are not sharp). Under low loads and exponential service times simulation of hypercube interconnections performed poorly while other interconnections did somewhat better.

4. Simulations of rings tend to have higher variance. This is understood by realizing that high workload in some network region does not easily disperse; the other topologies are better at spreading jobs around the network. This understanding of the phenomenon is re-enforced by Reed's observation[14] that concentrated chains of jobs tended to form in his simulations.

5. The form of lookahead used (Border or Full) has a smaller effect on performance than we anticipated. In this set of experiments the cheaper form of lookahead (Border) uniformly performed better, but this effect was secondary when compared to the effects of service time distribution and topology. We hasten to recall though that the mapping of queues to processors forces every queue to feed a proportionally large number of off-processor queues, so that the lookahead gained by collecting additional information from on-processor queues is overshadowed by the cost of collecting that information. We did study two variations on the lookahead calculation which only analyzes event lists. In one variation we dedicated a processor to the task of searching for this type of lookahead while all other processors did simulation work. This scheme had very little impact on the execution times. In a second variation we relied entirely on appointments computed by the auxiliary processor, and achieved comparatively poor speedups, even under high loads.

Two other points are of interest and are not shown in these graphs. Network size has some effect on performance; as expected, larger problems yield larger speedups, although the speedup still depends most heavily on the average queue length and the service time distribution. Secondly, we measured the number of times the lookahead analysis algorithm is called in the course of a simulation run. Under high loads ($\nu = 16$) the analysis routine is never called: the ordinary lookahead computed with every arrival to a queue sustains the progress of the simulation.

We reiterate the main conclusion that we can draw from this data: at least under limited circumstances it is possible to achieve good *real* speedups by using a conservative synchronization mechanism which exploits the problem being simulated.

5 Summary

The parallelization of discrete-event simulations has proven to be a difficult problem, due in large part to extensive and irregular synchronization requirements. One means of alleviating that synchronization burden is to have processors analyze their simulation state and compute *lookahead*, lower bounds on times at which they perform actions that directly affect the event lists of other processors. We illustrate this technique on the knotty problem of stochastic queueing network simulations. These simulations are particularly difficult because their intrinsic computation to synchronization cost ratio is so dis-advantageous. We show how the simulation can be re-organized to allow lookahead to be computed for FCFS queuing networks, discuss trade-offs between the quality of lookahead and the cost of providing it, and demonstrate the effectiveness of the method by implementation on several common queueing network topologies. This result stands in contrast with previous studies which used synchronization mechanisms that are largely unaware of the underlying simulation problem. Generality in a synchronization mechanism is a worthy goal, but the price of that goal may be poor performance.

References

[1] K. M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, September 1979.

[2] G. S. Fishman. *Principles of Discrete Event Simulation*. John Wiley and Sons, New York, 1978.

[3] R. M. Fujimoto. Performance measurements of distributed simulation strategies. In *Proceedings of the 1988 SCS Conference on Distributed Simulation*, pages 14–20, San Diego, CA, 1988.

[4] E.F. Gehringer, D.P. Siewiorek, and Z. Segall. *Parallel Processing: The Cm* Experience*. Digital Press, Bedford, Massachusetts, 1987.

[5] B. G. Groselj and C. Tropper. The time-of-next-event algorithm. In *Proceedings of the 1988 SCS Conference on Distributed Simulation*, pages 25–29, San Diego, CA, 1988.

[6] V. Holmes. *Parallel algorithms on multiple processor architectures*. PhD. thesis, Department of Computer Science, University of Texas at Austin, 1978.

[7] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[8] D.W. Jones. An empirical comparison of priority-queue and event-set implementations. *CACM*, 29(4):300–311, April 1986.

[9] B. D. Lubachevsky. Bounded lag distributed discrete event simmulation. In *Proceedings of the 1988 SCS Conference on Distributed Simulation*, pages 183–191, San Diego, CA, 1988.

[10] N. Matelan. The Flex/32 multicomputer. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 209–213, Computer Society Press, June 1985.

[11] D. M. Nicol. *The Performance of Synchronizing Networks*. Master's thesis, Department of Computer Science, University of Virginia, January 1984.

[12] D. M. Nicol, P. F. Reynolds, Jr. Problem Oriented Protocol Design. In *Proceedings of the 1984 Winter Simulation Conference*, pages 471–474, Dallas, Texas, December 1984.

[13] J. K. Peacock, E. Manning, and J. W. Wong. Synchronization of distributed simulation using broadcast algorithms. *Computer Networks*, 4:3–10, 1980.

[14] D. A. Reed, A.D. Maloney, and B.D. McCredie. Parallel discrete event simulation: a shared memory approach (extended abstact). In *Proceedings of the 1987 SIGMETRICS Conference*, pages 36–39, Banff, Alberta, Canada, May 1987. To appear in *IEEE Trans. on Soft. Eng.*

[15] P.F. Reynolds, Jr. A shared resource algorithm for distributed simulation. In *Proceedings of the Ninth Annual International Computer Computer Architecture Conference*, pages 259–266, Austin, Texas, April 1982.