Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

High performance parallelized discrete event simulation of stochastic queueing networks

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

ABSTRACT

Queueing network simulations provide a stress-test for the study of parallel discrete-event simulation because each event involves some synchronization overhead, and very little computation. In this paper we describe techniques for optimizing the performance of large queueing network simulations using a shared-memory multiprocessor. We give empirical data showing that good speedups can sometimes be achieved.

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* 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. Seminal work in parallel simulation (Chandy and Misra (1979)) 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 many probability distributions of interest are not bounded from below, implementations must choose ϵ to be very small. Performance studies by Holmes (1978) and Reed *et al.* (1988) have strongly suggested that this poor lookahead ability leads to dismal performance due to extremely high synchronization overhead.

Nicol (1984) 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. Nicol and Reynolds (1984) examined the effect that increased lookahead has on overall performance. Fujimoto (1988) re-examined the Chandy 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 (1988) also uses lookahead that is computable under 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 when the minimum service time is small.

Lookahead is the basis of any conservative synchronization protocol that avoids deadlock. Our thesis is that to achieve good performance in a parallel simulation, the programmer should provide the synchronization layer with the best lookahead that can be easily computed. This paper discusses techniques for computing lookahead in stochastic queueing network simulations. The utility of our techniques are proven with large, empirically measured speedups on a shared-memory multiprocessor.

2. STOCHASTIC QUEUEING NETWORKS

We now briefly describe the class of simulations considered in this paper. We assume the reader has an intuitive feel for the concept of a "queue" and its "server". For simplicity we simply refer to the aggregate queue and server as a queue.

Jobs arrive at a queue, eventually receive service, and then depart. In a stochastic simulation the amount of time a job receives service is drawn from a probability distribution associated with the queue. The queue's *queueing discipline* determines when jobs will receive service at the queue. Many queueing disciplines are non-preemptive—once placed in service, a job remains in service until its service requirements are met. When a job leaves the queue, a non-preemptive discipline decides which job among all those in queue will receive service. For example, the First-Come-First-Serve (FCFS) discipline chooses the job that has been in the queue the longest; the Longest-Job-First (LJF) discipline chooses the job with the largest service time. Preemptive disciplines are somewhat more complicated in that they allow a job in service to be yanked out in favor of another before its service requirement is met.

A queueing network is constructed from a collection of queues. When a job leaves one queue, it may be routed as an arrival to another queue. The network topology defines which queues may feed other queues. Typically, a given queue may route a job to one of only a small subset of the network queues. *Branching probabilities* are assigned to each of a queue's choices. A job's destination upon leaving a queue is chosen randomly in accordance with the branching probabilities.

3. LOOKAHEAD IN PARALLEL SIMULATION

In our view of parallel discrete-event simulation every processor 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 four queue network on three processors. Q_1 and Q_2 reside respectively on processors P_1 and P_2; both feed two queues Q_3 and Q_4 coresident in P_3. Q_1 has a number of jobs, labeled with their service requirements and branching destinations. Q_2 is empty, and its clock is 12.

To ensure simulation correctness we require that the sequence of events processed by a processor be monotonically increasing in simulation time. Consequently, even though P_3 has an event at time 18 to perform, it must be prohibited from doing so—it is possible for a job to arrive at Q_2, require only a small amount of service, and be routed to one of P_3's queues. The role of a synchronization algorithm is to inform P_3 when it is safe to process its next pending event.

Our ability to avoid this sort of blocking depends highly

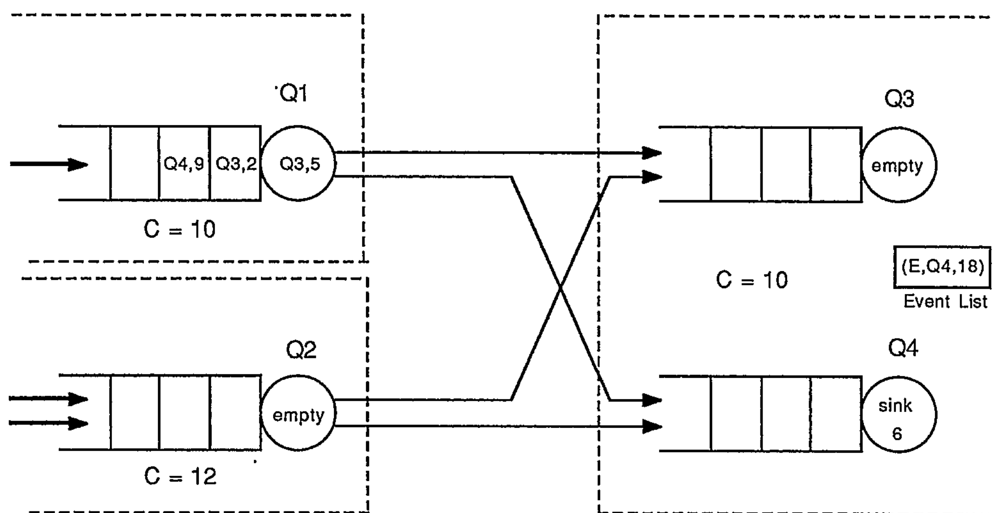

Figure 1: Parallel simulation of four queues

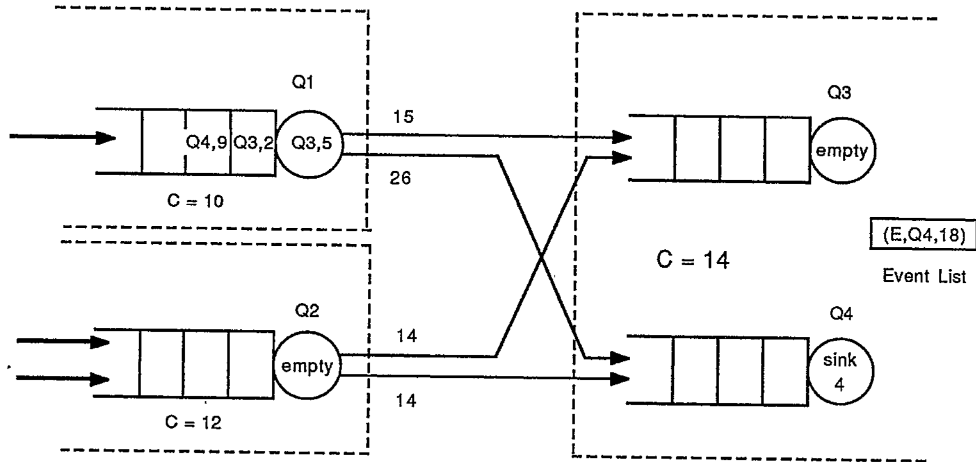

Figure 2: Minimum service times and FCFS queues

on our ability to predict the future behavior of the simulation. One model assumption facilitating this is that of a minimum service time. Consider—if we knew that an instantaneous job arrival at Q_2 will demand at least 2 units of service time, then Q_2 can promise P_3 that it will not route a job before time 14. This is clearly an improvement, but for our example is insufficient to unblock P_3. Nevertheless, when minimum service times are large, then this information can be exploited to achieve reasonablely good speedups.

Let us further suppose that each server's queueing discipline is FCFS. Figure 2 illustrates how Q_1 can bound future job arrival times from below (in this case, sharply) by analyzing the times and destinations of enqueued jobs. These bounds are illustrated on the appropriate links. Note still that Q_2's emptiness prohibits it from improving its bounds in this fashion. But suppose that Q_2's two writers promise not to route jobs before times 17 and 20, respectively. Q_2 can reason that no job will arrive before time 17, if one does it receives at least two units of service, so that no job will leave Q_2 before time 19. Supplied with this bound, P_3 no longer is blocked on Q_2 at time 14. However, due to the very real threat of a job arrival at Q_3 from Q_1 at time 15, P_3 remains unable to process the first job in its event list. This last difficulty is alleviated by the observation that under FCFS service a job's departure time is completely determined by its arrival time and the amount of work ahead of it in queue. As jobs arrive at Q_1, it can inform P_3 of future

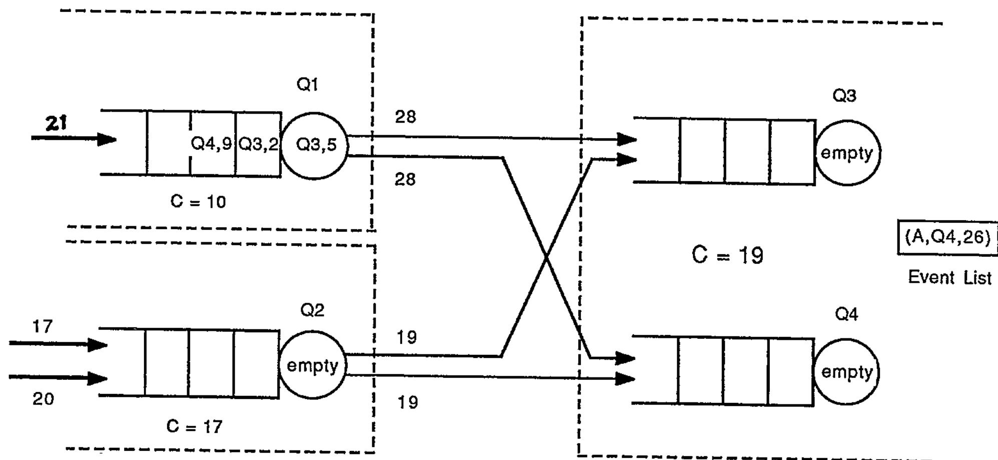

Figure 3: Minimum service times and lookahead scheduling in FCFS queues

job arrivals—these arrivals are recorded in P_3's event list. Figure 3 illustrates our example assuming this type of *lookahead scheduling*. Q_1's new bounds are based on the fact that a job arriving while Q_1 is not empty cannot receive service before time 26, and then will receive at least 2 units of service. Under the more general assumption of non-preemptive service, we can always perform lookahead scheduling at the point where a job enters service. Figure 3 also reflects the possibility that in wallclock time, the jobs still enqueued at Q_1 can have already been processed by their destination queue, Q_3. This is possible because P_3 is ahead of P_2 in simulation time.

P_3's ability to surge ahead in the examples above is very much dependent on the high incoming bounds enjoyed by Q_2. Were those bounds smaller, the large workload in P_3 would remain blocked. In order to optimize performance, the programmer must strive to maintain these bounds as high as possible. The object of this paper is to point out techniques for doing so.

4. THE FUTURE LIST

The network simulations we consider choose a job's service time in accordance with a probability distribution that is associated with the attendant queue. Traditional queueing network simulations do not sample service times until needed—for example, when a job arrives at the queue or when it enters service. Likewise, traditional simulations do not choose a job's destination until that job departs. In parallel simulations there is some advantage to choosing service times and branching destinations before the jobs ever arrive. The *future list* is a list of pre-sampled job characteristics, ordered in terms of the future jobs' (unknown) arrival times. When a job arrives at a queue, the first job in the future list is removed. The service time and branching destination of the new arrival are taken from the future list job. Suppose that the branching destination is Q_B. If the future list removal empties the list of jobs with destination Q_B, jobs with randomly sampled service times and branching destinations are appended to the end of the list until a job with destination Q_B is created. In this way the statistical integrity of the simulation is preserved.

Figure 4 illustrates how the future list can be used with FCFS queues to construct superior bounds to those already constructed. In particular, the future list allows us to construct non-trivial bounds from Q_2 to Q_3 and Q_4 (37 and 26 respectively) when Q_2 does not have any jobs enqueued for either queue. These bounds are predicated on the assumption that a pile of jobs do arrive at Q_2 at time 17. The bounds are constructed exploiting the order in which the jobs are serviced, the jobs' service times and branching destinations, and the practice of lookahead scheduling. Note that a bound is constructed by adding service times of jobs in queue (and the residual service of the one in service) to the clock value.

A future list can be used to advantage with other service disciplines. For example, suppose that Q_A employs Largest-Job-First, and seeks to construct a bound for Q_B. First con-

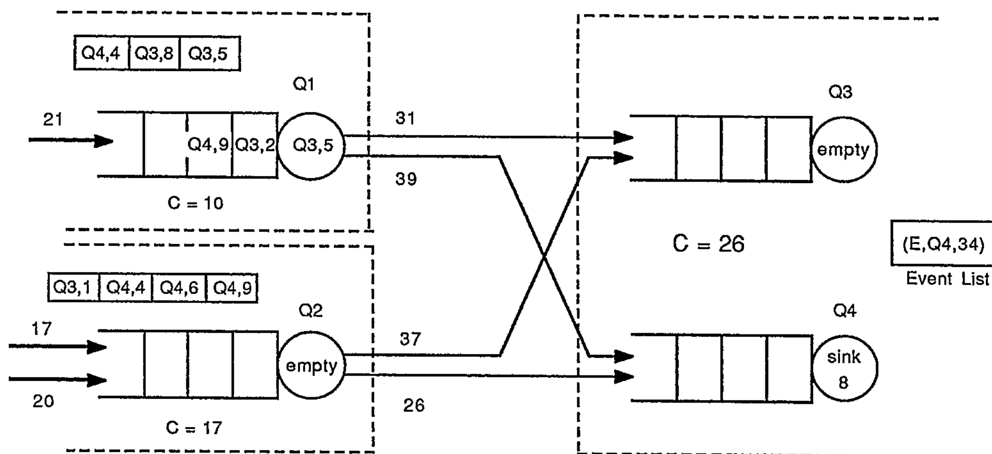

Figure 4: Future list and FCFS queues

sider the case where Q_A already has at least one job (one not receiving service—one receiving service has already been reported via lookahead scheduling) destined for Q_B; let the service time of the largest such be s_B. We construct a bound by adding s_B to the sum of service times of all jobs in queue with service requirements exceeding s_B—we are assured that each of these receives service before any job for Q_B does. We apply a similar technique using the future list if Q_A is devoid of jobs for Q_B. Let s_B be the service time of the first job j_B destined for Q_B in the future list. We add s_B to the sum of service times exceeding s_B of jobs which must arrive before j_B: jobs currently in queue, and jobs ahead of j_B in the future list.

As another example, suppose that jobs are prioritized by their branching destination. Jobs with a common destination are serviced in FCFS order. Consider the computation of a bound by Q_A for one of its readers, Q_B. If there is already a job in that queue destined for Q_B we construct a bound by summing the service times of all jobs already in queue with higher priorities. To this we add the service time of the first job in queue with destination Q_B, and as always, add the residual service time of the job in service, and the current simulation time. If Q_B has no jobs in queue we find the first job in the future list with destination Q_B, and sum the service times of all jobs with higher priority which have already arrived or must arrive before that job (jobs ahead of it in the future list).

We have so far considered only job service times that may be precisely pre-computed before the job enters service. Some service distributions are load-dependent—the amount of service a job requires depends on the the number of jobs in queue at the time it enters service. In these models a job entering service with i other jobs in queue tends to require less service than a job who enters service with $j > i$ jobs in queue. One way of modeling this is to let R be a random number from some distribution, let $g(i, r)$ be a function that increases in i and in r, and to give $g(i, R)$ service time to a job who enters service when i other jobs are enqueued. The future list can pre-sample R as before. If the random variable used to compute a job's service time is r, then $g(0, r)$ is a lower bound on the job's service time. The bounds we have outlined for FCFS, LFS, and prioritized queues all work using $g(0, r)$ as a lower bound on jobs' service times (note that our ability to perform lookahead scheduling on FCFS queues is affected).

These examples demonstrate that it is sometimes possible to compute substantial lookahead bounds between queues by using a future list. Details of the bounds calculation depend highly on the queueing discipline. Some disciplines do not admit easy calculation of large lower bounds. Difficult disciplines are those that admit the possibility of a very short job getting immediate service on arrival, even if the queue has a substantial number of jobs already enqueued. Common examples of these are Shortest-Job-First, and Processor-Sharing. We are currently exploring the possibility of using optimistic bound *approximations* for such disciplines. These approximations would not be absolute guareentees, but only values that are highly likely to be true bounds. A queue can easily estimate its aggregate arrival rate. It can then assume that arrivals occur much faster, and compute outgoing bound approximations using the pessimistic arrival pattern. At the user's option, the simulation might employ state-saving and rollback for late arrivals in the style of Time Warp (Jefferson (1985)), may compromise the simulation's statistics by ignoring such jobs, or may simply invalidate that run. This approach varies significantly from Time Warp in that we still expend substantial effort to prevent jobs from arriving in a queue's logical past.

5. THE APPOINTMENT PROTOCOL

The lookahead bounds we have discussed are called *appointments*, and form the basis of a simple synchronization mechanism. The appointment protocol was first proposed in Nicol (1984) and analyzed in Nicol and Reynolds (1984). We assume that a simulation is partitioned into communicating *Logical Processes*, or simply LPs. One may treat a queue as an LP; one may also aggregate a number of queues to be an LP. An LP has readers (LPs whom it can directly affect, e.g. route a job to), and writers (LPs which directly affect it). Each of LP_i's writers (e.g. LP_j) provides LP_i with an appointment time A_{ji}. When LP_i is ready to process the next event, it first checks to see if the time stamp of that event (say e_i) is no greater than LP_i's minimum incoming appointment m_i. If $e_i > m_i$, then LP_i asks each writer whose appointment is less than e_i to provide a better appointment. LP_i then checks to see if any if its own readers are requesting new appointments

from it. If so, these requests are serviced (if there is a possibility to increase the existing appointment). One key feature of the protocol is that event processing always takes precedence over appointment calculation. Another key feature is that any appointment calculations which are not automatic (e.g. appointments are automatically computed in a FCFS queue upon a job's arrival) are demand driven.

It is important to note that the appointments we calculate can be used by other synchronization protocols. For example, Lubachesky's (1988) method is synchronous, while most other protocols are asynchronous. Nevertheless, the correctness and performance of the method depends on the ability to extract lookahead from the model. Taken by themselves, appointments are merely lookahead values.

6. IMPLEMENTATION DETAILS

We implemented a queueing network simulation on a shared-memory multiprocessor, the Flex/32 (Matelan (1985)). The appointment values, simulation clocks, and event lists are all maintained in the shared memory. "Sending" an appointment consists of writing the new appointment value in its designated memory location. Any processor can post an event in any of the event lists. All shared data are protected by using spin-locks to enforce mutual exclusion.

Our implementation aggregates all queues assigned to one processor as a single LP. The notion of appointments between aggregate LPs extends naturally—the appointment from LP_A to LP_B is the minimum inter-queue appointment from a queue in LP_A to a queue in LP_B. An LP can consequently determine its minimum incoming appointment value by scanning a list of inter-queue appointments, all in shared-memory. This aggregation has the positive benefit of reducing scheduling overhead; on the other hand, it leads to complications due to "stale" inter-queue appointments. A full description of these problems and their solutions are beyond the scope of this paper and have been documented in Nicol (1988).

7. PERFORMANCE STUDIES

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 is FCFS, 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 $i \bmod n$, where n is the number of processors.

Speedup is the time required to solve the problem on an optimized 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 (denoted Maximum Possible) to better reflect how efficient the program is relative to its inescapable internal overhead.

The statistics collected by the network simulation 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 execution time measurements exclude the I/O time required to initially load the problem, but include all other I/O 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. For each set of experimental 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, $(\mu_s - \sigma_s)/(\mu_p + \sigma_p)$.

Figure 5 presents representative speedup intervals. More extensive data can be found in Nicol (1988). A number of observations stand out. Ordered roughly by importance, they are:

1. Under moderate to heavy simulation loads the performance reaches its optimal level (a speedup which tends to be close to eleven). These experiments prove that good speedups are sometimes possible. 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 (1988) experiments.

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

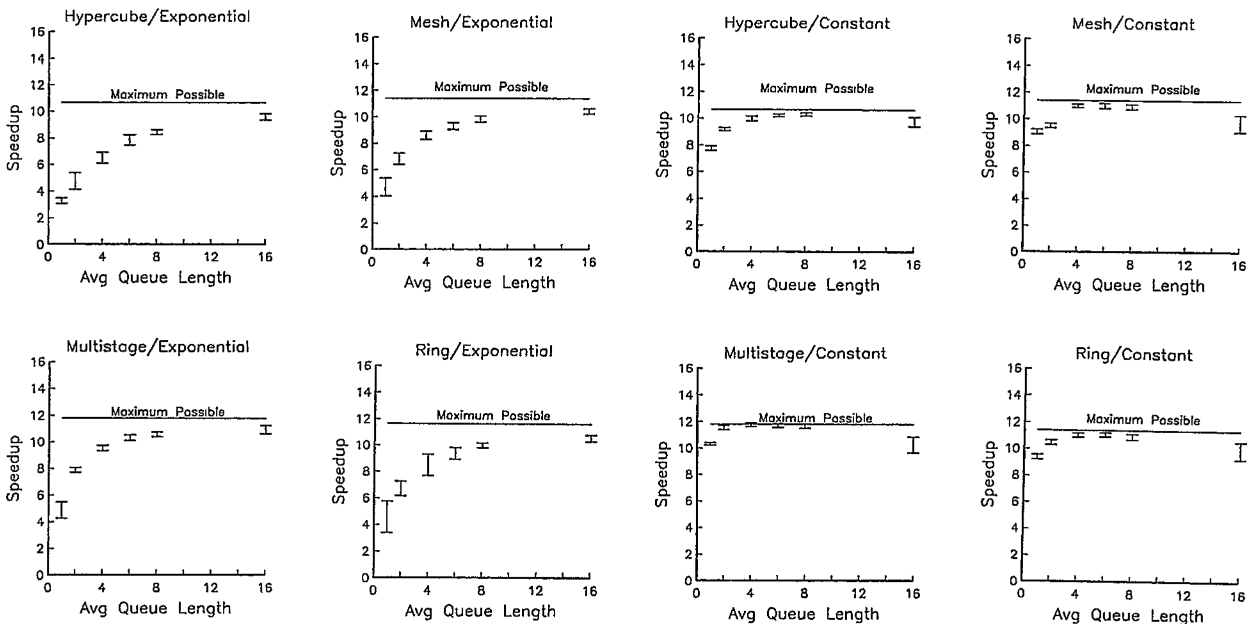

Figure 5: Representative speedup curves using 16 processors. Maximum algorithmic speedups are labeled.

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 that concentrated chains of jobs tended to form in his simulations.

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

8. 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 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 several different queuing disciplines, 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.

A simulation modeler is likely to be uninterested in uninterested in the implementation issues of parallel simulation. Because of this, it is important to develop software tools, languages, and environments that ease and automate the development process, and hide implementation details from a casual model builder. A first approach to this problem is discussed by Miller and Nicol (1988) in these proceedings.

References

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

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

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

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

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

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

Miller, K. and D. M. Nicol (1988). Implementing parallelized queueing network simulations using Fortran and data abstraction, In *Proceedings of the 1988 Winter Simulation Conference*, San Diego, CA., December 1988.

Nicol, D. M. (1984). *The performance of synchronizing networks.* Master's thesis, Department of Computer Science, University of Virginia, January 1984.

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

Nicol, D. M. (1988). Parallel discrete-event simulation of FCFS stochastic queuing networks, In *Proceedings of the ACM SIGPLAN PPEALS Conference*, pages 124-137, New Haven, CT., July 1988.

Reed, D. A., Maloney, A. D., and McCredie, B. D. (1988). Parallel discrete event simulation using shared memory. *IEEE Trans. on Soft. Eng.,* 14(4):541-553, 1988.

David M. Nicol received the B.A. in mathematics from Carleton College, Northfield, MN. in 1979, and the M.S. and Ph.D. degrees in computer science from the University of Virginia in 1984 and 1985. He was a programmer/analyst with Control Data Corp. from 1979 to 1982, and a staff scientist at the Institute for Computer Applications in Science and Engineering from 1985 to 1987. He is presently an assistant professor in the department of computer science at the College of William and Mary. He has published numerous articles in the area of parallel processing, and is a member of ACM, IEEE Computer Society, and ORSA/TIMS.

David M. Nicol
Department of Computer Science
College of William and Mary
Williamsburg, VA 23185
(804) 253-4748
e-mail: nicol@wmcs.cs.edu