EXPLOITING PARALLELISM IN SIMULATIONS

I MITRANI

Rapporteur: R de Lemos

Exploiting Parallelism in Simulations

Albert G. Greenberg
AT&T Bell Laboratories
Murray Hill, NJ 07974, USA

Isi Mitrani
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

Boris D. Lubachevsky
AT&T Bell Laboratories
Murray Hill, NJ 07974, USA

1. Introduction

The simulation of a discrete event system is traditionally regarded as the process of generating an operation path that represents the system state as a function of time. This normally entails the use of a global clock and an event list. In the last few years, much effort has been devoted to the task of splitting the simulation process into a number of sub-processes and executing the latter in parallel on different processors [1, 2, 3, 4, 5]. For example, when simulating a queueing network, the idea might be to allocate each processor to a node, or a group of nodes, and let it handle the corresponding events, taking care of possible interactions with other processors. At best, the degree of parallelism obtained by such an approach will be equal to the number of nodes, and in general may be much smaller [4, 5].

We propose new methods that do not limit the degree of parallelism in this way. The concepts of "time" and "event" are no longer present explicitly, and the necessity for the event list disappears. In section 3, we consider the problem of simulating a long run of a first in, first out (FIFO) G/G/1 queue [6], using P processors. A simple algorithm is presented for computing the arrival and departures times of the first N jobs in time proportional to N/P, for large N. The algorithm does not rely on any regenerative properties of the queue. We propose similar methods to obtain similar speedups for acyclic fork-join queueing networks (section 4.1), acyclic queueing networks (section 4.2), and series of queues with bounded buffers (section 4.3). Moreover, we obtain new serial simulation methods for these networks, which compare favorably with their event list counterparts. Some experimental results on the efficiencies of the serial and parallel simulations of the G/G/1 queue are reported. In the full paper, we will describe parallel simulation algorithms for some queueing systems subject to breakdowns or having multiple priority classes.

A key idea behind our simulation methods is to pose the simulation problem mathematically using recurrence relations. When the recurrence relations are of a certain type, the problem of solving them reduces to the parallel prefix problem [7]. By exploiting the connection to parallel prefix, the recurrences can be solved quite efficiently in parallel [8, 9]. A second basic problem we face in the simulations is merging two sorted lists. Fortunately,

this problem can also be solved efficiently on parallel processors [10]. A simple, practical parallel merge routine is described in the Appendix.

In the case of the G/G/1 queue, the recurrence relations define the sequences of arrival and departure instants. For this simple system, if the object of interest is the waiting time, it would also be possible to use Lindley's recurrences [6]. However, the latter do not contain enough information to reconstruct the sample path, and do not generalize to other systems. Recently, Baccelli and his coworkers have used recurrence relations in studies of a variety of stochastic systems, including acyclic fork-join networks [11] and certain generalized stochastic Petri nets [12]. In their work the recurrences serve as the starting point for the study of ergodicity (stability) conditions and for the derivation of stochastic orderings and bounds. Similar recurrences serve as the starting point for developing our parallel simulations.

Chandy and Sherman [13] pointed out parallel simulations might be written that decouple the physical system being simulated from the parallel processor, and thereby achieve speedups greater than the number of objects in the physical system. However, our methods and results do not overlap with theirs. In [13], a general parallel relaxation method for simulation was proposed. It could be that the simulation methods proposed here may be fruitfully combined with relaxation to handle queueing systems where our methods are not directly applicable.

All of the queueing systems considered here can viewed as if all (random) choices can be made at the times that jobs arrive, without regard to the state the system happens to be in. Specifically, a job's route through the network and service demands along the route do not depend on the disposition of other jobs in the network. It seems more difficult to obtain simulations with massive parallelism if, for example, the routing depends on the state the jobs see while in the network, as would be the case if routes are dynamically adapted in response to congestion. However, work in progress on simulating multiple access protocols such as Aloha [14] suggests it may be possible to achieve very large speedups for some systems where interactions between components are more complicated than considered here.

2. Parallel Processing Model

We shall assume the following simple parallel machine setup:

- The number of processors P available for the computation is fixed, independently of the parameters of the simulation problem.
- The processors share a common memory.
- There is a barrier synchronization mechanism; that is, a mechanism under program control that delays the start of a computation until all P processors have completed the previous computation. We shall use this mechanism sparingly.

These assumptions suit the Sequent Balance 21000, the machine available for our experiments.

However, our parallel simulation methods are applicable on a wide variety of other parallel architectures. The methods call on a small number of basic parallel processing

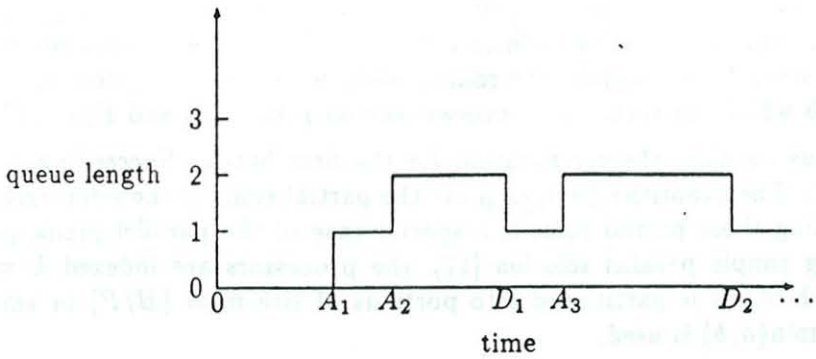

Figure 1: Example sample path.

operations: linear recurrence computation, parallel prefix computation, and parallel merging. These operations have been widely studied and programmed for a great variety of architectures: including systolic arrays, hypercubes, butterflies, ultracomputers, meshes, etc. [15].

3. The G/G/1 FIFO Queue

Consider the G/G/1 queue with interarrival times $(\alpha_i)_{i\geq 1}$ and service times $(\delta_i)_{i\geq 1}$. Let A_i denote the time the i^{th} job arrives and D_i the time the i^{th} job departs. Assuming the queue operates FIFO, for all $i \geq 1$,

$$A_i = A_{i-1} + \alpha_i \tag{3.1}$$

$$D_i = (D_{i-1} \vee A_i) + \delta_i \tag{3.2}$$

where $A_0 = D_0 = 0$ and $x \vee y$ denotes the maximum of x and y. Equation (3.1) restates that the α_i are the job interarrival times. Equation (3.2) states that the i^{th} job starts service either at time D_{i-1} or at time A_i, depending on whether the job sees a busy queue on arrival or an idle one, respectively. The i^{th} job's *sojourn time* is $D_i - A_i$. The queue length trajectory or *sample path* is obtained by merging the sequences $(A_i)_{i\geq 1}$ and $(D_i)_{i\geq 1}$, as shown in Figure 1.

We pose the problem of simulating the G/G/1 queue as that of computing, for a given integer $N \geq 1$, the quantities $(A_i)_{1\leq i\leq N}$ and $(D_i)_{1\leq i\leq N}$. We shall touch on the ancillary problems of generating the random variables $(\alpha_i)_{1\leq i\leq N}$ and $(\delta_i)_{1\leq i\leq N}$, computing sojourn time statistics, and computing queue length statistics.

Recurrences (3.1) and (3.2) give a simple, efficient serial (one processor) simulation method. Some results of experiments comparing the efficiency of this solution to that of the conventional event list method [16] are reported below.

In our parallel solution, an efficiency/memory tradeoff leads us to organize the computation into *batches*. Let $B \geq 1$ be an integer parameter, which we assume for simplicity divides N, the total number of jobs to simulate. First, we assign all P processors to simulate the first B jobs, computing $(A_i)_{1 \leq i \leq B}$ and $(D_i)_{1 \leq i \leq B}$. Second, we assign all P processors to simulate the next B jobs, computing $(A_i)_{B+1 \leq i \leq 2B}$ and $(D_i)_{B+1 \leq i \leq 2B}$, and so forth. It turns out the computation's efficiency suffers if B is too small compared to the number of processors P. To support the computation we need two vectors of memory locations of length B which are successively overwritten with the A_i's and D_i's of the current batch.

Let us consider the computation for the first batch. Succeeding batches are handled similarly. The quantities $(A_i)_{1 \leq i \leq B}$ are the partial sums of the interarrival times $(\alpha_i)_{1 \leq i \leq B}$. Computing these partial sums is a special case of the parallel prefix problem [7]. In the following simple parallel solution [17], the processors are indexed $k = 1, 2, \ldots, P$, and the batch of α's is partitioned into portions of size $m = \lceil B/P \rceil$ or smaller; the notation $a \wedge b = \min\{a, b\}$ is used.

1. In parallel, for each $k = 1, 2, \ldots, P$, let processor k compute the block of partial sums,

$$A'_i = \alpha_i + \alpha_{i-1} + \ldots + \alpha_{(k-1)m+1},$$

for $i = (k-1)m+1, (k-1)m+2, \ldots, (km) \wedge B$. If $(k-1)m \geq B$ then processor k is idle.

2. Compute the partial sums (see remark below),

$$A''_k = A'_{km} + A'_{(k-1)m} + \ldots + A'_m,$$

for $k = 1, 2, \ldots$, while $km < B$.

3. In parallel, for each $k = 1, 2, \ldots, P$, let processor k compute the block of final results,

$$A_i = A'_i + A''_{k-1},$$

for $i = (k-1)m+1, (k-1)m+2, \ldots, (km) \wedge B$. Here $A''_0 = 0$ by definition. Again, if $(k-1)m \geq B$ then processor k is idle.

Phases 1 and 3 require order B/P time. Phase 2 can be carried out in order $\log_2 P$ time using, for example, Stone's recursive doubling method [18, 17], Alternatively, phase 2 can be carried out in order P time using one processor. On machines (such as the one used in our experiments) where the cost of barrier synchronizing between the three phases is order P and P is small, there is nothing to gain by the $\log_2 P$ solution.

It remains to compute the departure times $(D_i)_{1 \leq i \leq B}$. The key is to observe that (3.2) is a linear recurrence in the semiring [19] on the real numbers where $\vee$ is the addition operator with identity $-\infty$, and $+$ is the multiplication operator with identity 0. For example, verifying that multiplication distributes over addition, we have

$$a + (b \vee c) = (a + b) \vee (a + c).$$

With the understanding that matrix multiplication is carried out in the $(\vee, +)$ semiring rewrite (3.2) as

$$\begin{bmatrix} D_i \\ 0 \end{bmatrix} = \begin{bmatrix} \delta_i & A_i + \delta_i \\ -\infty & 0 \end{bmatrix} \cdot \begin{bmatrix} D_{i-1} \\ 0 \end{bmatrix}, \tag{3.3}$$

a matrix linear recurrence [9, 8]. The computation of $(D_i)_{1 \le i \le B}$ may be handled similarly to that of $(A_i)_{1 \le i \le B}$, except that the partial sums are replaced by the partial products of the 2×2 matrices that arise in (3.3). We shall call on the identities,

$$\begin{bmatrix} x_1 & x_2 \\ -\infty & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 & y_2 \\ -\infty & 0 \end{bmatrix} = \begin{bmatrix} x_1 + y_1 & (x_1 + y_2) \vee x_2 \\ -\infty & 0 \end{bmatrix} \tag{3.4}$$

$$\begin{bmatrix} x_1 & x_2 \\ -\infty & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} (x_1 + y_1) \vee (x_2 + y_2) \\ y_2 \end{bmatrix}. \tag{3.5}$$

Note that both products require two standard additions and one maximization.

Let M_i denote the 2×2 matrix on the right hand side of (3.3), and again let $m = \lceil B/P \rceil$. The departure times $(D_i)_{1 \le i \le B}$ may be extracted from $(v_i)_{1 \le i \le B}$ where

$$v_i = \begin{bmatrix} D_i \\ 0 \end{bmatrix},$$

which in turn may be computed as follows.

1. In parallel, for each $k = 1, 2, \ldots, P$, let processor k compute via (3.4) the block of partial products,
$$M'_i = M_i \cdot M_{i-1} \cdot \ldots \cdot M_{(k-1)m+1},$$
for $i = (k-1)m+1, (k-1)m+2, \ldots, (km) \wedge B$. If $(k-1)m \ge B$ then processor k is idle.

2. Compute via (3.4) and (3.5),
$$v'_k = M'_{km} \cdot M'_{(k-1)m} \cdot M'_m \cdot v_0, \tag{3.6}$$
for $k = 1, 2, \ldots$, while $km < B$.

3. In parallel, for each $k = 1, 2, \ldots, P$, let processor k compute the block of final results,
$$v_i = M'_i \cdot v'_{k-1},$$
for $i = (k-1)m+1, (k-1)m+2, \ldots, (km) \wedge B$. Here $v'_0 = v_0$ by definition. If $(k-1)m \ge B$ then processor k is idle.

The time complexity is of the same order as in the computation of $(A_i)_{1 \le i \le B}$. The matrices $(M'_i)_{1 \le i \le B}$ consume an additional $2B$ memory locations; the second row of each such matrix is $[-\infty, 0]$, which need not be stored. An additional $O(P)$ memory locations supports the rest of the variables.

To summarize, the whole computation requires order

- $P + B$ memory locations,

- $B/P + \log_2 P$ time.

Note that, for $B > P \log_2 P$, the algorithm processes B jobs in order B/P time, which is optimal to within a constant factor.

Up to now we have assumed that the random variables $(\alpha_i)_{1 \le i \le N}$ and $(\delta_i)_{1 \le i \le N}$ were given. In many applications, these are assumed to be independent and so can be generated on the fly, on demand. Computation of sojourn-time statistics is easily added to phase 3 of the $(D_i)_{1 \le i \le N}$ computation. Just let each processor tally statistics for the D_i that it computes. At the end of the computation, combine the P tallies.

Given the $(A_i)_{1 \le i \le B}$ and $(D_i)_{1 \le i \le B}$ sequences, the sample path can be generated up to time A_B. To compute the sample path, merge the two sequences up to that time. A simple efficient parallel merge algorithm is described in the Appendix. The resulting merged sequence, (S_i), implicitly determines a sequence of $+1$'s and -1's marking the times when the queue length jumps up one (A_i's) and down one (D_i's). The partial sums of this ± 1 sequence give the queue lengths at the times of the jumps, and the differences between successive S_i give the times between jumps. With this information, queue length statistics can be computed efficiently in parallel. Computing the partial sums of the ± 1 sequence is yet another instance of the parallel prefix problem, and may be handled in the same way as the computation of $(A_i)_{1 \le i \le B}$. Departures $D_i > A_B$ must be held in store until the computation for the next batch.

Experiments

In our experiments, we chose the interarrival times $(\alpha_i)_{1 \le i \le N}$ and service times $(\delta_i)_{1 \le i \le N}$ be families of independent, identically distributed uniform random variables. We relied on the default random number generator supported by the machine, a Sun 3/260 for the serial experiments and a Sequent Balance 21000 for the parallel experiments. In the parallel experiments, each processor used a private copy of the generator initialized with a different seed. (See [20] for a more sophisticated way to provide the processors with quasi-independent random number streams.) In addition to computing $(A_i)_{1 \le i \le N}$ and $(D_i)_{1 \le i \le N}$, the average sojourn time, $1/N \sum_{i=1}^{N} (D_i - A_i)$, was tallied.

To contrast solving recurrences (3.1) and (3.2) to simulate the G/G/1 queue with the conventional event list method, we coded both methods in the SIMULA language. SIMULA provides convenient, low level support for the event list method. In all trials, the recurrence method turned out to be at least twice faster than the event list method.

A code implementing the parallel G/G/1 simulation method was written in the C language for the Sequent Balance 21000. As mentioned above, barrier synchronization takes $O(P)$ time on this machine, so we could perform phase 2 of the $(A_i)_{1 \le i \le N}$ and $(D_i)_{1 \le i \le N}$ computations serially in $O(P)$ time without significantly further lessening efficiency. In principle, the running time for simulating each batch of jobs should be on the order of $B/P + P$, and so the total time taken should be on the order of $(N/P)(1 + P^2/B)$. That is, the speedup should be linear with the slope increasing with B.

We carried out a series of experiments, varying the batch size B from 50 to 50000 and the number of processors P from 2 to 16. In each experiment a total of $N = 10^6$ jobs were simulated. Timing results are given in table 1. It is immediately clear that the algorithm is inefficient for small B, becomes efficient (doubling the number of processors almost halves

batch size	number processors			
	2	4	8	16
50	487	315	176	256
100	445	255	176	160
500	409	213	118	75
1000	405	207	109	64
5000	401	202	103	56
10000	401	202	104	55
50000	407	208	109	59

Table 1: Parallel G/G/1 Simulation Results: The table gives the running times (in secs) for the parallel simulation of $N = 10^6$ jobs, for a variety of batch sizes B and numbers of processors P.

the running time) for moderate B, and then becomes slightly less efficient for larger B.

The initial inefficiency is explained by the cost of processor coordination on the Sequent, a relatively coarse grained parallel processor. In our implementation the computation for each batch uses six barrier synchronizations, one for each of the three phases of the computations for A_i and D_i. Moreover, the total number of barrier synchronizations is larger for smaller values of B because the total number of batches to do, N/B, is larger. An execution profile revealed that even in efficient cases, such as $B = 10^5$ and $P = 16$, about 40% of a processors time was taken by the systems fork/join mechanism, which we used for barrier synchronization. About 10% was taken by random number generation, about 15% by the actual (double precision) computation, and the rest by a variety of overheads, including input and output. Efficiency dipped for very large B because of the added overhead in dynamic memory allocation. This dip disappears in experiments where N, the number of jobs simulated, is larger.

4. More Parallel Simulations

In this section we discuss parallel simulation methods for a variety of other queueing systems.

4.1. Acyclic Fork Join Networks

Acyclic fork-join (AFJ) queueing networks arise naturally in the performance analysis of parallel processing and flexible manufacturing systems [11]. Indeed, the parallel processing codes developed for this paper are simply and accurately modelled as AFJ networks. Queues in series and series-parallel queueing networks are special cases of AFJ networks.

An acyclic fork-join network consists of an acyclic network of FIFO queues, which serves a single stream of jobs whose interarrival times are given by the sequence $(\alpha_n)_{n\geq 1}$. An example of an AFJ network is depicted in figure 2. There are V nodes, indexed $i = 1, 2, \ldots, V$, connected via a set of directed edges E. As the network is assumed to be acyclic, we may label the nodes so that each edge (i, j) is such that $j > i$. If (i, j) is an edge, we say j is a successor of i and i is a predecessor of j. Nodes are assigned to integer *levels* as follows.

A node with no predecessors belongs to level 0. Any other node belongs to the level one greater than the maximum level of its predecessors.

A single copy of each job visits each node; at node i the n^{th} job's service requirement is a random variable δ_n^i. There is a single arrival stream, determined by the sequence of interarrival times $(\alpha_n)_{n\geq 1}$. On arrival, a copy of the n^{th} job is routed to each node at level 0. The edges leading out of each node model a *fork* and those leading in model a *join*. At every level except the last, after a node completes a job, a token for that job is routed to each of the node's successors. A copy of the job itself arrives to a given node instantaneously after the node has collected a token for that job from each of its predecessors. At the last level, no further work is propagated, though there is an implicit join. That is, a job's departure time from the whole system is the time that the last copy completes service at the last level.

More formally, let variables A_n^i and D_n^i denote the times that node i receives and completes the processing of its n^{th} job, respectively. All nodes i at level 0 receive their n^{th} job, for $n \geq 1$, at time

$$A_n^i = A_{n-1}^i + \alpha_n, \tag{4.1}$$

where $A_0^i = 0$. For each node j at any level > 0, the n^{th} arrival coincides with the maximum of the completion times of the n^{th} job from each predecessor of node j: for $n \geq 1$,

$$A_n^j = \bigvee_{(i,j)\in E} D_n^i. \tag{4.2}$$

Since the nodes operate FIFO, the variables D_n^i's satisfy the same recurrence as before:

$$D_n^i = (D_{n-1}^i \vee A_n^i) + \delta_n^i, \tag{4.3}$$

where $D_0^i = 0$. Letting F denote the set of nodes at the last level, the departure of the n^{th} job, for $n \geq 1$, from the network as a whole occurs at time

$$D_n = \bigvee_{i\in F} D_n^i. \tag{4.4}$$

To simulate the network, we solve the recurrence relations as functions of the interarrival times $(\alpha_n)_{n\geq 1}$ and service times $(\delta_n^i)_{n\geq 1, 1\leq i\leq V}$. This is straightforward to do serially: For $n = 1, 2, 3,...$ compute the variables describing the course of the n^{th} job starting at level 0 and increasing level by level, using the results of the computation for the $(n-1)^{st}$ job.

The parallel computation may be organized in a variety of ways, the simplest of which is to step through the network level by level, processing the nodes one by one at the current level, using the G/G/1 parallel simulation method. Specifically, let us compute arrival and departure times in batches of nominal size B. The time and memory required will each be on the order of V times that needed used to simulate the G/G/1 queue as described in section 3. Thus, for large B, the time requirement becomes optimal, order VB/P, at the cost of a memory requirement of VB. As will be clear, by exploiting the special structure of the network, sometimes the memory requirement can be much reduced. For example, if the network is a series of queues, just order B memory is needed.

In brief, the computation for the first batch may be performed as follows. Let K be the maximum level, and let F_k, $0 \leq k \leq K$ denote the set of nodes at level k. Assign all P processors to:

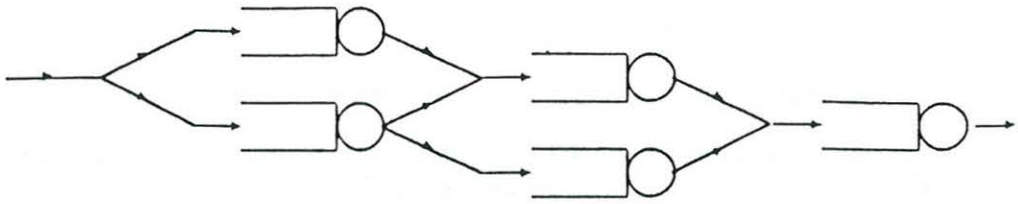

Figure 2: An acyclic fork-join queueing network.

1. Compute the arrivals $(A_n)_{1\le n\le B}$ as in the computation of the G/G/1 queue. This determines the arrivals $(A_n^i)_{1\le n\le B}$ for nodes $i\epsilon F_0$.

2. For $k = 1, 2, ..., K$, and for each node i at level k compute $(A_n^i)_{1\le n\le B}$ via (4.2). (If several nodes at level k have the same subset of nodes as their predecessors, the corresponding sequence of arrival instants needs to be computed only once.) Then compute $(D_n^i)_{1\le n\le B}$ via (4.3) as in the computation for the G/G/1 queue.

3. Compute the departure times $(D_n)_{1\le n\le B}$ via (4.4).

4.2. Acyclic FIFO Queueing Networks

In this section we consider acyclic queueing networks, of the usual (not fork-join) type, where jobs arrive at arbitrary nodes, follow arbitrary paths in the network, and then depart. Viewed separately, each node operates as a G/G/1 queue. We focus on networks of degree two–each node has at most two edges leading in and at most two leading out. The discussion easily generalizes to higher degree networks, though the time and memory needed for simulation increases with the degree.

There may be several classes of jobs, where each class has its own probabilistic routing rule controlling the choices of paths through the network. The routing rule is assumed to be state independent or oblivious, meaning that a job's route is a random function only of its class and index in the arrival sequence for that class. Unlike the fork-join networks of section 4.1, overtaking is possible. Jobs that arrive and depart from the same node but follow different paths may depart in a different order than the order in which they arrive.

Some important examples of acyclic queueing networks of this type are the networks used for routing in high speed switches and in parallel computers, such as the various Banyon networks [21, 22]. Figure 3 depicts an example of a three node acyclic network, which albeit simple, exhibits splitting and merging of job streams, and the possibility of overtaking. The nodes are labeled X, Y, and Z. We assume that there are two arrival class, determined by two sequences of random interarrival times. One class consists of jobs that enter at Y and follow the path YZ. The other class consists of jobs that enter at X and follow either the path XYZ or the path XZ; a job in this class takes the former path independently with probability p and the later with probability $1 - p$. We shall return to this example (specifying it further) below.

Now, consider a single node in an acyclic network, with two incoming edges, labeled i and j and two outgoing edges, labeled u and v, as shown in figure 4. Assume that each job carries the information whether it is to be routed out along u or along v. Let

- $(A_n^i)_{n\geq 1}$ and $(A_n^j)_{n\geq 1}$ denote the sequences of arrivals of jobs along edges i and j,
- $(A_n)_{n\geq 1}$ and $(D_n)_{n\geq 1}$ the sequence of arrivals of jobs as seen by the node in question and the corresponding sequence of departures, and
- $(D_n^u)_{n\geq 1}$ and $(D_n^v)_{n\geq 1}$ the subsequences of $(D_n)_{n\geq 1}$ of jobs routed along u and v.

Simulation of the job flow through this node can be done in three steps: merge, solve, split. The arrival sequence $(A_n)_{n\geq 1}$ is the merge of $(A_n^i)_{n\geq 1}$ and $(A_n^j)_{n\geq 1}$. Given $(A_n)_{n\geq 1}$, solving the G/G/1 recurrence (3.2) provides $(D_n)_{n\geq 1}$. With $(D_n)_{n\geq 1}$ in hand, we can split off the subsequences $(D_n^u)_{n\geq 1}$ and $(D_n^v)_{n\geq 1}$. It is straightforward to carry out these three step procedure serially, and to devise an efficient serial network simulation algorithm based on such a procedure.

Our ideas for a comparable parallel method are:

- Use a parallel merge procedure to form $(A_n)_{n\geq 1}$.
- Use the parallel method for solving a single G/G/1 queue to compute $(D_n)_{n\geq 1}$.
- Use a parallel enumerate/pack procedure (described below) to split out the subsequences $(D_n^u)_{n\geq 1}$ and $(D_n^v)_{n\geq 1}$.

Efficient parallel merging algorithms are known [10]. A simple, efficient parallel merging procedure appropriate for our purposes is described in the Appendix. Of course, we have already discussed the parallel computation of the job departure times for a G/G/1 queue. An enumerate/pack procedure takes as input a vector composed of two sequences, and produces as output two vectors holding the two subsequences separately. Splitting out the two subsequences may be accomplished by the following parallel "enumerate/pack" procedure. Suppose the jobs in the $(D_n)_{n\geq 1}$ sequence are held in the machine's memory in a buffer. Consider the jobs to be routed along edge u as marked 1 and those to be routed along edge v as unmarked. Use segmented parallel prefix computation (a simple variant of the original method, where unmarked inputs are skipped over) to determine the partial sums of this sequence and, for each $i \geq 1$, put the i^{th} job into the i^{th} slot of a memory buffer allocated for the subsequence $(D_n^u)_{n\geq 1}$. Reversing the ordering and applying the same method provides the jobs in $(D_n^v)_{n\geq 1}$ in another buffer.

To adapt these ideas to produce a realizable simulation method, we must deal with memory limitations. In this paper we propose a very simple parallel method similar to the method described in section 4.1 for AFJ networks. At a high level the idea is, as before, to work in iterations, first generating a batch of new arrivals for each class of jobs, and then pushing (most of) those arrivals through the network, level by level, node by node using parallel merge, solve, split routines at each node. Efficiency is gained by the parallel processing of large numbers of jobs per iteration. Though the method is adequate for small networks, it is rather wasteful of memory so more sophisticated methods (currently under investigation) are needed for large networks. We shall illustrate the method by giving the details for the three node example of Figure 3.

Returning to this example, let us suppose that the long term arrival rates of the two job classes are known: λ_X for the class entering the network at node X and λ_Y for the other. A job is represented as a record, with fields to hold the arrival time to the network (needed to tally sojourn times), the arrival time to the current node, and the departure time from the current node. The computation is organized in batches of nominal size B. We shall use several vectors of job records as buffers associated with the edges of the network:

- buffer J_X of size $B\lceil\lambda_X/(\lambda_X+\lambda_Y)\rceil$ associated with the edge leading into X,
- buffer J_Y of size $B\lceil\lambda_Y/(\lambda_X+\lambda_Y)\rceil$ associated with the edge leading into Y, and
- buffers J_{XY}, J_{XZ}, J_{YZ}, and J_Z, each of size B, associated with the edges XY, XZ, YZ, and the edge leading out of Z.

At each iteration, these vectors will be overwritten. At most B jobs will be processed per iteration. The sizes of buffers J_X and J_Y were chosen taking into account the arrival rates for the two job classes, so that, for large B, the number processed will be close to B. It is *not* crucial that each of the other buffers be large enough to hold all B jobs, but it makes the algorithm simpler.

The first iteration proceeds as follows.

1. (Generate Arrivals) Fill buffers J_X and J_Y with jobs by solving the G/G/1 recurrence (3.1) using the appropriate interarrival times. Let f be the minimum of the arrival time of the last job in J_X and the arrival time of the last job in J_Y. All jobs with arrival time $\leq f$ shall be processed in this iteration; the others are held back for the next iteration and are not considered further in this iteration.
2. (Process Node X) Solve the G/G/1 recurrence (3.2) to compute the departure times for the jobs in J_X. Toss a coin with bias p for each job in J_X to determine whether it takes the edge to Y or the edge to Z, and use the enumerate/pack procedure to split out the two resulting job streams into buffers J_{XY} and J_{XZ}, sorted by their departure times from node X.
3. (Process Node Y) Merge the jobs in J_{XY} (keying on the departure times from X) with the jobs in J_Y (keying on the arrival times to Y), storing the resulting sequence in buffer J_{YZ}. Solve the G/G/1 recurrence (3.2) for these jobs to compute their departure times from node Y.
4. (Process Node Z) Merge the jobs in J_{YZ} with those in J_{XZ} (keying on the newly computed departure times), and store the result in J_Z. Compute the departure times for these jobs, and tally network sojourn times.

At any subsequent iteration, the only additional problem is including the jobs left over from the previous iteration in one of the buffers J_X or J_Y. A simple solution is to manage those buffers circularly so that, without any special bookkeeping, each can be filled to capacity at each iteration. We chose the sizes of buffers J_X and J_Y taking into account the arrival rates for the two job classes, so that about B jobs on average should be processed per iteration.

We are developing serial and parallel simulation codes for this example network. Preliminary results tell the same basic story as that for the G/G/1 simulation codes.

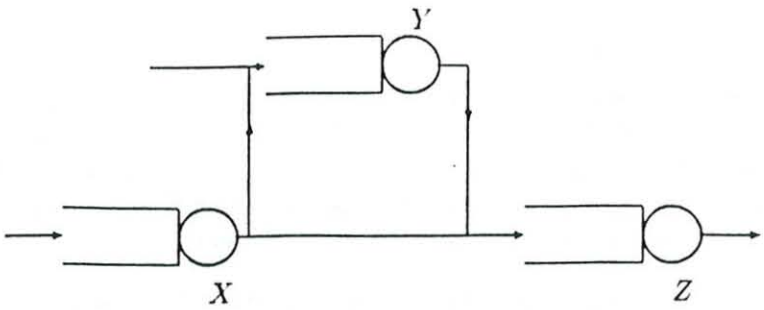

Figure 3: An acyclic queueing network.

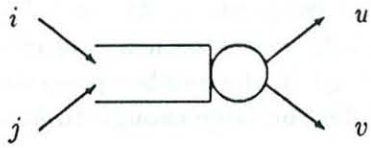

Figure 4: One node in an acyclic queueing network.

4.3. FIFO Queues in Series with Bounded Buffers

In this section, we treat series of queues with bounded buffers, motivated mainly by the study of production lines [23]. An example of four queues in series is depicted in Figure 5. The k^{th} server serves the k^{th} buffer in FIFO order, eventually forwarding the jobs to the $(k+1)^{st}$ buffer. This server may block (stop working) if, in particular, the k^{th} buffer is empty or the $(k+1)^{st}$ buffer is full. Several specific blocking rules can be modelled; for concreteness, we consider only *transfer* blocking and *minimal* blocking [23].

Assume that there are K queueing stations, indexed $k = 1, 2, ..., K$. Station k consists of a FIFO server and a buffer with finite capacity $C_k \geq 1$. An infinite buffer precedes station 1 and acts as the source for jobs. An infinite buffer follows station K and acts as the sink for jobs. We label these two infinite buffers 0 and $K+1$ respectively. In a moment, we will define transfer blocking and a corresponding system of recurrences, describing the system's sample path. We will then do the same for minimal blocking. It turns out, once again, that the recurrences are linear in the $(\vee, +)$ semiring, but are higher *order* than the recurrences for the G/G/1 queue. The order increases with the number of queues and the maximum buffer size.

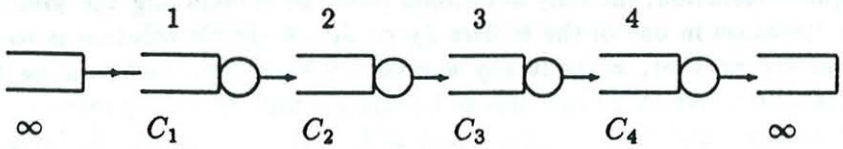

Figure 5: Four queues in series

Transfer Blocking

Under the *transfer* blocking rule, the server at each station k continuously attempts to serve jobs from buffer k, blocking if buffer k is empty or buffer $k+1$ is full. Specifically, after each service completion, if buffer $k+1$ is not full, then the newly completed job is instantaneously transferred to buffer $k+1$. Otherwise, the server is blocked until the instant a slot opens in buffer $k+1$, at which point the job is instantaneously transferred.

Let us define variables A_n^k, δ_n^k, and T_n^k, referring to the n^{th} job handled at station k, as the time the job arrives to the buffer, the service time requirement at this station, and the time service is completed at this station, respectively, for $n = 1, 2, \ldots,$ and $k = 1, 2, \ldots, K$. The following recurrences [23] determine these variables (and thereby the entire sample path):

$$A_{n+1}^k = T_{n+1}^{k-1} \vee A_{n+1-C_k}^{k+1} \tag{4.5}$$

$$T_{n+1}^k = (A_{n+1}^k \vee A_n^{k+1}) + \delta_{n+1}^k \tag{4.6}$$

where by definition $T_n^0 = 0$ and $A_n^{K+1} = T_n^K$. Equation (4.5) states that the $(n+1)^{st}$ job arrives at station k when either the job is completed at the preceding station (if station k is unblocked at that time), or when the $(n+1-C_k)^{th}$ job arrives to the next station. The justification of (4.6) is that the $(n+1)^{st}$ job goes into service at station k either the moment the job arrives (if the server is free at that moment), or the moment the n^{th} job moves to the next buffer.

Substituting (4.6) into (4.5), we are led to

$$A_{n+1}^k = (A_{n+1}^{k-1} + \delta_{n+1}^{k-1}) \vee (A_n^k + \delta_{n+1}^{k-1}) \vee A_{n+1-C_k}^{k+1} \tag{4.7}$$

Upon solving (4.7) for the A_n^k's, substituting into (4.6) gives the T_n^k's. To solve (4.7) we require an ordering of index pairs that puts $(n+1,k)$ higher in the order than the indices (n,k), $(n+1,k-1)$, and $(n+1-C_k,k+1)$ appearing on the right hand side of (4.7). A little thought leads to the pairing function $J(n,k)$ that achieves the desired effect economically by ordering index pairs along diagonals as illustrated in Table 2. Moreover, $J(n,k)$ is easily computable. Under this ordering, (4.7) becomes a linear recurrence in the $(\vee,+)$ semiring, of order $M \leq KC_{max}$ where $C_{max} = C_1 \vee \ldots \vee C_K$ is the maximum buffer capacity. That is, the i^{th} term in the recurrence depends only on terms $i-1$, $i-2$, ..., $i-M$.

Minimal Blocking

Minimal blocking uncouples serving jobs from transferring jobs between buffers and thereby achieves higher throughput [23]. Whereas under transfer blocking, the buffer at station k holds only jobs awaiting service at this station, under minimal blocking the buffer may also hold jobs that have already completed service at this station. Completed jobs are held until slots open up in the next buffer. Specifically, if after a service completion at station k buffer $k+1$ is full, then the job remains in buffer k. However, server k is set to work on the next uncompleted job in its buffer, if there is one. The instant a slot opens up in buffer $k+1$, the oldest completed job present at station k is transferred to fill the slot, without interrupting the server at station k.

Under minimal blocking, the recurrence relation for completion times becomes

$$T_{n+1}^k = (A_{n+1}^k \vee T_n^k) + \delta_{n+1}^k, \tag{4.8}$$

n \ k	1	2	3	4	5
1	1	2	4	7	11
2	3	5	8	12	16
3	6	9	13	17	21
4	10	14	18	22	26
5	15	19	23	27	31
6	20	24	28	32	36
⋮	⋮	⋮	⋮	⋮	⋮

Table 2: Pairing function $J(n,k)$, for $K = 5$, $1 \leq k \leq K$, and $n \geq 1$.

and the recurrence relation for A_n^k is unchanged. The justification for (4.8) is that service for job $n+1$ starts either the moment the job arrives (if the server is idle at that moment), or the moment the previous job completes service at this station.

The two recurrences are now interdependent so one set of terms cannot be eliminated by substituting one equation in the other as we did for transfer blocking. However, using the pairing function $J(n,k)$ to totally order the terms of each recurrence and applying a standard transformation, we can obtain a single linear recurrence of order about $2KC_{max}$, which determines both the A's and the D's.

Simulation

It is straightforward to solve the recurrences serially, following the order $J(n,k)$ depicted in Table 2. This gives a simple, efficient serial simulation method.

To obtain a comparable parallel method, we need only a fast parallel method for solving M^{th} order linear recurrences. As in section 3, where we described an algorithm for first order recurrences, this problem can be reduced to solving a first order linear matrix recurrence [8]. Adapting the algorithm of [8], the first N terms of such a recurrence can be computed using P processors in time proportional to

$$\frac{NM^{w-1}}{P} + M^w$$

where w is the exponent of the matrix multiplication algorithm used: $w = 3$ for the standard algorithm, $w = 2.81$ for Strassen's algorithm. Once again, to limit the memory locations needed to a nominal number $O(B)$, the computation can be organized in N/B iterations, where B terms of the recurrence are computed in each.

5. FINAL REMARKS

Posing simulation problems using recurrence relations opens up possibilities for new approaches to proving that some systems cannot be simulated with a high degree of parallelism. For example, Kung [24] has shown that in general a recurrence of algebraic degree > 1 can be sped up by at most a constant factor, regardless of how many processors are

available. Working backwards, if one can show that simulating a given discrete event system solves a hard recurrence in the sense of [24], then there can be no efficient parallel simulation of that system.

APPENDIX

In this section we sketch a simple, practical parallel method for merging a sorted list of length N with another of length M, in time proportional to $(N+M)/P+\log_2(N+M)$. In our applications, we can often control N and M so that the maximum of the two is as large as we like, and the $\log_2(N+M)$ term is swamped. We are grateful to Andrew Odlyzko for proposing the method.

Suppose the input lists are presented as vectors A and B of N and M locations respectively, sorted in increasing order. Assume for simplicity that all elements of the two lists are distinct. A vector F of $N+M$ locations is to hold the merge of A and B, also sorted in increasing order. Let $m = \lceil (N+M)/P \rceil$. The key observation is that given the P percentiles of F,

$$F[m], F[2m], F[3m], ..., F[Pm \wedge (N+M)],$$

it remains only to carry out P independent, standard serial merges of sublists of A and B filling in the $\leq m$ elements between percentiles. In brief,

1. In parallel, for each $k = 1, 2, ..., P$, let processor k determine $f = F[(km) \wedge (N+M)]$, and determine the indices of the largest elements of A and B that are $\leq f$.

2. In parallel, for each $k = 1, 2, ..., P$, let processor k fill in

$$F[(k-1)m+1], F[(k-1)m+2], ..., F[km \wedge (N+M)]$$

using the information computed in step 1.

Step 2 is carried out in time proportional to m, via the standard serial merge procedure. A little thought shows that the task of each processor k at step 1 is essentially the same as that of searching two sorted lists of equal length j for the j^{th} largest element in the merge of the two lists, which can be done via a binary search method in time proportional to $\log_2 j$. The time for step 1 is proportional to $\log_2(N+M)$.

References

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

[2] B.D. Lubachevsky. Efficient distributed event-driven simulation of multiple-loop networks. *Communications of the ACM*, 32(1):111, January 1989.

[3] J. Misra. Distributed-discrete event simulation. *ACM Computing Surveys*, 18(1):39–66, March 1986.

[4] D.M. Nichol. Parallel discrete-event simulation of fcfs queueing networks. In *Parallel Programming: Experience with Applications, Languages, and Systems*, pages 124–137. ACM SIGPLAN, July 1988.

[5] D.B. Wagner and E.D. Lazowska. Parallel simulation of queueing networks: Limitations and potentials. In *1989 ACM Sigmetrics and Performance Evaluation Review and Performance '89, International Conference on Measurement and Modelling of Computer Systems (sponsored by ACM Sigmetrics and IFIP W.G. 7.3), special issue 17,1*, pages 146–155. ACM Press, May 1989.

[6] L. Kleinrock. *Queueing Systems, Volume 1.* Wiley, 1975.

[7] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[8] A.G. Greenberg, R.E. Ladner, M. Paterson, and Z. Galil. Efficient parallel algorithms for linear recurrence computation. *Information Processing Letters*, 15(1):31–35, August 1982.

[9] L. Hyafil and H.T. Kung. The complexity of parallel evaluation of recurrences. *Journal of the ACM*, 24:513–521, 1977.

[10] C.P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, TC-32:942–946, 1983.

[11] F. Baccelli, W. Massey, and D. Towsley. Acyclic fork-join queueing networks. *Journal of the ACM*, 36(3):615–642, July 1989.

[12] F. Baccelli. Ergodic theory of stochastic petri nets. Technical Report 1037, INRIA-Sophia Antipolis, INRIA-Sophia 06565 Valbonne, France, May 1989.

[13] K.M. Chandy and B. Sherman. The conditional event approach to distributed simulation. In *Distributed Simulation 1989*. The Society for Computer Simulation, 1989.

[14] N. Abrahamson. Development of the alohanet. *IEEE Transactions on Information Theory*, IT-31(2):119–123, March 1985.

[15] T. Leighton. An introduction to the theory of networks, parallel computation and vlsi design, 1989. draft.

[16] I. Mitrani. *Simulation Techniques for Discrete Event Systems.* Cambridge University Press, 1982.

[17] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10), October 1985.

[18] A.G. Greenberg and B.D. Lubachevsky. A simple efficient parallel prefix algorithm. In *1987 International Conference on Parallel Processing*, pages 66–69, 1987.

[19] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[20] O.E. Percus and M.H. Kalos. Random number generation for mimd parallel processors. *Journal of Parallel and Distributed Computing*, 6(3):477–498, June 1989.

[21] T.Y. Feng. A survey of interconnection networks. *Computer*, 14:12–27, 1981.

[22] H.J. Siegel, W. Nation, C.P. Kruskal, and L.M. Napolitano. Uses of the multistage cube network topology. to appear in the Proceedings of the IEEE, 1989.

[23] D. Mitra and I. Mitrani. Control and coordination policies for systems with buffers. In *1989 ACM Sigmetrics and Performance Evaluation Review and Performance '89, International Conference on Measurement and Modelling of Computer Systems (sponsored by ACM Sigmetrics and IFIP W.G. 7.3), special issue 17,1*, pages 156–164. ACM Press, May 1989.

[24] H.T. Kung. New algorithms and lower bounds for the parallel evaluation of certain rational expressions. In *6th Annual ACM Symposium on Theory of Computing*, pages 323–333. ACM, 1974.

DISCUSSION

Rapporteur: Rogério de Lemos

Professor Henderson asked what were the implications that storage had, for example in a computation involving 10^{16} jobs. Dr Mitrani replied that as far as shared memory architectures were concerned memory had no effects as all the processors accessed the same memory, and that simulation was carried out in batches. Dr Mitrani went on to say that in such shared memory systems only the batch size could influence the required memory size.

Mr Kerr asked if such simulation method could be used in a system like a toll gate with the purpose to obtain, for example the number of cars of a certain type that pass the gate. Dr Mitrani replied by saying that once the different classes of vehicles were defined, then there would not exist any restriction in using the method.

Professor Randell pointed out that special machines have been built for simulation, and asked if the simulation method presented could be an alternative to those machines. Dr Mitrani answered that the method could not be applied to all kind of problems, for example logic simulation; its application is restricted to those considered in the lecture, such as the problems where the jobs arrival times and departure times are related. Professor Randell also asked if there were other types of architectures, apart from the shared memory systems where this method could be applied. Dr Mitrani answered that a large simulations have been realized on a connection machine, but, at the moment, he had no measures to show.