Efficient Distributed Simulation of a Communication Switch with Bursty Sources and Losses

A.S.McGough and I. Mitrani
Computing Science Department, University of Newcastle,
Newcastle upon Tyne, NE1 7RU
a.s.mcgough@ncl.ac.uk isi.mitrani@ncl.ac.uk

Abstract

Algorithms for simulating an ATM switch on a distributed memory multiprocessor are described. These include parallel generation of bursty arrival streams, along with the marking and deleting of lost cells due to buffer overflows. These algorithms increase the amount of computation carried out independently by each processor, and reduce the communication between the processors. When the number of cells lost is relatively small, the run time of the simulation is approximately $O(N/P)$, where N is the total number of cells simulated and P is the number of processors. The cells are processed in intervals of fixed length; that length affects the structure and the performance of the algorithms.

1. Introduction

Consider an ATM switch with transmission capacity C cells / second (i.e. a service time of $c = 1/C$). The switch contains a finite buffer of size Q cells, which is filled from arrivals generated by merging M independent bursty sources of the "on"/"off" type, see figure 1 below. Suppose that the performance measure of interest is the cell loss probability, i.e. the long–term fraction of cells that are lost due to buffer overflow. If the "on", "off" and cell inter-arrival intervals for the different sources are different and generally distributed, that quantity cannot normally be determined by analysis. On the other hand, estimating the loss probability by simulation tends to be a very time-consuming task because the overflow events are usually rare and so a large number of cell arrivals and departures have to be generated in order to obtain an accurate result.

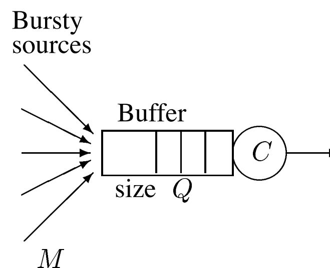

Figure 1. ATM Switch with multiple sources

In order to reduce these large simulation times, considerable effort has gone into exploring parallel computation techniques. In particular, the parallel simulation of Multiplexers, as used in ATM networks, has attracted much attention over recent years. Space-parallel simulation techniques, where different nodes of the network are allocated to different processors, have been proposed in [6, 7, 8]. These techniques do not help in speeding up a long simulation of a single ATM node. Nikolaidis [9] and Fujimoto [10] presented a method of simulating an ATM multiplexer at the level of bursts. Since individual cells are not simulated, this approach can only provide approximate results or ones that require particular assumptions. Wang and Abrams [11] presented another approximate method for the parallel simulation of the G/G/1/n queue. That method becomes exact when the service times are constant but it requires the computation of all departure times as well as all arrivals. A similar approach to ours is adopted by Andradóttir and Ott [13]. They apply time-parallel and relaxation techniques to the simulation of queuing networks. However, they do not handle bursty arrivals and do not report the achieved speedup in either shared-memory or distributed environment. Chen [3] presents an alternative approach to the parallel simulation of finite buffers, based on longest-path algorithms to compute departure times. That approach does not apply easily to our model.

A parallel simulation algorithm for the present model was described in [5]. It used the parallel prefix approach

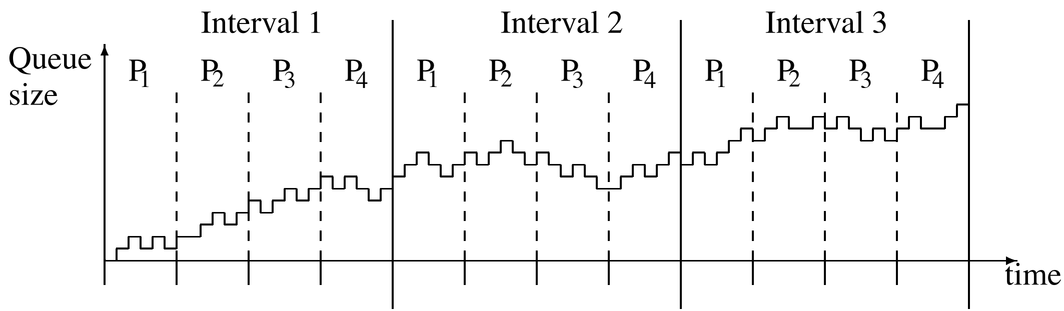

Figure 2. The division of the sample path into intervals

of [1, 12], together with parallel merge and relaxation techniques for deciding which cells are lost. That algorithm achieved almost linear speedup on a shared-memory multiprocessor. However, if it is run without modification on a distributed memory multiprocessor, such as a cluster of workstations connected by a fast Ethernet, the benefits of parallelism are overcome by the communication overheads. While generating the merged arrival stream, large amounts of data have to be passed around among the processors. In that environment we have found that increasing the number of processors (up to eight) does not reduce significantly the simulation execution time.

That is why it is necessary to develop different and more efficient parallel simulation algorithms, which is the subject of this paper. The emphasis of the new approach is to use a method for generating arrivals which allows the majority of cells to be generated and handled on the correct processor. This allows the merging of streams and the marking of lost cells to be done locally, thus significantly reducing the communication costs.

1.1. Outline of the distributed algorithm

The total simulation period is broken up into intervals of B seconds. Within each interval the work is divided approximately equally between the P processors. Figure 2 illustrates a sample path where a simulation over the interval 0 to $3B$ is split among four processors.

If the chosen value for B is great enough then the average amount of work performed by each processor will be approximately the same. This coupled with low communication costs between processors allows the simulation to reach almost linear speed–up.

The actions taken by the processors in parallel are:

- **1. Generate all arrivals for the next B seconds.** Processor k first computes the “on”and “off”periods for arrival source i that fall within its own sub-interval of the time line. It can then compute the list of arrivals from that source which occur during this sub-interval. These two steps are performed using the modified version of the parallel prefix algorithm described in section 2 below. Some arrivals may be generated on the incorrect processor due to the random nature of the arrival sources. These arrivals need to be communicated to the correct processor. Provided that the number of such cells is small, in comparison with the total number of arrivals generated, this should have little effect on the performance of the simulation. Repeating these steps for all sources and merging the resulting cells produces the total arrival stream that processor k needs to handle.

- **2. Mark and remove lost cells.** The algorithm used for this task is an adaptation of an algorithm introduced in [5]. It can be used with an arbitrary sized collection of arrivals, and works by generating a step function of the queue size over a given time interval. Each processor generates a portion of the sample path corresponding to its own sub-interval, assuming some initial conditions for the state of the queue. That step is iterated using new initial conditions obtained from the previous processor and passing new initial conditions to the next one. This process, known as ’relaxation’, continues until two consecutive iterations produce identical sample paths. If the cell loss fraction is small then the number of iterations should be small.

The technique of using relaxation to refine the current state of knowledge of individual processors was discussed, in general context, by Chandy and Sherman [4]. Relaxation does not always help, but it can be implemented efficiently in our case.

To obtain a point estimate and a confidence interval for the cell loss probability, it is enough to compute the number of cells, L, that are lost during the simulation period. It should be pointed out, however, that with simple modifications the above algorithms can generate other performance measures for the ATM switch, such as average buffer occupancy or average cell response time.

It is also worth pointing out that these algorithms can be modified to accommodate more general models, including cells of different priority types, reservation of buffer space for higher priority cells, and sources with dependent “on”and “off”periods (provided that the “on”periods are large compared to the inter-arrival times).

Finally, it should be mentioned that although we have considered a continuous time model (arrival instants are real and transmissions can start at arbitrary points) for this work, the method can easily be adapted to a discrete–time model.

The following sections provide a more detailed description of the stages described above.

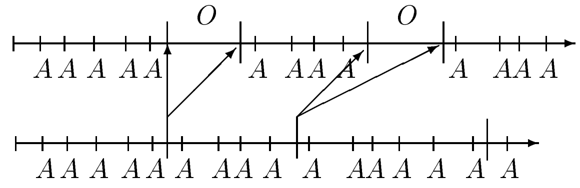

Figure 3. Unadjusted and adjusted arrival instants

2. Generation of arrival instances

It is assumed that the bursty nature of each source can be simulated by an alternating sequence of “on” periods during which cells arrive, and “off” periods without arrivals. All sources start with an “on” period. The n th “on” , “off” and inter-arrival periods for source i are denoted by $\xi_{i,n}$, $\eta_{i,n}$ and $\alpha_{i,n}$, respectively. These are sequences of i.i.d. random variables with general distributions.

The generation of a sequence of arrivals during an interval of length B for source i is carried out in two steps. First, the “off” periods are ignored and an ‘unadjusted’ arrival sequence is calculated as if the source was “on” all the time (see the lower part of figure 3, where the “off” periods have been condensed to 0).

The ‘unadjusted’ arrival time of cell n from source i, $a_{i,n}$, satisfies the following recurrence relation:

$$a_{i,n+1} = a_{i,n} + \alpha_{i,n+1} \ \ ; \ \ n = 1, 2, \dots \ , \tag{1}$$

where $\alpha_{i,n+1}$ is the inter-arrival interval between cells n and $n+1$. These recurrences can be solved in parallel by applying the parallel prefix algorithm (see [1]). A total of N_i arrival instants can be calculated on P processors in time on the order of $O(N_i/P)$ when N_i, the number of arrivals that are generated from source i over the interval of length B, is much larger than P.

The second step consists of adjusting $a_{i,n}$ by inserting the missing “off” periods in the appropriate positions (see upper part of figure 2, where the inserted “off”periods are denoted by O). To ascertain how many “off” periods occurred before a particular ‘unadjusted’ arrival time, the index of the “on” period during which $a_{i,n}$ occurs must be determined. Given the lengths of the consecutive “on” periods for source i, $\xi_{i,j}$, we need to find, for each $a_{i,n}$, an index l such that

$$\sum_{j=1}^{l-1} \xi_{i,j} < a_{i,n} \leq \sum_{j=1}^{l} \xi_{i,j} \ , \tag{2}$$

where an empty sum is 0 by definition.

Having solved the inequalities (2) for l, the actual arrival instant of cell n from source i, $A_{i,n}$, is obtained from

$$A_{i,n} = a_{i,n} + \sum_{j=1}^{l-1} \eta_{i,j} \ . \tag{3}$$

The adjustment procedure described above assumes that the realizations of $\xi_{i,j}$ and $\eta_{i,j}$ have been pre-computed. Since the total number of “on” and “off” periods that are generated during the simulation is typically much smaller than the total number of cells, we treat that pre-computation as an overhead. Of course, the sequences of partial sums for $\xi_{i,j}$ and $\eta_{i,j}$ can also be obtained by means of the parallel prefix algorithm, and infact are in this case.

Solving (2) and (3) is essentially equivalent to merging the two sequences of arrival and “off” instants. Since there are many more arrival instances than “on”/ “off”periods, that operation together with the calculation of the actual arrival times, can be carried out on P processors in time approximately equal to $O(N_i/P)$.

2.1. Generation of “off”and “on”periods

The first step of the new algorithm for generating the arrival instances is to compute the prefix sums of “on”and “off”periods involved in (2) and (3). Each of the P processors computes all “on”and “off”instants that occur within a sub-interval of length B/P. For processor k that sub-interval is $[(k-1)B/P, kB/P]$, $k = 1, 2, ..., P$. All processors follow the four stages of the algorithm outlined below. For simplicity the algorithm only describes the computation of the first interval of the simulation. All other intervals are processed in a similar manner.

1. Processor k assumes that the beginning of its sub-interval, $(k-1)B/P$, is the start of an “on”period for every source. It then computes sets of partial sums $\Xi'_{i,j}$ and $\Theta'_{i,j}$ according to the following equations.

$$\Xi'_{i,j} = \xi_{i,1} + \xi_{i,2} + ... + \xi_{i,j} \ \ j = 1, 2, 3, ... \ ; \tag{4}$$

$$\Theta'_{i,j} = \eta_{i,1} + \eta_{i,2} + ... + \eta_{i,j} \quad j = 1, 2, 3, ... \quad . \quad (5)$$

where i indexes the source and j indexes the "on"/"off"pair. Partial sums are computed until

$$(k-1)B/P + \Xi'_{i,j} + \Theta'_{i,j} > kB/P \quad i = 1, 2, ..., M. \quad (6)$$

Denote the values of $\Xi'_{i,j}$ and $\Theta'_{i,j}$ which satisfy (6) by $\Xi_i^{(k)}$ and $\Theta_i^{(k)}$ respectively.

2. Processor k sends $\Xi_i^{(k)}$ and $\Theta_i^{(k)}$ for each i to processors $k+1, k+2, ..., P$. It also receives similar values from all the processors $1, 2, ..., k-1$. Processor k computes

$$\Xi_i = \Xi_i^{(1)} + \Xi_i^{(2)} + ... + \Xi_i^{(k-1)} \quad i = 1, 2, ..., M \quad ; \quad (7)$$

$$\Theta_i = \Theta_i^{(1)} + \Theta_i^{(2)} + ... + \Theta_i^{(k-1)} \quad i = 1, 2, ..., M \quad . \quad (8)$$

3. The true "on"and "off"starting points are computed as

$$\Xi_{i,j} = \Xi'_{i,j} + \Xi_i \quad j = 1, 2, 3, ... \quad ; \quad (9)$$

$$\Theta_{i,j} = \Theta'_{i,j} + \Theta_i \quad j = 1, 2, 3, ... \quad . \quad (10)$$

4. Find the last pair $(\Xi_{i,j}, \Theta_{i,j})$ which satisfies

$$\Xi_{i,j} + \Theta_{i,j} < kB/P. \quad (11)$$

Copy that, and all subsequent pairs $(\Xi_{i,j}, \Theta_{i,j})$, to processor $k+1$. Receive similar pairs from processor $k-1$. This is necessary for computing the burst of arrivals that may straddle the boundary between the kth and the $k+1$st sub-intervals. Processor k then renumbers its (increasing) sequences $\Xi_{i,j}$ and $\Theta_{i,j}$ such that $\Xi_{i,1} + \Theta_{i,1} < (k-1)B/P$. Thus the first "on"-"off"cycle for processor k in fact starts before the beginning of its sub-interval.

2.2. Distributed generation of arrival instances

In this section we present the algorithm for generating the arrival instances from all sources. To do that for source i, start by eliminating all corresponding "off"periods and consider the "on"periods joined end to end. On this 'compressed' time line the start of the k'th sub-interval, for source i, moves from $(k-1)B/P$ to

$$s_i^{(k)} = \begin{cases} \Xi_{i,1} & \text{if } (k-1)B/P \text{ is in an "off"period} \\ (k-1)B/P - \Theta_{i,1} & \text{otherwise.} \end{cases}$$

The arrival generation algorithm proceeds as follows.

1. Processor k starts by assuming that there is an arrival instance for source i at time $s_i^{(k)}$. It computes the partial sums

$$a'_{i,n+1} = a'_{i,n} + \alpha_{i,n+1} \quad ; \quad n = 1, 2, ... \quad , \quad (12)$$

until

$$s_i^{(k)} + a'_{i,n} > s_i^{(k+1)} \quad . \quad (13)$$

Denote the first value $a'_{i,n}$ which satisfies (13) by $a_i^{(k)}$.

2. Processor k sends $a_i^{(k)}$ for each i to processors $k+1, k+2, ..., P$. It also receives similar values from all the processors $1, 2, ..., k-1$. Processor k computes

$$a_i = a_i^{(1)} + a_i^{(2)} + ... + a_i^{(k-1)} \quad i = 1, 2, ..., M \quad . \quad (14)$$

3. The arrival instances for source i on the 'compressed' time line are computed as

$$a_{i,n} = a'_{i,n} + a_i \quad n = 1, 2, 3, ... \quad . \quad (15)$$

4. Any arrival instances which satisfy

$$s_i^{k+r} < a_{i,n} < s_i^{(k+r+1)}. \quad (16)$$

are sent to processor $k+r$, where they will finish their processing.

5. The true arrival time $A_{i,n}$ can now be computed by first finding the index l of the relevant "on"period. That index satisfies

$$\Xi_{i,l-1} < a_{i,n} < \Xi_{i,l} \quad . \quad (17)$$

Adjust $a_{i,n}$ by adding to it the sum of all previous "off"periods:

$$A_{i,n} = a_{i,n} + \Theta_{i,l-1} \quad . \quad (18)$$

In practice, when the sub-interval length B/P is large compared to the inter-arrival times, step 4 only requires arrivals to be passes to processor $k+1$.

The arrivals from all M sources are then merged to produce the full list of arrivals within the kth sub-interval.

3. Mark and remove lost cells

Denote, for convenience, the merged arrival instants in the current sub-interval by A_n, $n = 1, 2, \ldots$ (in practice, the numbering carries on sequentially from one sub-interval to the next). It is now necessary to determine, in parallel, which cells are accepted into the buffer and which are lost as a result of finding it full. An algorithm to achieve this result is presented below.

3.1. Acceptance Algorithm

Processor k now computes the queue size at the arrival instants in its sub-interval, assuming some initial conditions. The latter are then refined in subsequent iterations. For the purpose of determining the lost cells, it is only necessary to calculate some of the departure instants.

For cell n within a sub-interval, let q_n be the queue size 'just before' A_n; this is the queue size 'seen' by the incoming cell. Also, let d_n be the time of the last departure before A_n if $q_n > 0$; otherwise $d_n = A_n$. For the first cell, assume initially that $q_1 = 0$ and $d_1 = A_1$. This definition of d_n is illustrated in figure 4. The actual departure times D_n are marked on for clarity.

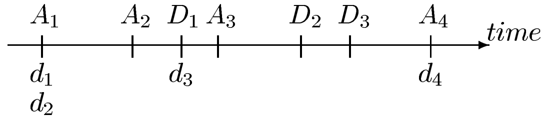

Figure 4. The values of d_n

Clearly, cell n is accepted if $q_n < Q$ and is lost otherwise. Denote by σ_n the indicator of that event:

$$\sigma_n = \begin{cases} 1 & \text{if } q_n < Q \\ 0 & \text{if } q_n = Q. \end{cases}$$

Let δ_n be the number of cell transmissions that can be completed in the interval (d_n, A_{n+1}):

$$\delta_n = \left\lfloor \frac{A_{n+1} - d_n}{c} \right\rfloor ,$$

where $\lfloor x \rfloor$ is the largest integer not exceeding x.

Now, the values of q_n and d_n are computed by means of the following recurrence relations:

$$q_{n+1} = \max(q_n + \sigma_n - \delta_n, 0) , \tag{19}$$

$$d_{n+1} = \begin{cases} d_n + \delta_n c & \text{if } q_{n+1} > 0 \\ A_{n+1} & \text{if } q_{n+1} = 0. \end{cases} \tag{20}$$

These equations rely on the fact that cell service times are constant. If that is not the case, they would be modified in a straightforward manner, but would still remain recurrences.

Processor k solves (19) and (20) for its sub-interval, using known or assumed initial values of q_1 and d_1 (in the case of processor 1, these are known from the previous interval; the other processors start by assuming that their first cell arrives into an empty buffer). The computed values of q_n and d_n for the last cell in the sub-interval are then passed to processor $k+1$ and serve as the latter's new initial values. This procedure is iterated until the new initial conditions of all processors are the same as the old ones. In the worst case P iterations are required, but when the number of losses is small, fewer iterations suffice.

There are several strategies that can be employed to reduce the amount of computation performed by each processor during an iteration. They are based on the following ideas:

1. Let I_k be the total idle time during sub-interval k, as computed by processor k in one of the iterations:

$$I_k = \sum_n I(q_n = 0)[A_n - d_{n-1} - c(q_{n-1} + \sigma_{n-1})] , \tag{21}$$

where the summation is over all arrival instants in the sub-interval, and $I(x) = 1$ if the event x occurs, 0 otherwise. Suppose that $I_k > cQ$, i.e. a full buffer can be cleared during an interval of length I_k. Then the index of the last accepted cell in the sub-interval, and the queue size seen by that last cell, are independent of the initial conditions. Hence, the new initial conditions for processor $k+1$ are correct and it can perform its final iteration, regardless of the future state of processor k.

2. More generally, if for a given iteration the increase of the initial queue size (passed from processor $k-1$) does not exceed the old value of I_k/c, then the new initial conditions for processor $k+1$ will be the same as the old ones.

3. If j consecutive cells, $\{n+1, n+2, \ldots, n+j\}$, have the property that none of them are lost and none of them, except perhaps the first, finds an empty buffer, then that collection can be treated as a single 'packet' for the purpose of calculating the evolution of the queuing process. Instead of computing j pairs of recurrences (19) and (20), a single pair is evaluated:

$$q_{n+j} = q_n + j - \delta_{n,j} , \tag{22}$$

$$d_{n+j} = d_n + \delta_{n,j} c , \tag{23}$$

where

$$\delta_{n,j} = \left\lfloor \frac{A_{n+j} - d_n}{c} \right\rfloor .$$

The effectiveness of this technique has not been evaluated empirically. However, it appears to have the potential for reducing the amount of computation significantly.

Figure 5 illustrates the application of **1** & **2** above, where the interval is 20 seconds long and the buffer size Q is 3, with the work distributed over four processors. During the first iteration each processor computes the queue size trajectory for all of the cells in its sub-interval. Note that the algorithm only computes the queue size immediately before and after a cell arrival, the departures have been added to the

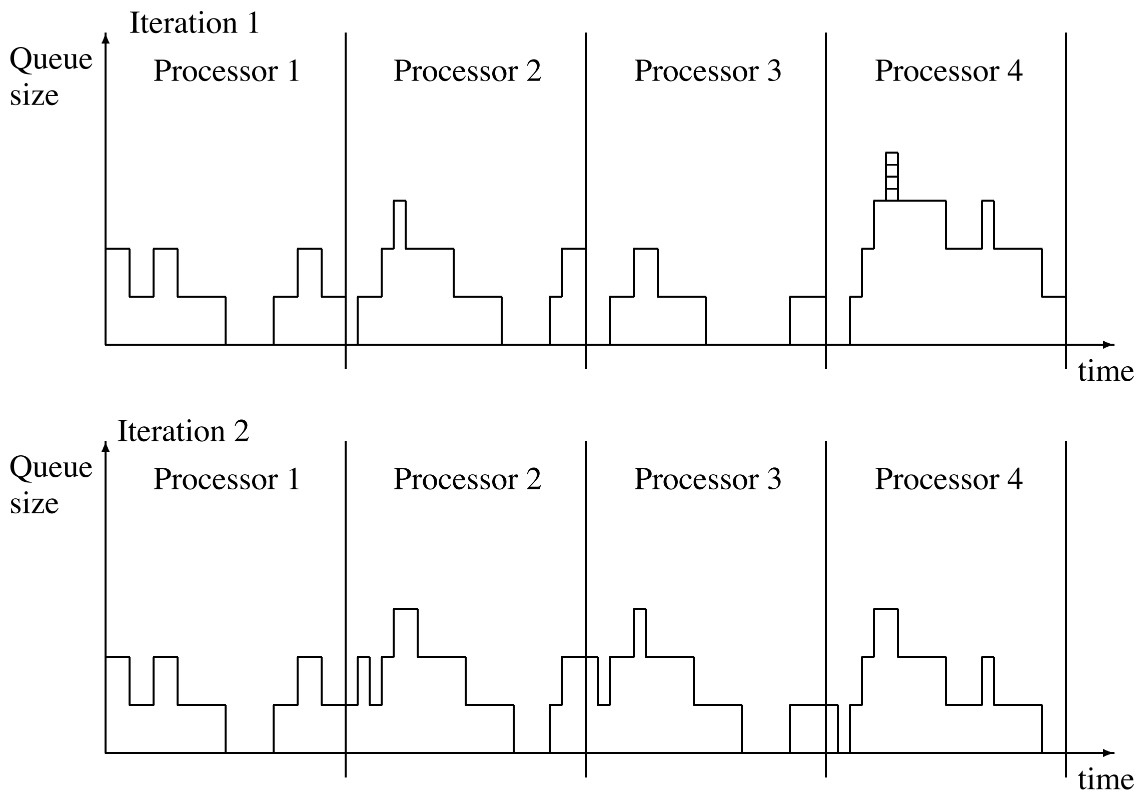

Figure 5. Example graph of queue sizes

diagram for clarity. In the case of processor 4 its fourth arrival will exceed the queue size, marked in the diagram as a shaded block. However in a future iteration this cell may be accepted, thus it is marked as lost here but is still kept in the list of arrivals.

Processor 2 contains enough idle time to absorb one cell in the queue from the previous processor. Processor 3 contains enough idle time to absorb any valid queue size from the previous processor. All cells that arrive to processor 3 after cQ idle time can be computed as final and the end state of its sub-interval is finalized.

In iteration 2 the end state (queue size after the last arrival and time of the last departure) from each processor is passed onto the next processor as the start state. Processor 2 receives a queue of size 1 from processor 1 and needs only to compute those cells that arrive before the first point where the queue size reaches zero, likewise for processor 3. Processor 4 receives a queue size of one and marks cell 4 as lost. It is removed from the arrival list and does not appear in the diagram for the final iteration. The end conditions are now passed from processor k to processor $k+1$, as these new start conditions are identical for all processors the iterations terminate.

4. Experimental Results

The results were generated from running the test program on a cluster of eight PentiumII 233Mhz workstations, connected by fast Ethernet. The simulation was written using the LAM [14], implementation of MPI [15], running under Linux. Experimental results were also produced from the same cluster with the addition of a shared memory quad processor system running at 450MHz. This allows the number of processors to be increased to twelve.

Varying numbers of processors were used to produce results for two ATM systems, the first having eight bursty sources and the second having 24 sources. The offered load was chosen to ensure that the fraction of lost cells was just under 10^{-4}. Each simulation run represents 10^7 seconds of simulated time. During that time, approximately 1.2×10^7 cell arrivals occured in all cases. For simplicity, the cell inter-arrival times, the “on”and “off”periods were assumed to be exponentially distributed, with the cell transmission time assumed to be constant.

Since the object of this study is to examine the efficiency of the parallel simulation algorithms and the speedups that can be achieved, the only metric plotted is the ratio T_0/T_n, where T_0 is the execution time of the best *sequential* sim-

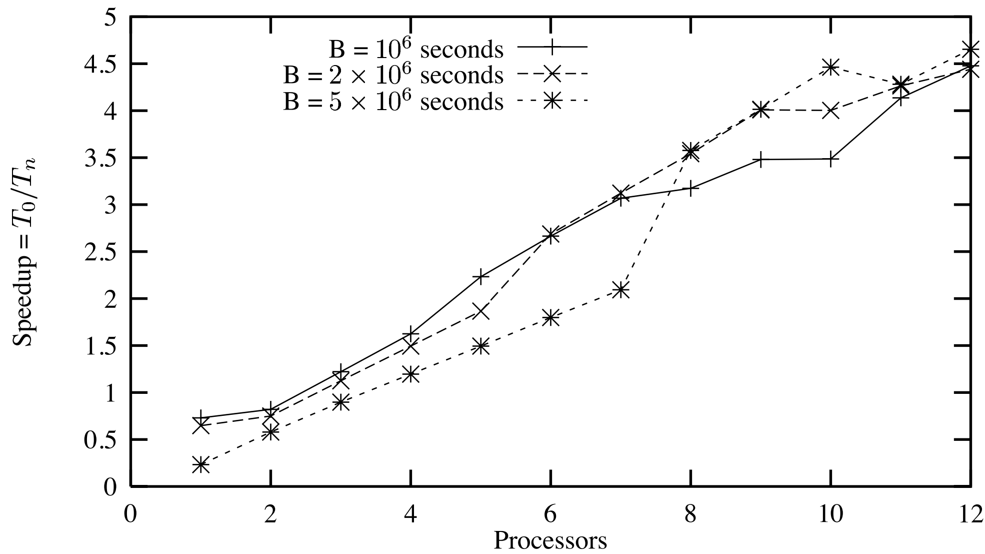

Figure 6. Results from a cluster with 8 input sources

ulation run on a single processor and T_n is the execution time of the parallel simulation run on n processors. This ratio is commonly known as the 'speedup' achieved by the algorithm. Note that T_1, the execution time of the parallel simulation run on 1 processor, is normally larger than T_0. If T_0/T_n is proportional to n, the parallel algorithm is said to achieve 'linear speedup'. Statistics about the lost cells were collected, but neither the point estimates nor the confidence intervals are displayed in the following graphs.

Figures 6 and 7 illustrate the speedup achieved as a function of the number of processors. Each figure shows the results for running the simulation with different interval sizes B. Figure 6 illustrates the situation with eight input sources, showing almost linear speedup for all interval lengths. Altering the interval length appears to have little effect on the overall speedup of the simulation. For very large intervals there is a difference, although slight. There is an apparent jump in performance between seven and eight processors, when the batch size is 5×10^6. This seems to be a consequence of the removal of page swapping as the amount of data handled by each processor decreases as the processor count increases.

Processors nine to twelve are included by using the use of the quad processor workstation. These processors are faster than the others and therefore compute their sections of the simulation path quicker. However, since there is no attempt at load balancing, the trend of the speedup remains as before.

Figure 7 shows similar results for the case of 24 bursty input sources. Here again we observe an almost linear speedup. The larger jump between seven and eight processors for $B = 5 \times 10^6$ is present here too, and probably has the same explanation.

5. Conclusion

We have demonstrated that a large sample path for a nontrivial communication system can be simulated in parallel on a distributed cluster of processors. Moreover, the parallelization is efficient, in the sense that a linear speedup is achieved. The major obstacle that has been overcome by the algorithms presented here is the large amount of data communicated between the processors. Normally the communication overheads swamp the benefits of parallelism. We have been able to eliminate most of the overheads by delegating more intelligence to the individual processors. Each processor is now able to decide accurately which arrivals to generate, so that they can be handled locally.

Dependencies between sub-intervals, due to lost cells, require some repetition of work performed by processors. However these iterations can be carried out efficiently without destroying the benefits of parallel simulation.

The relaxation techniques described here are not restricted to this model. The idea that each processor can work on an interval of the sample path, subsequently refining its knowledge in light of information received from other processors, can be applied to many different systems. However, the details of that allocation can have an important effect on performance. Unless all intervals converge quickly to their final states, the advantage of parallel processing can be lost.

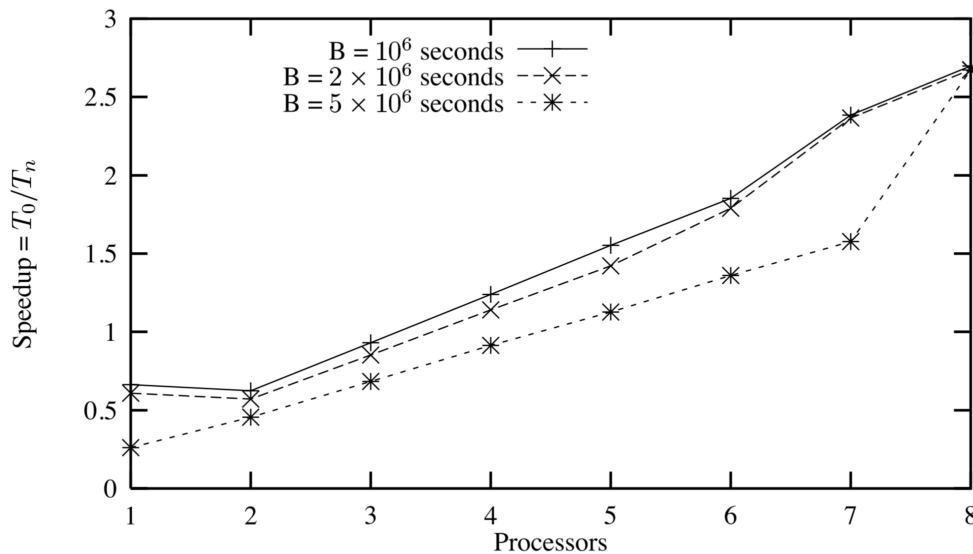

Figure 7. Results from a cluster with 24 input sources

References

[1] A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani. Algorithms for Unboundedly Parallel Simulations, ACM TOCS, 91(9):201-221, August 1991.

[2] C.P. Kruskal, L. Rudolph and M. Snir. The Power of Parallel Prefix, IEEE Trans. Comp., 85(34):965-968, October 1985.

[3] L. Chen. Parallel Simulation by multi-instruction, longest-path algorithms, Queueing Systems, 97(27 no 1-2):37-54, 1997.

[4] K.M. Chandy, R. Sherman. Space-Time and Simulation, Proceedings of the SCS Multiconference on Distributed Simulation, Tampa, Florida, Society for Computer Simulation, 89:53-57, July 1989.

[5] A.S. McGough, I. Mitrani. Parallel Simulation of ATM Switches using Relaxation, IFIP ATM'98, 98(54), July 1998.

[6] Z. Xiao, B. Unger, R. Simmonds, J. Cleary. Scheduled Critical Channels in Conservative Parallel Discrete Event Simulation, PADS '99, 99:20-28, May 1999.

[7] C. Williamson, B. Unger, Z. Xiao. Parallel Simulation of ATM Networks: Case Study and Lessons Learned, CCBR '98, 98:78-88, June 1998.

[8] C.D. Carothers, K.S. Perumalla. Efficient Optimistic Parallel Simulations using Reverse Computation, PADS '99, 99:126-135, May 1999.

[9] I. Nikolaidis, R. Fujimoto, C.A. Cooper. Time-Parallel Simulation of Cascaded Statistical Multiplexers, ACM Sigmetrics, 94:231-239, May 1994.

[10] R.M. Fujimoto, I. Nikolaidis, C.A. Cooper. Parallel Simulation of Statistical Multiplexers, Discrete Event Dynamic Systems-Theory and Applications, 95(5):115-140, April 1995.

[11] J.J. Wang, M. Abrams. Approximate Time-Parallel Simulation of Queuing systems with losses, 1992 Winter Simulation Conference, 92:700-708, December 1992.

[12] F. Baccelli, M. Canales. Parallel Simulation of Stochastic Petri Nets Using Recurrence Equations, ACM Transactions on Modeling and Computer Simulation, 93(Vol. 3, No. 1):20-41, January 1993.

[13] S. Andradóttir, T.J. Ott, Time-Segmentation Parallel Simulation of Networks of Queues with Loss or Communication Blocking, ACM Transactions on Modeling and Computer Simulation, 95(Vol. 5, No. 4):269-305, October 1995.

[14] LAM / MPI Parallel Computing, http://www.mpi.nd.edu/lam/, 09/1999.

[15] Message Passing Interface Forum, http://www.mpi-forum.org/, 09/1999.