Parallel Simulation of ATM Switches using Relaxation

A.S. McGough and I. Mitrani

Computing Science Department, University of Newcastle,
Newcastle upon Tyne, NE1 7RU
E-mail: a.s.mcgough@ncl.ac.uk isi.mitrani@ncl.ac.uk

Abstract

Algorithms for simulating an ATM switch on a number of parallel processors are described. These include parallel generation and merging of bursty arrival streams, marking and deleting of lost cells due to buffer overflows, and, in one version of the algorithm, computation of departure instants. When the number of lost cells 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 batches of fixed size; that size affects both the structure and the performance of the algorithms.

1 Introduction

Consider an ATM switch with transmission capacity C cells/second, a buffer of size Q cells and an input stream formed 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 interarrival 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.

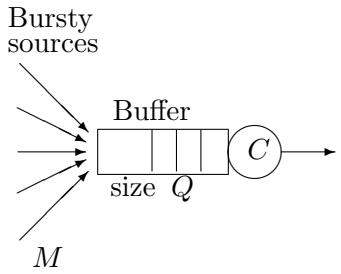

Figure 1: ATM Switch with multiple sources

We attack the above problem by developing an efficient parallel simulation algorithm for the ATM switch. This enables several processors to be employed simultaneously, thus increasing the computing power and reducing the total required time. The algorithm manages to achieve an almost linear speed-up, in the sense that if a total of N cells are to be simulated on P processors in parallel, and the buffer overflow events are reasonably rare, then the run time of the simulation is roughly $O(N/P)$.

Our approach dispenses with the normal concepts of event scheduling and event list. Instead, the simulation task is reduced to that of solving a set of recurrence relations, computing the

sequences of arrival and departure instants, and then modifying them in order to take account of lost cells. That approach is based on ideas which were introduced in [1], in the context of open queueing networks without losses.

The novel aspects of the present development arise from the buffer overflows. Cell losses introduce possible dependencies between the processors. These are handled by means of relaxations.

The speed-up of the algorithm is a consequence of the fact that the following two operations can be implemented efficiently on a number of processors (e.g., see [2, 3, 4]):

- Parallel Prefix
- Parallel Merge

On the other hand, some degradation in performance is caused by the iterative treatment of cell losses.

Cells are processed in batches of size B, in parallel by all P processors. The actions taken are:

1. Generate B cell arrival instants for each source. The "on"and "off" periods for source i are generated first; then the consecutive arrival instants, $A_{i,n}$, $n = 1, 2, \ldots, B$. This step uses the Parallel Prefix algorithm.

2. Merge the sources. This is done in parallel, until a total of B arrival instants have been obtained. Any arrivals left over are saved for the next batch. The merging algorithm is based upon the one presented by [4].

3. Mark and remove lost cells. Two algorithms are presented for this stage: the first requires that the batch size, B, is equal to the buffer size, Q. That requirement, together with the sequence of departure instants from the previous batch, simplifies the process of deciding which cells in the current batch are lost. The P processors may need to exchange information about the cells in their sub-batches in order to make those decisions. Because of that, the algorithm may need to be iterated. However, when there are few losses, both the number of iterations and the amount of work per iteration are small. After finalizing the accepted arrivals, this algorithm computes the sequence of cell departure instants (again using Parallel Prefix), ready for the next batch.

The technique of using relaxation to refine the current state of knowledge of individual processors was discussed, in a general context, by Chandy and Sherman [6]. Chen [5] presents an alternative approach to parallel simulation of finite buffers, based on longest-path algorithms to compute departure times. That approach does not apply easily to our model.

Our second algorithm allows cells to be handled in batches of arbitrary size. It constructs a space-time graph of the queue size over the period of B arrivals. Each processor builds its own sub-section of the graph, assuming some initial conditions for the state of the queue. Relaxation is then used to finalize each sub-section. Again if the losses are low then the number of iterations is small.

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

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

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

2 Generate B arrival instants for each source

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 during which they do not. All streams start with an "on" period. The n th "on" , "off" and interarrival 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 B arrival instants 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 2, 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 interarrival 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 B arrival instants can be calculated on P processors in time on the order of $O(B/P)$ when 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). 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 an index k such that

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

where an empty sum is 0 by definition.

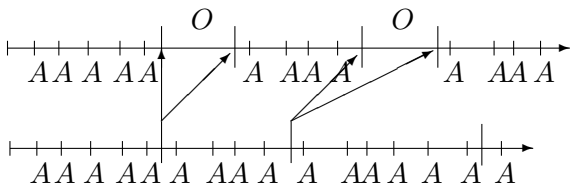

Figure 2: Unadjusted and adjusted arrival instants

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

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

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.

Solving (2) and (3) is essentially equivalent to merging the two sequences of arrival and "off" instants. Since the second is much shorter than the first, that operation, together with the calculation of the actual arrival times, can be carried out on P processors in time approximately equal to $O(B/P)$.

2.1 Merging of sources

Having computed B arrival instances from each of the M sources, these are merged to produce the next batch of B arrivals into the ATM buffer. When B is large compared to M and P, this can be done in parallel on P processors in time $O(B/P)$, using existing algorithms with appropriate modifications (see [2, 4]). Some care needs to be exercised in order to ensure that the work is divided approximately equally among the processors. After some binary searching, each of the source sequences provides an appropriate sub-sequence to each of the processors. The latter then merge their respective subsequences in parallel.

Typically, only a fraction of the arrival instances from each source are used in the merged batch. The remainder (of which there are $(M-1)B$ in total) are kept for the following batch.

3 Mark and remove lost cells

Denote, for convenience, the merged arrival instants in the current batch by A_n, $n = 1, 2, \ldots, B$ (in practice, the numbering carries on sequentially from one batch 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. Two algorithms that achieve this objective by different methods are presented below.

3.1 Algorithm 1.

The batch size, B, is chosen to be equal to the size of the buffer, Q. This algorithm also requires that the departure instants of the previous B accepted cells have already been computed. Denote those instants by D_n, $n = 1-B, 2-B, \ldots, 0$ (again, in practice the departure instants are numbered sequentially throughout the simulation).

Given this information, the acceptance/loss of certain cells can be decided independently of the others. More precisely:

1. If cell n arrives after the departure of cell $n-B$, i.e. if $A_n \geq D_{n-B}$, then it finds space in the buffer and is therefore accepted.

2. If cell n arrives before any of the cells from the previous batch have departed, i.e. if $A_n \leq D_j$, $j = 1-B, 2-B, \ldots, 0$, then it finds the buffer full and is lost.

If neither of these conditions is satisfied, then

$$D_{n-B-j} < A_n < D_{n-B} \ , \tag{4}$$

for some $0 < j < B$. Let j be the *smallest* integer for which (4) holds. In that case, cell n is accepted if at least j preceding arrivals from the current batch are lost, otherwise it is lost.

This situation is illustrated in figure 3, where $B = Q = 5$. Cells 1 and 2 are lost, according to criterion 2; cell 5 is accepted, by criterion 1. What happens to cell 3 cannot be determined without knowing the outcome for cells 1 and 2: since $D_{-4} < A_3 < D_{-2}$, cell 3 is accepted if both earlier cells are lost (which is in fact the case). Similarly, since $D_{-2} < A_4 < D_{-1}$, cell 4 is accepted if at least one of cells 1, 2, 3 is lost (which is the case).

The B arrivals in the current batch are divided into subsequences numbered $1, 2, \ldots, P$, in chronological order. These are processed in parallel by processors $1, 2, \ldots, P$, respectively. Given the depardure instants of the previous batch, processor k applies criteria 1 and 2, and classifies each cell in its subsequence as (a) accepted, (b) lost and (c) possibly accepted. In case (c), a note is made of the number, j_n, of cells *in previous subsequences* that have to be lost in order for cell n to be accepted (case (c) does not arise in processor 1, which has enough information to determine the acceptance/loss of all cells in its subsequence). In addition, processor k computes the total number of lost cells, l_k, and the total number of not accepted cells, L_k, in its subsequence.

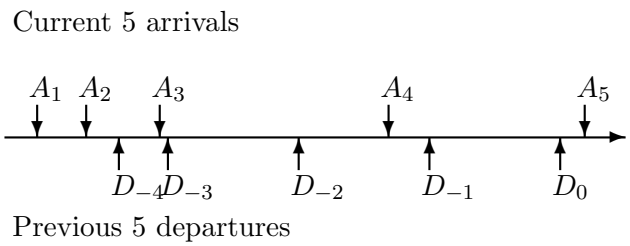

Figure 3: Determining cell acceptance and loss; $B = Q = 5$

Next, the partial sums

$$ll_k = \sum_{i=0}^{k-1} l_i \quad ; \quad LL_k = \sum_{i=0}^{k-1} L_i \ , \tag{5}$$

are evaluated and passed to processor k, for $k = 2, 3, \ldots, P$. This allows the latter to reassess the classification of the possibly accepted cells: if $j_n \leq ll_k$ then cell n is reclassified as accepted; if $j_n > LL_k$ then cell n is reclassified as lost; otherwise it remains possibly accepted. The values of l_k and L_k are updated in the light of this reassessment.

The above steps are iterated until $l_k = L_k$ for all $k = 1, 2, \ldots, P$ (i.e., until there are no uncertain cells). In the worst case, this requires P iterations (subsequence 2 is finalized on the 2nd iteration, subsequence 3 on the 3rd, etc), making the algorithm effectively sequential. However, if the number of lost cells is small, then a small number of iterations – 1 or 2 – suffices.

After removing the lost cells, the batch is replenished with new or stored arrival instants until it contains B accepted arrivals. This task, which can be treated as overhead, is carried out sequentially by one processor.

Having determined the arrival instants, A_n, of the B accepted cells in the current batch, their departure instants, D_n are calculated by solving the following recurrences:

$$D_{n+1} = \max(A_{n+1}, D_n) + c \ , \tag{6}$$

where $c = 1/C$ is the time to transmit one cell. This is a known problem whose solution is reduced to a parallel prefix operation by the introduction of a special matrix product in the (max,+) algebra (see [1]).

Thus, when B is large compared to P and the cell loss probability is small, the run time of algorithm 1 is on the order of $O(B/P)$. However, the requirement $B = Q$ limits the achievable speed-up if the buffer size of the ATM switch is not large. The following algorithm removes that restriction.

3.2 Algorithm 2

This algorithm allows cells to be handled in arbitrarily large batches. Again the B arrivals in the batch are divided into P subsequences and are processed in parallel by the P processors, but processor k now computes the queue size at the arrival instants in its subsequence, 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 subsequence, 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.

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

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

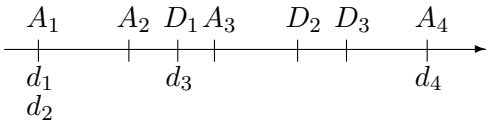

Figure 4: The values of d_n

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

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

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:

$$q_{n+1} = \max(q_n + \sigma_n - \delta_n, 0) , \tag{7}$$

$$d_{n+1} = \begin{cases} d_n + \delta_n c & \text{if } q_{n+1} > 0 \\ A_{n+1} & \text{if } q_{n+1} = 0 \end{cases} . \tag{8}$$

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 (7) and (8) for its subsequence of arriving cells, using known or assumed initial values of q_n and d_n (in the case of processor 1, these are known from the previous batch; 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 subsequence 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. As with algorithm 1, 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 subsequence k, as computed by processor k in one of the iterations:

$$I_k = \sum_n I(q_n = 0)[A_n - d_{n-1} - c(q_{n-1} + \sigma_{n-1})] , \tag{9}$$

where the summation is over all arrival instants in the subsequence, 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 subsequence, 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 new 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, $\{n, n+1, \ldots, n+j\}$, have the property that none of them are lost and none of them finds an empty buffer, then that collection can be treated as a single 'packet' for the putpose of calculating the evolution of the queueing process. Instead of computing j pairs of recurrences (7) and (8), a single pair is evaluated:

$$q_{n+j} = q_n + j - \delta_{n,j} c , \tag{10}$$

$$d_{n+j} = d_n + \delta_{n,j} c , \tag{11}$$

where

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

This technique, together with simple tests for deciding that a group of cells constitutes a packet, can reduce the amount of computation significantly.

4 Experimental Results

The algorithms described in the previous sections were implemented on an Encore Multimax 520 MIMD computer system with 12 processors. Different sets of processors were used for the simulation of two ATM systems – one with 6 and one with 24 input sources. The offered load was chosen so that the fraction of lost cells was between 0.008 and 0.009. Each run simulated a total of 1000000 cells. For simplicity, the cell interarrival times, the "on"periods and the "off"periods were assumed to be exponentially distributed. Of course, those assumptions could be changed very easyly. The cell transmission times were 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 performance measure considered is the elapsed time of a simulation run (measured in seconds). Statistics about the lost cells were collected, but neither the point estimates nor the confidence intervals are displayed in the following graphs

Figure 5 illustrates the performance of algorithm 1 ($B = Q$) for different buffer sizes and 6 input sources. For comparison, the run time of a serial simulation run on 1 processor is shown as a horisontal line. As expected, the smallest batch size yields the worst speed-up. The fact that a batch size of 50000 is slightly worse than that of 10000 appears counter-intuitive, but can be explained by the increased overheads associated with virtual memory and paging.

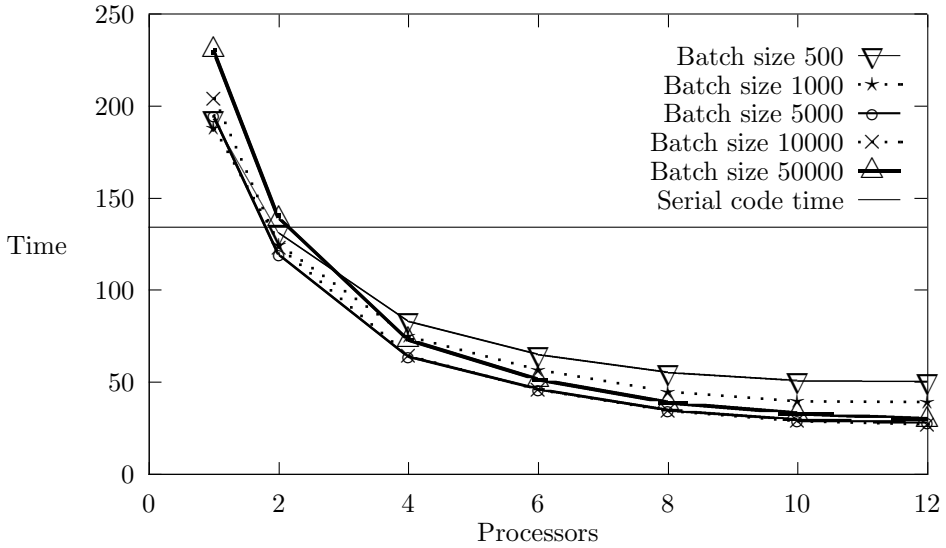

Figure 5: Algorithm 1; 6 input sources

Figure 6 shows a similar set of results for algorithm 2. The buffer size is kept constant, $Q = 1000$, while the batch size varies. The extra curve added to this figure represents the performance of algorithm 1 for $B = Q = 1000$. It can be seen that, with more than 6 processors, algorithm 1 slightly outperforms algorithm 2 on the same batch size, despite having to generate all departure instants. This is due to the fact that it is more efficient at quickly recognising lost and accepted cells. However, the advantage of being able to work with larger batches, thereby reducing communication overheads between processors, makes algorithm 2 ultimately preferable. Another feature of this figure is that, for $B = 500$, 12 processors take longer to

run the simulation than 10. This is an illustration of the fact that the penalty of additional communications may outweigh the benefit of more parallelism.

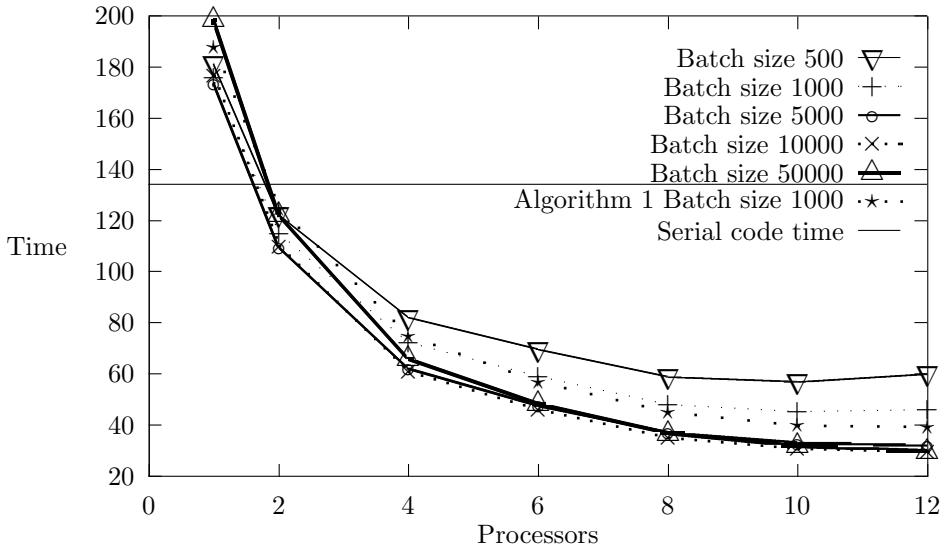

Figure 6: Algorithm 2; 6 input sources

Increasing the number of input sources to 24 (figures 7 and 8) has the obvious effect of slowing down the simulations performed on few processors, since more time is spent on generating and merging arrivals. Indeed, in order to beat the serial simulation, it is generally necessary to use 3 or more processors. On the other hand, with more than 8 processors and large batch sizes, the elapsed times are not very different from those in figures 5 and 6.

Comparing the performance of the two algorithms we see again that, when the buffer size is small or moderate, using algorithm 2 with a larger batch size is better than algorithm 1.

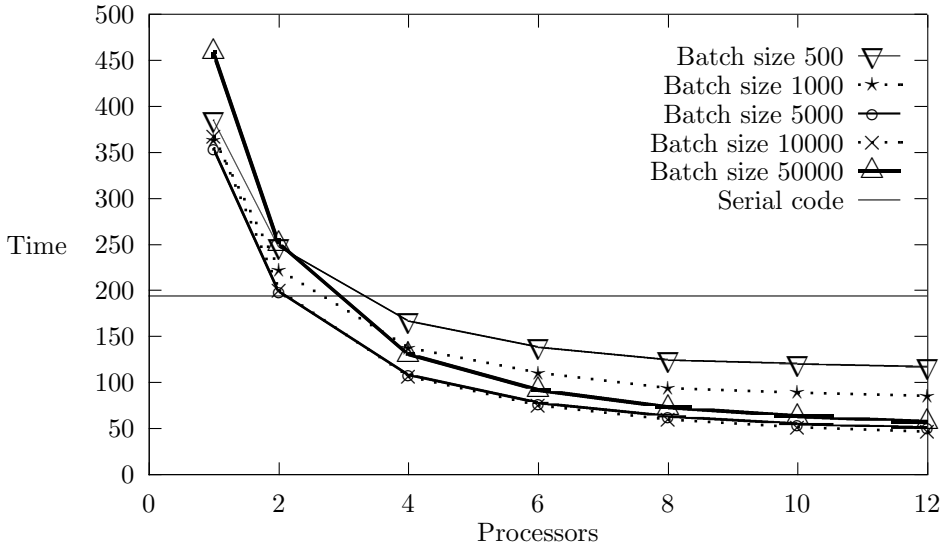

Figure 7: Algorithm 1; 24 input sources

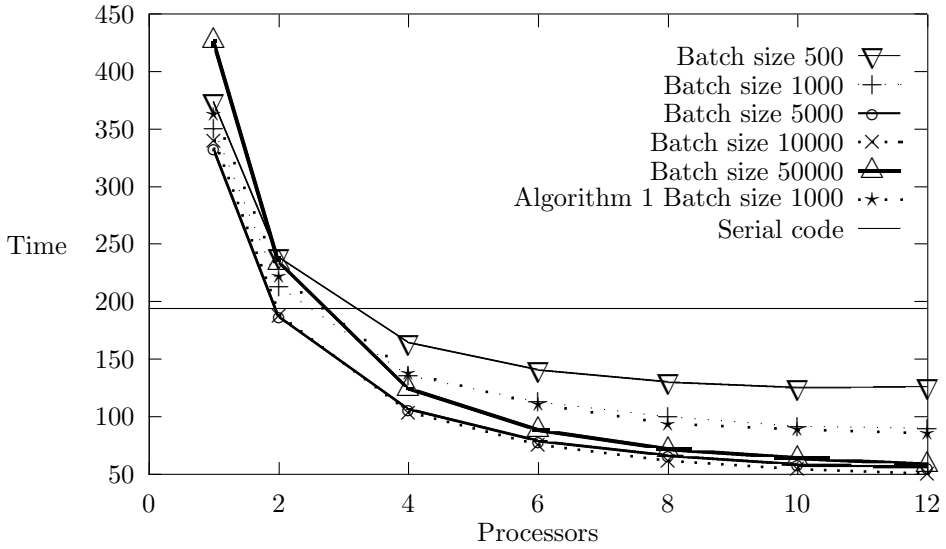

Figure 8: Algorithm 2; 24 input sources

5 Conclusions

Our experiments show that it is indeed possible to exploit efficiently the parallelism inherent in the simulation of ATM switches. Dependencies between different sections of the sample path, caused by lost cells, make it necessary for individual processors to iterate and repeat work already done. Nevertheless, signifficant speedups can be obtained in the pressence of loss probabilities on the order of 1%. This is a higher rate of buffer overflow than would be tolerated in most real ATM switches.

Two algorithms for marking and removing lost cells are proposed. The first uses fewer iterations, in general, but requires that the batch size is equal to the buffer size. The second tends to be less efficient in handling the dependencies between processors, but allows arbitrarily large batch sizes. It is therefore recommended that algorithm 1 should be used when the buffer size is reasonably large, whereas algorithm 2 should be applied in systems with small buffers.

Obviously, both algorithms can be implemented in a distributed environment where processors communicate with each other by means of messages. Then their performance is likely to be subject to different trade-offs. For instance, sharing the information about all departure instants from the previous batch (as required by algorithm 1), can become an expensive task. These issues need to be investigated further.

It is clear that the relaxation approach to parallel simulation is not restricted to the particular model considered here. The idea that each processor can work on a portion of the sample path, subsequently refining its knowledge in the 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 portions converge quickly to their final versions, the advantage of parallel processing can be lost.

References

[1] A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani, "Algorithms for Unboundedly Parallel Simulations", ACM TOCS, **9**, 201-221, 1991

[2] C.P. Kruskal, "Searching, Merging and Sorting in Parallel Computation", IEEE Trans. Comp., **TC-32**, 942-946, 1983

[3] C.P. Kruskal, L. Rudolph and M. Snir "The Power of Parallel Prefix", IEEE Trans. Comp., **TC-34**, 965-968, 1985

[4] Abali B., Bataineh A., "Balanced Parallel Sort on Hypercube Multiprocessors", IEEE Trans. Parallel and Distributed Systems, **4**, 572-581, 1993

[5] Chen L., "Parallel Simulation by multi-instruction, longest-path algorithms", Queueing Systems, **27**, 37-54, 1997

[6] Chandy K.M., Sherman R., "Space-Time and Simulation", In "Distributed Simulation 1989", Society for Computer Simulation, 1989, pp. 53-57.