Parallel Simulation of ATM Switches

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

The parallel simulation of discrete event systems, by the means of recurrence relations solved by the parallel prefix algorithm, allow almost linear speed-up with respect to the number of processors used. This paper presents a method for simulating an Asynchronous Transfer Mode switch (ATM), which includes parallel generation and merging of bursty arrival streams, marking and deleting of lost cells due to buffer overflows, and and computation of departure instants. In the case where the number of cells lost is relatively small, the time requirement for this algorithm is approximately $O(M/P + MP/B + M\log^2(B)/B)$, where M is the total number of cells simulated, B is the buffer size and P is the number of processors.

1 Introduction

Consider an ATM switch with transmission capacity C cells/second, a buffer of size B cells and an input stream formed by merging N 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.

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 M cells are to be simulated on P processors in parallel, and the buffer overflow events are rare, then the run time of the simulation is roughly $O(M/P)$.

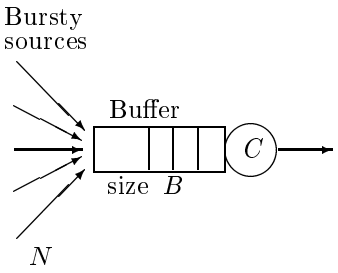

Figure 1: ATM Switch with multiple sources

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 propagated by means of iterations.

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.

The simulation algorithm handles cells in batches of size B - the number that can fit in the buffer. That number is assumed to be quite large. Each batch is processed in parallel by all processors, relying on information obtained from the previous batch. The actions taken are:

1. Generate B cell arrival instants for each source. The "on"and "off" periods are generated first; then the consecutive arrival instants, A_n, are obtained by solving the recurrences

$$A_{n+1} = A_n + a_{n+1} \ ,$$

where a_{n+1} is the interval between the n'th and the $n+1$'st arrivals; whenever an "on" period boundary is crossed, an "off" period is added to the subsequent arrival instants. 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 simulation block. The merging algorithm is based upon the one presented by [4].

3. Mark and remove lost cells. This procedure uses the already computed departure times, D_i, for the previous batch of B cells. If the arrival time of cell n satisfies $A_n > D_{n-B}$, then that cell finds room in the buffer and is accepted. Otherwise, it may or may not be lost, depending on the number of cells preceding it in the current batch that are lost. This step may have to be iterated several times. In the worst case, the number of iterations is equal to the number of processors; however, if cell losses are rare, one or two iterations suffice. Having removed lost cells, the batch is filled with further arrivals until its size is B.

4. Compute departure instants for the current batch. The sequence $\{D_n\}$ satisfies the recurrence relations

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

where $c = 1/C$ is the time to transmit one cell. The solution of these recurrences is reduced to a Parallel Prefix operation by the introduction of a special matrix product in the $(\max, +)$ algebra [1].

To obtain a point estimate and a confidence interval for the cell loss probability, it is enough to store the total number of arrivals, A, needed to fill each batch ($A - B$ is the number of lost cells). It should be pointed out, however, that the above algorithm effectively generates 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. For example, the response time of cell n is equal to $D_n - A_n$.

Finally, it should be pointed out 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.

Below is given a detailed description of the stages required to perform this simulation.

2 Generate B cell arrival instants for each source

Presented below is an algorithm for generating the arrival times for cells from N independent bursty streams of "on"/"off" type. The time requirement for these arrivals is of order $O(M/P + MP/B + V)$ where V is an initial overhead associated with the calculation of the "on"/"off" periods. That overhead depends on the number of streams, N, and on the total number of cells that are generated.

For simplicity it has been assumed that the bursty arrival property of each arrival stream, i, can be simulated by a sequence of "on" periods during which cells can arrive, and "off" periods during which no cells can arrive, and that each stream will start with an "on" period. During "on" periods, the intervals between consecutive cell arrivals are independent and identically distributed random variables; the nth such interval is denoted by $a_{i,n}$. In the experiments, all inter-arrival time distributions were assumed to be exponential, but of course that is not a requirement. The consecutive pairs of "on" and "off" periods for source i are also general i.i.d. random variables; the jth such pair for source i is denoted by $(q_{i,j} \ , \ r_{i,j})$ (again, in the experiments those periods were assumed to be exponentially distributed).

2.1 Calculating Arrival times

To generate in parallel the sequence of arrival instants for source i, it is necessary to perform two operations. The first is to ignore the "off" periods and to calculate an 'unadjusted' arrival sequence as if the source was "on" all the time (see figure 2, where the "off" periods, marked by O, have been condensed to 0).

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

$$U_{i,n+1} = U_{i,n} + a_{i,n+1} \ , \qquad (1)$$

$a_{i,n+1}$ being the inter-arrival interval between cells n and $n+1$.

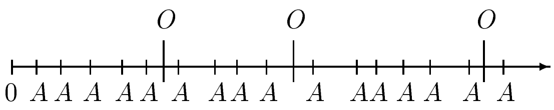

Figure 2: Arrivals during "on" periods

The second stage is to adjust $U_{i,n}$ by inserting the missing "off" periods in the appropriate positions (see figure 3).

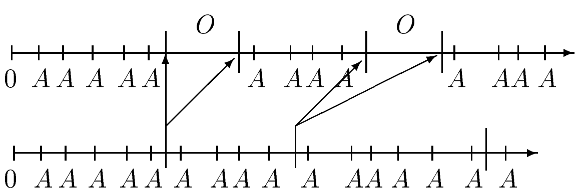

Figure 3: Adjusted arrival instants

To ascertain how many "off" periods occured before a particular 'unadjusted' arrival time, $U_{i,n}$, the index of the "on" period during which $U_{i,n}$ occurs must be determined. Given the lengths of the consecutive "on" periods for source i, $q_{i,j}$, we need to find an index k such that

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

where an empty sum is 0 by definition.

Having solved the inequalities (2) for k, $U_{i,n}$ can be adjusted in order to determine the actual arrival instant of cell n from source i, $A_{i,n}$. That adjustment consists of adding $k-1$ "off" periods to $U_{i,n}$:

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

(An empty sum is 0 by definition.)

The calculation of the 'unadjusted' arrival instants, $U_{i,n}$ is performed according to the parallel prefix algorithm presented in [1]. A total of M arrival instants can be calculated on P processors in time $O(M/P + P)$. However, since we are working with batches of B cells at a time, this becomes $O(M/P + MP/B)$.

The adjustment procedure described above assumes that the realisations of $q_{i,j}$ and $r_{i,j}$ have been precomputed. 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 $q_{i,j}$ and $r_{i,j}$ can also be obtained by means of the parallel prefix algorithm. Denoting the start instant of the j th "on" period (with the "off" periods eliminated) by $Q_{i,j}$, and of the j th "off" period (with the "on" periods eliminated) by $R_{i,j}$, we have the recurrences

$$Q_{i,j+1} = Q_{i,j} + q_{i,j+1} \ , \tag{4}$$

$$R_{i,j+1} = R_{i,j} + r_{i,j+1} \ . \tag{5}$$

The implementation of (4,5) can be done on P processors in time

$$O(s/P + P) = V \tag{6}$$

where s is the total number of "on"/"off" periods required. That number is typically much smaller than the number of cells, M.

The solution of the inequalities (2) is essentially equivalent to merging the two sequences $U_{i,n}$ and $Q_{i,j}$. 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 $O(M/P + MP/B)$. It should be pointed out that B is typically much larger than P, so that this is roughly on the order of $O(M/P)$.

2.2 Estimating the number of "on" and "off" periods

To ensure that (i) enough "on" and "off" periods are computed for the simulation, and (ii) not too much time is spent unnecessarily calculating periods that are not going to be used, it is desirable to produce a reasonably accurate estimate of the number of periods that will be required for each source. That estimate should take into account the different parameters of the N sources.

Denote by ξ_i and η_i the average lengths of the "on" and "off" periods for source i: $\xi_i = E(q_{i,j})$; $\eta_i = E(r_{i,j})$. Then $\xi_i + \eta_i$ is the average length of a source i "on"–"off" cycle. Let α_i be the average inter-arrival interval for cells from source i during "on" periods, $\alpha_i = E(a_{i,n})$. The average number of cells arriving from that source during one "on"–"off" cycle, β_i, is equal to $\beta_i = \xi_i/\alpha_i$.

Suppose that the simulation runs for time T. The average number of "on"–"off" cycles that will occur in source i during that interval, s_i, is given by

$$s_i = \frac{T}{\xi_i + \eta_i} . \tag{7}$$

The total average number of cells arriving from source i during time T, m_i, is equal to

$$m_i = s_i \beta_i = \frac{T \xi_i}{\alpha_i (\xi_i + \eta_i)} . \tag{8}$$

Normally we simulate not for a fixed amount of time, T, but for a fixed total number of cells from all sources, M. In view of (8), those two quantities are related (approximately) as follows:

$$M = T \sum_{i=1}^{N} \frac{\xi_i}{\alpha_i (\xi_i + \eta_i)} . \tag{9}$$

From (7) and (9) it follows that the average number of "on"–"off" cycles occurring in source i during a simulation where a total of M cells are generated from all sources, is equal to

$$s_i = \frac{M}{\xi_i + \eta_i} \left[\sum_{i=1}^{N} \frac{\xi_i}{\alpha_i (\xi_i + \eta_i)} \right]^{-1} . \tag{10}$$

As a safety margin, in all simulations performed, the computed value for the number of "on"/"off" periods were doubled. However, the experiments showed that the computed values were correct to within 5%.

2.3 Optimization of arrival generation and timing

As a general principle, the less communication there is between processors, the faster the simulation will run. Thus, each processor should preferably be allocated the largest possible piece of work that can be done independently of the others. When the number of sources is greater than the number of available processors, it is better to allocate one or more whole arrival streams to each processor, rather than use all processors to generate the arrivals from one stream at a time. The latter should be done with the sources (if any) that are left over from the original allocation. This approach leads to lower simulation run times. It is also more efficient when the the number of cells arriving from one or more sources is relatively small.

The time requirements for the parallel prefix algorithm are $O(B/P + P)$ where B is the number of cells to be generated and P is the number of processors used. Over the simulation this will be computed for each block, giving a total simulation time of $O(M/P + MP/B)$. The time to generate the "on"/"off" periods is computed above in (6) giving an overall time of

$$O(M/P + MP/B + s/P + P) \tag{11}$$

where s is the number of "on"/"off" periods to compute.

3 Merging of sources

All arrival streams now contain the times for their next B arrival instances, described below is an algorithm based on [4] which will perform a parallel merge of the first B arrival instances from these arrival streams. The algorithm presented by [4] provides a method for load balancing each processor to deal with B/P cells by first computing where each arrival stream should be split across the processors in order to give an even amount of work to each. Thus the number of cells to merge can be set to B and the last processor will compute the positions in each stream that correspond to this total.

Two simple serial algorithms are required to perform a parallel merge, these are the binary search and serial merge algorithms. The standard forms of these algorithms are presented below for clarity. The binary search algorithm will, given an ordered list S and a value x to search with, return the location of the value x in the list or the location of the last value in the list, that is smaller than x.

Binary Search(S, x, k)

(S **is an ordered list of size** n, x **is the value to search for and k is the returned position)**

$bottom = 1$

$top = n$

$k = 0$

WHILE $bottom < top$ **DO**

middle $= \lfloor (top + bottom)/2 \rfloor$

IF x $= S_m$ **THEN**

$k = middle$

$bottom = top + 1$

ELSE IF $x < S_m$ **THEN**

$top = middle$

ELSE

$bottom = middle + 1$

ENDIF

ENDWHILE

END Binary Search

The serial merge algorithm merges the contents of K ordered lists $(A_1, ..., A_K)$ and forms a new ordered list C.

Serial Merge$(A_1, ..., A_K, C)$

(A_i is an ordered list of r_i elements, C is the returned ordered list of $\sum_{i=1}^{K} r_i$ elements, K is the number of ordered lists)

FOR EACH ordered list A_i

(Set the position for the next item to merge the current list from)

$pos_i = 0$

ENDFOR EACH

FOR $l = 0$ **TO** $\sum_{i=0}^{K} r_i$ **DO**

p = index of list where A_{p,pos_p} **= min**$(A_{1,pos_1}, ..., A_{K,pos_K})$

$C_l = A_{p,pos_p}$

$pos_p = pos_p + 1$

ENDFOR

END Serial Merge

The positioning of the split points is achieved by the use of a function called FindPos which, given a value n, the index of the last cell that will be merged by a given process, will return the positions into each of the arrival streams that will provide a total of n cells. If each process calls the function FindPos it is then possible to sub-divide the block of B cells up into B/P cells for each process. The number of cells for each process may not be exactly B/P since although each stream cannot have repeated values (ie each stream is a constantly increasing stream of values) it is possible for different streams to have the same values. In this case it is not possible to have the number of cells for each process to differ from the expected by more than P cells, where P is the number of processes.

FindPos$(S, pos, number)$

(S is an array of arrival streams, pos is an array of end positions, and $number$ is the number of cells to find)

$UB_i = B$

$LB_i = 0 \qquad i = (0, 1, ..., N)$

(UB and LB are the upper bounds, and lower bounds for each arrival stream)

WHILE (TRUE)

FOR each Stream i

$$middle_i = 0 \ if \ UB_i < LB_i$$
$$= S_{i,(UB_i+LB_i)/2} \ otherwise$$

ENDFOR

$guess =$**median**$(middle)$

now we can use binary searches on each of the arrays to calculate number less than $guess$

FOR each Stream i

$pos_i = BinarySearch(S_i, guess)$

ENDFOR

$sumPositions = \sum_{i=0}^{N} pos_i$

IF $sumPositions = number$

we have found the number we want so exit

ELSE IF $sumPositions > number$

too many found so this is now a new upper bound set all upper bounds to be equal to the positions found

FOR each Stream i

$UB_i = pos_i$

ENDFOR

ELSE IF $sumPositions < number$

too few found so this is now a new lower bound set all lower bounds to be equal to the positions found

FOR each Stream i

$LB_i = pos_i$

ENDFOR

ENDIF

check for ends caused by identical values

IF $\sum_{i=0}^{N} UB_i \geq number$

Return lower bounds as positions

FOR each Stream i

$pos_i = UB_i$

ENDFOR

ELSE IF $\sum_{i=0}^{N} LB_i \geq number$

Return Lower Bounds as positions

FOR each Stream i

$pos_i = LB_i$

ENDFOR

ENDIF

ENDWHILE

END FindPos

The last two **IF** comparisons in the FindPos function above deal with the cases where multiple cells with the same arrival times occur from different streams, in each case the sum of the number of cells returned from the streams is different from that passed in *number* but will be as close as possible.

The parallel merge can now be broken down into distinct stages:

- **Stage : 1** Each processor i $(i < P)$ uses the FindPos function to find the positions in the arrival streams that correspond to the first Bi/P cells and stores these values in a two dimensional array $pos_{i+1,j}$ where j is the index of an arrival stream. Process 0 also stores the values $pos_{0,j} = 0 \ \ (0 \leq j < N)$.
- **Stage : 2** Each processor i can then perform a serial merge of all the cells from the N arrival streams with indexes from

$$(pos_{i,j}, pos_{i+1,j}) \qquad 0 \leq j < N \qquad (12)$$

At the end of this stage the first B cell arrivals from all N arrival streams have been merged, note that the last process will continue to merge until B cells have been merged thus making up for any deficit from runs of arrivals at the same time. The last processor will then have the positions into each arrival stream which indicate where the last cell was removed, these pointers can be copied to indicate where the arrays start from for the next B arrival to be merged.

3.1 Time requirements

The merging operations of this merge method will operate on B/P arrivals and will thus require $O(NB/P)$ time to compute using the sequential merge method given above, as each merge requires the comparison of N times. The time requirement of the binary search algorithm given above will be $O(\log B)$, where B is the size of the ordered list that is required to be searched, in this case the length of an arrival stream.

The findPos algorithm will perform repeated calls to the binary search algorithm, after each call the search space will be approximately halved, thus the number of times the findPos algorithm will loop will be approximately $\log B$. At each loop in the findPos algorithm there are N calls to the binary search algorithm, requiring $O(N \log B)$ time. There is also a computation of the median value of the N middle values to compute the guess value; this may be performed by first ordering the N values, which requires time of the order $O(N \log N)$ and then picking the middle value which can be done in unit time. The computation of the sums of the N positions will also require $O(N)$ time to compute. This will give an overall time requirement for the findPos algorithm of

$$O(\log B(N \log B + N \log N + N)), \qquad (13)$$

Which is a small constant compared to M. This gives an approximate time requirement for the merging algorithm of $O(NB/P)$. Over the whole simulation this will require time $O(MN/P)$.

4 Mark and remove lost cells

Let the arrival instances of the next B cells, generated by the merge algorithm, be A_i $(i = 1, 2, ..., B)$. It is now necessary to determine which of these cells will be accepted and which will be lost due to the buffer overflow. The last B departure times for the cells that were accepted, and have already departed from the server, will be referred to as D_j $(j = 1 - B, 1 - B + 1, ..., 0; D_j \leq D_k \ if \ j \leq k)$.

For some cells it is possible to decide immediately that they will be accepted. In other cases it is possible to decide that the cell will be rejected. There are also cases when such decisions cannot be made without iterations. More precisely, we consider the following three cases:

- Cell i is definitely accepted if the cell $i - B$ before it in the previous batch has already departed. This can be generalised further. A cell i can definitely be accepted if

$$A_i \geq D_{i-j} \qquad where \ 0 \leq j \leq B \qquad (14)$$

  It is normally sufficient to say that cell i is accepted if $A_i \geq D_{i-B}$, as this requires less computation. Eg in figure 4 job 5 is accepted as $A_5 \geq D_0$.

From this it can be seen that it is not possible to handle more than B cells at a time by this method otherwise a cell's acceptance would be dependent upon a cell that hasn't definitely been accepted yet. Also if cells are lost before the cell i this does not alter cell i's acceptance as the equation (14) is still valid.

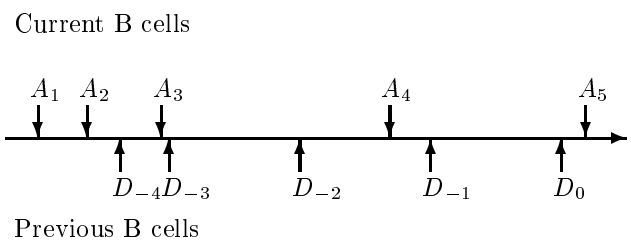

Figure 4: eg of determining cell loss when B is 5

- Cell i is definitely lost if cell $i-B$, and all preceding cells from the previous batch are still in the system. That is, cell i is certainly lost if $A_i \leq D_j$, for all $j \leq i-B$. Eg in figure 4 jobs 1 and 2 are definitely lost.
- cell i may or may not be lost if cell $i-B$ is still in the system at time A_i, but some of the preceding cells from the previous batch have departed. In particular, if the last departure before A_i is that of cell $i-B-j$ and j cells preceding i from the current batch are lost, then cell i will be accepted. Eg in figure 4 cell 3 may be accepted if 2 cells are lost before it from the current buffer. Cell 4 may be accepted if 1 cell is lost before it from the current buffer.

It is thus necessary to store a copy of the previous B departure times from the simulation in order to determine the acceptance of the next B arrivals. It is then possible for each process apart from process 0, to ascertain for each arrival whether or not it will be accepted firmly, possibly or not at all and store this state. If a cell i is possibly accepted, then the number of cells that must be lost for cell i to be accepted is also stored as L_i. Process 0 can compute exactly for each cell whether it is accepted or not, as all cells before the current are already determined.

It is now necessary to iterate over the following steps to determine for the cells that are still only possibly accepted, whether they are accepted or lost.

Each process p stores two values, the number of cells definitely lost in its own block of cells $lost_p$ and the number of cells either definitely lost or possibly accepted $possible_p$. Thus giving a maximum and minimum for the number of cells that can be lost in each processor block. The summation of these values for all blocks before a given processor block will thus give the maximum and minimum number of cells that can be lost before the current processor block. This then allows those cells that are still marked as possible accept and have a loss requirement either less than or greater than the range of minimum to maximum to be marked as either definite accepted or definite loss respectively.

WHILE there are still cells that are possibly accepted

FOR each process p calculate

$min = \sum_{k=0}^{p-1} lost_k$

$max = \sum_{k=0}^{p-1} possible_k$

ENDFOR

This gives both the maximum and minimum number of cells that are lost before the current block.

FOR each cell i in the block of a processor

IF the cell is possibly accepted and the number of cells that must be lost for it to be accepted, L_i, is less than min THEN

mark the cell as accepted

$possible_p = possible_p - 1$

If the cell is possibly Accepted and the number of cells that must be lost for it to be accepted L_i is greater than max THEN

mark the cell as lost

$lost_p = lost_p + 1$

ENDFOR

ENDWHILE

The while loop above in the worst case will finalise the cells in one process on each iteration, thus will terminate in at most P iterations. However as it is assumed that the number of cells lost is small, thus the number of iterations will be small.

4.1 Optimisation and timings

The algorithm above will iterate through all the B/P cells in each block and can take up to P iterations to solve giving an order of operations of $O(B)$ is required plus $O(P^2)$ to perform the calculations of the min/max sums at the start. The computation of the B/P values for L_i can be performed in unit time if the cell is accepted at the start but will require time of an order equivalent to the number of cells lost on average to compute. This gives an overall approximate time requirement of $O(B + B/P + P^2)$. In an attempt to optimise the time this stage takes only the sub-list of cells at each process that are still 'unsure' are checked.

If the simulation is performed where the cell-loss is low then the time requirement for the above stage is low, and will be approximately $O(B/P)$, as the loss iteration loop will normally not be used. Over the entire simulation

$$O(M/P) \tag{15}$$

will be required to handle the M/B blocks. See results in section 6 below.

5 Calculation of Departure times

Once the list of accepted cells has been finalised it is then possible to convert these times into the departure times for those cells. The departure times are calculated by use of the parallel prefix algorithm given by [1] and the equation (16) given below which is converted into the special matrix product in the (max, +) algebra given in [1]. Again as each process will hold approximately B/P cells each process will deal with it's own sub-section of the buffer.

$$D_{i+1} = max(A_{i+1}, D_i) + 1/C \tag{16}$$

The time requirements for this stage will be of the order $O(B/P+P)$ as given by [1], with some reduction due to lost cells. The overall time requirements for dealing with the whole simulation of M/B blocks will thus be

$$O(M/P + MP/B). \tag{17}$$

Once the cells are finalised, the buffer is topped up to B cells using the serial merge, acceptance and departure algorithms. This is assumed to be efficient enough for this stage as the expected number of lost cells will be small.

6 Achieved Results

The algorithms described in this paper have been implemented on a Encore Multimax 520 MIMD computer system with 14 processors. A range of processor numbers were used for the simulations and two different arrival stream numbers, six and twenty-four. In all cases the expected number of lost cells was kept small as to the requirements of the program. In all cases the times given are for the run-times of the programs in seconds when computing the first 1000000 cell arrivals at the ATM switch.

The table 1 below gives the results for the case of six input streams. As can be seen from the table the results show, as expected, almost linear speed- up from increasing the number of processes. As the number of losses increase the linear increase is less clear.

The table 2 below gives the results for the case of 24 input streams. Again the results show the expected linear increase in speed with number of processes.

Table 1: Simulation times for a 6 input ATM switch
Buffer size B, no of processors P
s = sequential version L = average cell loss

B	500	1000	5000	10000	50000
P					
s	151.01	150.17	151.73	151.16	152.63
1	193.36	189.86	194.53	202.58	228.76
2	131.55	125.72	122.52	124.14	138.51
4	84.13	75.14	66.53	65.24	73.25
6	66.58	57.86	49.58	46.86	51.65
8	56.99	46.46	37.55	35.51	39.21
10	52.82	41.03	31.52	29.60	33.20
12	51.37	40.91	29.05	27.49	30.03
14	67.10	54.23	32.08	30.04	31.76
L	10000	7600	2100	1100	100

Table 2: Simulation times for a 24 input ATM switch
Buffer size B, no of processors P
s = sequential version L = average cell loss

B	500	1000	5000	10000	50000
P					
s	211.13	215.62	211.15	212.33	215.48
1	383.81	361.90	354.35	365.11	456.40
2	242.55	217.93	197.35	200.25	250.08
4	161.04	134.26	108.53	107.30	131.43
6	133.79	106.60	79.17	76.56	92.87
8	119.44	92.05	63.95	60.75	73.71
10	117.06	86.80	56.22	52.53	63.82
12	106.86	80.87	50.52	46.72	57.57
14	158.12	120.96	72.29	51.37	61.16
L	7000	4775	760	690	130

The results displayed in the tables above are the average times for 20 runs of the simulation. Also a serial version of the algorithms presented here was run, to give a speed comparison for the parallel versions. For all queue sizes the expected cell loss was also computed and is displayed at the bottom of each table. The confidence intervals for these results were also computed but not displayed here due to space requirements. However to give some indication of the size of these confidence intervals, the graph in figure (6) gives the relation between number of processors and simulation run time along with the corresponding confidence intervals for the given times. That graph is for the six input stream case with buffer size of 5000.

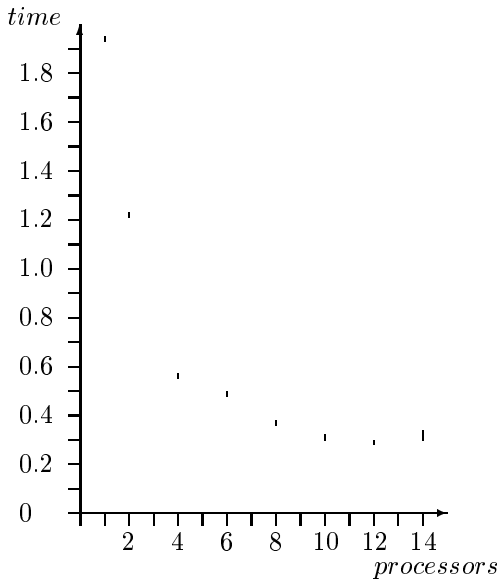

Figure 5: Graph of Run time against processors for 6 input streams and block size of 5000

7 Conclusion

Our experiments show that it is indeed possible to obtain approximately linear speedup when applying the algorithms described in this paper to the parallel simulation of ATM switches. This gain can be maintained even with loss rates as high as 1%, which is much more than would be tolerated in a real ATM switch.

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, **Vol. 4 No.5**, 572-581, May 1993