Parallel Simulations Using Recurrence Relations and Relaxation

Andrew Stephen McGough

Ph.D. Thesis

Department of Computing Science
University of Newcastle Upon Tyne

September 2000

Abstract

This thesis develops and evaluates a number of efficient algorithms for performing parallel simulations. These algorithms achieve approximate linear speed-up, in the sense that their run times are in the order of $O(n/p)$, where n is the size of the problem and p is the number of processors employed.

The systems that are being simulated are related to ATM switches and sliding window communication protocols. The algorithms presented first are concerned with the parallel generation and merging of bursty arrival sources, marking and deleting of lost cells due to buffer overflows, and computation of departure instants. They work well on shared memory multiprocessors. However, different techniques need to be employed in order to achieve similar speed-ups on a distributed cluster of workstations. The main obstacle is the inter-process communication overhead. To overcome it, new algorithms are developed that reduce considerably the amount of information transferred between processors. They are applied both to the ATM switch and to the sliding window protocol with feedbacks.

In all cases, the methodology relies on reducing the simulation task to a set of recurrence relations. The latter are solved using the techniques of parallel prefix computation, parallel merging and relaxation.

The effectiveness of these algorithms is evaluated by comparing their run times with that of an optimized sequential algorithm. A number of experiments are carried out on a 12-processor shared memory system, and also on a distributed cluster of 12 processors connected by a fast Ethernet.

Acknowledgements

I would like to express my sincere gratitude to several people who have contributed in various ways to the completion of this thesis.

First and foremost, I would like to thank my supervisor, Professor Isi Mitrani. His help and guidance throughout this work has been invaluable. The enthusiasm and encouragement he has consistently shown throughout my Ph.D. studies have been essential for the completion of this thesis.

Many thanks to DERA in Malvern (United Kingdom) and the Engineering and Physical Science Research Council (United Kingdom) for providing the finance to support this research.

Thanks are due to all of the staff in the department of Computing Science at the University of Newcastle, who provided an environment in which to work. Particular mention goes to Mrs. S. Craig (for finding those obscure references), Dr. C. Phillips, Dr. N. Speirs, Dr. P. Watson.

I would like to thank colleagues who have helped in various ways during my Ph.D. studies, in particular Dr. G. Morgan, Mr. I. Welch and Mr. B. Arief.

Finally, I would like to thank my mother, for support and understanding over the years.

Contents

Abstract i

Acknowledgments ii

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Background 1

1.2 Aims and Objectives 4

1.3 Summary of Original Work and Publications 5

1.4 Overview of Thesis 6

2 Parallel Simulation 7

2.1 Forms of Parallel Simulation 8

2.2 Space-Parallel Simulation 10

2.2.1 Conservative Space-Parallel Simulation 12

2.2.2 Optimistic Space-Parallel Simulation 15

2.3 Time-Parallel Simulation 18

2.3.1 Regenerative Simulation 20

2.3.2 Relaxation 22

2.3.3 Recurrence Equations and Parallel Prefix 26
2.3.4 Longest Path 28
2.4 Multiple Replication 30
2.5 Space-Time-Parallel Simulation 31
2.6 Conclusion 33

3 Time-Parallel Simulation Techniques **34**
3.1 Parallel Prefix 34
3.1.1 Parallel Prefix Method 1 ($p \geq n/2$) 37
3.1.2 Parallel Prefix Method 2 ($p \geq n-1$) 39
3.1.3 Parallel Prefix Method 3 ($p < n$) 40
3.1.4 Parallel Prefix Method 4 ($p < n$) 41
3.2 Batch Simulation 43
3.3 Simulation of a G/G/1 server using Parallel Prefix 44
3.3.1 Implementation of the Parallel simulation 45
3.4 Parallel Merging 46
3.4.1 Parallel Merge Method 1 47
3.4.2 Techniques for merging when $p < n$ 49
3.4.3 Parallel Merge Method 2 51
3.4.4 Parallel Merge Method 3 52
3.4.5 Parallel Merge Method 4 55
3.5 Parallel Split 57
3.6 The Acyclic Queueing Network 58
3.6.1 The acyclic network model 59
3.6.2 Parallel simulation of the acyclic network 60
3.6.3 Parallel Implementation of the acyclic network 61
3.7 Conclusion 63

4 Simulating the ATM Switch in shared memory 64

4.1 The Model 65

4.2 Related Work 65

4.3 Outline of Solution 66

4.4 Generation of Bursty Arrivals 68

4.5 Estimating the number of "on" and "off" periods 70

4.6 Merging of Arrival Sources 71

4.7 Cell Acceptance Algorithms 74

4.7.1 Fixed buffer size Algorithm 74

4.7.2 Variable buffer size Algorithm 78

4.7.3 In the case where c is not constant 85

4.8 Implementation Results 86

4.9 Conclusion 90

5 Simulating the ATM Switch on a distributed system 93

5.1 Introduction 93

5.2 Efficiency of the shared memory algorithm on a distributed cluster 94

5.3 Outline of the distributed algorithm 98

5.4 Generation of bursty Arrival Sources 99

5.4.1 Generation of "off" and "on" periods 100

5.5 Generation of Arrivals 101

5.6 Mark and remove lost cells 103

5.7 Implementation Results 103

5.8 Conclusion 110

6 Simulating a Sliding Window Protocol 112

6.1 Introduction 112

6.2 Related Work 114

6.3 The Model 114
6.4 Parallel simulation algorithms 116
6.4.1 Algorithm 1 119
6.4.2 Algorithm 2 120
6.5 Experimental Results 124
6.6 Conclusion 126

7 Conclusion **128**
7.1 Summary 128
7.1.1 ATM Switch 129
7.1.2 Sliding Window 130
7.2 Future Work 130

Bibliography **133**

List of Figures

2.1 An example system . . . 7

2.2 An example sample path . . . 8

2.3 Space-Parallel Partitioning . . . 9

2.4 Time-Parallel Partitioning . . . 9

2.5 Space-Time-Parallel Partitioning . . . 9

2.6 Space-Parallel Partitioning of a system . . . 11

2.7 Space-Parallel Partitioning of a sample path . . . 12

2.8 The network in deadlock . . . 14

2.9 State of LP_C at time of straggler arriving . . . 18

2.10 Time-Parallel Partitioning of a sample path . . . 19

2.11 Regenerative Simulation . . . 21

2.12 Generating a sample path using over and underflow regenerative points . 22

2.13 Using relaxation to generate the correct sample path . . . 23

2.14 Simulating both extremes to determine coupling . . . 25

2.15 Shift effects of a different iteration . . . 26

2.16 A single unbounded FIFO G/G/1 server . . . 26

2.17 Path view of Departure times . . . 28

2.18 Arrangement of nodes for a finite server . . . 29

2.19 Replicated simulation . . . 30

2.20 Space-Time-Parallel Partitioning of a sample path . . . 31

2.21 A simple circuit switched network 32
2.22 Allocation of processors to the Circuit Switched Network 32

3.1 The result of the up sweep 38
3.2 The result of the down sweep 38
3.3 Iteration stages of parallel prefix method 2 39
3.4 Parallel prefix When $p = 3, \quad n = 15$ 41
3.5 The Harmony Technique on 3 processors : $n = 7$, $m = 1$ 43
3.6 Partitioning simulation time into batches 44
3.7 A Single G/G/1 Server 44
3.8 Computing the queue size 45
3.9 Speed-up for the G/G/1 Server using Parallel Prefix 46
3.10 The Comparison unit 48
3.11 Generating Odd-Even Mergers 48
3.12 A merging circuit for two lists size 4 49
3.13 The acyclic network 59
3.14 Speed-up for the Acyclic network using Parallel Prefix, merge and split . . 62

4.1 ATM Switch with multiple sources 65
4.2 Unadjusted and adjusted arrival instants 69
4.3 Determining cell acceptance and loss; $B = Q = 5$ 75
4.4 Example of computing cell acceptance with algorithm 1 76
4.5 The values of d_n 79
4.6 Example graph of queue sizes 82
4.7 An example of Collections of cells 84
4.8 The effect of new initial queue size and departure time 84
4.9 Algorithm 1; 6 input sources 87
4.10 Algorithm 2; 6 input sources 88

4.11 Algorithm 1; 24 input sources 89
4.12 Algorithm 2; 24 input sources 90
4.13 The effect of service time (c) on execution time 91
4.14 The effect of service time (c) on iterations 92

5.1 Breakdown of execution time 95
5.2 Time lines for processors running distributed algorithm 96
5.3 An interval of time lines for processors running distributed algorithm . . . 97
5.4 Results from a cluster with 8 input sources 104
5.5 Results from a cluster with 24 input sources 105
5.6 The effect of using clusters on speed-up 106
5.7 The effect of using clusters on speed-up, no overlapping 106
5.8 Amount of time spent computing accepted cells using clusters 107
5.9 Percentage of clusters that are decomposed 107
5.10 An efficient simulation time line 109
5.11 The average size of a cluster 110
5.12 The percentage of time spent in MPICH Calls 110

6.1 The Sliding window protocol 115
6.2 Simulation speed-ups for $W = 4$ 125
6.3 Simulation speed-ups for $W = 8$ 126

List of Tables

2.1 Allocation of PPs to processors 10
2.2 Minimum lookahead between PPs 13
2.3 Messages between LPs 14
2.4 Time parallel cache simulation by relaxation 24

3.1 Allocation of values to merge in method 3 52
3.2 Computing the ranges over four processors 55
3.3 Allocation of values to merge in method 4 55
3.4 Allocation of values to merge in method 5 57

4.1 Arrival instances in each source 73
4.2 Proposed values for e_k at each iteration 73
4.3 Arrival times to be merged by processors 73
4.4 Values of U and u for Clusters 83

Chapter 1

Introduction

1.1 Background

The world is now, more than ever in the past, dependent upon systems. The way we work and the way we live are all dependent on systems. These can be as natural as the weather or as artificial as the Internet. Our desire to understand and affect these systems is great. To be able to ascertain the properties of a system is of great value, especially if they can be provided in a timely manner.

The three main techniques for determining performance measures of a system are observation, mathematical modeling and simulation. The first of these allows measurements to be taken whilst the system is running. Results obtained in this way represent real system performance. However, observation is impossible if the system has not been built. Even existing systems may be difficult to monitor in detail.

A mathematical model can capture the essential properties of the system by means of assumptions and equations. Estimates of performance measures can then be obtained cheaply and quickly. The disadvantage of relying on mathematical models is that they are always approximate. Moreover, the quality of the approximation tends to be inversely related to the ease of the solution.

Simulation is the process of imitating the system behaviour using a computer pro-

gram. Estimates of performance measures are obtained from statistics collected during one or more runs. A simulation model can in principle be made as accurate as desired by including a sufficient level of detail. Also, it can be used to predict the behaviour of a system that cannot be observed.

The use of simulations to obtain performance measures of a system has become more widespread over the years. They are used extensively in fields such as transport, engineering, manufacturing, computing and communications. This has lead to the development of a vast range of simulation languages and packages, e.g. GPSS, MODSIM, Simula, SimScript, SimLab, Sunulink, Matlab, Simul8 and ns.

Very roughly speaking, the accuracy of an estimate is proportional to the square root of the effort put into obtaining it. Thus, to double the accuracy, one needs to do four times as much work.

To overcome the amount of work that needs to be performed in generating accurate performance measures the following approaches to speeding up a simulation are possible.

- **The use of more powerful computing systems**. The computational power of computer systems is currently growing at an exponential rate. According to Moore's Law it is doubling every eighteen months. Thus it is possible in most situations, to acquire a computer that will improve the performance of your simulation. This however will only provide small increases in performance in the short term. It also assumes that Moore's Law holds in the future. This is not considered an effective solution to speeding up a simulation.

- **The use of alternative programming paradigms / algorithms**. It is sometimes possible to speed up a simulation by using more efficient data structures or programming techniques. For example, the complexity of inserting an event into a list of size n is on the order of $O(n)$. This complexity can be reduced to $O(\sqrt{n})$ by using an indexed structure (Franta and Maly [20]), or to $O(\log n)$ by using a heap (Gonnet [33]).

The applicability of such techniques is by no means universal, and the gains may be limited.

- **Exploiting parallelism**. The use of parallel computation has been widely proposed as a way of speeding up simulations. The idea is to share the work amongst a number of processing elements. In the most optimal case, each time the number of processing elements is doubled, the execution time for the problem is halved. This is commonly referred to as *linear speed-up*. The use of this technique does not preclude that of the other two. However, the development of parallel simulation programs is a difficult undertaking, with only certain problem areas showing great increases in performance.

The work presented in this thesis is aimed at the parallel solutions to simulation problems, with an emphasis on developing techniques that produce almost linear speed-up.

Much work has already been done in the area of *space-parallel* simulation. In this case the system to be simulated is divided amongst the available processors, with each processor having a sub-system to simulate. Interactions between the sub-systems are handled by inter-process communications.

For example, if the system consists of a network of service nodes, each node could be allocated to a different processor in the parallel computer. Communication between the nodes in the network would be achieved by inter-process communication.

This works well when the system can be broken down in such a way that each processor's sub-system is largely independent of the rest of the system. However, if the simulation cannot be broken up into enough sub-systems to allocate work to each processor, or the partitioning leads to high levels of communication between processors, then achieving satisfactory speed-up is not normally possible.

An alternative approach to the parallelising of simulation problems is the *time-parallel* method, where the simulation time is divided into intervals. Each processor will then

simulate the entire system for one of these intervals. This allows all processors to be used in the solution of simulation problems regardless of system structure. The problem of parallelisation becomes one of determining the starting conditions for a given interval of the simulation before the finishing conditions of the previous interval have been determined. Techniques for obtaining starting conditions exist for systems that exhibit particular characteristics. However, this is only a small subset of all simulation problems.

A simple example of a time parallel simulation is the so called *regenerative* simulation. It can be applied in situations where it is possible to identify portions of the simulation run that are statistically identical, and independent of each other. Such portions can be assigned to different processors. The difficulty is that, in complex systems, regenerative points are rare.

1.2 Aims and Objectives

The overall objective of all parallel computation is to achieve a better performance as the number of processors increases. This may be by the reduction of the execution time to solve the problem or in the improved accuracy of the results produced in a given time period. Suppose that the optimal version of the sequential simulation requires time T_0 to execute, then the run time of the parallel simulation, T_p, on p processors, satisfies:

$$T_p \geq T_0/p, \tag{1.1}$$

with equality rarely being achieved due to communication costs between processors.

If the run time of the parallel simulation is inversely proportional to the number of processors used, i.e.

$$T_p \approx aT_0/p, \tag{1.2}$$

where $a \geq 1$ is a constant, then the parallel solution is said to achieve linear speed-up.

A parallel simulation technique that exhibits linear speed-up for some simple queuing

models, was developed by Greenberg et al. [35]. The idea is to reduce the simulation task to the solution of a set of recurrence equations. That solution can, under certain conditions, be parallelised by employing an existing algorithm known as *parallel prefix* (see chapter 3 for more details). That approach performs very well, but as it stands, its applicability is limited to a narrow class of systems.

The aim of this thesis is to develop more general algorithms that have wider applicability, while continuing to achieve linear, or near-linear speed-up. In addition to parallel prefix, these algorithms will rely on different versions of parallel merge, and on relaxation.

1.3 Summary of Original Work and Publications

Outlined below are the major contributions made in this thesis.

- Two parallel simulation techniques are developed for a class of systems with bursty sources and losses due to buffer overflows. These algorithms achieve almost linear speed-up but rely on fast communications between processors via a shared memory. Those developments gave rise to two papers. The first was presented at the 1997 conference of the United Kingdom Simulation Society, U.K. Sim. '97, Keswick U.K. [57]. The second was presented at the sixth IFIP workshop on Performance Modelling and Evaluation of ATM Networks, ATM '98, Ilkley U.K. [58]. An extended version of that paper was published in the Journal of Performance Evaluation [61].

- The above algorithms do not perform well in a distributed memory environment because of the high communication costs. To overcome this a different approach to the generation and merging of arrivals is developed. The modified algorithm also achieves a near-linear speed-up when executed on a cluster of workstations. This work was presented at the 14th workshop on Parallel and Distributed Simulation, PADS 2000, Bologna Italy [59].

- The ideas of parallelising a simulation by means of recurrence relations and parallel prefix are applied to a different class of models involving a finite window protocol with feedback. Again, two distinct algorithms are developed and their performance evaluated. This work was presented at the eighth IFIP workshop on Performance Modelling and Evaluation of ATM and IP Networks, ATM & IP 2000, Ilkley U.K. [60].

1.4 Overview of Thesis

The rest of this thesis is organised as follows.

Chapter 2 presents a background to the area of parallel simulation, describing most of the currently known techniques.

In chapter 3 the basic techniques for performing time-parallel simulations are outlined and evaluated.

Chapter 4 presents two methods for simulating an ATM switch on a shared memory multiprocessor. The problems addressed are concerned with the generation of bursty arrival sources, the merging of these sources and determining the acceptance and / or loss of individual cells resulting from the finite capacity of the buffer.

Chapter 5 goes on to describe how this approach can be adapted for use in a distributed memory environment, overcoming the excessive communication cost that is present in the previous version.

Chapter 6 deals with the simulation of a sliding window protocol, in which packets of information are sent to a receiver and acknowledged before further packets may be sent. Two techniques are presented for this. The first computes the arrival times at the source and the service times in parallel, with the dependencies between packets being resolved sequentially. The second uses only parallel prefix computations.

An overall conclusion to the work is presented in chapter 7, where the perceived benefits of these techniques are discussed.

Chapter 2

Parallel Simulation

A survey of current techniques for performing parallel simulations is presented in this chapter. These techniques are compared and illustrated by the use of a running example illustrated in figure 2.1. The five nodes are each unbounded FIFO queueing servers. Packets of data may move between nodes in the directions of the arrows. Nodes A and B receive data from external sources, whilst packets leave the system from nodes D and E. Nodes A and C will select a destination for their packets based upon a random routing algorithm, independent of the current state of the system.

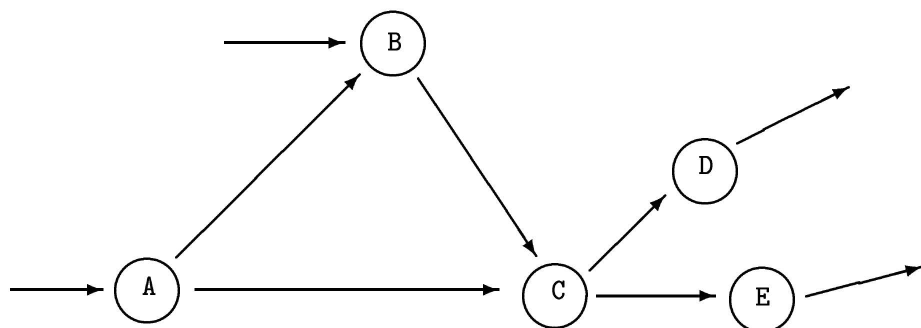

Figure 2.1: An example system

A simulation program generates a sample path for the system by computing the state of the system, $S(t)$, where t represents time. Figure 2.2 below illustrates a typical sample

path, for the example system, which would be generated by a simulation. For each node the queue size, as time progresses, is plotted. Notice that the state, at each node, is often more complicated than a single property.

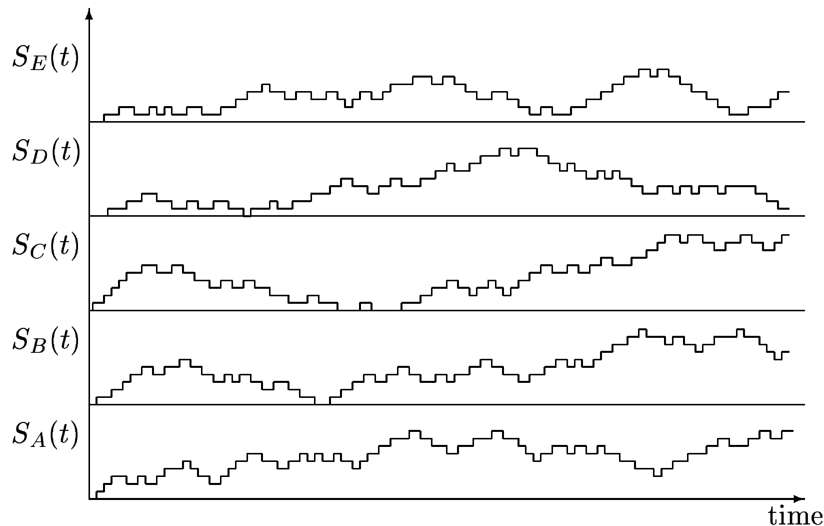

Figure 2.2: An example sample path

2.1 Forms of Parallel Simulation

Parallel simulations may be categorised in terms of how the original simulation problem is partitioned between processors. These smaller problems in general will not be independent of each other. The way in which the problem is partitioned and how these dependencies are dealt with can be used to categorise these simulations.

Parallel simulations may be partitioned in terms of *space*, *time* or potentially both. The simulation problem may be viewed as the computation of state variables over a simulation period (Bagrodia et al. [6]). This can be illustrated by means of figures 2.3, 2.4 and 2.5 in which the vertical axis represent the state variables (often referred to as *simulation space*) and the horizontal axis represent simulation time. A parallel simulation will then use multiple processors to "fill in" the space-time graph.

If the simulation is partitioned in terms of space the system to be simulated will be

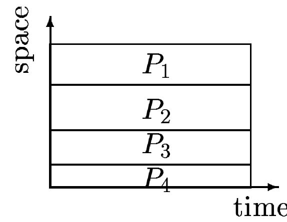

Figure 2.3: Space-Parallel Partitioning

decomposed into separate sub-systems each of which will be simulated concurrently on different processing units (figure 2.3). Each processor will be responsible for simulating its own sub-system for the entire simulation period. Synchronisation is required between processing units to deal with the dependencies.

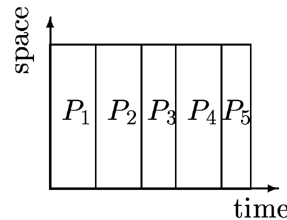

Figure 2.4: Time-Parallel Partitioning

The simulation may also be decomposed in terms of time (figure 2.4). In this situation the time period for the simulation is divided up into intervals. Each processing unit is allocated one (or more) of these intervals and will compute all of the state variables for the whole system during these intervals.

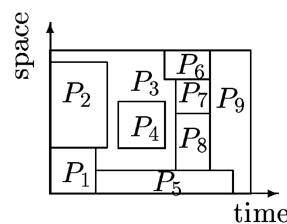

Figure 2.5: Space-Time-Parallel Partitioning

In Space-Time-Parallel simulation the space-time graph is partitioned into a collection

of arbitrary, non-overlapping, regions that cover the graph (figure 2.5). A processor is allocated to each of these regions where it can compute the local state variables.

An alternative approach, is multiple replication, in which a standard sequential simulation program is executed on each of the processors independently. The final results from these runs can be combined to give averages, and confidence intervals, for the performance measures. This approach is considered here as a special case of time-parallel simulation.

The following are more detailed descriptions of the above classifications.

2.2 Space-Parallel Simulation

In Space-Parallel Simulation (sometimes called Distributed Simulation) the idea is to allocate part of the physical system to each processor. Processors then simulate their subsystem for the entire simulation period. Physical processes (called PPs) are simulated by logical processes (LPs), interactions between PPs are modelled by time stamped event messages. Note that multiple LPs may be assigned to a processor in the parallel computer. Each LP contains the local state of the system for the PP it is simulating.

The example system may be partitioned over three processors as illustrated in figure 2.6. The grouping of PPs onto processors will be dependent on how tightly coupled the PPs are. A balance is needed between the amount of communication between LPs and the load balancing across the processors. Table 2.1 indicates which PPs are allocated to which processor. Processor 2 will require information from processor 1 in order to compute the

Table 2.1: Allocation of PPs to processors

Processor	PPs
1	A
2	B
3	C,D,E

correct simulation path, with processor 3 requiring information from processors 1 and 2.

Figure 2.7 illustrates the allocation of subsets of the sample path to processors.

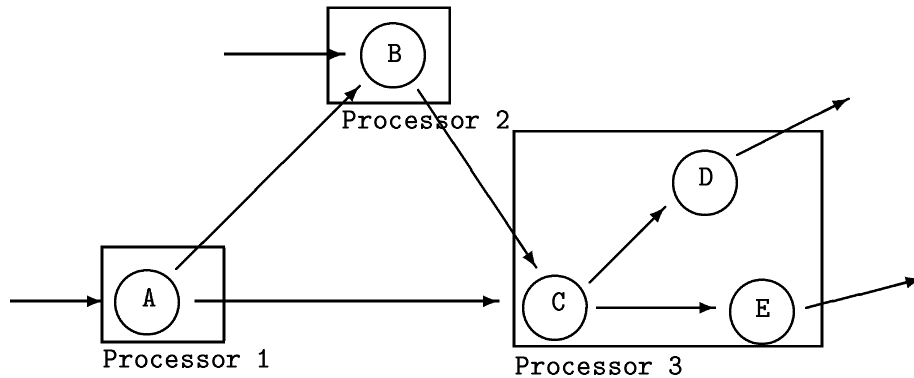

Figure 2.6: Space-Parallel Partitioning of a system

Each LP has a local ordered event list containing events that will occur within its own part of the system. Each event consists of a time stamp, for when the event is scheduled, along with the action to be performed. The processing of an event can change the state of the system being simulated and / or cause other events to be scheduled in the future. New events that are scheduled for parts of the system held on the local processor are entered into the appropriate event queue. When an event is scheduled on a LP which is not on the local processor a message is sent to the processor containing the appropriate LP. The message consists of a time stamp for the event and the action to perform.

By selecting which LPs are allocated to each processor the number of these messages can be reduced, leading to more efficient simulations.

To ensure correctness of the simulation it is essential that each processor deals with messages sent from other processors in time stamp order. This introduces synchronisation into the simulation process, thus reducing the potential for speed-up. The speed-up is also bounded by the number of PPs. However, it is usually much lower than this (Wagner and Lazowska [74]).

The techniques for ensuring that messages are dealt with in timestamp order can be classified into two categories, *Conservative* and *Optimistic* Space-Parallel simulation. These are outlined below.

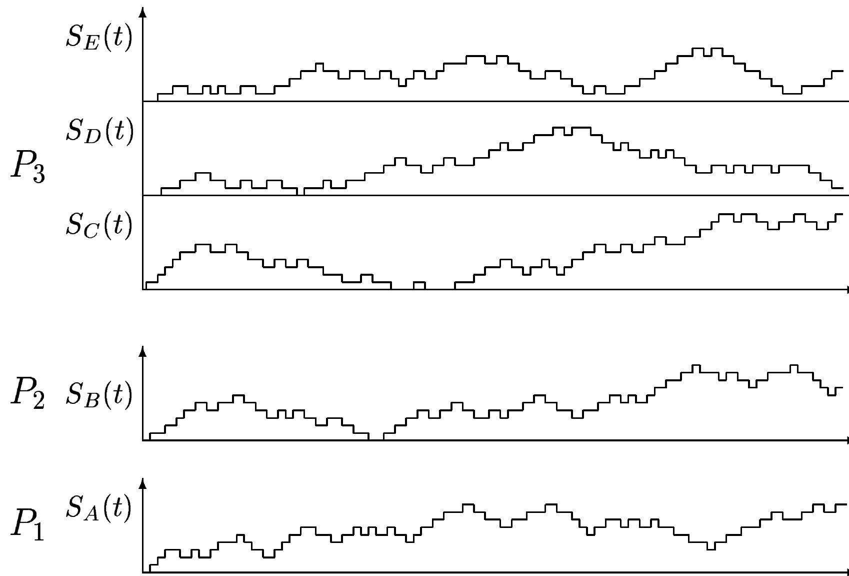

Figure 2.7: Space-Parallel Partitioning of a sample path

2.2.1 Conservative Space-Parallel Simulation

Conservative Space-Parallel Simulation works by assuming the worst-case scenario; messages may only be processed once it is known that no other message may arrive with an earlier time stamp. This technique requires that the number of PPs and the possible links between PPs are known at the start of the simulation. Communication of messages between processors must preserve order. Blocking may occur if an LP cannot guarantee that a message received in the future will not have a lower time stamp than the next message to be processed.

The first conservative algorithm was independently developed by Chandy, Misra [12] and Bryant [10]. A processor waits until it has at least one message from all processors that can send it messages. Then messages may be processed in increasing time stamp order until one source has no messages waiting. This process can easily end up in *deadlock* if all processors are waiting for messages from other processors. Null messages are introduced allowing the simulation to continue. These null messages contain a time stamp for the earliest time that a message could have arriving from a given LP. This

time is computed as the current simulation time on the source LP plus the value L often referred to as *lookahead*. The lookahead is derived from properties of the system being simulated. This may include the minimum time to move between nodes and / or time required for processing events at nodes. This prevents deadlocking. However, if the lookahead values are too small, this can lead to *lookahead creep*, where many null messages are exchanged to avoid a single deadlock.

The solution presented by Candy, Misra [12] and Bryant [10] have much in common. However, for the solution proposed by Bryant each LP may transmit a message at any time, thus requiring (potentially) infinite buffers at each LP. If infinite buffers are allowed, deadlocking is avoided in all situations apart from loops, where each LP in the loop is awaiting messages from a previous LP in the loop.

Table 2.2: Minimum lookahead between PPs

Node Links	lookahead
$A \rightarrow B$	1
$A \rightarrow C$	1
$B \rightarrow C$	1
$C \rightarrow D$	0.5
$C \rightarrow E$	0.75

In the case of the running example, suppose that the minimum lookahead values are as specified in table 2.2. Suppose that messages $\{A_1, A_2, A_3, A_4\}$ arrive at node A at times $\{1,2,3,4\}$ and messages $\{B_1, B_2, B_3, B_4\}$ arrive at node B at times $\{1.1, 2.4, 2.8, 3.9\}$ with the second arrival at node A being routed to node B and all others routed to C. The table 2.3 illustrates the progression of the simulation given that each node can only store one arrival from each of its sources. The table indicates the times at which packets leave each of the links. Values placed in braces indicate null messages, with the line running across the table indicating where the system would first deadlock in the absence of null messages.

This deadlock is illustrated in figure 2.8. Message A_1 has departed from node C as it

Table 2.3: Messages between LPs

Link	Packet							
	A_1	B_1	A_2	B_2	B_3	A_3	B_4	A_4
$A \to B$	*[3]*	–	4	–	–	*[5]*	–	*[6]*
$A \to C$	3	–	*[4]*	–	–	5	–	6
$B \to C$	–	3.1	7.4	4.4	5.4	–	6.4	–
$C \to D$	4.5	5.5	*[10.5]*	6.5	*[7.5]*	*[11.5]*	*[8.5]*	8.5
$C \to E$	*[4.75]*	*[5.75]*	10.75	*[6.75]*	7.75	11.75	8.75	*[8.75]*

was deemed safe to service when message B_1 arrived at C. Node C is now blocked until it receives a message from node A. However node A is blocked trying to send message A_3 to node B. Node B is blocked trying to send message B_2 to node C.

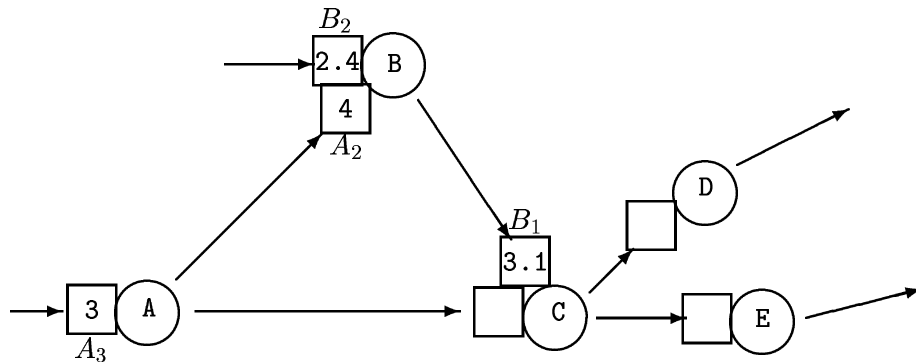

Figure 2.8: The network in deadlock

One solution to breaking the deadlock would be to increase the buffer size at each node. However, this would eventually lead to deadlock at some later stage in the simulation. Alternatively if each time a message is sent out along a link, a null message is sent out on all other output links, then a null message would have been sent on link $A \to C$ when message A_2 was sent on link $A \to B$. This null message would have timestamp 4 thus avoiding the deadlock as message B_1 would be safe to process. In general null messages will be sent out at the same times as all other messages as it is difficult to predict when they will be needed.

Chandy and Misra [13] showed in later work that the use of null messages to prevent

deadlocking can be inefficient. They presented an alternative approach in which the simulation is allowed to run until it deadlocks. A separate process is used to determine when the system has deadlocked and initiate a sequential stage that will determine a simulation time for which all outstanding messages before this time are safe to process. This can be extended further to allow distributed computation of a lower bound on time stamps that will remove both local and global deadlocks (Misra [62]).

Other approaches to preventing deadlock have included barrier synchronisation, where all messages before a barrier time are considered safe to process (Lubachevsky [53]). Conservative time windows may be used, in which each LP computes a window of time where all messages are safe to process (Ayani [4]).

Conservative simulation works well in situations where the amount of independent work each processor can perform is large (and approximately equivalent between processors). In situations where the amount of work that each processor can perform independently is small, a high proportion of the simulation time is wasted as processors determine a safe time to which they can progress.

2.2.2 Optimistic Space-Parallel Simulation

Optimistic simulation does not prevent the system from entering an erroneous state due to messages being processed out of order, but identifies these situations and corrects the simulation accordingly. An LP will process all received messages in time stamp order. If a *straggler* message is received at a point where messages that have greater time stamps have already been processed, the LP will 'undo' the erroneous simulation steps and continue the simulation from the straggler message.

There is no restriction on the communication between LPs provided that messages are reliably communicated. LPs may also be dynamically created and destroyed during the simulation run.

Jefferson [41] presented the Time Warp mechanism for achieving optimistic simu-

lations. In this approach each LP will process messages until a straggler message is received, at which point the state of the LPs simulation is "rolled back" to the last correct state stored before the straggler message. This requires that the simulation state is periodically stored, normally after processing each message. Any new messages that may have been generated during the erroneous computation will also need to be 'undone'. An *antimessage* is sent out to remove each of the erroneous messages that have been sent. If a message has not yet been processed and its antimessage arrives, the two messages cancel themselves out. However, if the antimessage arrives after the processing of the message then the LP must perform a roll back to remove the erroneous message. This may lead to a cascading roll back of LPs.

As the Time Warp simulation progresses the memory requirements will grow due to state saving. A system referred to as *fossil collection* is used to remove states that will never be restored. To determine these states a lower bound on the simulation times at each of the LPs and all unprocessed messages is computed. This is often referred to as Global Virtual Time (GVT). Each LP can safely discard all, except the last, states saved with time stamps less than GVT. Computations of GVT have been proposed by Fujimoto and Hybinette [26] for a shared memory system. Mattern [55] presented a technique of using distributed snapshots to compute GVT.

Memory requirements may still be excessive with the use of fossil collection. Rather than saving the state after each message is processed the system state can be stored periodically to reduce memory overheads. Rolling back will need to restore the last saved state before the straggler message, thus requiring work to be repeated. However, if rollbacks are infrequent enough this will not adversely effect performance. Lin et al. [52] presents a mechanism to determine the correct interval to perform state saving. Quaglia [71] introduced a mechanism to save the states that are most likely to require rolling back. Carothers et al. [11] suggest an alternative approach to saving states by reversing the computation performed in changing the states. Jefferson [40] proposes a

system to artificially introduce roll back, called cancelback, to prevent processors which are simulating events far ahead of the other processors from exhausting available memory. Lin and Preiss [51] present a general and easier to implement version of this approach.

Attempts have been made to limit the optimism of the time warp approach to reduce the chances of cascading roll backs. These techniques have included restricting processors to processing events within a time window [GVT, GVT + W] (Sokol et al. [73]). Messages may be buffered up at their source until they are known to be correct, this will prevent the need to generate antimessages and remove the possibility of cascading roll backs (Reynolds [67]).

The optimistic method can exploit more parallelism present in a system than conservative simulations. This is achieved by allowing computation to proceed when a dependency may exist but is later discovered not to. The simulation is less dependent on system specific knowledge, although this knowledge may be used to optimise the simulation.

However, the requirement to store states reduces the efficiency of the technique and greatly increases the storage requirements. Implementation of an optimistic simulation is considered to be more complex than that of the conservative approach.

In the case of the example system each processor will process messages as they arrive and store copies of the system state after each arrival. Suppose LP_C receives messages in the following order $\{3, 3.1, 4.4, 5.4, 6.4, 5, 7.4, 8\}$. The sixth message is a straggler and will cause roll back of the system to the state saved after the completion of message at time 4.4. Figure 2.9 illustrates the state of the system as the straggler arrives.

LP_C will generate antimessages for the two messages sent to LP_E. This may potentially cause LP_E to roll back. The simulation process will now restart with the next message to be processed being 5.

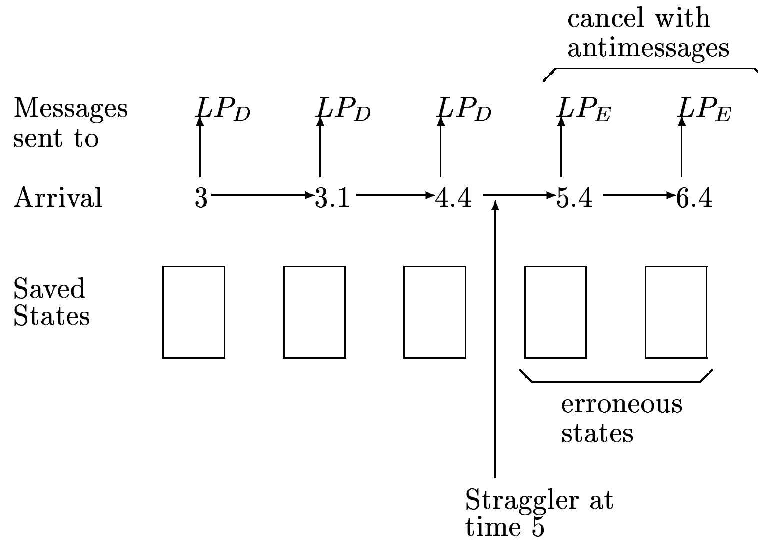

Figure 2.9: State of LP_C at time of straggler arriving

2.3 Time-Parallel Simulation

Time-Parallel simulations divide the simulation time interval $[0, t)$ into sub-intervals $[0, t_1)$, $[t_1, t_2), \ldots, [t_n, t)$. Each processor is allocated a sub-interval, say $[t_p, t_{p+1})$, and is responsible for simulating the entire system during this interval. The simulation can, in general, be divided amongst an arbitrary number of processors. The problem with parallel simulation now becomes one of determining the starting conditions for interval $[t_p, t_{p+1})$ before the completion of interval $[t_{p-1}, t_p)$. If this can be achieved then near optimal speed-up is possible. Figure 2.10 shows the sample path divided between three processors.

Techniques have been developed to determine the system state at the start of an interval or enable the interval to be computed without this information. These may only be applied to systems that exhibit certain properties. Thus the techniques described below are more a methodology for solving specific problems rather than a general approach to time parallel simulation.

Lin and Lazowska [50] presented a categorisation of time-parallel simulations in terms

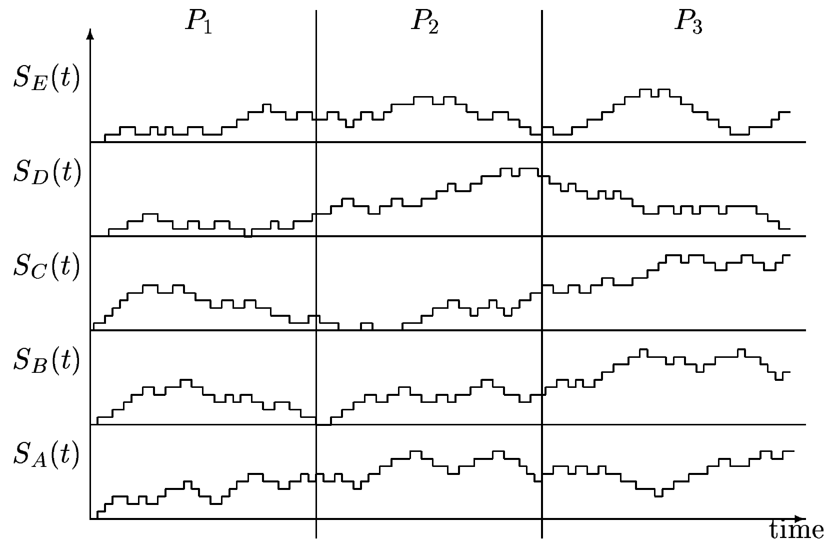

Figure 2.10: Time-Parallel Partitioning of a sample path

of two of the properties of the simulation. The first was the method used to partition time and the second is the method used to deal with the fact that the initial state of a sub-interval is unknown. The categorisation is described below and used to classify the techniques described later in this section.

Time partitioning may be carried out by:

- Dividing the time interval so that each sub-interval contains the same number of arrival events. The first n arrivals are allocated to P_1, the next n to P_2, etc. This division may be used with recurrence relations using parallel prefix and relaxation.
- Dividing time by state matching. Each processor will simulate until a given state is reached. This is the regenerative approach.
- Dividing the time interval into arbitrary sub-intervals. This can be used for the relaxation approach.

The methods for dealing with the lack of initial state of each sub-interval are:

- Part of the interval may be computed independently of the initial state. The rest may be computed once the initial state is known. A fixed number of stages will

be required to compute the true trajectory. This is the recurrence relations with parallel prefix, longest path or regenerative approach.

- The initial state of the interval has a transient effect on the sub-interval. Thus in general if the sub-interval is long enough then the end state will be independent of the starting state. Often a small number of iterations will be required to compute the true trajectory. This is an efficient use of the relaxation approach.

- The initial state may affect the whole of the interval. In this case if the initial state changes the entire interval will need recomputing. The number of iterations required to compute the true trajectory will be high, bounded by the number of processors. This is the relaxation approach.

An approach to time-parallel simulation of the example system is presented in chapter 3.

2.3.1 Regenerative Simulation

Regenerative simulation (often referred to as state matching) is a technique for splitting simulation time into intervals that are independent of each other. The aim is to find points in the sample paths for which all future states are independent of the states before this point. Each processor may then simulate an interval independently of all other processors to generate the final sample path. Figure 2.11 illustrates this approach. First the regenerative points are determined, then each processor simulates one of the intervals in-between.

This technique requires that such states can be found and occur at regular enough intervals.

Fujimoto et al. [28] used this technique in a high-level simulation of a statistical multiplexor (such as an ATM switch). There are N identical arrival sources each of which may be either "on" or "off". During an "on" period cells arrive at a constant rate. The server can send C cells per unit time.

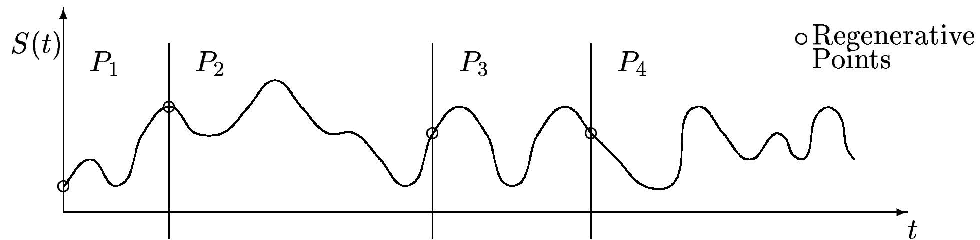

Figure 2.11: Regenerative Simulation

Assume that the “on” and “off” periods for each stream are already available. Each processor is allocated an interval of the time period. This interval is searched for a regenerative point from which to start simulating. There are two forms or regenerative points:

- Guaranteed overflow. If enough sources are active for a long enough period then the queue will become full.
- Guaranteed underflow. If only a small number of sources are active for a long enough period then the queue will become empty.

Each processor can then safely simulate the interval from this point onwards. At the end of a processor's interval it passes its end state onto the next processor. These end states can be used to compute the first part of the interval before the regenerative point. This approach requires that regenerative points can be found within each interval.

Figure 2.12 illustrates this technique when the simulation is run over four processors. It can be observed that the regenerative points may not be the first time that the queue becomes empty or full during an interval, rather the first time this is guaranteed to happen.

Nikolaidis et al. [65] continued this work to perform a parallel simulation of cascading multiplexors. Each level of multiplexors was computed independently using the above approach.

Lin and Lazowska [50] proposed the idea of partial state matching. This can be

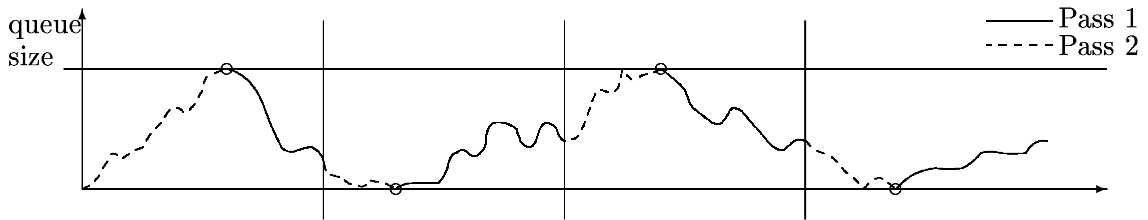

Figure 2.12: Generating a sample path using over and underflow regenerative points

applied if the system state can be separated into a regenerative substate and a non-regenerative substate such that the two substates are independent. The simulation is performed as a regenerative simulation. A fix up may be required afterwards for the non-regenerative substate.

2.3.2 Relaxation

In many situations it may not be possible to predict the starting conditions for a particular interval of a simulation. Chandy and Sherman [14] have suggested using relaxation as a way of dealing with these uncertainties. That approach can be applied to either time-parallel or space-time-parallel simulations.

The simulation period is broken up into arbitrary intervals and a processor assigned to each. The starting state for each interval may be unknown. In that case the processor makes an assumption about the initial conditions and proceeds to simulate the interval. In general the end state from processor p's interval will not match the starting condition for processor $p + 1$, see figure 2.13. The latter will then re-simulate its interval using the end conditions from processor p as a new set of starting conditions. This process is iterated until two consecutive iterations produce identical sample paths. Figure 2.13 illustrates the case where two iterations are required.

If the final state of an interval is independent of the initial state of the interval then this technique will normally perform well. However, if there is a dependency, then the approach will require up to p iterations to converge to the true sample path when using

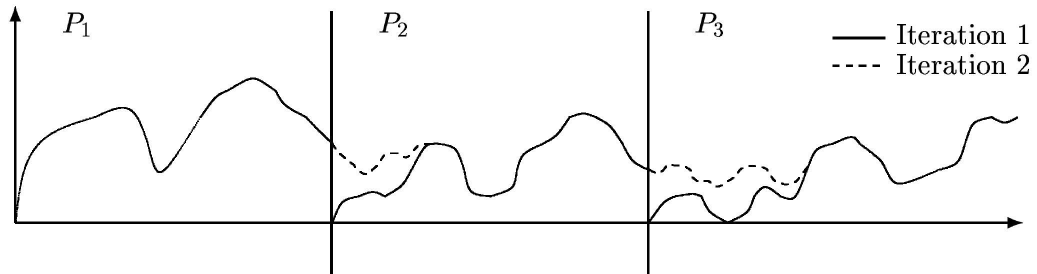

Figure 2.13: Using relaxation to generate the correct sample path

p processors.

Heidelberger and Stone [39] used this approach for a trace-driven simulation of a processor's cache. Consider the simulation of a single processor cache capable of storing four separate address values from a memory range of seven over an interval of twenty-four address requests. A request for an address may result in either a hit, indicating that the value resides in the cache, or a miss. In the case of a miss the address is loaded from main memory and, if the cache is full, one of the previously stored addresses is discarded. Many replacement policies exist, in this case the least recently used (LRU) address is discarded.

Table 2.4 shows an example of the relaxation approach applied to the LRU policy. The twenty-four trace-driven access requests are divided equally among three processors. Notice that for clarity the cache locations are sorted in order of usage.

The first stage is to check whether the requested address (A) exists already in one of the four cache locations $\{c_1, c_2, c_3, c_4\}$. Otherwise the address is retrieved from main memory and placed into the cache. Notice that the table shows the result after the operation has been performed. Each processor assumes initially that it will start with an empty cache and simulates its interval. After the first iteration all but the first processor will need to re-compute its interval. Each uses the final cache state from the previous processor as the starting state and simulates until the new simulation state matches the equivalent state from the previous iteration. It was found that if the intervals were long

enough, two iterations would suffice.

Table 2.4: Time parallel cache simulation by relaxation

	Processor 1								Processor 2								Processor 3							
A	3	1	3	2	7	2	4	6	1	3	1	4	5	2	2	4	7	1	2	3	6	1	6	7
	Iteration 1																							
c_1	3	1	3	2	7	2	4	6	1	3	1	4	5	2	2	4	7	1	2	3	6	1	6	7
c_2	-	3	1	3	2	7	2	4	-	1	3	1	4	5	5	2	-	7	1	2	3	6	1	6
c_3	-	-	-	1	3	3	7	2	-	-	-	3	1	4	4	5	-	-	7	1	2	3	3	1
c_4	-	-	-	-	1	1	3	7	-	-	-	-	3	1	1	1	-	-	-	7	1	2	2	3
	Iteration 2																							
c_1									1	3	1	4	5				7	1	2	3				
c_2				idle					6	1	3	1	4	match			4	7	1	2	match			
c_3									4	6	6	3	1				2	4	7	1				
c_4									2	4	4	6	3				5	2	4	7				

Andradóttir and Ott [3] proposed a technique for determining the position where the system state becomes independent of the starting conditions. Suppose the performance measure of interest is the queue size at a finite capacity server. Each processor conducts two simulations, the first starting the interval with an empty queue and the second with a full queue. If both of these simulations use the same arrival streams it is clear that the true trajectory (and all other possible trajectories) must lie between these two trajectories. Eventually these two simulations will couple (come together) and remain so. From this point on, the simulation will be independent of the starting conditions. The rest of the interval is performed as a single simulation, with the correct final state of the interval being passed to the next processor.

Figure 2.14 shows an example of three processors using this technique to generate the sample path. After the first iteration each processor knows that the starting conditions passed from the previous processor are correct and can produce the final sample path.

If the two sample paths do not couple before the end of a processor's interval, the two end states are passed onto the next processor which will simulate both cases until

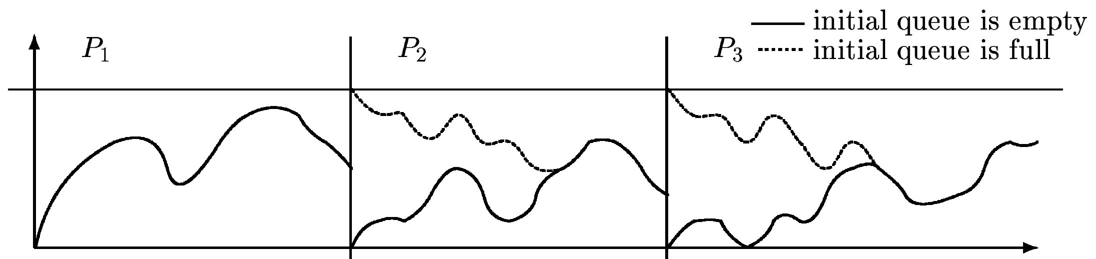

Figure 2.14: Simulating both extremes to determine coupling

either they couple or that interval ends.

Lin [48] proposed a method for placing bounds on the losses from a G/G/1/K queue. If the first pass of the relaxation algorithm starts with a queue size of 0 then this will give a lower bound on the number of cells lost. In order to determine an upper bound on the number of cells lost the sample path from the first pass is divided into clusters where the server is active and idle periods. Figure 2.15 illustrates three clusters $\{C_1, C_2, C_3\}$ separated by idle periods.

During future iterations the idle periods can be used to service extra cells that may be in the queue. During clusters the sample path is moved up by the number of extra cells in the queue at the start of the cluster and the departure instances are 'shifted' left. See figure 2.15 where β_i extra cells are in the queue at the start of cluster i with all departure times shifted δ_i left, where δ_i is the amount of time remaining in service for the active cell. An equation is presented for computing an upper bound on the number of cells lost during each cluster, along with an estimate for the number of cells left in service at the end of the interval.

Subsequent iterations of the relaxation algorithm presented by Lin compute these upper bounds for the cells losses, thus reducing the computation time for the iteration stages.

Lin extends this idea to bounding the number of cells lost in a G/G/1/K server with priority scheduling policy [49].

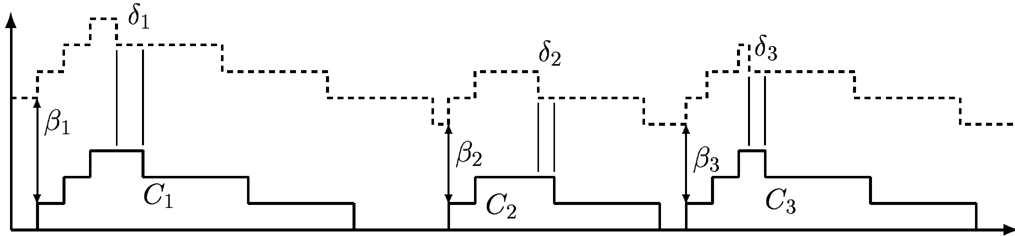

Figure 2.15: Shift effects of a different iteration

2.3.3 Recurrence Equations and Parallel Prefix

Greenberg et al. [34, 35] presented an alternative approach to solving the time-parallel simulation problem. This solution does not rely on the ability to pre-compute "known" points in the simulation or the correction of simulation states in light of new starting conditions. The approach offers almost unlimited speed-up for parallel simulation.

Many simulation problems may be formulated as a set of recurrence equations that describe the new system state based upon the previous state. If these equations can be written in terms of associative operations then the system states may be computed by use of the parallel prefix method. This method can compute all n prefix products over p processors in $O(n/p + \log_2 p)$ time.

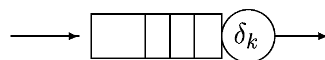

Figure 2.16: A single unbounded FIFO G/G/1 server

Take for example a single G/G/1 server (figure 2.16). Packets of data arrive at the server and are stored in an unbounded FIFO queue awaiting service. The inter-arrival time between packets $i-1$ and i is α_i. The service time for packet i being δ_i. The arrival, A_i, and departure, D_i, times for packet i may be computed from the following recurrence relations:

$$A_i = A_{i-1} + \alpha_i, \tag{2.1}$$

$$D_k = \max(A_i, D_{i-1}) + \delta_i. \tag{2.2}$$

Equation 2.1 may be computed directly as a parallel prefix sum as addition is associative. However, equation 2.2 will require re-writing as a matrix multiplication in the (max,+) semi-ring in order to be performed as an associative operation.

A complete solution to this problem, including a discussion of parallel prefix algorithms is presented in chapter 3. The chapter also includes a solution to a simplified version of the acyclic network that has been used as a running example.

The paper by Greenberg et al. illustrated the solution to other queueing networks solved by recurrence relations and parallel prefix such as acyclic fork-join networks and certain types of tandem networks with bounded buffers and blocking. The use of merging and splitting of arrival and departure lists are also used to solve acyclic networks. Finally the use of relaxation was added to solve networks with feedback. An extended version of this paper was published [36] developing these solutions further.

The use of parallel prefix computations to solve simulations of petri nets was shown by Baccelli and Canales [5]. In this work the evolution of the petri net was described by a matrix in (max,+) algebra. The simulation of the petri net could be performed using parallel prefix computations if the size of the matrix was small in comparison to the number of processors. However, if the size of the matrix exceeded the number of processors, performing the simulation as a set of parallel matrix vector multiplications was more appropriate.

The approaches of relaxation and recurrence relations are brought together by Wang and Abrams [75], who developed approximate simulations of finite capacity G/G/1/K and G/D/1/K servers.

To simulate the G/G/1/K server, first the equivalent G/G/1 server is simulated using the approach by Greenberg et al. Each processor is allocated an interval of arrivals and relaxation is used to determine the cell losses. Each processor assumes that the initial queue size is the smaller of the queue size from the G/G/1 simulation or K. Departure

times of lost cells and subsequent effects on departures are not dealt with in this case leading to approximate solutions.

The G/D/1/K server simulation first computes the arrival instances using parallel prefix computations. Again relaxation is used to compute the queue size, with the queue size at the start of each interval being set initially to 0. To reduce the time required for relaxation to converge the time left in service for the first cell in each interval is artificially set to a pre-defined value. This again leads to approximate results.

An alternative approach to simulating trace-driven cache simulations using parallel prefix computations was proposed by Nicol et al. [63]. This allowed various cache sizes to be simulated concurrently.

2.3.4 Longest Path

Chen [15] presented an approach that can be used to simulate the departure times for a G/G/1 server. It is based on computing the latest time that a departure may occur (considered to be the longest path).

Figure 2.17: Path view of Departure times

Consider the graph in figure 2.17, the arrival times A_i in row 1 are pre-computed along with the service times S_i. The departure times D_i in row 2 are to be computed.

The departure times for packet i may be computed using the recurrence relation:

$$D_i = \max(A_i + S_i, D_{i-1} + S_i). \tag{2.3}$$

This may be re-written to eliminate D_{i-1} by substitution:

$$D_i = \max(A_i + S_i, A_{i-1} + S_{i-1} + S_i, D_{i-2} + S_{i-1} + S_i). \tag{2.4}$$

These substitutions may be continued until all departure times are removed, giving:

$$D_i = \max(A_j + \sum_{l=j}^{i} S_l : j = 1, \ldots, i). \tag{2.5}$$

An algorithm is presented for computing $MP+1$ departures in batches of size $P+1$ on P processors in $O(M(1+\log_2 P))$ time. If the processor count is high then this algorithm will perform as well as the parallel prefix approach. However, if the processor count is small then the performance will suffer due to the synchronisation required between batches.

The paper goes further to illustrate the use of this algorithm for simulating a finite capacity G/G/1/K queue. This requires that an infinite capacity G/G/1 queue precedes the finite queue. Figure 2.18 illustrates this with node 1 a G/G/1 server with infinite capacity and node 2 a finite capacity G/G/1/K server. The batch size, b, thus the number of processors P, must now be less than K.

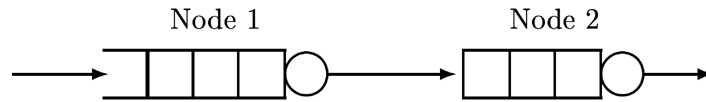

Figure 2.18: Arrangement of nodes for a finite server

If the times of the last b departures D'_j to depart node 2 are known then packet i will depart node 1 and be accepted at node 2 if $D_i \geq D'_{v-b}$, where v is the number of departures previously accepted into node 2. All other departures from node 1 will be lost. Note that these comparisons will require sequential computation.

Simulation algorithms are also presented for simulating networks of nodes and nodes with finite capacity that block the service at prior nodes rather than losing packets.

2.4 Multiple Replication

In multiple replication each processor independently performs a sequential simulation of the system. The results generated can then be used to compute an estimate and a confidence interval for the performance measures of interest. This can be one of the quickest techniques to implement if a sequential simulation program is already available.

For the example system each processor would be given the complete system to model, as in figure 2.19, producing multiple sample paths that can be used to generate estimates and confidence intervals for performance measures.

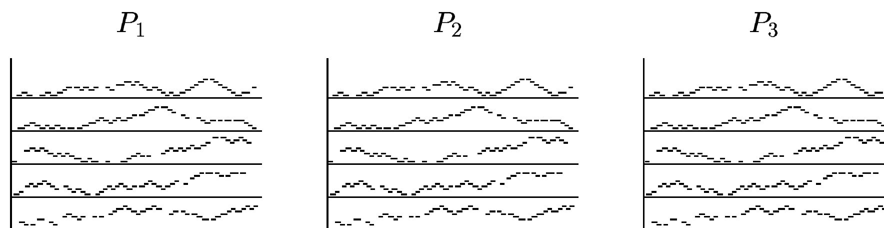

Figure 2.19: Replicated simulation

If the performance measures of interest are for properties of the system that may be determined within a relatively short time period (often refereed to as transient or terminating), replication simulation is well suited. Examples of such systems include the expected time before the queue size exceeds a certain level or the number of packets of data that can be processed in a given time interval. This will lead to near optimal use of the available processors (some degradation may be present due to deletion of the initial start up states of each simulation). However, work by Kelton and Law [42] and Whitt [76] shows that the best results will be achieved if the number of replications is kept low (typically no more than 5 to 10).

The use of this technique can also lead to biased estimates if there is a limit on the amount of time a processor may simulate the system. This may be due to the effects of work in progress at the end of the time period or how the estimates are generated. For further details see Glynn and Heidelberger [32].

2.5 Space-Time-Parallel Simulation

In space-time-parallel simulation the space-time region is decomposed into non-overlapping regions that offer the best option for massive speed-up. Chandy and Sherman [14] proposed the first partitioning method, allowing the space-time graph to be decomposed in some arbitrary manner (figure 2.20). Communications that flow between PPs across horizontal lines are modelled by time stamped messages between processors. Unknown starting conditions along vertical edges are "guessed" with relaxation being used to determine final states.

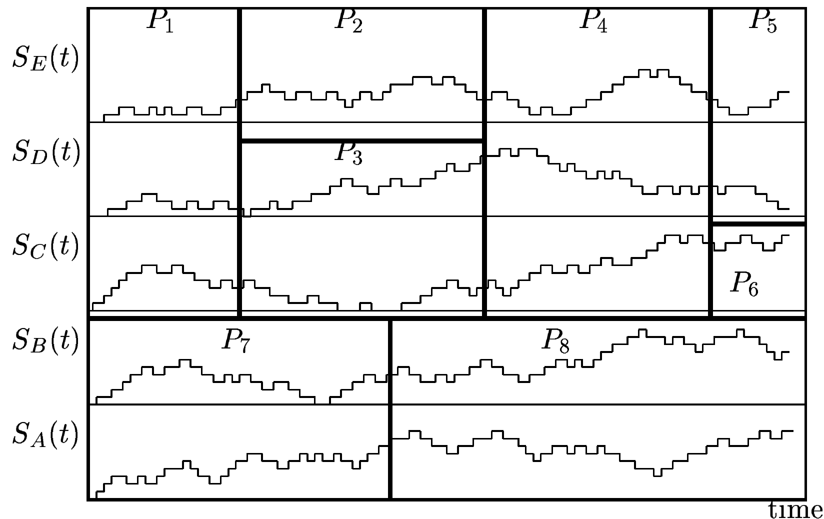

Figure 2.20: Space-Time-Parallel Partitioning of a sample path

Bagrodia et al. [6] generalised the concept of space-time. The state of PP j at time t is $S(j,t)$. A message passed between PPs i and j at time t is $M(i,j,t)$. The process of simulating a system is a case of solving $S(j,t)$ and $M(i,j,t)$ for all PPs in the interval $[0,T)$. Descriptions of how time-warp and conservative algorithms may be expressed in this notation were given. However, no general solution was proposed, though the authors alluded to possible approaches.

Circuit-Switched networks, such as the telecommunications networks offer scope for space-time-parallel simulation due to the massive scale of such problems. The AT&T

network consists of approximately 5000 links capable of handling 1 million calls at a time with simulations processing at least 10^9 calls. A call that is placed between two nodes i and j can either be offered, routed via another node v or blocked.

Two space-time-parallel approaches have been proposed for simulating circuit switched networks, Eich et al. [17] and Gaujal et al. [29, 30]. In both cases it is assumed that the number of processors exceeds the number of links. The simulation progresses in time intervals $[t, t + \delta)$. Each link in the network is allocated a number of processors which divide the time interval amongst themselves. For example figure 2.21 illustrates a simple network of three nodes. Figure 2.22 indicates how eight processors could be allocated over the links during interval $[t, t + \delta)$.

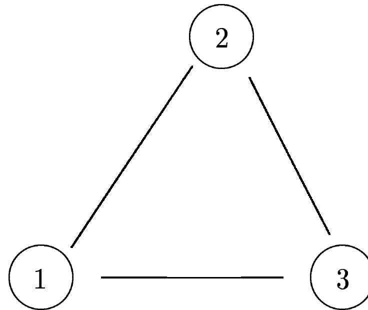

Figure 2.21: A simple circuit switched network

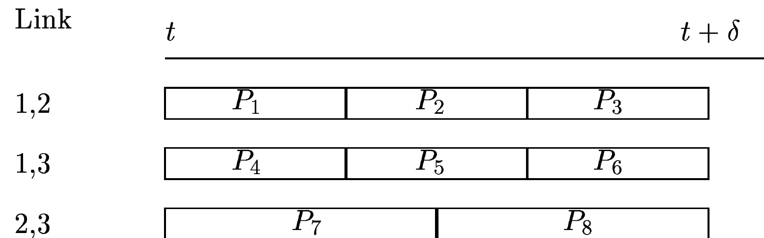

Figure 2.22: Allocation of processors to the Circuit Switched Network

The two algorithms differ in the way that the intervals are simulated. In both cases a list of call arrival times and lengths are pre-generated. Eich et al. use relaxation to compute which calls are accepted and along which links.

Gaujal et al. tentatively offer both the direct and indirect routes for each call. Iterations are used to compute which events are the true events. Each processor cancels out routes if they can't be provided until only the offered calls remain.

2.6 Conclusion

The overall objective of this thesis is to perform simulations with near-linear speed-up. Time-parallel simulations have the potential to achieve near-linear speed-up, although not for all simulation problems. The approaches of parallel prefix computation with recurrence relations are used as these have the potential to offer linear speed-up. In addition to this, relaxation will be used to deal with parts of the simulation that cannot be described as associative operations.

Much of the background material for the space-parallel and some for time and space-time-parallel simulations has come from a collection of survey papers written by Fujimoto [21, 22, 23, 24] and a paper written by Fujimoto and Nicol [27]. This work has now been presented in a book by Fujimoto [25]. Readers interested in space-parallel simulation are referred to these references.

Chapter 3

Time-Parallel Simulation Techniques

In this chapter the techniques used in this thesis for performing time-parallel simulations are described along with examples of their use.

3.1 Parallel Prefix

Suppose that there are n elements:

$$x_1, x_2, ..., x_n,$$

and a binary associative operation $\odot$. The intention is to compute the partial products,

$$\begin{aligned}
&x_1, \\
&x_1 \odot x_2, \\
&x_1 \odot x_2 \odot x_3, \\
&\ldots, \\
&x_1 \odot x_2 \odot \ldots \odot x_n.
\end{aligned}$$

This is called the 'prefix operation' (sometimes referred to as the scan operation), and has applications in many algorithms. In particular the solution of recurrence relations which arise in simulations (see section 2.3.3) can be reduced to a prefix operation. Consider, for example, the computation of arrival instants by means of the recurrence (2.1):

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

Given $\alpha_1, \alpha_2, \ldots, \alpha_n$ and operation $+$ compute:

$$\begin{aligned} A_1 &= \alpha_1, \\ A_2 &= \alpha_1 + \alpha_2, \\ A_3 &= \alpha_1 + \alpha_2 + \alpha_3, \\ &\ldots, \\ A_n &= \alpha_1 + \alpha_2 + \ldots + \alpha_n. \end{aligned}$$

The computation of the departure instants is less straightforward, but can also be reduced to a prefix operation. Rewrite the recurrence (2.2) as:

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

This equation can be further rewritten as a 'product' recurrence, by introducing the following matrices and column vectors:

$$M_i = \begin{bmatrix} \delta_i & A_i + \delta_i \\ -\infty & 0 \end{bmatrix}, \qquad i = 1, 2, 3, \ldots, \tag{3.3}$$

$$V_i = \begin{bmatrix} D_i \\ 0 \end{bmatrix}, \qquad i = 0, 1, 2, \ldots. \tag{3.4}$$

The matrix multiplication is in the $(\max, +)$ algebra, where the max operation represents 'addition' and is denoted by $\oplus$; the $+$ operation represents 'multiplication' and

is denoted by $\otimes$. The 'zero' element of $\oplus$ is $-\infty$ (since $\max(x, -\infty) = x$ for any x), and the 'unit' element of $\otimes$ is 0 (since $x + 0 = x$ for any x). If $A = (a_{i,j})$ and $B = (b_{i,j})$ are two matrices with N rows and M columns and K columns and M rows respectively, then the elements of their product, $C = A \odot B$, in the $(\max, +)$ algebra, are computed according to:

$$c_{i,j} = \bigoplus_{k=1}^{M} (a_{i,k} \otimes b_{k,j}) = \max_{1 \le k \le M} (a_{i,k} + b_{k,j}) \; . \tag{3.5}$$

In this algebra, (3.2) becomes:

$$\begin{bmatrix} D_i \\ 0 \end{bmatrix} = M_i \odot \begin{bmatrix} D_{i-1} \\ 0 \end{bmatrix} . \tag{3.6}$$

The solution of (3.6) is:

$$\begin{bmatrix} D_i \\ 0 \end{bmatrix} = M_n \odot M_{n-1} \odot \ldots \odot M_1 \odot \begin{bmatrix} D_0 \\ 0 \end{bmatrix} . \tag{3.7}$$

In other words, the computation of successive departure instants reduces to that of the prefix:

$$\begin{aligned} & M_1, \\ & M_2 \odot M_1, \\ & M_3 \odot M_2 \odot M_1, \\ & \ldots, \\ & M_n \odot M_{n-1} \odot \ldots \odot M_1. \end{aligned}$$

Because of its importance, much effort has gone into the parallelising of the prefix operation.

Ladner and Fischer [47] presented one of the first optimal techniques for solving the prefix problem. The solution required $O(\lceil \log_2 n \rceil)$ time when the number of available processors is not greater than 4n. The approach generates a circuit recursively, with a

depth $\lceil \log_2 n \rceil$ giving the order of operations. This approach is not considered further due to the large number of processors required.

In the rest of this section four methods for computing parallel prefix are outlined, two require a minimum number of processors.

3.1.1 Parallel Prefix Method 1 ($p \geq n/2$)

The prescan operation computes, given the same initial elements as the prefix operation, and the binary associative operation $\odot$:

$$
\begin{aligned}
&e,\\
&x_1,\\
&x_1 \odot x_2,\\
&\ldots,\\
&x_1 \odot x_2 \odot \ldots \odot x_{n-1},
\end{aligned}
$$

where e is the identity element.

The approach presented by Blelloch [9] computes the prescan. To generate the prefix result the product of all elements is also required, this is computed as a byproduct of this prescan method.

To solve the prescan problem construct a balanced binary tree of depth $\lceil \log_2 n \rceil$ and place an element at each of the leaves. This assumes that the value of n is a power of 2, if this is not the case then an incomplete tree is used with the appropriate changes to the following algorithm. Consider for example the values $\{1,3,5,9,1,3,2,7\}$ and the associative operation $+$. The values have been placed into the leaves of the tree in figure 3.1. The algorithm is as follows.

- Step 1: Up sweep

  For vertices at depth $\lceil \log_2 n \rceil - 1, \ldots, 1$ compute in parallel. The value at a vertex is the product ($\odot$) of its two children.

- Step 2: The value at the top of the tree is the product of all elements. Store this and replace it with the identity element.

- Step 3: Down Sweep

  For vertices at depth $1, \ldots, \lceil \log_2 n \rceil - 1$ compute in parallel. Each vertex passes its own value to its left child and the product ($\odot$) of its own value and the value of its left child in the up sweep to its right child.

Figures 3.1 and 3.2 illustrate the prescan algorithm for $n = 8$. Figure 3.1 indicates the result of the up sweep (step 1) and figure 3.2 the result of the down sweep (step 3).

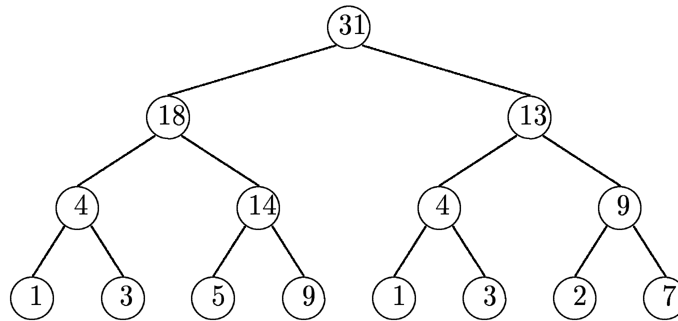

Figure 3.1: The result of the up sweep

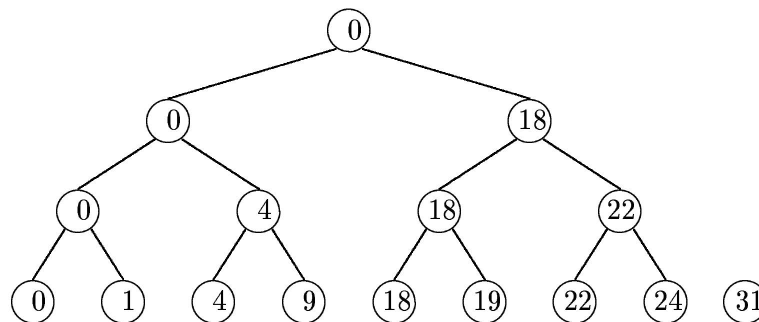

Figure 3.2: The result of the down sweep

The above algorithm will compute the prescan in time $O(\log_2 n)$, provided that there are at least $n/2$ processors. To convert the prescan into a scan or a prefix operation,

the value stored in step 2 is used as the last partial product. Notice that only when computing vertices at depth $\lceil \log_2 n \rceil - 1$ will all processors be utilised.

3.1.2 Parallel Prefix Method 2 ($p \geq n-1$)

Kogge and Stone [44] present an alternative approach, called recursive doubling, requiring a larger number of processors, although computing the prefix problem directly. A second list, x'_i, of size n is required along with step 2 to ensure that the new values of x_i are not overwritten before the old values are read. The algorithm is outlined below:

for $j = 0$ to $\lceil \log_2 n \rceil - 1$

- Step 1: In parallel compute

$$x'_i = x_{i-2^j} \odot x_i \qquad 2^j + 1 \leq i \leq n.$$

- Step 2: In parallel compute

$$x_i = x'_i \qquad 2^j + 1 \leq i \leq n.$$

end **for**.

If the number of processors is at least $n-1$ then steps 1 and 2 can be computed in unit time. This leads to an overall time requirement of $O(\log_2 n)$ to compute all prefix sums. Only when $j = 0$ will all processors be used. Figure 3.3 illustrates the case when $n = 8$. The arrows indicate which two values are combined at each iteration.

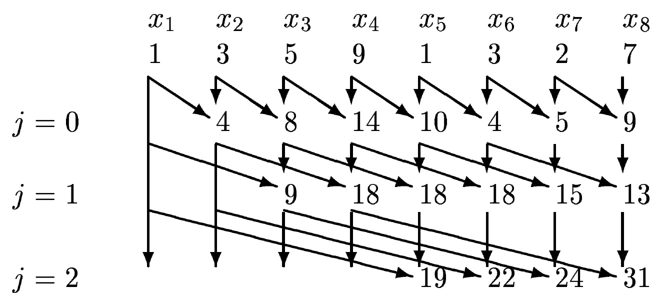

Figure 3.3: Iteration stages of parallel prefix method 2

Kruskal et al. [46] presented a version of this algorithm for the case where the x_is are stored in a linked list. Lubachevsky and Greenberg [54] produced an asynchronous version of the algorithm that can cope with input data arriving at arbitrary times and non homogeneous processors.

3.1.3 Parallel Prefix Method 3 ($p < n$)

Blelloch [9] and Kruskal et al. [45] (using the Kogge and Stone [44] algorithm) present a similar modification of the parallel prefix algorithm when the number of processors is much smaller than n. These two approaches can be summarised as follows:

Assume that the processors are numbered $k = 1, \ldots, p$.

- Step 1: In parallel, processor k computes the partial products for the elements whose indices are in the range $(k-1)n/p, \ldots, kn/p - 1$:

$$\begin{aligned} & x_j, \\ & x_j \odot x_{j+1}, \\ & x_j \odot x_{j+1} \odot x_{j+2}, \\ & \ldots, \\ & x_j \odot x_{j+1} \odot \ldots \odot x_{j+n/p-1}, \qquad \text{where } j = (k-1)n/p. \end{aligned}$$

  Denote the last partial product in this group by x'_k, $k = 1, 2, \ldots, p-1$.

- Step 2: Calculate the parallel prefix problem with input set $\{x'_1, \ldots, x'_{p-1}\}$ using either method 1 or 2 above to give:

$$\begin{aligned} & x''_1 = x'_1, \\ & x''_2 = x'_1 \odot x'_2, \\ & x''_3 = x'_1 \odot x'_2 \odot x'_3, \\ & \ldots, \\ & x''_{p-1} = x'_1 \odot x'_2 \odot \ldots \odot x'_{p-1}. \end{aligned}$$

- Step 3: In parallel, processor k generates partial products:

$$x_j = x''_{k-1} \odot x_j \qquad (k-1)n/p + 1 \leq j \leq kn/p.$$

The above algorithm will require $O(n/p)$ time to compute the first and last stages giving an overall $O(n/p + \log_2 p)$ time requirement. If the processor count is very small it may be more efficient to compute step 2 sequentially. This may be the case if the overheads of performing the parallel prefix are high.

Figure 3.4 illustrates the use of this algorithm when $n = 15$ and $p = 3$ under addition (+).

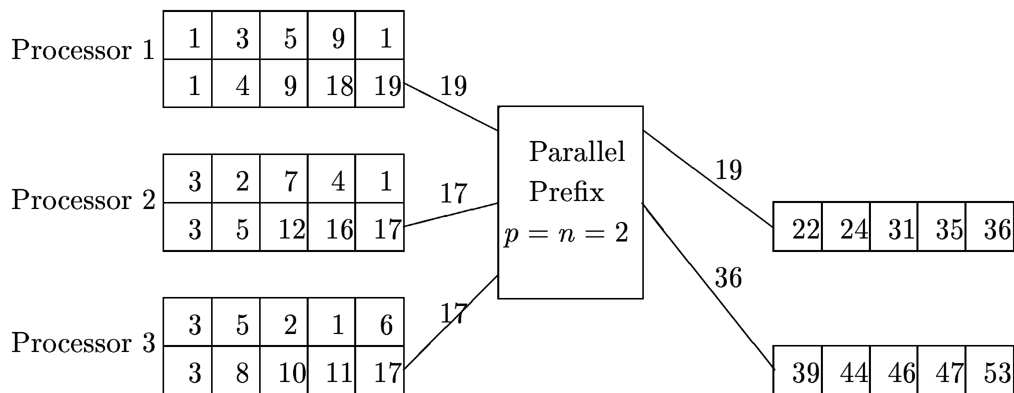

Figure 3.4: Parallel prefix When $p = 3, \quad n = 15$

3.1.4 Parallel Prefix Method 4 ($p < n$)

An alternative approach, called the harmony method, to solving the prefix problem when $p < n$ is presented by Nicolau and Wang [64]. The algorithm attempts to minimise the number of intermediate operations and fully utilise the processors. The algorithm assumes that $n \geq p(p+1)/2$. For simplicity of the description assume that n is of the form $n = m\delta + 1$, where:

$$\delta = p(p+1)/2.$$

Intermediate values are stored in elements $t_1, t_2, \ldots, t_\eta$, where:

$$\eta = (p-1)p/2.$$

The algorithm is as follows:

for $i = 0, \ldots, m-1$

- Step 1: In parallel compute:

$$x_{i\delta+2} = x_{i\delta+1} \odot x_{i\delta+2},$$

$$t_{i\eta+j+1} = x_{i\delta+j(j+1)/2+2} \odot x_{i\delta+j(j+1)/2+3} \quad 1 \leq j < p;$$

- Step r ($r = 2, \ldots, p$): In parallel compute:

$$x_{i\delta+r(r-1)/2+2} = x_{i\delta+r(r-1)/2+1} \odot x_{i\delta+r(r-1)/2+2},$$

$$x_{[i\delta+r(r-1)/2+2+l} = x_{i\delta+r(r-1)/2+1} \odot t_{i\eta+r+(2p-2-l)(l-1)/2} \quad 1 \leq l < r,$$

$$t_{i\eta+(2p-r)(r-1)/2+j+1} = t_{i\eta+(2p-r+1)(r-2)/2+j+2} \odot x_{i\delta+(j+r)(j+r-1)/2+r+2}$$

$$1 \leq j \leq (p-r);$$

end **for**.

The above algorithm requires $O(n/(p+1))$ time and is shown to be optimal.

As an illustration, take $p = 3$. Thus $\delta = 6$, $\eta = 3$ and $n = 6m+1$. The algorithm now reduces to:

for $i = 0, \ldots, m-1$

- Step 1: In parallel compute:

$$x_{6i+2} = x_{6i+1} \odot x_{6i+2},$$

$$t_{3i+1} = x_{6i+3} \odot x_{6i+4},$$

$$t_{3i+2} = x_{6i+5} \odot x_{6i+6};$$

- Step 2: In parallel compute:

$$x_{6i+3} = x_{6i+2} \odot x_{6i+3},$$

$$x_{6i+4} = x_{6i+2} \odot t_{3i+1},$$

$$t_{3i+3} = t_{3i+2} \odot x_{6i+7};$$

- Step 3: In parallel compute:

$$x_{6i+5} = x_{6i+4} \odot x_{6i+5},$$

$$x_{6i+6} = x_{6i+4} \odot t_{3i+2},$$

$$x_{6i+7} = x_{6i+4} \odot t_{3i+3};$$

end **for**.

This can be visually represented as in figure 3.5. Notice that at each step of the computation the number of parallel operations matches the number of processors.

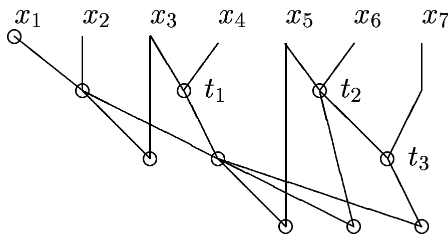

Figure 3.5: The Harmony Technique on 3 processors : $n = 7$, $m = 1$

3.2 Batch Simulation

Time parallel simulations are often required to generate results for long runs. This can require more memory than is physically available within the parallel computer. To overcome this restriction simulation time is partitioned into batches. Each batch is then simulated in parallel using all processors. The state of the system at the end of each batch is then used as the starting conditions for the next batch. The batches may be chosen of equal time length or equal number of arriving events.

This may be illustrated by figure 3.6 in which the entire simulation time has been decomposed into three batches, with each batch being simulated in parallel by all four

processors. It should be noted that the figure is just a rough illustration; the state of a system is not always described by a single variable.

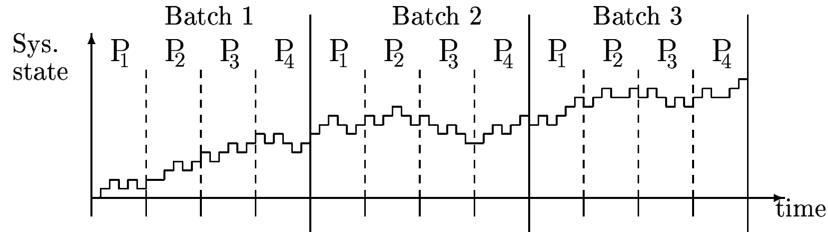

Figure 3.6: Partitioning simulation time into batches

3.3 Simulation of a G/G/1 server using Parallel Prefix

Consider a general single server system with an incoming stream of jobs that arrive at random intervals, see figure 3.7. Jobs arriving at the server are queued, in an infinite FIFO buffer, while awaiting service. The interarrival time between jobs $i-1$ and i is α_i. When a job i reaches the head of the queue it requires service of length δ_i, before being released.

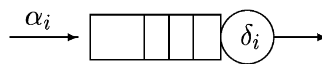

Figure 3.7: A Single G/G/1 Server

The sequence of arrival instances, A_i, and departure instances, D_i, constitute the sample path for this system. All performance metrics can be derived from this stream. The simulation task consists of computing A_i and D_i, given the inter-arrival and service times α_i and δ_i.

Equations 3.1 and 3.2 describe how to generate the arrival and departure times A_i and D_i. The solution of these using prefix computations was presented in section 3.1.

The results generated from equations (3.1) and (3.2) above can be used to determine

performance measures for the server. For example the response time for job i may be computed from:

$$W_i = D_i - A_i,$$

and the average response time is:

$$W = 1/n \sum_{i=1}^{n} W_i.$$

The queue size may be computed by merging the arrival and departure streams together and generating the step function as illustrated in figure 3.8. The average queue size may be computed from the area under this graph.

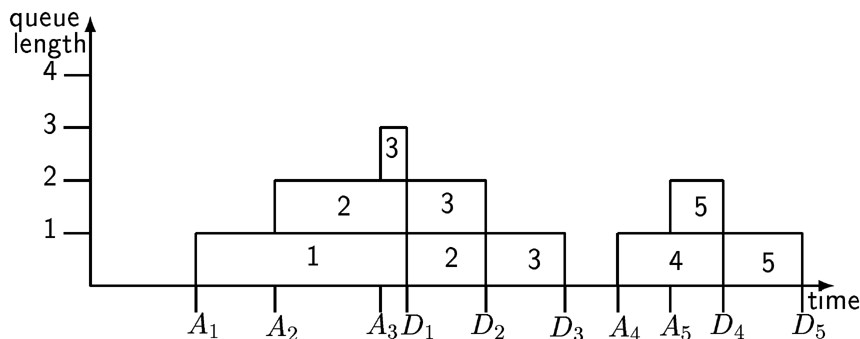

Figure 3.8: Computing the queue size

3.3.1 Implementation of the Parallel simulation

The simulation was implemented on a 14 processor Encore Multimax 520. As the processor count was small the third parallel prefix algorithm was chosen. This gives an overall execution time of $O(n/p + b \log_2 p)$. Where n is the total number of arrivals, p is the number of processors used and b is the number of batches.

When computing the matrix product, for equation (3.7), the second row will always remain constant. Thus the matrix multiplication will be carried out by two computations.

The performance measure of interest is the achieved speed-up. Other performance

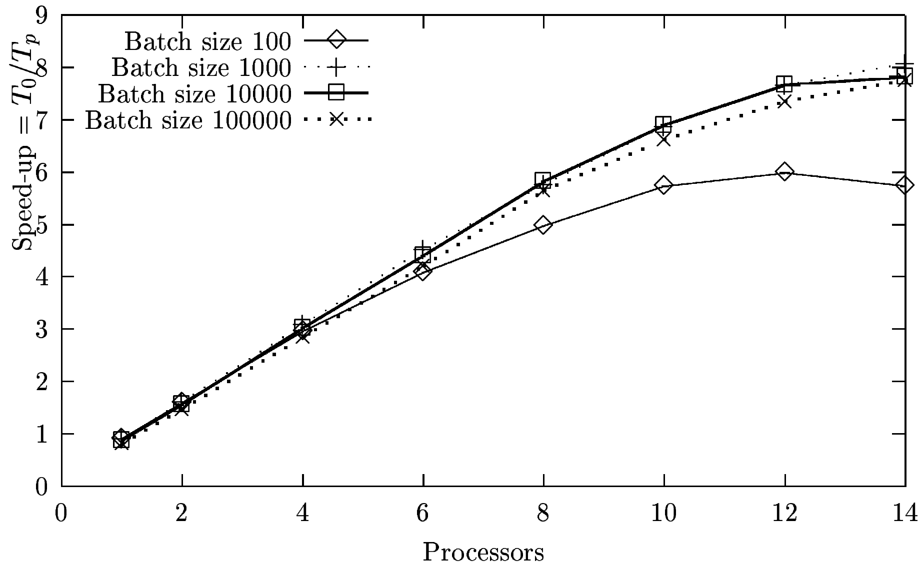

Figure 3.9: Speed-up for the G/G/1 Server using Parallel Prefix

measures were observed, though not plotted here. In this thesis speed-up S_p is defined to be T_0/T_p, where T_0 is the execution time of the optimal sequential version on one of the parallel processors and T_p is the execution time of the parallel version using p processors.

Figure 3.9 illustrates the results for the G/G/1 server. The graphs indicate the speed-up achieved as the processor count increased. If the batch size is small the simulation will not achieve optimal speed-up as the processor count increases. The results also appear to become less linear as the processor count increases. This is due to the fact that the system was a time share computer with other users. This made it almost impossible to gain exclusive access to all processors.

3.4 Parallel Merging

In this section several algorithms for performing parallel merging are described and compared.

When the two, strictly non-decreasing lists:

$$A = \{a_1, a_2, ..., a_r\},$$

$$B = \{b_1, b_2, ..., b_s\},$$

are merged a single ordered list is produced with $r + s$ elements. This new list will also be in strictly non-decreasing order. In the worst case the number of operations to perform this on a sequential computer will be O(s) when $s = r$. Thus the lower bound for a parallel merging algorithm will be $O((s)/p)$ where p is the number of processors available.

Parallel merging is of use in many areas, it is of interest in parallel simulation for merging streams of arrivals. Consider the example system presented in chapter 2 (figure 2.1). To simulate the whole system each node in the acyclic network may be simulated using the G/G/1 simulation presented in 3.3. Arrivals to the nodes may either be external arrivals generated from the recurrence equation 3.1 or from the departures from the proceeding nodes, as appropriate. Nodes B and C receive their arrivals from two separate sources, these sources will require merging in order to generate the departure times.

As each arrival stream to a node will consist of a sequence of non-decreasing arrival times the technique of parallel merging may be used to combine them.

3.4.1 Parallel Merge Method 1

Batcher [8] presented two techniques for generating merging circuits, the odd-even and bitonic mergers. Both are based upon the comparison unit illustrated in figure 3.10. The comparison unit takes two inputs (A and B) and generates two outputs, L is the smaller of A and B, and H the larger.

It is assumed for simplicity of the explanation that $r = s$. The odd-even merger is generated recursively from figure 3.11. Each odd-even merge is an implementation of

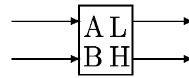

Figure 3.10: The Comparison unit

figure 3.11 until only two inputs are required, in which case a single comparison unit may be used. If the lists are not the same size then the smaller is padded out with dummy values.

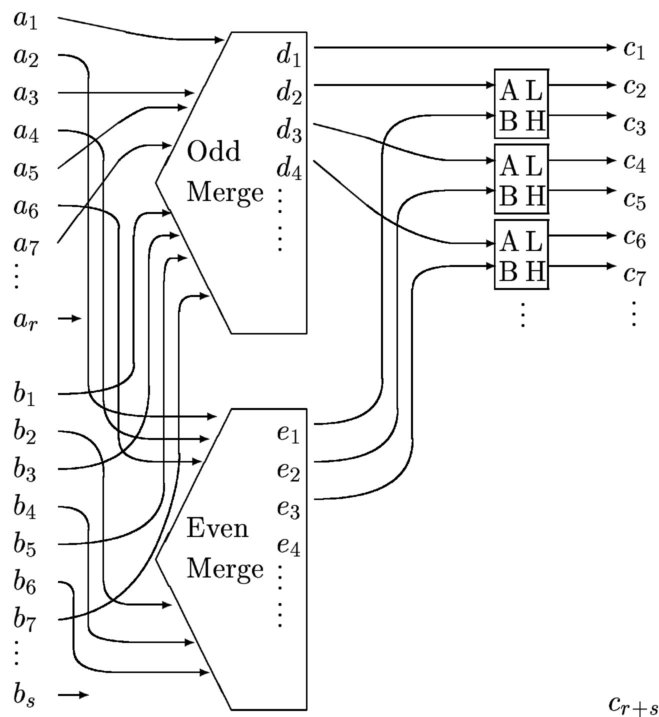

Figure 3.11: Generating Odd-Even Mergers

The depth of the circuit will be $1/2\lceil\log_2 n\rceil(\lceil\log_2 n\rceil + 1)$, where $n = \max(r, s)$. Note that for each n there is a different circuit. Figure 3.12 illustrates the circuit for the case $n = 4$.

Cole [16] illustrates how this can be implemented on a parallel computer requiring $n/2$ processors and computing the result in $O(\log_2^2 n)$ time, where $n = \max(r, s)$. Cole goes further to optimise the implementation of this algorithm by noticing that once the

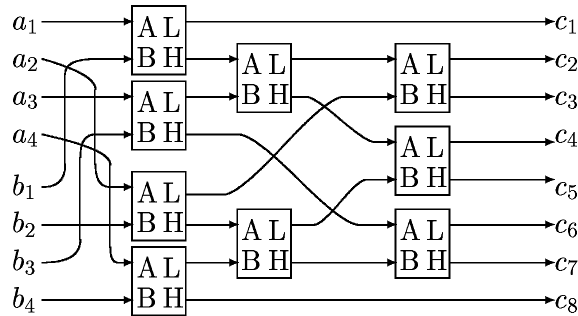

Figure 3.12: A merging circuit for two lists size 4

odd indexed elements have been computed this restricts the positions of the even indexed elements.

The Bitonic sorter presented by Batcher [8] is a more general approach. The circuit is capable of merging two lists up to the size of the circuit. There is also the opportunity to compose large circuits from smaller circuits.

Again a recursive circuit design is used. The two lists are concatenated together, the first list ascending and the second descending – giving a bitonic list. This list is recursively split into two equal sized bitonic lists, with the elements in the second list being larger than those in the first. The depth of the circuit remains the same, although the total number of comparison units is greater.

3.4.2 Techniques for merging when $p < n$

In the following methods the number of available processors is far less than the sizes of the lists to merge. To achieve a parallel merge in these cases the following general algorithm is used.

- Step 1: Partition A and B into sub-lists $A_1, \ldots, A_p$ and $B_1, \ldots, B_p$ respectively. Such that all elements in A_i and B_i are greater than all elements in A_{i-1} and B_{i-1}.

- Step 2: Each pair (A_i, B_i) of these sub-lists can be merged independently on one of the processors.

The choice of the algorithm for step 1 will depend upon the requirement to balance the distribution of work for stage 2 over the processors and the ability for processors to access the data.

The following algorithms rely on the sequential algorithms for merging lists and performing binary searches. These are common algorithms which may be found in such references as Knuth [43].

The sequential merge take two input arrays and merges them in non-decreasing order to produce one ordered array of size $r + s$ where each of the input arrays have size r and s respectively. The time requirement for this algorithm will be $O(\max(r, s))$.

The Binary search algorithm will, given an ordered list and a value x to search with, return the location of the value x in the list if present. The algorithm can easily be adapted to return the number of elements in the ordered list that would be before x, if x is not present.

At each iteration, the binary search algorithm will halve the number of elements under consideration. Therefore in the worst case it will require $O(\log_2 n)$ time to find the correct value, or location, where $r = s = n$.

If list A is a good approximation to the final list then binary searches can be used as a simple merging algorithm. List A is split into p sections of length r/p. Then processor k finds the last element less than rk/p in list B. These indices can be used to merge subsections of A and B. As the lists to be merged here may be very different this approach is not considered further.

Kruskal [46] presents a parallel binary search algorithm requiring $O(\log_2(n+1)/\log_2(p+1))$ time, which may be used to improve the above method. Kruskal goes further to perform repeated parallel binary searches to position elements of list B into list A requiring $O(n/\log_2 n \log_2 \log_2 r + n + 1)$ time. This assumes that there are $\lfloor s^{1-1/n} r^{1/n} \rfloor$ processors available and $r \geq s$.

3.4.3 Parallel Merge Method 2

A more elaborate method is suggested by Shiloach and Vishkin [72] which improves the balancing across processors by using elements from both A and B to estimate the final merged list. The algorithm is as follows.

- Step 1: In parallel select $p-1$ elements from A to evenly subdivide A into p sections, and place these $p-1$ elements into A'. Perform the same on B placing the elements into B'. $a'_k = a_{k\lceil r/p \rceil}$, $b'_k = b_{k\lceil s/p \rceil}$ $(1 \le k < p)$.

- Step 2: Merge A' and B' into a sequence $V = \{v_1, v_2, ..., v_{2p-2}\}$ where v_i is a triple consisting of an element of either A' or B' followed by its position in A' or B' followed by the name of the list it originally came from. This may be performed sequentially, using Batcher's [8] parallel circuits or using the following steps to find the correct position of a'_i:

  If j is the smallest index such that:

  $$a'_i < b'_j,$$

  then $v_{i+j-1} = (a'_i, i, A)$. If a suitable value cannot be found for j then $v_{i+p-1} = (a'_i, i, A)$. Values from B' can be places in a similar fashion.

- Step 3: Processor, k, computes a pair, q_k, of indices, in A and B, respectively. If the element stored in v_{2k-2} came from list A then $q_k = (k\lceil r/p \rceil, j)$, where j is the smallest index such that $b_j > a'_k$. Likewise if v_{2k-2} came from B then $q_k = (j, k\lceil s/p \rceil)$, where j is the smallest index such that $a_j > b'_k$.

  The individual sections for merging can now be computed as:

  $$A_k = \{a_{q_{k-1,1}}, \ldots, a_{q_{k,1}}\},$$
  $$B_k = \{b_{q_{k-1,2}}, \ldots, b_{q_{k,2}}\},$$

where $q_0 = (0,0)$ and $q_p = (r,s)$.

- Step 4: Processor k perform a sequential merge using the intervals A_k and B_k.

Example of method 2. Merge the lists

$$A = \{1, 3, 7, 11, 15, 19, 21, 35, 47\},$$
$$B = \{17, 21, 28, 29, 33, 37, 42, 67, 81\},$$

using three processors. Step 1: $A' = \{7, 19\}$ $B' = \{28, 37\}$, Step 2: $v_1 = (7, 1, A)$ $v_2 = (19, 2, A)$ $v_3 = (28, 1, B)$ $v_4 = (37, 2, B)$, Step 3: $q_1 = (1, 1)$ $q_2 = (6, 2)$ $q_3 = (9, 6)$.

This leads to the values being merged as in table 3.1

Table 3.1: Allocation of values to merge in method 3

Processor 1	Processor 2	Processor 3
1,3,7,11,15	19, 21, 35	47
17	21, 28, 29, 33	37, 42, 67, 81

It can be seen from the example that the balancing of the number of elements each processor will merge can vary considerably. Step 1 can be computed in constant time. Step 2 consists of a binary search which will take $O(\log_2 p)$ time. Step 3 consists of binary searches requiring $O(\log_2 \max(r, s))$ time. Merging at step 4 will normally take $O((r + s)/p)$ time. In the worst case, when $r = s = n$, the overall time requirement will be $O(n/p + \log_2 n + \log_2 p)$.

3.4.4 Parallel Merge Method 3

In this and the following method the lists to be sequentially merged are balanced in size. This requires more computation time for the partitioning stage, though if the lists are significantly different the performance will not be adversely affected.

Akl [2] presented a method for parallel merging of two lists on an exclusive read - exclusive write (EREW) parallel computer.

The method relies on the ability to determine the median of the final merged list before the lists are merged. The following algorithm returns a pair of indices (x, y) which satisfy the following two properties:

1) Either a_x or b_y is the lower median of the final merged list. I.e. a_x or b_y is larger than precisely $\lceil (r+s)/2 \rceil - 1$ elements of the final list and smaller than $\lfloor (r+s)/2 \rfloor$ elements.

2) If a_x is the lower median then b_y is either the largest element of B such that $b_y \leq a_x$, or the smallest element of B such that $b_y \geq a_x$. Likewise if b_y is the lower median.

The algorithm to compute the pair of indices is as follows:

- Step 1: $l_A = 1, \quad l_B = 1, \quad h_A = r, \quad h_B = s,$

  $t_A = r, \quad t_B = s, \quad u = \lceil r/2 \rceil, \quad v = \lceil s/2 \rceil.$

  while $t_A > 1$ and $t_B > 1$ do

  - Step a: **if** $a_u \geq b_v$

    $t_A = u - l_A + 1, \qquad t_B = h_B - v,$

    $h_A = u, \qquad l_B = v + 1.$

    else

    $t_A = h_A - u, \qquad t_B = v - l_B + 1,$

    $l_A = u + 1, \qquad h_B = v.$

    end **if**.

  - Step b: $u = l_A + \lceil (h_A - l_A - 1)/2 \rceil,$

    $v = l_B + \lceil (h_B - l_B - 1)/2 \rceil.$

  end **while.**

- Step 2: Return the indices of the pair from $\{a_{u-1}, a_u, a_{u+1}\} \times \{b_{v-1}, b_v, b_{v+1}\}$ which satisfies (1) and (2) above. If multiple pairs satisfy (1) and (2) return the pair for which $x + y$ is the smallest.

This will require $O(\log_2(\min(r, s)))$ time. In the worst case when $r = s = n$ this becomes $O(\log_2 n)$.

The parallel merge method can now be described as follows:

- Step 0: Send the ranges $A[1, r]$ and $B[1, s]$ to processor 1.
- Step $j (1 \leq j \leq \log_2 p)$: Processor k receives the range $A[e, f]$ and $B[g, h]$ and computes the median pair for these ranges (x,y). Compute the ranges to communicate as:
  - Step a: **if** a_x is the median

    $p_1 = x, \quad q_1 = x + 1.$

    if $b_y \leq a_x$

    $p_2 = y, \quad q_2 = y + 1.$

    else

    $p_2 = y - 1, \quad q_2 = y.$

    end **if**.

    else

    $p_2 = y, \quad q_2 = y + 1.$

    if $a_x \leq b_y$

    $p_1 = x, \quad q_1 = x + 1.$

    else

    $p_1 = x - 1, \quad q_1 = x.$

    end **if**.

    end **if**.
  - Step b: Communicate the ranges $A[e, p_1]$ and $B[g, p_2]$ to processor $2k - 1$.
  - Step c: Communicate the ranges $A[q_1, f]$ and $B[q_2, h]$ to processor $2k$.
- Step $\log_2 p + 1$: Each processor k receives the ranges $A[e, f]$ and $B[g, h]$. These two ranges may be safely merged independently of all other processors.

The overall time requirement for this method is $O(n/p + \log_2 p \log_2 n)$. The following example uses the same data as the previous example. For ease the merge is performed over four processors.

Example of method 3. Table 3.2 indicates which ranges are held on the processors at the end of each step. Table 3.3 indicates the final set of data that will be merged on each of the four processors.

Table 3.2: Computing the ranges over four processors

Processor	1	2	3	4
Step 0	$A[1,9]B[1,9]$			
Step 1	$A[1,7]B[1,2]$	$A[8,9]B[3,9]$		
Step 2	$A[1,5]B[1,0]$	$A[6,7]B[1,2]$	$A[8,8]B[3,6]$	$A[9,9]B[7,9]$

Table 3.3: Allocation of values to merge in method 4

Processor 1	Processor 2	Processor 3	Processor 4
1,3,7,11,15	19, 21	35	47
	17, 21	28, 29, 33, 37	42, 67, 81

The approach of using the median to partition the data was also used by Plaxton [70] for merging on a hypercube. In this approach each time the median is determined the hypercube is partitioned into two, a lower and upper half. The values greater than the median are sent to the upper hypercube and the values less than the median are sent to the lower hypercube. This process is applied recursively until all values are in order.

3.4.5 Parallel Merge Method 4

In this method the exact indices are computed such that each processor will merge $(r + s)/p$ elements. The method was presented by McGough [56] and is based on a parallel sorting algorithm by Abali et al. [1]. The method determines pairs of indices

(x, y) such that $x + y = k(r + s)/p$ $(1 \leq k \leq p)$ and:

$$a_x < b_j \quad \forall j > y,$$
$$b_y < a_i \quad \forall i > x.$$

Given that v is the index in the final merged list that is desired, then the following algorithm will determine appropriate values for (x, y):

- Step 1: $u_A = r, \quad u_B = s, \quad l_A = 1, \quad l_B = 1, \quad x = 0, \quad y = 0.$

  while $x + y \neq v$

  - Step a: $m_A = (u_A + l_A)/2, \qquad m_B = (u_B + l_B)/2,$

    $g = \min(a_{m_A}, b_{m_B}).$

  - Step b: Find the smallest values for x and y such that $g \leq a_x$, and $g \leq b_y$ respectively. These can be performed using a binary search.

  - Step c: **if** $x + y > v$

    $u_A = x - 1, \qquad u_B = y - 1.$

    end **if**.

  - Step d: **if** $x + y < v$

    $l_A = x, \qquad l_B = y.$

    end **if**.

  end **while**.

The above algorithm will require $O(\log_2^2 n)$ time in the worst case (r=s=n). Thus the parallel merge will now become:

- Step 1: Processor k $(1 \leq k < p)$ computes (x_k, y_k) such that $x_k + y_k = k(r + s)/p$ using the above algorithm. Set $(x_0, y_0) = (1, 1)$, $(x_p, y_p) = (r, s)$.

- Step 2: Processor k can perform the sequential merge on the interval:

$$A_k = \{a_{x_{k-1}}, \ldots, a_{x_k}\},$$
$$B_k = \{b_{y_{k-1}}, \ldots, b_{y_k}\}.$$

Step 1 will require $O(\log_2^2(r+s))$ time, with step 2 requiring $O((r+s)/p)$ time. This gives an overall worst case $(r = s = n)$ time of $O(n/p + \log_2^2 n)$.

Example of method 4. Using the same data as the previous examples and merging over three processors.

The values computed for (x_k, y_k) are $(x_0, y_0) = (1,1)$, $(x_1, y_1) = (5,2)$, $(x_2, y_2) = (8,6)$, $(x_3, y_3) = (9,9)$. Table 3.4 indicates the values that will be merged on each processor.

Table 3.4: Allocation of values to merge in method 5

Processor 1	Processor 2	Processor 3
1,3,7,11,15	19, 21	35, 47
17	21, 28, 29, 33	37, 42, 67, 81

3.5 Parallel Split

If a stream of arrivals requires routing on departure from a node, this may be performed by a parallel splitting algorithm in which the departure times are placed into one of several lists for delivery to the input for another node. The routing may be either state dependent, in which case the choice of destination is dependent on the current state of the system, or state independent. Parallel splitting is only required for the acyclic network in section 3.6 below. In this case the routing is performed in a state independent manner.

The algorithm described below is state independent and assumes that each departures destination may be chosen at random. For simplicity it is assumed that there are only

two destinations B and C, however this may easily be increased.

- Step 1: In parallel for each element i in A select a destination at random. Mark each departure with its destination.
- Step 2: Perform a parallel prefix operation to determine the index of each departure sent to B. For each i define the value ρ_i as follows,

$$\rho_i = \begin{cases} 1 & \text{if } i \text{ is sent to } B \\ 0 & \text{otherwise.} \end{cases}$$

  Perform the parallel prefix method from section 3.1 with addition as the associative operation. Store the result for element i in j_i.

  If element i is to be sent to B then $b_{j_i} = a_i$.

- Step 3: Repeat this process for elements to be sent to C.

Step 1 will require $O(1)$ time if $p \geq n$, otherwise $O(n/p)$ time. The time requirement for step 2 will depend on the parallel prefix method chosen.

3.6 The Acyclic Queueing Network

All components required to perform a parallel simulation of an acyclic network are now available. This section gives the outline of a time-parallel simulation of the network shown in figure 3.13.

A time-parallel simulation of a network may be performed in two ways. The entire network may be simulated on each processor in parallel. Alternatively the network may be decomposed into sub-systems. Each sub-system is then simulated in a time-parallel manner.

To simulate the entire of the acyclic network in figure 3.13 using recurrence relations would require equations capable of dealing with the re-ordering of packets due to al-

ternative routes. This is not a trivial matter as packets may be arbitrarily re-ordered. Alternatively the network can be decomposed into sub-systems and solved using the techniques of parallel prefix, parallel merge and parallel split.

3.6.1 The acyclic network model

Each of the nodes A, B and C represent a single server with an infinite FIFO buffer. Packets may enter the network at nodes A and B. The ith interarrival intervals for nodes A and B are $\alpha_{A,i}$ and $\alpha_{B,i}$ respectively, each being generally distributed. Each node requires a generally distributed random interval to process packet j of length $\delta_{A,j}$, $\delta_{B,j}$ and $\delta_{C,j}$ as appropriate.

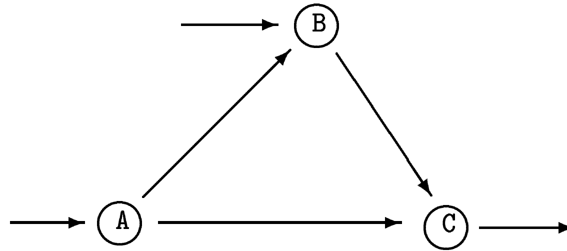

Figure 3.13: The acyclic network

Departure of packet i from node A may either progress to node B, with probability p_i, or to node C with probability $1 - p_i$. It is assumed that p_i is independent of the system state. The time for communicating packets between nodes is assumed to be zero. This may be relaxed by adding the transmission times to departure times.

As each node represents a single G/G/1 server the equations presented in section 3.3 may be used to describe each node. With arrivals at nodes B and C requiring merging and departures at node A requiring routing (splitting) to B and C.

The arrival time at A or B and departure time from C may be used to determine the average response time. The values for arrivals and departures at each node may be used to compute the average queue size.

3.6.2 Parallel simulation of the acyclic network

As with the simulation of the G/G/1 server the simulation will be split into batches. However, in this case there is a dependency between batches. For example suppose that a packet arrives at node A just before the end of a batch. If the packet is routed via B then its time of arrival at C may be within the next batch. Thus later packets, in the next batch, routed directly from A to C may arrive before this packet.

To deal with this complication before each critical stage of the simulation is performed the last packet which is safe to process is computed. Any packets after this are saved for the next batch of the simulation. Each batch will generate up to L arrivals for each external arrival source.

Below is described the steps to perform in simulating the network.

- Step 1: Generate arrivals at node A so that there is a total of L available, call these arrivals A_A. This may be performed using equation (3.1) and parallel prefix.

- Step 2: Generate departures for node A, called D_A. This is performed using equation (3.2) and parallel prefix.

- Step 3: Generate external arrivals at node B so that there is a total of L available, call these A_B. This is performed using equation (3.1) and parallel prefix.

- Step 4: Compute f as the minimum of the last departure time from D_A and the last arrival time from A_B. All packets arriving at B with time stamps less than f will be processed in this batch, all remaining packets will be retained for the next batch.

- Step 5: For the packets from D_A with time stamps less than f use the parallel split algorithm to generate the two lists D_{AB}, departures routed to B, and D_{AC}, departures routed for C.

- Step 6: Merge the packets from A_B, with time stamps less than f, with D_{AB}. This new list of arrivals is called A_{AB}. The merge can be performed using one of the parallel merge algorithms.

- Step 7: Compute the list of departures D_B from A_{AB} using (3.2) and parallel prefix.

- Step 8: Compute g as the minimum of the last departure time in D_{AC} and the last departure time in D_B. Again packets with time stamps greater than g will be saved for the next batch.

- Step 9: Merge the packets from D_{AC} and D_B with time stamps less than g to give A_C the arrivals at node C.

- Step 10: Compute the departure times from node C (D_C).

The above algorithm can be repeated until the required number of packets have been simulated.

3.6.3 Parallel Implementation of the acyclic network

The parallel simulation was implemented on an Encore Multimax 520 using Encore Parallel Threads. A total of 14 processors were available for experimentation. Again the third parallel prefix method was chosen due to the small processor count. The forth parallel merge algorithm was chosen as this allowed good load balancing across the processors.

The parallel prefix operations, to compute arrival and departure times, will require $O(n/p + \log_2 p)$ time. The parallel merges will require $O(n/p + \log_2^2 n)$ time and the parallel split $O(n/p + \log_2 p)$. The computations of f and g can be computed in $O(1)$ time. Thus the overall simulation will be performed in $O(n/p + b \log_2^2 n + b \log_2 p + 1)$ time, where b is the number of batches.

Various performance measures were obtained from the simulations. However, as with the G/G/1 simulation only the speed-up of the simulation with respect to the number of processors is presented here.

Figure 3.14 illustrates the speed-up obtained for the simulation when run with different processor counts. The speed-up is almost linear for the larger batch sizes. If the batch size is particularly small the speed-up decreases rapidly from linear. This decrease is greater than in the G/G/1 simulation. This is likely to be caused by the extra synchronisation from the larger number of steps, the fact that each batch will now not be complete and the $O(\log_2^2 n)$ time required to partition the merges. The $O(\log_2^2 n)$ will become more significant as the batch size decreases.

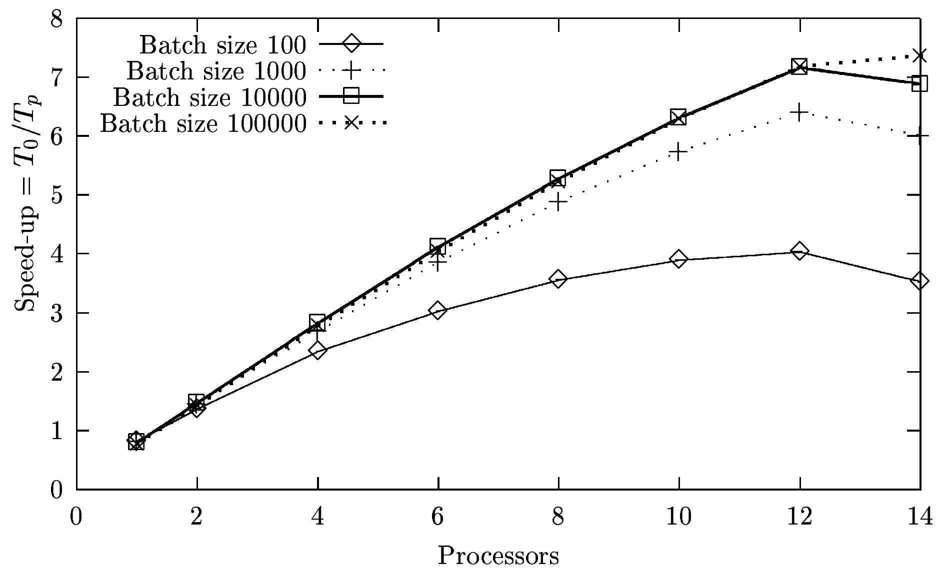

Figure 3.14: Speed-up for the Acyclic network using Parallel Prefix, merge and split

As with the G/G/1 simulations the results for 14 processors is worse than expected due to the time-sharing nature of the system in use.

3.7 Conclusion

Both the parallel prefix and parallel merge techniques are well developed areas of computing science. The algorithms presented here only cover a small portion of those available. The interested reader is referred to Blelloch [9] for parallel prefix methods and Akl [2] and Cole [16] for parallel merging. However, the use of these methods, together with sets of recurrence relations, in computing simulation sample paths, is not well known.

The choice of the parallel prefix method will depend on the nature of the parallel computer in use. The number of available processors is the main consideration. Likewise for parallel merging algorithms, the number of processors and the desire for load balancing will affect the choice of algorithms.

For the case of the single server queue the speed-up obtained is as hoped for, in terms of the order of operations. The increase in speed of the code for larger numbers of processors provides a good approximation to the operation count. The best speed-up of 8.08 is observed in the case of 14 processors with batch size of 1000. Each simulation of the single server, with a fixed number of processors appears to have an optimal batch size, for which the execution time of the simulation is minimised.

The acyclic network shows the same properties as the single server queue, in that the execution time of the code appears to be approximately proportional to n/p. This is not obvious from the order of operations for the simulation as the number of packets processed per batch is not constant. In the same manner the execution times also appear to have an optimal batch size for a given number of processors. This again is not obvious from the operation count.

Chapter 4

Simulating the ATM Switch in shared memory

The performance evaluation of ATM switches has been of great interest for some time, due to its importance in the new backbone for the internet. Much effort has been expended in the simulation of such systems in an effort to provide optimal performance. It is desired to have a very low cell loss probability, often quoted in the order of 10^{-9}. Losses occur in ATM switches due to buffer overloads in the multiplexor, which is used to combine a number of sources. To adequately simulate such a system, to give accurate estimates on the loss probabilities, it is normal to simulate on the order of 10^{12} arrival instances. This is due to the fact that in such system the losses tend to be grouped together.

In this chapter two algorithms are presented for simulating an ATM switch on a shared memory multiprocessor. If the buffer overflow events are reasonably rare then the run time of the system is roughly $O(n/p)$ in both cases.

4.1 The Model

Consider an ATM switch with transmission capacity C cells/second, a buffer of size Q cells and an input source formed by merging M independent bursty sources of the "on"/"off" type, see figure 4.1 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.

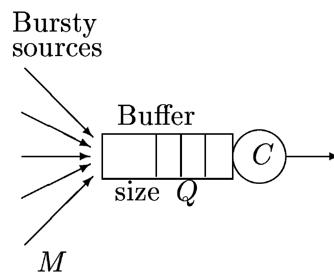

Figure 4.1: ATM Switch with multiple sources

4.2 Related Work

In order to reduce the large simulation times, considerable effort has gone into exploring parallel computation techniques. In particular, the parallel simulation of Multiplexors, as used in ATM networks, has attracted much attention over recent years. Space-parallel simulation techniques have been proposed by Xiao et al. [78], Williamson et al. [77] and Carothers and Perumalla [11], for simulating networks of switches.

Fujimoto et al. [28] presented an approximate time-parallel simulation of an ATM switch. This work was extended to a network of cascading switches by Nikolaidis et al. [65].

Wang and Abrams [75] presented an approximate method for simulating a G/G/1/K queue. An alternative method for generating upper and lower bounds on the cells lost was developed by Lin [48, 49].

Chen [15] 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 this model.

An approach to determine the correct end state for the relaxation approach was presented by Andradóttir and Ott [3]. This technique could be adapted for use in the work presented.

4.3 Outline of Solution

The two algorithms presented in this chapter work in a time-parallel manner and achieve almost linear speed-up.

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 by Greenberg et al. [35], in the context of open queueing networks without losses (see section 3.1).

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 chapter 3):

- 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:

Step 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.

Step 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 Abali et al. [1].

Step 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-intervals 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 finalising the accepted arrivals, this algorithm computes the sequence of cell departure instants (again using Parallel Prefix), ready for the next batch.

The 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 a portion of the sample path corresponding to its own sub-section, assuming some initial conditions for the state of the queue. Relaxation is then used to finalise each portion. 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 this work has 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.

4.4 Generation of Bursty Arrivals

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 arrive. All sources start with an "on" period. The j th "on" and "off" periods for source i are denoted by $\xi_{i,j}$ and $\eta_{i,j}$ respectively. The n th interarrival time for source i is denoted $\alpha_{i,n}$. 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 4.2, 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:

$$a_{i,n+1} = a_{i,n} + \alpha_{i,n+1} \quad ; \quad n = 1, 2, \dots \ , \tag{4.1}$$

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 section 3.1). 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.

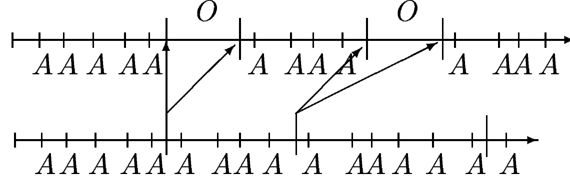

Figure 4.2: Unadjusted and adjusted arrival instants

The second step consists of adjusting $a_{i,n}$ by inserting the missing "off" periods in the appropriate positions (see upper part of figure 4.2, 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, $\xi_{i,j}$, an index l for each $a_{i,n}$ needs to be found such that:

$$\sum_{j=1}^{l-1} \xi_{i,j} < a_{i,n} \leq \sum_{j=1}^{l} \xi_{i,j} , \tag{4.2}$$

where an empty sum is 0 by definition.

Having solved the inequalities (4.2) for l, the actual arrival instant of cell n from source i, $A_{i,n}$ is obtained from:

$$A_{i,n} = a_{i,n} + \sum_{j=1}^{l-1} \eta_{i,j} . \tag{4.3}$$

The adjustment procedure described above assumes that the realisations 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, this pre-computation can be treated 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 (4.2) and (4.3) 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 $O(B/P)$.

4.5 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: $\xi_i = E(\xi_{i,j})$; $\eta_i = E(\eta_{i,j})$. 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, $\alpha_i = E(\alpha_{i,n})$. 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:

$$s_i = \frac{T}{\xi_i + \eta_i} \; . \tag{4.4}$$

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

$$m_i = s_i \beta_i = \frac{T\xi_i}{\alpha_i(\xi_i + \eta_i)} \; . \tag{4.5}$$

Normally a simulation will run not for a fixed amount of time, T, but for a fixed total number of cells from all sources, N. In view of (4.5), those two quantities are related

(approximately) as follows:

$$N = T \sum_{i=1}^{M} \frac{\xi_i}{\alpha_i(\xi_i + \eta_i)} \ . \tag{4.6}$$

From (4.4) and (4.6) it follows that the average number of "on"-"off" cycles occurring in source i during a simulation where a total of N cells are generated from all sources, is equal to:

$$s_i = \frac{N}{\xi_i + \eta_i} \left[\sum_{i=1}^{M} \frac{\xi_i}{\alpha_i(\xi_i + \eta_i)} \right]^{-1} \ . \tag{4.7}$$

4.6 Merging of Arrival 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 section 3.4). Some care needs to be exercised in order to ensure that the work is divided approximately equal among the processors. After some binary searching, each of the sources provides an appropriate sub-section to each of the processors. The latter then merge their respective sub-sections 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.

To determine the sub-sections of arrivals that processor k will merge, it is necessary to compute values v_k such that there are a total of kB/P cells with arrival times before v_k. Then processor k can sequentially merge all cells that satisfy:

$$v_{k-1} < A_{i,n} \leq v_k \quad ; \ k = 1, 2, ..., P,$$

from the M arrival sources ($v_0 = 0$). All remaining cells with arrival times greater than v_P are kept for the following batch. The values v_k can be computed independently by

each processor, by computing estimates e_k, and iteratively improving on these estimates until there are kB/P cells that arrive before e_k.

The algorithm is similar to that of a binary search. An upper and lower bound for v_k is computed for each source. Denote the index of the cell from source i that is the lower bound for v_k by $g_{i,k}$, and the index for the upper bound by $G_{i,k}$. Initially, $g_{i,k} = 1$ and $G_{i,k} = B$. At each iteration processor k computes the next estimate for v_k as:

$$e_k = \frac{1}{M} \sum_{i=1}^{M} A_{i,\lfloor (g_{i,k}+G_{i,k})/2 \rfloor}. \tag{4.8}$$

If $g_{i,k} > G_{i,k}$ for some source, the latter is excluded from the above calculation. The number of cells ρ_k with arrival times before e_k from all arrival sources is then computed. This value can then be used to improve on the estimation of e_k. More precisely:

1. If $\rho_k < kB/P$ then e_k is too small an estimate for v_k. $g_{i,k}$ now become the index of the first cell with arrival time greater than e_k.

2. If $\rho_k > kB/P$ then e_k is too large an estimate for v_k. $G_{i,k}$ now become the index of the last cell with arrival time less than e_k.

3. If $\rho_k = kB/P$ then e_k is the final value for v_k.

In cases 1 and 2 above, new estimates are computed using equation (4.8) and new values are computed for ρ_k. The process is repeated until final values are computed for each v_k.

In the absence of identical arrival times the above algorithm can be used to compute values for v_k. However if identical values do occur the above algorithm will reach a state where the $g_{i,k}$ and $G_{i,k}$ do not change from one iteration to the next. In this situation the last value of e_k can be chosen for v_k as this will be within M cells of the correct value.

The example in tables 4.1, 4.2 and 4.3 below show the calculation of v_k for the case

of four processors with a batch size of 12. The lists of arrival times for each source are shown in table 4.1 below.

Table 4.1: Arrival instances in each source

Source 1	1, 2, 5, 8, 9, 12, 13, 20, 22, 23, 24, 25
Source 2	3, 4, 6, 7, 10, 14, 15, 16, 17, 18, 19, 20
Source 3	9, 11, 14, 16, 26, 28, 30, 31, 32, 33, 34, 35

Table 4.2: Proposed values for e_k at each iteration

iteration	1	2	3	4	5	6	v_k
e_1	18	$9\frac{2}{3}$	6	$2\frac{1}{2}$	4	3	3
e_2	18	$9\frac{2}{3}$	6				6
e_3	18	$9\frac{2}{3}$	6	8	$8\frac{1}{3}$		$8\frac{1}{3}$
e_4	18	$9\frac{2}{3}$	$13\frac{2}{3}$	11			11

Table 4.3: Arrival times to be merged by processors

Processor	1	2	3	4	Left for the next batch
source 1	1,2	5	8	9	12,13,20,22,23,24,25
source 2	3	4,6	7	10	14,15,16,17,18,19,20
source 3				9,11	14,16,26,28,30,31,32,33,34,35

From table 4.2, processors 2 and 4 computed valid values for v_k in 3 and 4 iterations respectively. Processor 3 fails to find a valid value for v_k as there are two arrivals at time 9. This fact is observed and v_3 is chosen as $8\frac{1}{3}$. Processor 1 takes six iterations to converge on a valid value for v_1, however it can be noted that from iteration 4 the estimates for e_1 give results of ρ_1 close to B/P. It is possible to take advantage of this fact in order to reduce the number of iterations to be performed, at the expense of processors no longer merging the same number of cells. Iterations could now be set to terminate as soon as ρ_k is within some specified range of kB/P. Processor 3 could have also taken advantage of this property, allowing it to terminate before the repeated values are detected.

4.7 Cell Acceptance Algorithms

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.

4.7.1 Fixed buffer size Algorithm

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 at least one free 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 $A_n \leq D_m$, $m = 1 - B, 2 - B, \ldots, 0$, then it finds the buffer full and is lost.

3. If neither of these conditions is satisfied, then:

$$D_{n-B-s} < A_n < D_{n-B} \ , \tag{4.9}$$

for some $0 < s < B$. Let s be the *smallest* integer for which (4.9) holds. In that case, cell n is accepted if at least s preceding arrivals from the current batch are lost, otherwise it is lost.

This situation is illustrated in figure 4.3, 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).

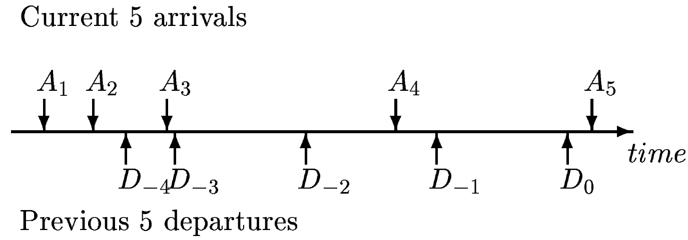

Figure 4.3: Determining cell acceptance and loss; $B = Q = 5$

The B arrivals in the current batch are divided into sub-intervals numbered $1, 2, \ldots, P$, in chronological order. These are processed in parallel by processors $1, 2, \ldots, P$, respectively. Given the departure instants of the previous batch, processor k applies criteria 1, 2 and 3, and classifies each cell in its sub-section as (a) accepted, (b) lost and (c) possibly accepted. In case (c), a note is made of the number, s_n, of cells *in previous sub-intervals* 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 sub-section). 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 sub-section.

Next, the partial sums:

$$ll_k = \sum_{i=0}^{k-1} l_i \quad ; \quad LL_k = \sum_{i=0}^{k-1} L_i \; , \tag{4.10}$$

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 $s_n \leq ll_k$ then cell n is

reclassified as accepted; if $s_n > LL_k$ 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 (sub-section 2 is finalised on the 2nd iteration, sub-section 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.

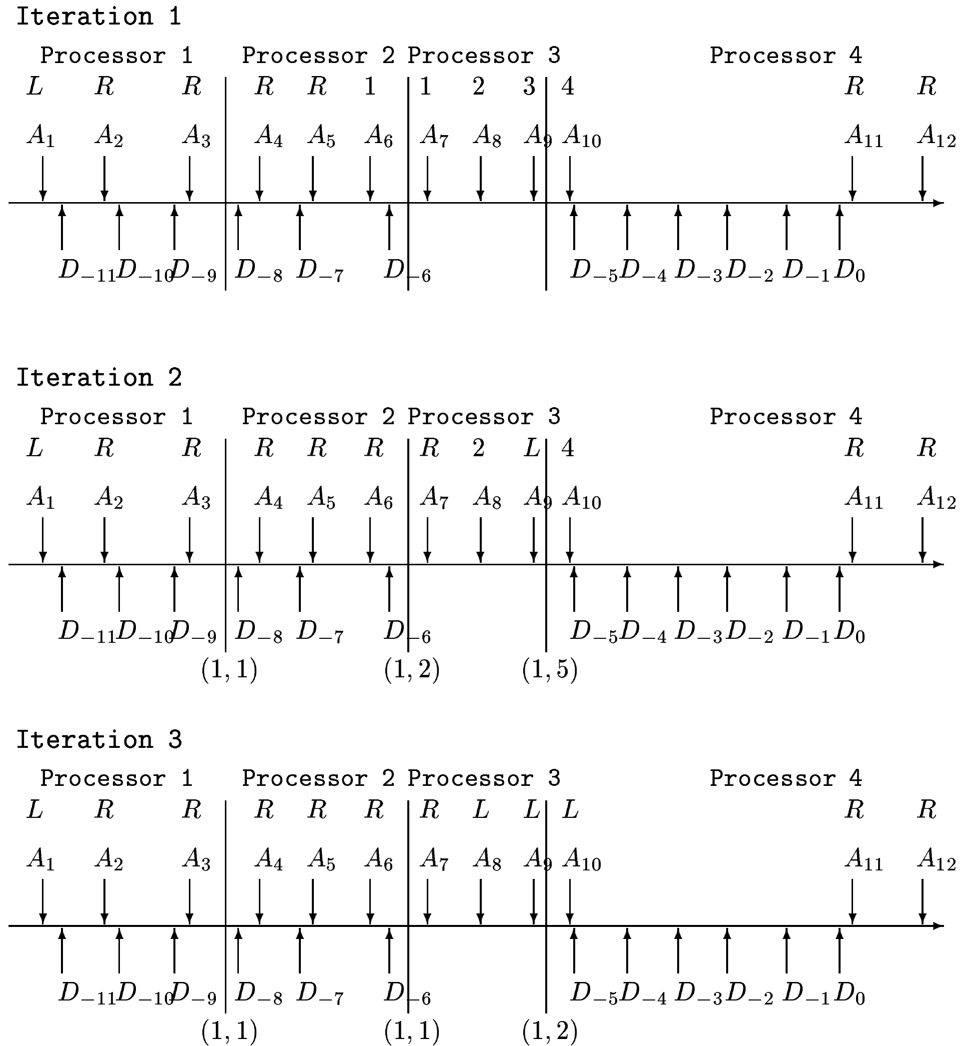

Figure 4.4: Example of computing cell acceptance with algorithm 1

The example in figure 4.4 shows how these iterations compute the accepted and lost cells in the case where $B = Q = 12$ with four processors. Processor 1 can compute the exact state for its cells during iteration 1 (marking lost cells in the figure with an 'L' and accepted cells with a 'R'), however all other processors use criteria 1, 2 and 3 above to determine those cells that can definitely be lost or accepted at this stage. Those arrivals that satisfy 1 and 2 above, for processors $k > 1$, are also marked as 'L' or 'R' respectively. Those arrivals which satisfy 3 are marked with the number of cells that must be lost in order for that arrival to be accepted. Processor 2 can accept all its cells apart from the last cell (arrival A_6) which from equation (4.9) requires at least one cell loss for cell 6 to be accepted. Processor 3 cannot directly accept any of its cells and computes that cells $7, 8$ and 9 require at least $1, 2$ and 3 cells to be lost respectively. Processor 4 can directly accept the last two cells and computes that it can only accept cell 10 if at least 4 cells are lost.

For the second iteration the values of ll_k and LL_k are computed and displayed underneath the separators between each processor in brackets (ll_k, LL_k). Processor 2 now knows that exactly one cell is lost before its first arrival, thus it can accept cell 6. For processor 3 the cell loss is bounded between 1 and 2, thus arrival 9 is lost as this would require at least 3 cells to be lost. Processor 4's cell loss is bounded between 1 and 5 thus cannot improve its state at this iteration.

In the third iteration the (ll_k, LL_k) pairs are re-computed, which allows processor 3 to accept cell 7 and reject cell 8 as only one cell is lost before this point. Cell 10 is also rejected at this stage as processor 4 knows that a maximum of 3 cells are lost. At this stage all cell states are known and $ll_k = LL_k$ for all processors, thus iterations cease.

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:

$$D_{n+1} = \max(A_{n+1}, D_n) + c \,, \tag{4.11}$$

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 section 3.1 for the solution to this recurrence). For ease all cells that were lost are removed, and the list of remaining cells are re-ordered such that all remaining cells are numbered consecutively.

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.

4.7.2 Variable buffer size Algorithm

This algorithm allows cells to be handled in arbitrarily large batches. Again the B arrivals in the batch are divided into P sub-intervals and are processed in parallel by the P processors, but processor k now computes the queue size at the arrival instants in its sub-intervals, 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 4.5. Note the actual departure times D_n are marked on for clarity.

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

A_1 A_2 D_1 A_3 D_2 D_3 A_4

d_1 d_3 d_4

d_2

Figure 4.5: The values of d_n

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

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:

$$q_{n+1} = \max(q_n + \sigma_n - \delta_n, 0) , \tag{4.12}$$

$$d_{n+1} = \begin{cases} d_n + \delta_n c & \text{if } q_{n+1} > 0 \\ A_{n+1} & \text{if } q_{n+1} = 0. \end{cases} \tag{4.13}$$

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 (4.12) and (4.13) for its sub-interval 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 sub-section

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 sub-interval k, as computed by processor k in one of the iterations:

$$I_k = \sum_n I(q_n = 0)[A_n - d_{n-1} - c(q_{n-1} + \sigma_{n-1})] \ , \tag{4.14}$$

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-section, 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 t consecutive cells, $\{n+1, n+2, \ldots, n+t\}$, 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', referred to as a *cluster*, for the purpose of calculating the evolution of the queueing process. Instead of computing t pairs of recurrences (4.12) and (4.13), a single pair is evaluated:

$$q_{n+t} = q_n + t - \delta_{n,t} \ , \tag{4.15}$$

$$d_{n+t} = d_n + \delta_{n,t} c \ , \tag{4.16}$$

where:

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

The effectiveness of this technique is evaluated in chapter 5, the results there should also apply to the shared memory algorithm presented here.

Figure 4.6 is an example of **1** & **2** above where the block size B is 20 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-section. Note that the algorithm only computes the queue size immediately before and after a cell arrival, the departures have been added to the 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-block is finalised.

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.

To apply the simplification **3** above, it is necessary to group sequences of consecutive

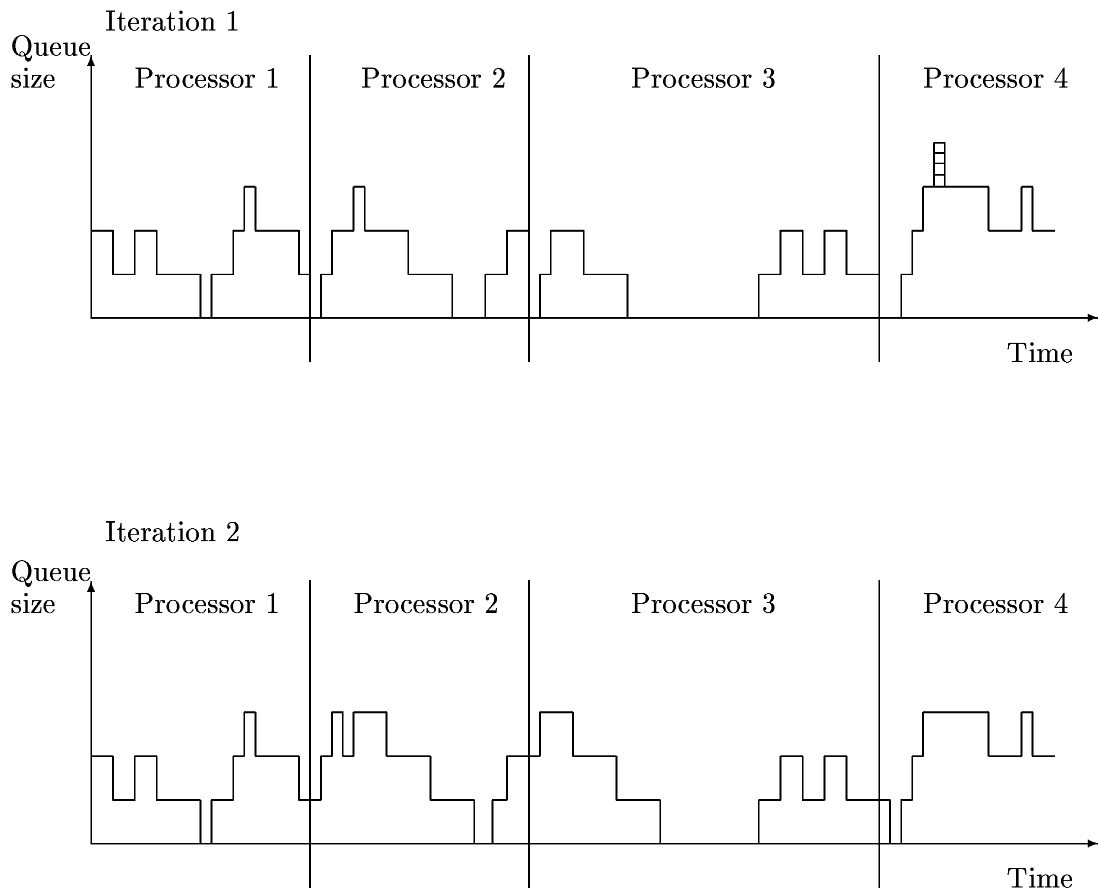

Figure 4.6: Example graph of queue sizes

cells into clusters, such that no cell arrives to an empty queue (except possibly the first cell). The clusters are also limited to T cells in size.

During the first iteration processors 2, ...,P generate tentative clusters based on the above definition assuming all cells are accepted. The amounts by which the queue can grow and decrease during a cluster are recorded as U and u respectively. Those amounts include the uncertainties due to the initial departure time.

At any future stage, where the queue size q_n before the cluster changes, it is possible to compute an upper and lower bound on the queue size during the cluster. The upper bound is computed as:

$$V = q_n + U. \tag{4.17}$$

The lower bound is computed in a similar manner:

$$v = q_n - u. \tag{4.18}$$

Thus to determine at any iteration whether a collection of cells is a valid cluster, it is necessary to show that v, V are both bounded within [1,Q]. Otherwise the collection may either lose cells or the queue may empty. In either of these two cases the cluster is decomposed into its individual cells.

The limitation given above that a cluster should not exceed T cells is intended to limit the effect of a large cluster being decomposed due to a single cell loss.

The diagram below (fig 4.7) illustrates how cells are grouped into clusters with the following diagram (fig 4.8) giving the effect of one potential queue size and departure time combination from the previous processor. Here a cluster is limited in size to five cells. Cluster 1 (O_1) consists of one cell, as the queue size reaches 0 after the first arrival. Cluster 2 is of size 2 as the queue size reaches zero after the arrival of cell 3. Clusters 3 and 4 contain the maximum of 5 cells, whilst cluster 5 reaches only size 2 before the processor runs out of cells.

By inspection cluster O_1 grows by 1 (due to the first arrival) and decreases by 0, as the cluster ends immediately after the arrival. The table below (4.4) shows the values for these growths and decreases for all clusters along with the values for U and u. The values for U and u are always one greater than the value for growth or decrease, the need for this can be explained by looking at figure 4.8. In figure 4.8 the cluster O_4 grows now

Table 4.4: Values of U and u for Clusters

	growth by inspection	decrease by inspection	U	u
Cluster O_1	1	0	2	1
Cluster O_2	2	0	3	1
Cluster O_3	4	0	5	1
Cluster O_4	1	1	2	2
Cluster O_5	1	0	2	1

by two as the departure that originally occurred between the first and second arrival cells now happens after the second arrival. Likewise for cluster O_5 the last arrival happens just after a departure allowing the decrease in the cluster to reach one. To deal with the fact that under certain starting values for a cluster the queue size may grow or decrease by one more than seen in the initial cluster sample path all the values of U and u are defined to be one greater than those found by inspection of the individual clusters.

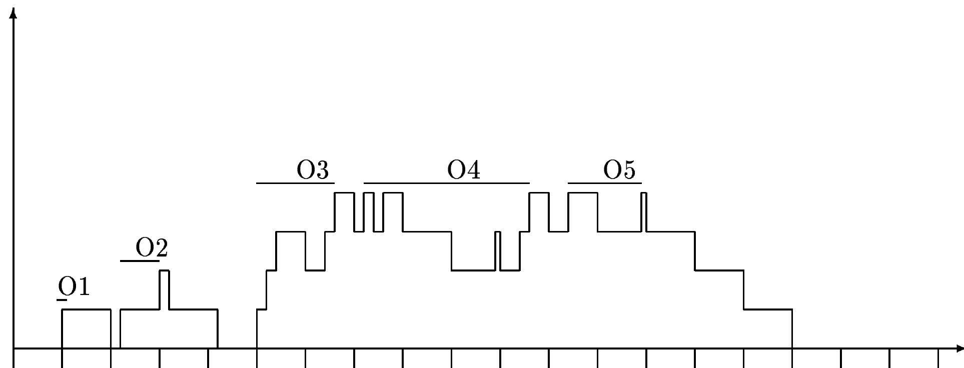

Figure 4.7: An example of Collections of cells

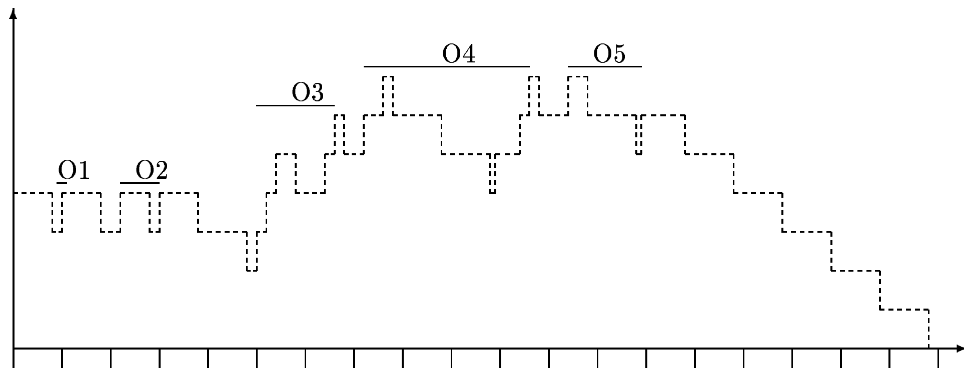

Figure 4.8: The effect of new initial queue size and departure time

Suppose, for example, that $Q = 8$. Then, according to (4.17,4.18), O_1 and O_2 can remain a clusters provided the initial queue sizes satisfy $1 \leq q_n \leq 6$, $1 \leq q_n \leq 5$ respectively. Likewise clusters O_3, O_4 and O_5 will remain clusters provided that $1 \leq q_n \leq 3$, $2 \leq q_n \leq 6$ and $1 \leq q_n \leq 6$.

4.7.3 In the case where c is not constant

For the fixed buffer size algorithm (section 4.7.1) there are no assumptions made as to the service times for cells. However, for the variable buffer size algorithm, constant service times are used to simplify the equations. In the rest of this section a generalised version of the latter algorithm is presented for handling variable service times.

Assuming the same notation as in section 4.7.2, the computation of cell acceptance, σ_n, is unaffected. The number of cells transmitted during the interval (d_n, A_{n+1}), is now dependent on the service times. Thus, δ_n is the largest value satisfying:

$$d_n + \sum_{j=f_n+1}^{f_n+\delta_n} \eta_n \leq A_{n+1}, \tag{4.19}$$

where f_n is the index of the last cell to depart at, or before, d_n, and η_n is the service time of cell n.

Computation of the queue size after cell n arrives (equation 4.12) is unchanged, though the computation of the time of the last departure becomes:

$$d_{n+1} = \begin{cases} d_n + \sum_{j=f_n+1}^{f_n+\delta_n} \eta_n & \text{if } q_{n+1} > 0 \\ A_{n+1} & \text{if } q_{n+1} = 0. \end{cases} \tag{4.20}$$

With the index of the last cell serviced before cell n arrives, f_{n+1} computed as:

$$f_{n+1} = f_n + \delta_n. \tag{4.21}$$

At the end of each iteration the queue size, q_n, time of the last departure, d_n, and the service times for the remaining cells are passed onto the next processor.

The strategies proposed for reducing the computation times may be adapted as follows.

- The total idle time during an interval is now computed as:

$$I_k = \sum_n I(q_n = 0)[A_n - d_{n-1} - \sum_{j=f_{n-1}+1}^{f_{n-1}+\delta_{n-1}} \eta_j]. \tag{4.22}$$

  Denote the sum of the service times passed from processor $k-1$ as E_{k-1}. Then if $E_{k-1} < I_k$ the end state of processor k's interval will be the same as the old state.

- Clusters may be generated as described above. The computation of the queue size at the end of a cluster remains unchanged, with the computation of the time of the last departure becoming:

$$d_{n+t} = d_n + \sum_{j=f_n+1}^{f_n+d_{n,t}} \eta_j, \tag{4.23}$$

  where $\delta_{n,t}$ is computed as the largest value satisfying:

$$d_n + \sum_{j=f_n}^{f_n+\delta_{n,t}} \eta_j \leq A_{n+t}. \tag{4.24}$$

  The index of the last departure after the cluster f_{n+t} is computed as:

$$f_{n+t} = f_n + \delta_{n,t}. \tag{4.25}$$

  With variable service times it is no longer possible to tightly define the maximum and minimum queue sizes during future iterations. A cluster will remain valid only if $q_n + t$ is less than Q and $q_n - \delta_{n,t}$ is greater than 0 where $\delta_{n,t}$ is re-computed before the cluster is checked.

4.8 Implementation Results

The algorithms described in the previous sections were implemented on an Encore Multimax 520 MIMD shared memory 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. If the cell loss fraction was smaller than this the simulation time would be slightly reduced due to the reduction of iterations required to accept the cells. 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 easily. The cell transmission times were constant.

Since the object of this study is to examine the efficiency of the parallel simulation algorithms and the speed-ups 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 4.9 illustrates the performance of algorithm 1 ($B = Q$) for different buffer sizes and 6 input sources. 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

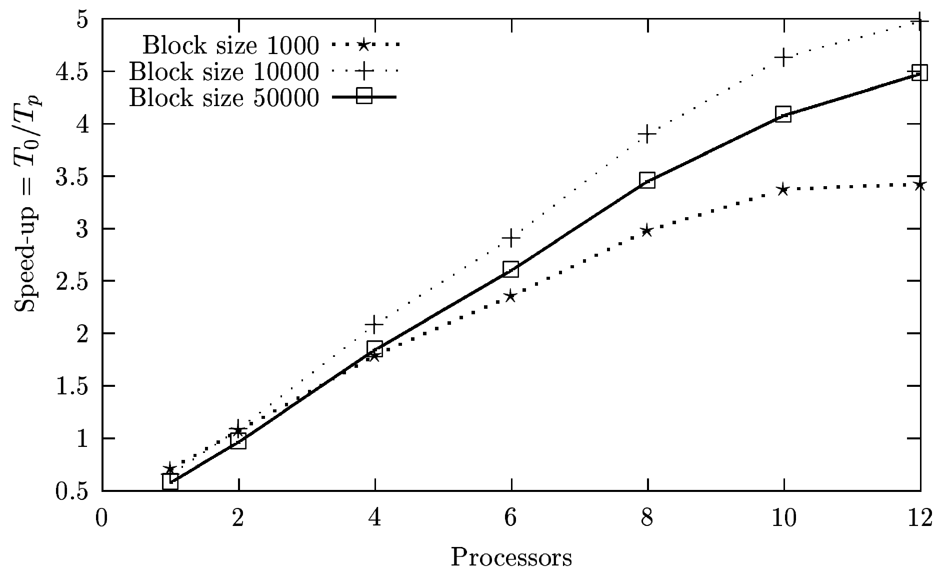

Figure 4.9: Algorithm 1; 6 input sources

counter-intuitive, but can be explained by the increased overheads associated with virtual memory and paging.

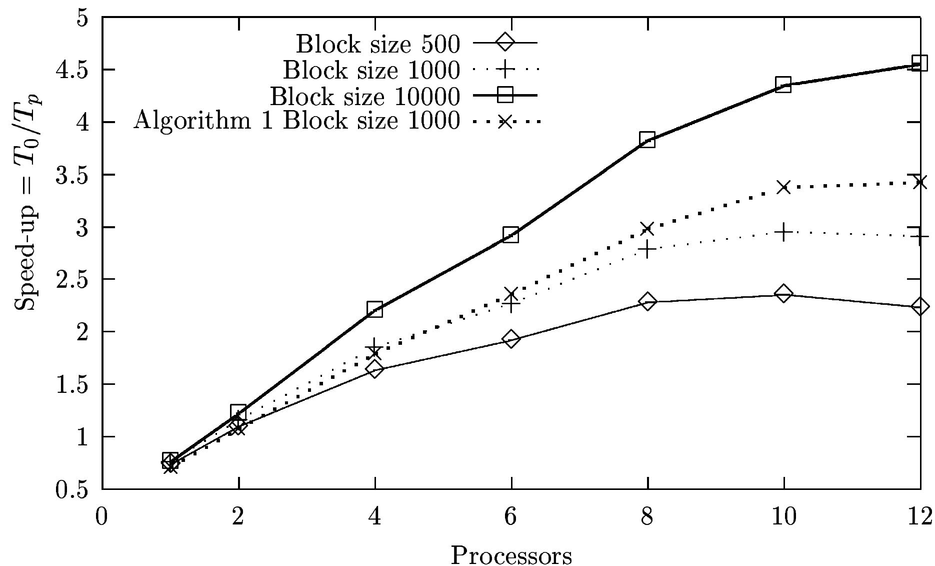

Figure 4.10: Algorithm 2; 6 input sources

Figure 4.10 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.

Increasing the number of input sources to 24 (figures 4.11 and 4.12) 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,

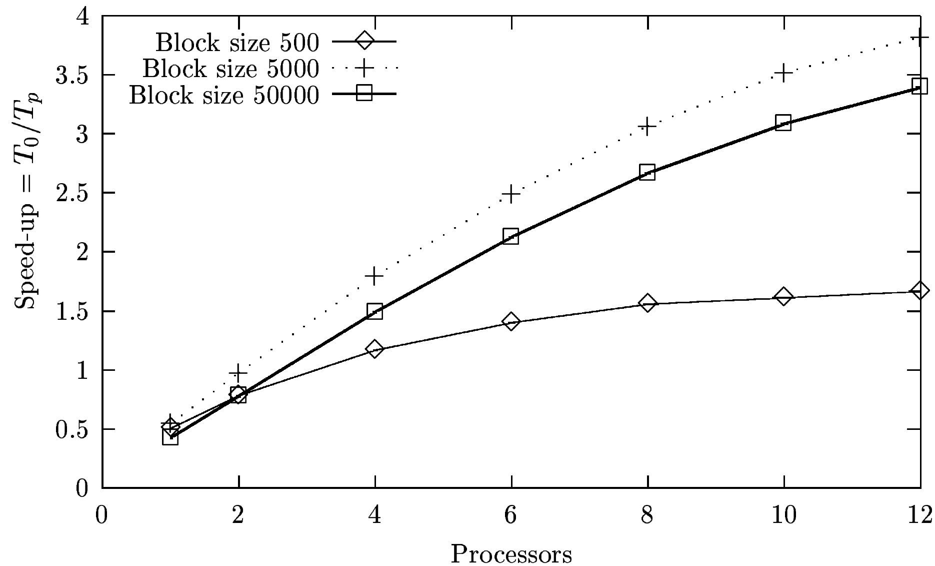

Figure 4.11: Algorithm 1; 24 input sources

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 4.9 and 4.10.

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

The following two figures (4.13 and 4.14) illustrate the effect on the two algorithms of increasing the cell loss ratio. This was done by increasing the cell service time c. A service time of 1 generates approximately 1% cell loss, whilst a service time of 2 causes almost 50% of cells to be lost. It can be seen from figure 4.13 that algorithm 1 suffers badly as the service time is increased. This is due to two factors. The batches in algorithm 1 are replenished sequentially to make up for those cells that are lost. Also the number of iterations required to finalise each batch (see figure 4.14) increases rapidly as the service time increases.

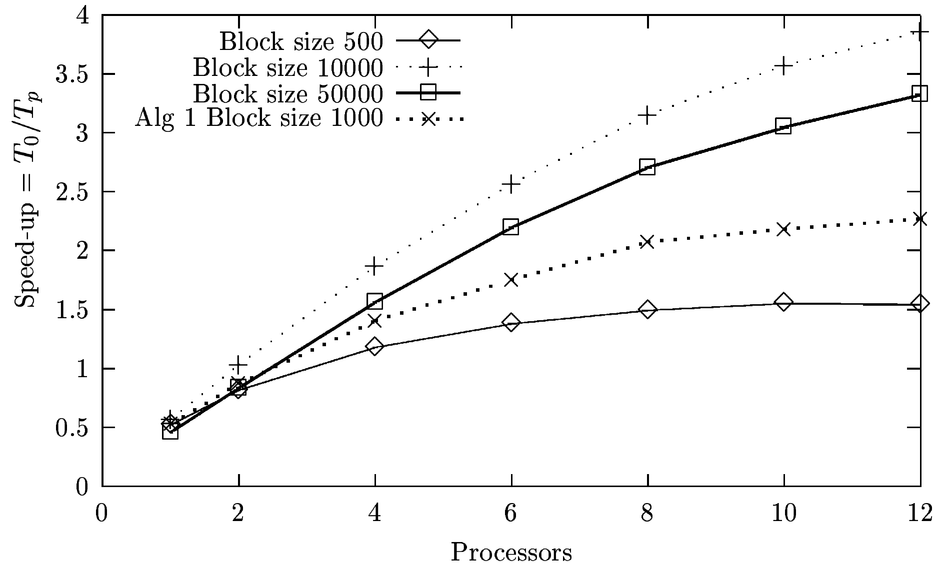

Figure 4.12: Algorithm 2; 24 input sources

For algorithm 2 the execution time for the simulation vary little as the service time is increased. In the case where the batch size B is 1000 the execution time remains almost constant. Figure 4.14 shows that for this batch size the number of iterations is almost maximal for all service times. For a batch size of 50000 figure 4.14 shows that the iterations don't become maximal until the service time reaches 1.2. This can be seen in figure 4.13 where the execution time increases until 1.2.

4.9 Conclusion

The 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, significant speed-ups can be obtained in the presence 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.

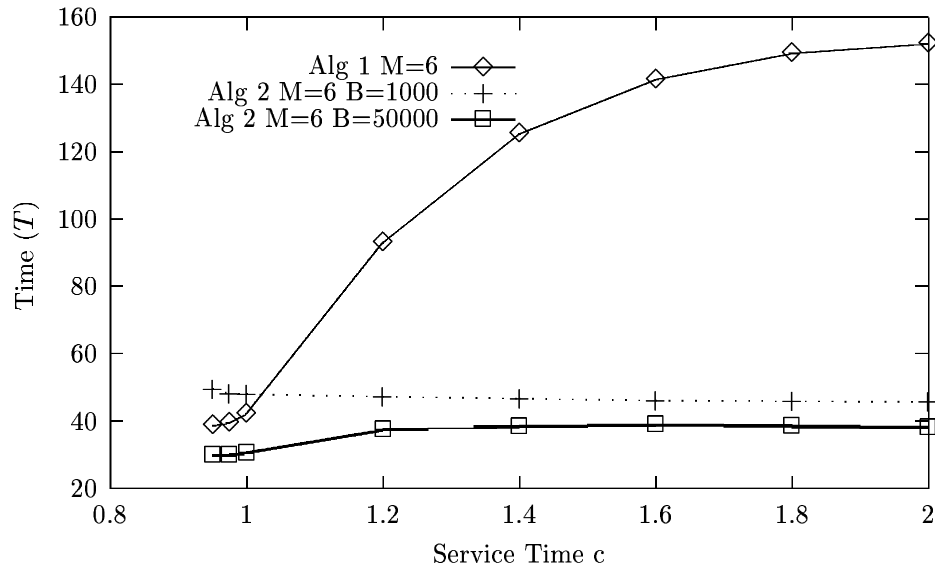

Figure 4.13: The effect of service time (c) on execution time

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 are investigated in the next chapter.

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

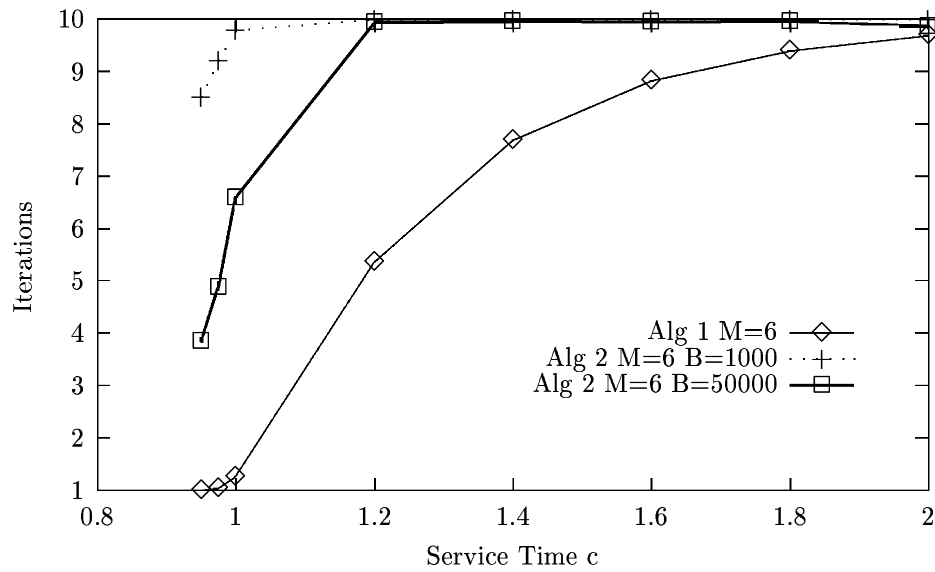

Figure 4.14: The effect of service time (c) on iterations

that allocation can have an important effect on performance. Unless all portions converge quickly to their final states, the advantage of parallel processing can be lost.

Chapter 5

Simulating the ATM Switch on a distributed system

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.

5.1 Introduction

A parallel simulation algorithm for the present model was described in chapter 4 (see also [61]). It used the parallel prefix approach presented in chapter 3, together with parallel merge and relaxation techniques for deciding which cells are lost. That algorithm achieved almost linear speed-up 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 it was 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 chapter. 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.

5.2 Efficiency of the shared memory algorithm on a distributed cluster

The algorithms described in chapter 4, using the variable buffer size technique, were implemented on a distributed cluster of eight PentiumII 233Mhz workstations, connected by fast Ethernet. Various distributed parallel programming paradigms were used to determine if the original algorithms could achieve a similar performance increase. The distributed programming environments used were LAM [66] and MPICH [37, 38] implementations of MPI [19], along with PVM [31].

The results of all implementations failed to achieve speed-up over the sequential implementation. It can be seen from figure 5.1 that the time to generate the arrivals, send and receive the data in the merge, and accept the arrivals are all decreasing as the processor count increases. Unfortunately the time to generate the "on" and "off" periods and find the merge points is increasing. This leads to an overall simulation time which after 4 processors remains approximately constant at 80 seconds in comparison to a sequential version that runs in 50 seconds.

Figure 5.2 indicates where the time is spent during a simulation run on five processors

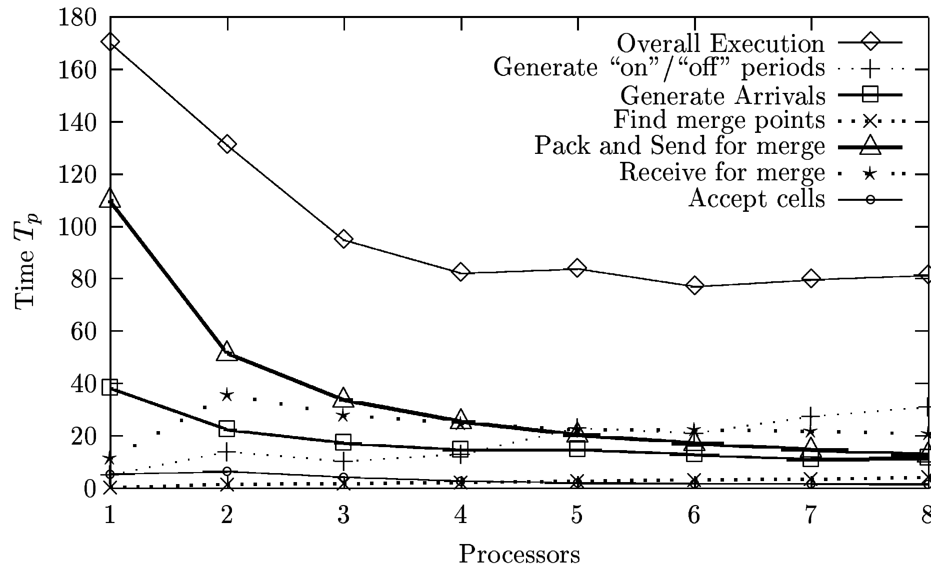

Figure 5.1: Breakdown of execution time

using MPICH. Each processor has a horizontal line representing the flow of time. If the line is covered by a box this indicates that the processor is executing a MPICH function, if the line is not covered then the processor is computing user code. It can be seen from the figure that a large proportion of the execution time is spent within MPICH functions, in fact the only functions illustrated in the figure are those for receiving data (RECV) and all processors exchanging data (ALLGATHER). Thus in the true simulation the time available for computation is less than would appear from the figure. In the figure repeated calls to MPI functions, with small amounts of user code in-between, will appear as a solid block. Figure 5.3 illustrates a small proportion of the whole simulation. From this individual MPI calls can be seen along with the small amount of time available for computation.

A total of 81% of the execution time is spent within MPI function calls with the bulk of the execution time spent processing calls to receive data and exchange values between processors (all gather), each taking approximately 33% of the execution time. These calls are used largely in the computation of merge points for processors and moving arrival

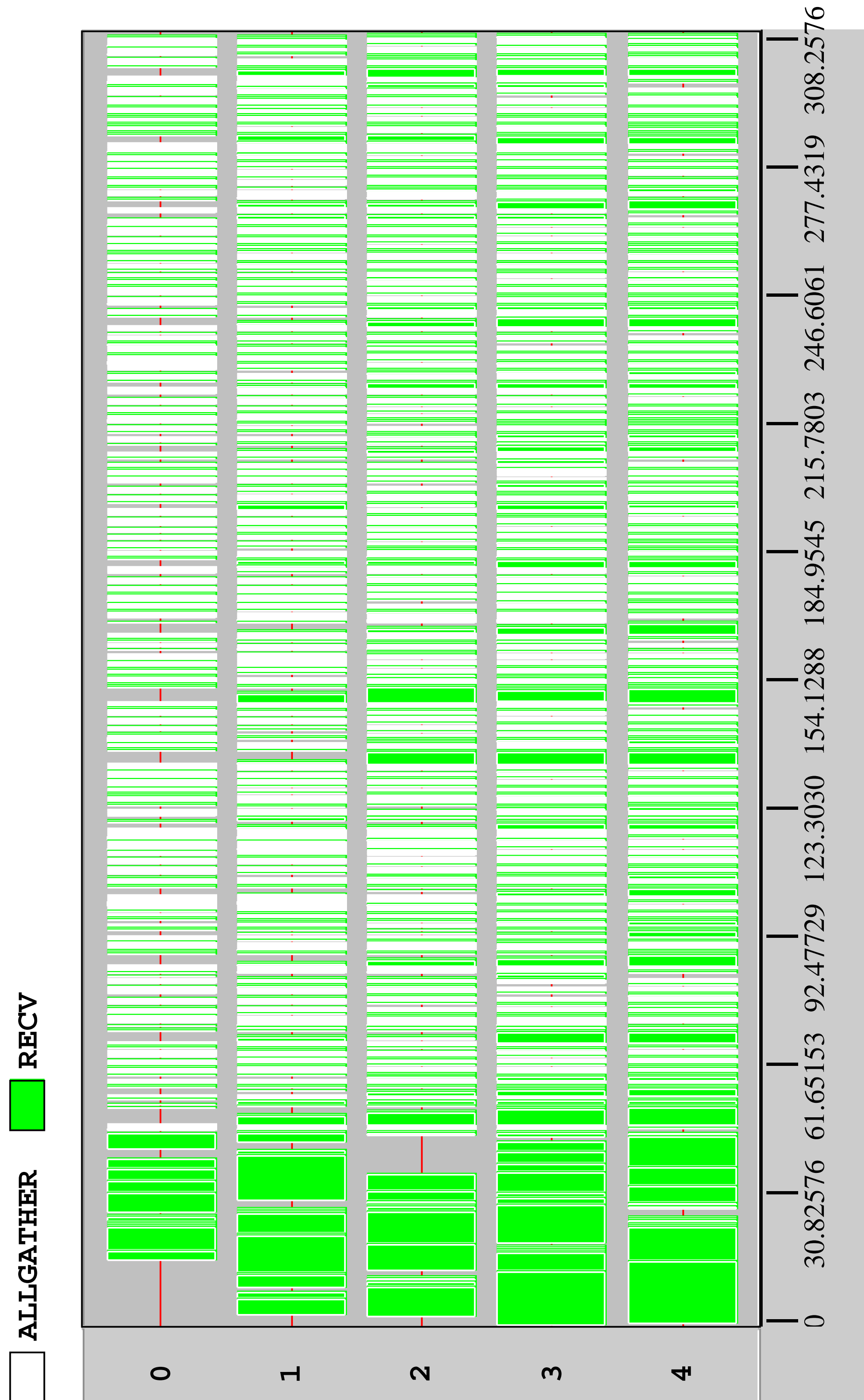

Figure 5.2: Time lines for processors running distributed algorithm

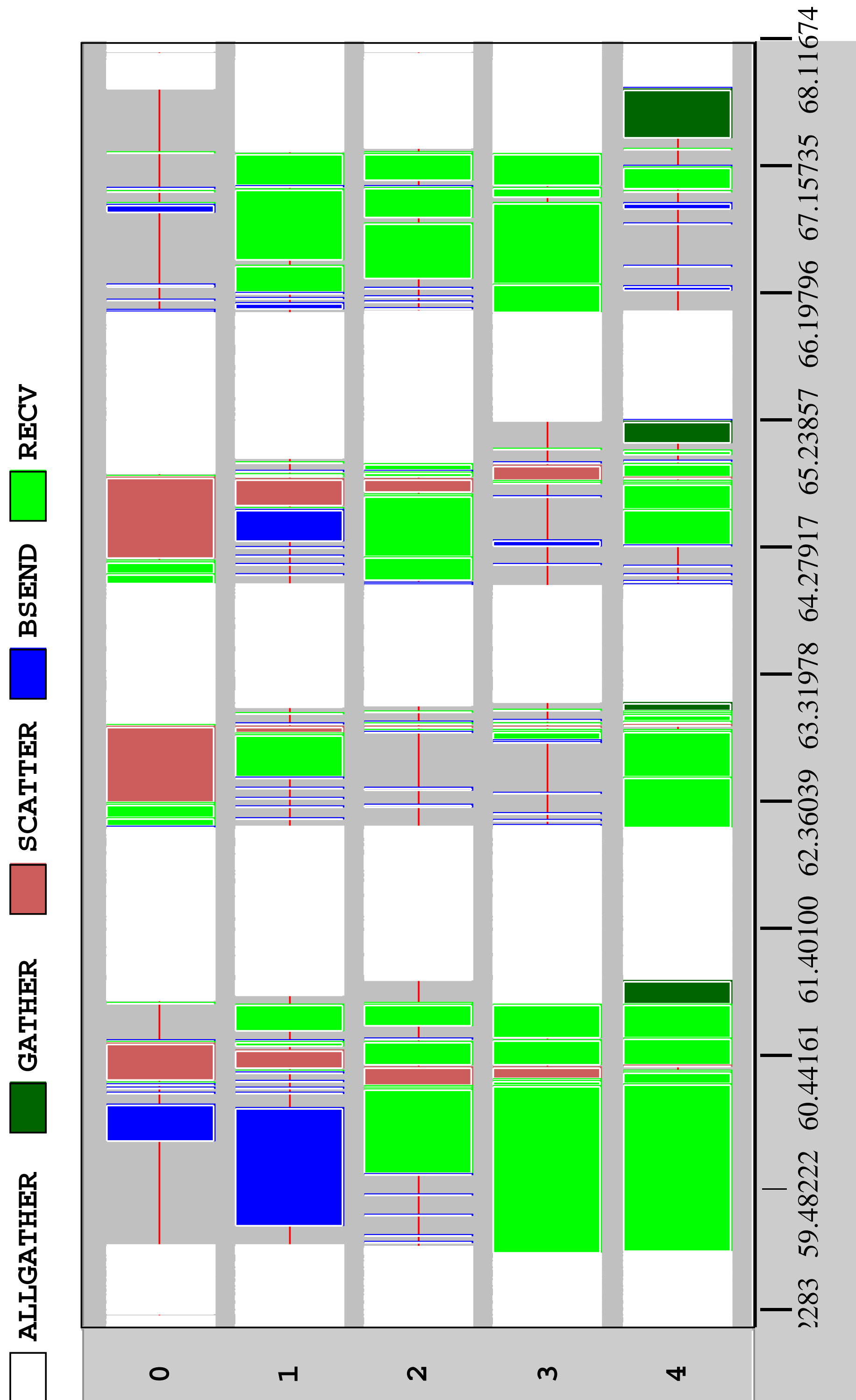

Figure 5.3: An interval of time lines for processors running distributed algorithm

and "on"/"off" periods between processors. A version of the program in which the "on"/ "off" periods were computed independently on each processor was developed. This reduced the overall execution time, though not significantly.

5.3 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.

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:

- **Step 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 5.4 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.

- **Step 2: Mark and remove lost cells.** The algorithm used for this task is an adaptation of an algorithm introduced in section 4.7.2. 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 [14]. Relaxation does not always help, but it can be implemented efficiently in this 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 the continuous time model (arrival instants are real and transmissions can start at arbitrary points) is considered 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.

5.4 Generation of bursty Arrival Sources

The overall approach to generating the bursty arrival sources was presented in section 4.4. In this section the distributed algorithm for performing this is described.

5.4.1 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 (4.2) and (4.3). 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.

- **Step 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:

$$\Xi'_{i,j} = \xi_{i,1} + \xi_{i,2} + ... + \xi_{i,j} \quad j = 1, 2, 3, ... \;; \tag{5.1}$$

$$\Theta'_{i,j} = \eta_{i,1} + \eta_{i,2} + ... + \eta_{i,j} \quad j = 1, 2, 3, ... \;, \tag{5.2}$$

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

$$(k-1)B/P + \Xi'_{i,j} + \Theta'_{i,j} > kB/P \quad i = 1, 2, ..., M. \tag{5.3}$$

  Denote the values of $\Xi'_{i,j}$ and $\Theta'_{i,j}$ which satisfy (5.3) by $\Xi_i^{(k)}$ and $\Theta_i^{(k)}$ respectively.

- **Step 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:

$$\Xi_i = \Xi_i^{(1)} + \Xi_i^{(2)} + ... + \Xi_i^{(k-1)} \quad i = 1, 2, ..., M \;; \tag{5.4}$$

$$\Theta_i = \Theta_i^{(1)} + \Theta_i^{(2)} + ... + \Theta_i^{(k-1)} \quad i = 1, 2, ..., M \;. \tag{5.5}$$

- **Step 3:** The true "on" and "off" starting points are computed as:

$$\Xi_{i,j} = \Xi'_{i,j} + \Xi_i \quad j = 1, 2, 3, ... \quad ; \tag{5.6}$$

$$\Theta_{i,j} = \Theta'_{i,j} + \Theta_i \quad j = 1, 2, 3, ... \quad . \tag{5.7}$$

- **Step 4:** Find the last pair $(\Xi_{i,j}, \Theta_{i,j})$ which satisfies

$$\Xi_{i,j} + \Theta_{i,j} < kB/P. \tag{5.8}$$

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 re-numbers 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.

5.5 Generation of Arrivals

Algorithms are presented in this section 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:

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

The arrival generation algorithm proceeds as follows:

- **Step 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:

$$a'_{i,n+1} = a'_{i,n} + \alpha_{i,n+1} \quad ; \quad n = 1, 2, ... \quad , \tag{5.9}$$

until:

$$s_i^{(k)} + a'_{i,n} > s_i^{(k+1)} \quad . \tag{5.10}$$

Denote the first value $a'_{i,n}$ which satisfies (5.10) by $a_i^{(k)}$.

- **Step 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:

$$a_i = a_i^{(1)} + a_i^{(2)} + ... + a_i^{(k-1)} \quad i = 1, 2, ..., M \quad . \tag{5.11}$$

- **Step 3:** The arrival instances for source i on the 'compressed' time line are computed as:

$$a_{i,n} = a'_{i,n} + a_i \quad n = 1, 2, 3, ... \quad . \tag{5.12}$$

- **Step 4:** Any arrival instances which satisfy:

$$s_i^{k+r} < a_{i,n} < s_i^{(k+r+1)}, \tag{5.13}$$

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

- **Step 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:

$$\Xi_{i,l-1} < a_{i,n} < \Xi_{i,l} \quad . \tag{5.14}$$

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

$$A_{i,n} = a_{i,n} + \Theta_{i,l-1} \quad . \tag{5.15}$$

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.

5.6 Mark and remove lost cells

The algorithm used here is the one presented as algorithm 2 form chapter 4. This algorithm requires little communication between processors. The first algorithm is inappropriate as each processor needs access to an arbitrary list of arrivals and departures.

5.7 Implementation 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 [66], implementation of MPI [19], 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 occurred in all cases. For simplicity, the cell inter-arrival times, the "on" and

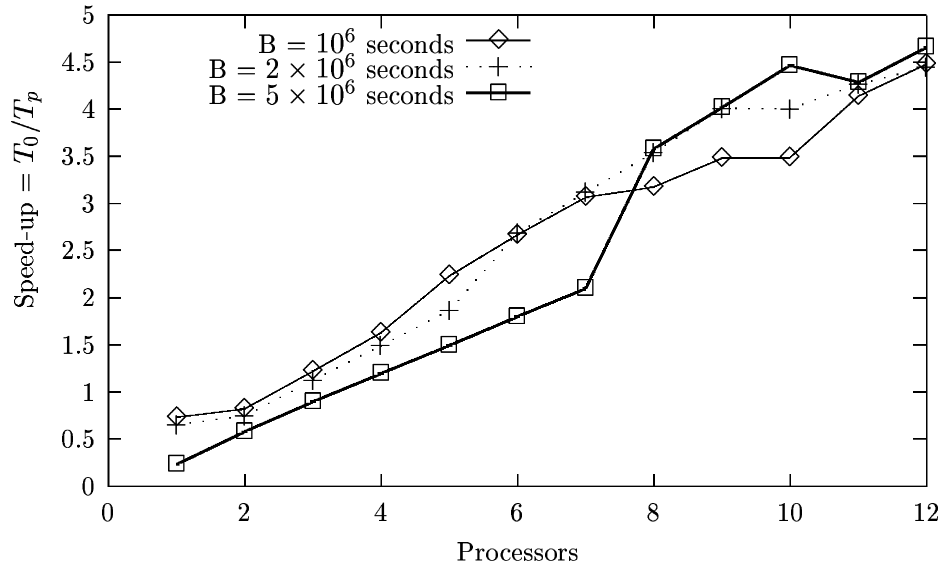

Figure 5.4: Results from a cluster with 8 input sources

"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 speed-ups that can be achieved, the only metric plotted is the ratio T_0/T_p, where T_0 is the execution time of the best *sequential* simulation run on a single processor and T_p is the execution time of the parallel simulation run on p processors. This ratio is commonly known as the 'speed-up' 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. Statistics about the lost cells were collected, but neither the point estimates nor the confidence intervals are displayed in the following graphs.

Figures 5.4 and 5.5 illustrate the speed-up achieved as a function of the number of processors. Each figure shows the results for running the simulation with different interval sizes B. Figure 5.4 illustrates the situation with eight input sources, showing almost linear speed-up for all interval lengths. Altering the interval length appears to have little effect on the overall speed-up of the simulation. For very large intervals there

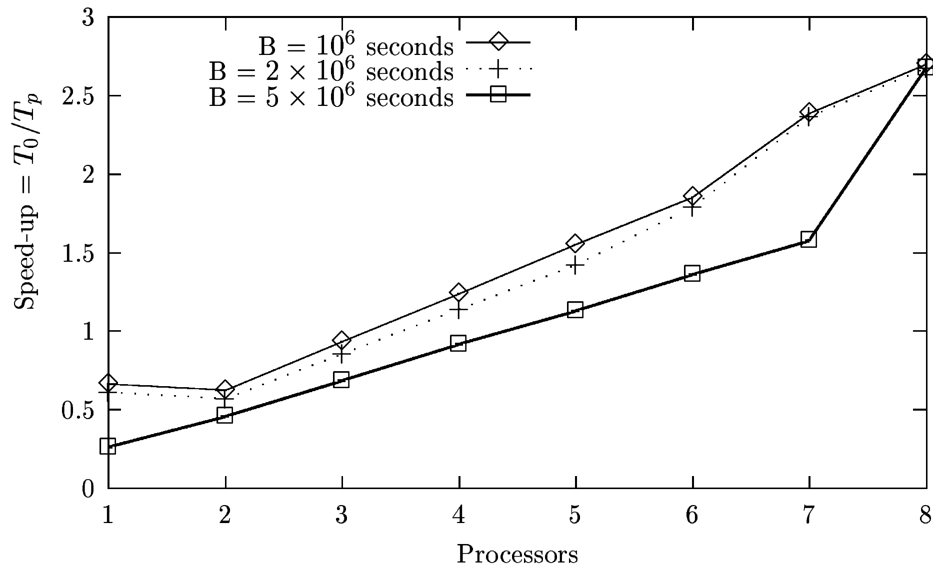

Figure 5.5: Results from a cluster with 24 input sources

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 speed-up remains as before.

Figure 5.5 shows similar results for the case of 24 bursty input sources. Here again an almost linear speed-up is observed. The larger jump between seven and eight processors for $B = 5 \times 10^6$ is present here too, and probably has the same explanation.

Experiments were also carried out into the efficiency of using clusters of cells when computing cell acceptances. Figure 5.6 illustrates the effect of performing simulations with and without clusters. It can be seen from this figure that the speed-up achieved for almost all processor counts is better than for a simulation without clusters. This

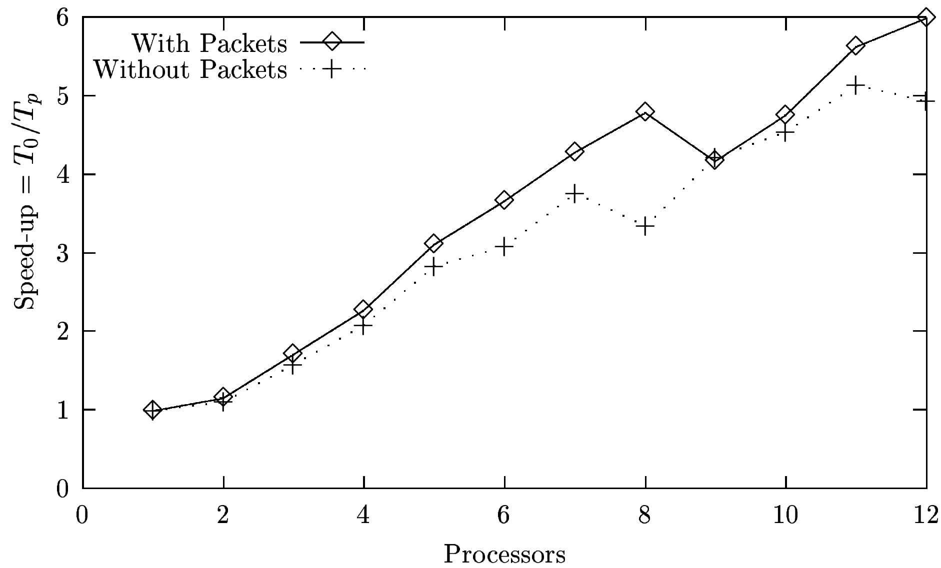

Figure 5.6: The effect of using clusters on speed-up

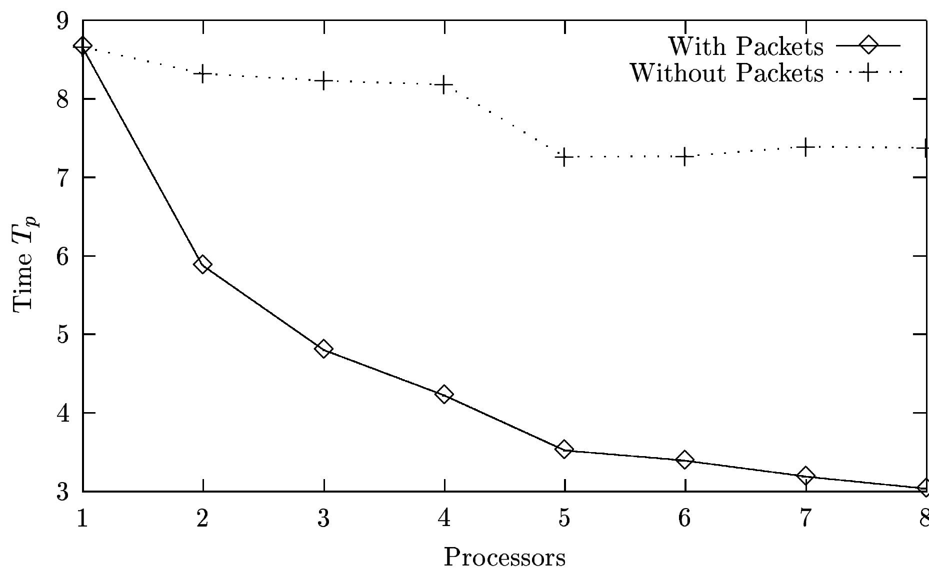

Figure 5.7: The effect of using clusters on speed-up, no overlapping

figure does not show the full improvement due to interleaving of work between processors. Figure 5.7 shows the amount of time required on each processor to perform cell acceptance. Both algorithms start with the same time for one processor, as clusters are

not used in the single processor case. Thereafter the cluster algorithm quickly achieves a faster execution time, with the non-cluster algorithm's execution time reducing slowly as the processor count increases.

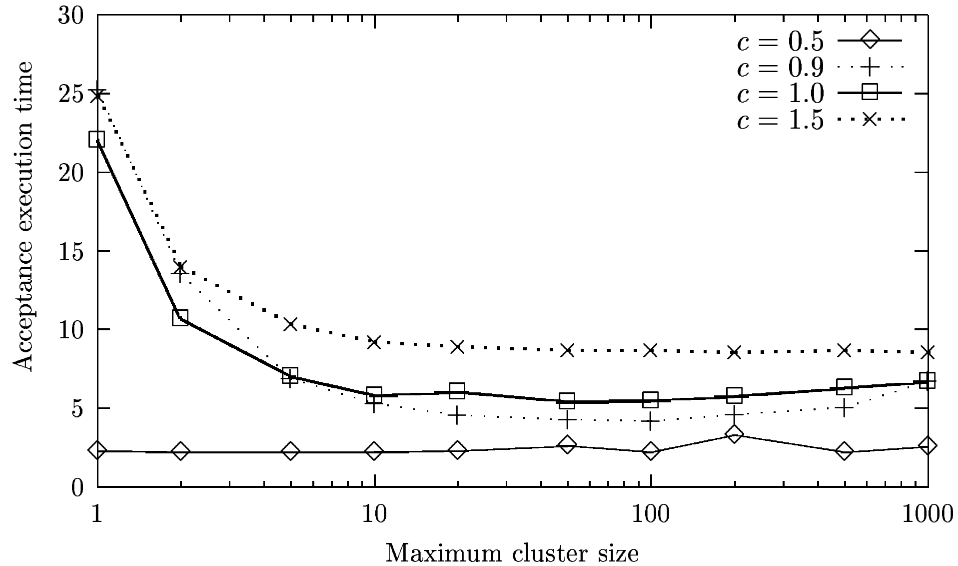

Figure 5.8: Amount of time spent computing accepted cells using clusters

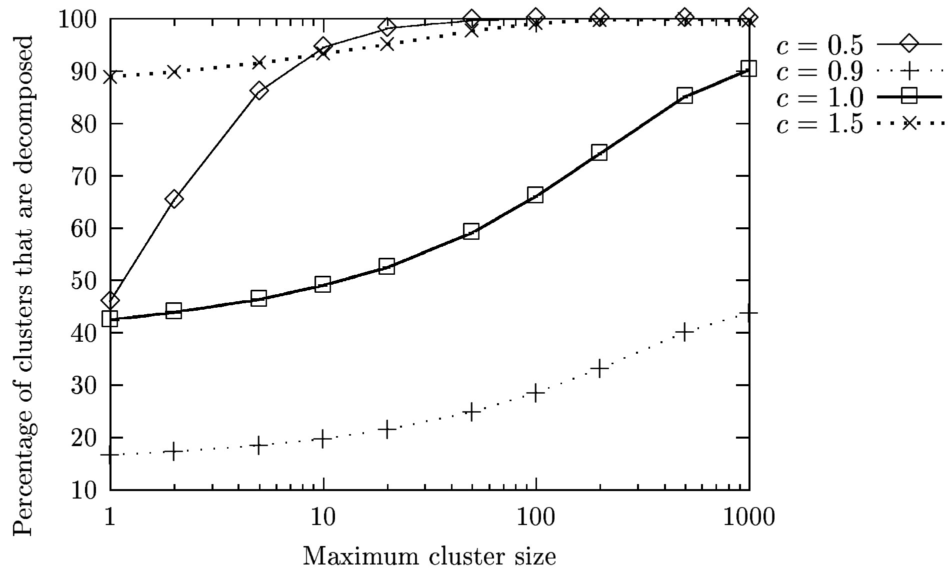

Figure 5.9: Percentage of clusters that are decomposed

Varying the cluster sizes altered the execution time as illustrated in figure 5.8. The efficiency of the different cluster sizes will depend on the arrival and departure properties of the simulation. In the figure different service times are illustrated. For each of these service times there is an optimal cluster size for which the execution time is minimal. When the service time is 0.5 then the server buffer empties quicker than it fills. In general this will make the end state of each interval independent of the starting state. As only a small portion of the interval is ever re-computed, and clusters are small due to the queue becoming empty, the use of clusters in this case is inefficient.

These small cluster sizes can be seen from figure 5.11 in which the average cluster size is plotted against the maximum cluster size. In most cases the cluster size is the maximum size that it can be. However, for $c = 0.5$ the queue becomes empty quickly reducing the average queue size to approximately 3.

Figure 5.9 can be used to help explain the execution times for the simulations. For each service time a line is drawn for the percentage of clusters that need to be decomposed due to potential losses or the queue size becoming zero. As the cluster size increases, the percentage of clusters that are decomposed increases. However, as the cluster size is increasing the amount of work performed (due to the larger cluster size) during each iteration is decreasing. Therefore an optimal cluster size can be found by balancing these two effects.

Figure 5.10 illustrates the proportion of simulation time spent in MPICH calls when eight processors are in use. It can be seen from this figure that far less time is spent performing MPI function calls than in the implementation in section 5.2. It can be seen from figure 5.12 that the percentage of execution time spent in MPICH calls is around 30% to 40% with a small increase as the processor count increases.

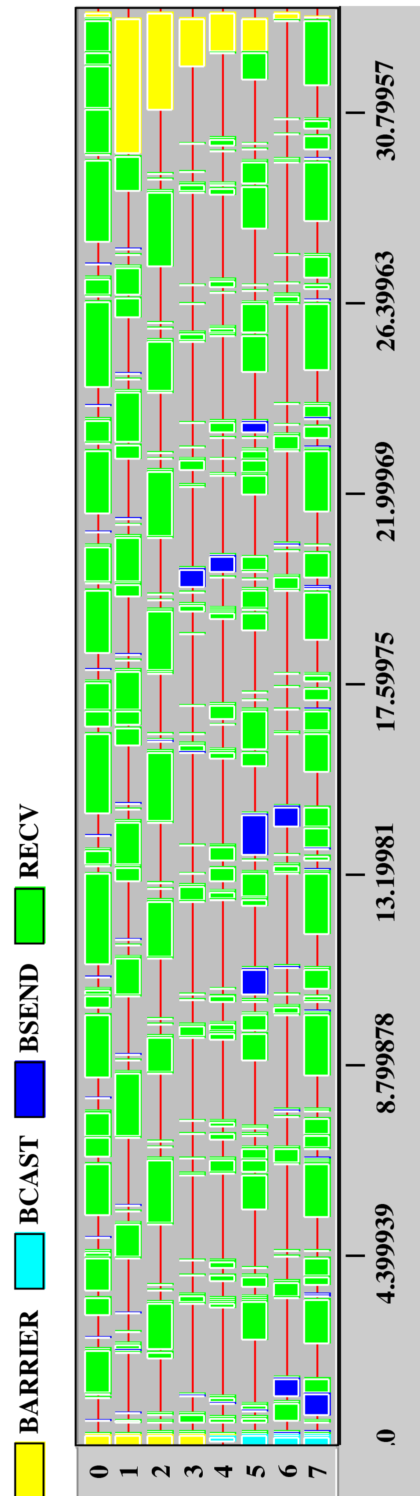

Figure 5.10: An efficient simulation time line

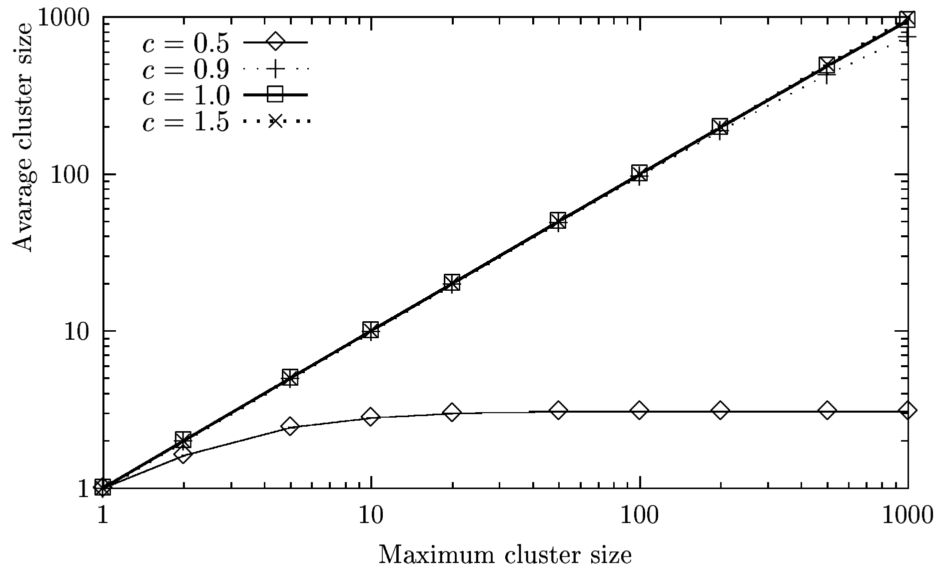

Figure 5.11: The average size of a cluster

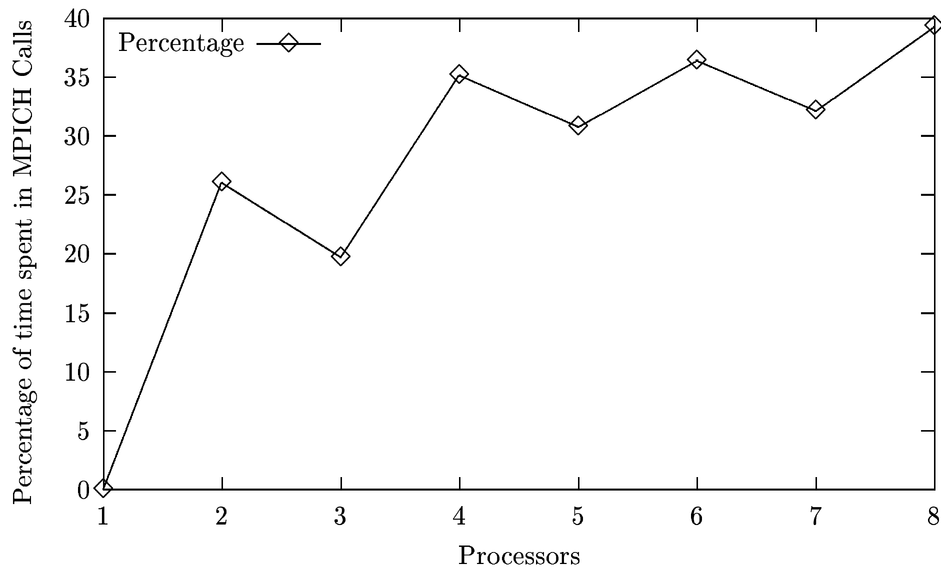

Figure 5.12: The percentage of time spent in MPICH Calls

5.8 Conclusion

It has been shown that a large sample path for a non-trivial communication system can be simulated in parallel on a distributed cluster of processors. Moreover, the paralleliza-

tion is efficient, in the sense that a linear speed-up 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. Almost all of the overheads have been eliminated 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.

Relaxation works best in situations where the number of iterations required to settle on the true values is small, in comparison to the number of processors. This is achieved in the case of the algorithms presented here. The acceptance algorithm does not always settle to the true values in a small number of iterations. However, as the proportion of the execution time spent accepting cells is small, and the fact that each iteration can be computed quickly due to the use of clusters, efficient execution times are maintained.

Chapter 6

Simulating a Sliding Window Protocol

Two parallel algorithms for simulating a sliding window communication protocol are described. The first exploits the parallel generation of interarrival, service and transfer times, whilst the second is based on parallel prefix computations. Both algorithms allow the processors to work largely asynchronously with each other. The efficiency of the two approaches are evaluated experimentally, using an implementation on a cluster of 12 processors connected by fast Ethernet.

This chapter demonstrates that the ideas of recurrence relations and parallel prefix presented in the previous chapters can successfully be applied to other systems than the ATM switch. The use of relaxation, although not used in the work presented here, could be used to implement a more general form of the simulation.

6.1 Introduction

All variants of the TCP communication protocol use some form of sliding window to regulate the flow of data. New packets arriving at the sending node are queued until the acknowledgements for certain previous packets have been received. The size of the

sliding window, i.e. the total number of packets and acknowledgements in transit, in service at the source, and waiting or in service at the destination, is controlled. When the arrival rate for acknowledgements falls below a given threshold, the window size is reduced sharply, and is then allowed to grow back. However, these occurrences are usually rare compared to the processing times of packets.

The basic model consists of two nodes–a source and a destination—which communicate across an arbitrary network by means of a sliding window protocol. Two simulation algorithms are described. Both are time-parallel, in the sense that they assign different processors to the computation of different portions of the sample path. Moreover, both use the parallel prefix algorithm (e.g., see chapter 3) to generate the sequence of external arrivals at node 1. The difference between the algorithms is in the way they handle the sending and receiving of packets and acknowledgements.

The first approach is very simple-minded and quite general. Beyond the sequence of external arrival instants, the only parallelism it exploits is in the generation of the random service times at the two nodes and transfer times of packets and acknowledgements in the network. The putting together of the different portions of the sample path is done sequentially.

The second method works by reducing the entire simulation task to the solution of sets of recurrence relations involving only the operations max and $+$. That solution is then expressed in terms of partial matrix products in the $(\max, +)$ algebra, which again allows the application of the parallel prefix algorithm. This is an extension of the work presented in section 3.1. This approach is more restrictive: it constrains us to assume that the queue at node 2 is unbounded and that the window size is fixed (possible generalisations are discussed in the conclusion).

Empirical comparisons of the two algorithms reveal that the first is more efficient when the number of processors employed is small, but its speed-up reduces when that number increases. The second algorithm, while requiring each processor to perform more

computations, has a potentially unlimited degree of parallelism; it displays a fully linear speed-up which, given a sufficiently large number of processors, will clearly overtake that of the first algorithm.

The model and the notation are introduced in section 6.3. The two parallel simulation algorithms are described in section 6.4, while section 6.5 presents the results of several experiments.

6.2 Related Work

Under general assumptions about interarrival, service and transfer times, the performance characteristics of such systems cannot be determined analytically. Some efforts in that direction have been made by making considerable simplifications (see, for example, Padhye et al. [69, 68], where the input queue is assumed always full and the round-trip time and the error rate are fixed and independent of congestion). It is, of course, always possible to estimate those characteristics by simulation. Indeed, there are simulation packages designed precisely for that purpose (e.g., ns, described in [18]). However, obtaining long-term estimates by means of sequential simulations at the packet level tends to be a very expensive and time-consuming task.

Attempts are made to speed up these simulations by performing them in parallel on several processors. This is a non-trivial undertaking, since most events associated with consecutive packets are dependent on each other. Thus, the standard discrete-event simulation algorithms do not lend themselves to efficient parallelization. Special purpose algorithms are required.

6.3 The Model

A stream of packets arrive externally into node 1 and join an unbounded FIFO queue (see figure 6.1). The consecutive interarrival intervals are i.i.d. random variables with

some arbitrary distribution function, $A(x)$. Consecutive service times at nodes 1 and 2 are i.i.d. random variables with arbitrary distribution functions, $S_1(x)$ and $S_2(x)$, respectively. Transfer times between node 1 and node 2 are i.i.d. random variables with distribution function, $T_1(x)$. When a packet has been processed at node 2, an acknowledgement is sent back to node 1. Transfer times between node 2 and node 1 are i.i.d. random variables with distribution function, $T_2(x)$. All transfer times are independent of the number of packets and/or acknowledgements that are in transit.

Numbered instances of the above random variables will be denoted by the corresponding lower case letter, plus an index. Thus, the nth external interarrival interval is denoted by a_n; the nth service time at node 1 is denoted by $s_{1,n}$, etc.

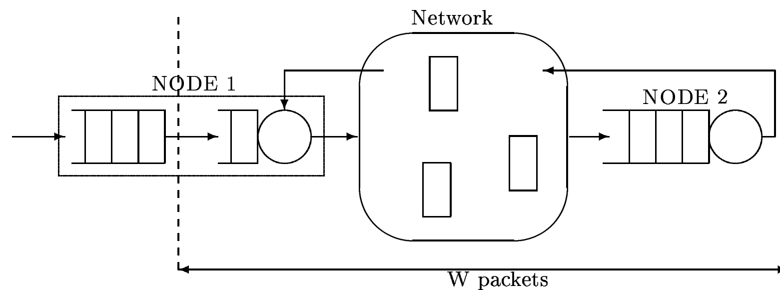

Figure 6.1: The Sliding window protocol

A packet which is in service at node 1 (i.e., being processed for sending), in transit between node 1 and node 2, waiting or in service at node 2, or one whose acknowledgement is in transit from node 2 to node 1, is said to be 'in the sliding window'. As soon as an acknowledgement for a packet arrives back at node 1, that packet is deemed to have departed from the window (i.e., the processing of acknowledgements is instantaneous). Thus, the number of packets in the window increases by 1 when a packet starts service at node 1; that number decreases by 1 when an acknowledgement is received at node 1.

Suppose that the size of the window is W. Consider the arrival of a packet, say packet n, into the system. If the external queue is empty at that moment, and server 1 is idle, and packet $n - W$ has already departed, then packet n enters the window immediately and starts service at node 1. Otherwise, it joins the external queue. When packet n

eventually reaches the head of that queue, it enters the window at the next instant such that server 1 has completed a service and packet $n - W$ has left.

The queue at node 2 is ordered according to packet indices, rather than times of arrival. Packets may overtake each other while in transit; e.g., packet $n - 1$ may arrive at node 2 after packet n, but will be served before it. Moreover, if packet $n + 1$ is not present when the service of packet n is completed, server 2 will wait, even though there may be other packets in the queue. For each packet that completes service at node 2, an acknowledgement is sent back to node 1.

The queue size at node 2 may be bounded, in which case incoming packets which find it full are lost. Packets may also be lost in transit in the network. In both cases, a loss results in a lack of acknowledgement, which causes a timeout at node 1 and a retransmission. Network losses, if they are independent of the numbers of packets in queues and in transit, can be incorporated in the original model assumptions by modifying appropriately the distribution function of transfer times, $T_1(x)$ and $T_2(x)$. Losses due to buffer overflow are more difficult, and are handled only by the first of the two simulation algorithms presented in the next section.

Different TCP protocols employ different strategies for the dynamic control of the window size, W. Again, the first algorithm is able to simulate changes in W; the second assumes that W is fixed.

6.4 Parallel simulation algorithms

A sample path for the system described above is determined by the following sequences of events: external arrivals of packets into node 1; service completions at node 1; arrivals of packets into node 2; service completions at node 2; arrivals of acknowledgements into node 1; changes in the value of W. Hence, the principal task of the simulation program is to compute the instants when those events occur. They provide sufficient information to determine the performance measures of interest.

We shall use the following notation:

A_n is the arrival time of packet n into node 1;

B_n is the service completion time of packet n at node 1;

C_n is the arrival time of packet n into node 2;

D_n is the service completion time of packet n at node 2;

G_n is the arrival time of the acknowledgement for packet n into node 1;

These sequences satisfy certain recurrence relations. Thus:

$$A_n = A_{n-1} + a_n \ ; \ \ n = 1, 2, \dots \ , \tag{6.1}$$

where a_n is the inter-arrival interval between packets $n-1$ and n, and $A_0 = 0$ by definition.

Since packet n starts service at node 1 either immediately upon arrival, or when packet $n-1$ completes its service, or when the acknowledgement for packet $n-W$ arrives, the service completion time B_n is computed as:

$$B_n = \max(A_n, B_{n-1}, G_{n-W}) + s_{1,n} \ ; \ \ n = 1, 2, \dots \ , \tag{6.2}$$

where $s_{1,n}$ is the required service time for packet n at node 1; B_0, and G_i when $i \leq 0$, are defined to be 0; W is the current value of the window size.

Similarly, since packet n starts service at node 2 either upon arrival, or when packet $n-1$ completes:

$$D_n = \max(C_n, D_{n-1}) + s_{2,n} \ ; \ \ n = 1, 2, \dots \ , \tag{6.3}$$

where $s_{2,n}$ is the required service time for packet n at node 2; $D_0 = 0$ by definition.

If node 2 does not reject packets due to buffer overflows, then:

$$C_n = B_n + t_{1,n} \ ; \ \ n = 1, 2, \dots \ , \tag{6.4}$$

where $t_{1,n}$ is the transit time for packet n from node 1 to node 2. In the presence of network losses, that interval may include timeouts and retransmissions.

If node 2 rejects incoming packets when its queue reaches a given size, then timeouts and further instances of $t_{1,n}$ may have to be added to B_n in order to obtain C_n. Note that the queue at node 2 at any given time, t, consists of those packets which arrived before t and will complete service after t. Hence, the current queue size is equal to the number of packet indices, i, such that:

$$C_i < t < D_i \ . \tag{6.5}$$

In evaluating the queue size seen at node 2 by packet n, it is only necessary to check (6.5) for i satisfying $|n - i| < W$. Moreover, if a packet with an index higher than n has arrived at node 2 before packet n, then it is certain to be in the queue; its service completion time is not needed.

The arrival instants for acknowledgements are given by:

$$G_n = D_n + t_{2,n} \ ; \ \ n = 1, 2, \dots \ , \tag{6.6}$$

where $t_{2,n}$ is the transit time for packet n's acknowledgement from node 2 to node 1. Again, in the presence of network losses, that interval may include node 1 timeouts and retransmissions.

Various performance characteristics can be derived from the above sequences. Evaluation of queue sizes has already been mentioned. That can also be used to estimate the probability that a queue size will exceed a given threshold. The response time of packet n is of course equal to $G_n - A_n$. The packet throughput is obtained by monitoring the

number of G_n instants per unit time.

The problem now is to parallelise the computation of A_n, B_n, C_n, D_n and G_n by means of the above relations.

This is easily done in the case of the arrival sequence. The solution of (6.1) is given by:

$$A_n = \sum_{i=1}^{n} a_i \ ; \ \ n = 1, 2, \dots \ . \qquad (6.7)$$

Since the a_is are i.i.d. instances of a random variable with a given distribution, they can be generated in parallel on different processors. The computation of the partial sums (6.7) is a special case of the well-known 'parallel prefix' problem, whose solution is described in the literature (see section 3.1). In the present case, if N consecutive arrival instants are to be computed, and there are P processors available, then each processor produces a subset of N/P consecutive arrivals. The algorithm achieves linear speed-up, in the sense that its run time is on the order of $O(N/P)$.

The rest of the sample path is more difficult to compute in parallel. The general idea is to divide the simulation into equal-sized batches of arriving packets, as discussed in section 3.2. Within each batch, the different processors are assigned approximately equal portions of the sample path.

Two approaches towards the implementation of that idea are described below. The presentation concentrates on the simulation of the first batch. All subsequent batches are handled in the same manner.

6.4.1 Algorithm 1

The first step is to solve (6.1) by applying the parallel prefix algorithm: processor k computes the kth portion of the external arrival instants, A_n, in the current batch.

For each packet in its portion, processor k generates and stores instances of the random variables $s_{1,n}$, $t_{1,n}$, $s_{2,n}$ and $t_{2,n}$, from the appropriate distributions. Since those instances are independent of each other, their generation can be performed in parallel

with other processors.

Processor 1 computes the sequences B_n, C_n, D_n and G_n for the first portion of packets, using relations (6.2), (6.4), (6.3) and (6.6). It then passes the final state of the first portion's sample path, i.e. the last elements of the B_n, C_n and D_n sequences, and the last W elements of the G_n sequence, to processor 2. These are the initial conditions which are used by processor 2 to compute those sequences for the second portion of packets. This process continues until processor P completes the sample path for the current batch. The final state of the sample path in this batch becomes the initial state for the next one.

Thus, algorithm 1 is only partially parallel. Important parts of the sample path are computed sequentially, causing processor k to remain idle whilst waiting for data from processor $k-1$. On the other hand, because each processor is given the correct initial conditions when it starts to compute its portion of the sample path, this algorithm can cope with state-dependent decisions such as rejecting packets at node 2 when the buffer is full, or changing the value of W according to traffic conditions and protocol strategy.

The utilisation of the processors implementing this algorithm can be improved by overlapping their work on different batches. Thus, when processor k has finished computing its portion of the sample path for the current batch, and has passed the final state to processor $k+1$, it can start generating the arrival times, service times and transfer times for the next batch. There may still be waiting for initial conditions from processor $k-1$, but the simulation will progress faster.

6.4.2 Algorithm 2

Suppose that the window size is fixed, and the queue at node 2 is unbounded (i.e., incoming packets are never rejected). Then a simpler set of recurrence relations is obtained by substituting (6.6) and (6.4) into (6.2) and (6.3), respectively. The time for completion

of service at node 1 now becomes:

$$B_n = \max(A_n, B_{n-1}, D_{n-W} + t_{2,n-W}) + s_{1,n} \ ; \ ; n = 1, 2, \dots , \tag{6.8}$$

and:

$$D_n = \max(B_n + t_{1,n}, D_{n-1}) + s_{2,n} \ ; \ n = 1, 2, \dots . \tag{6.9}$$

Remember that A_n, $s_{1,n}$, $s_{2,n}$, $t_{1,n}$ and $t_{2,n}$ are known constants for the purpose of these relations.

The above equations involve the operations max and +. With a suitable renaming of those operations, (6.8) and (6.9) can be transformed into a single equation in terms of a matrix multiplying a vector (that technique has been used in other contexts, e.g. Baccelli and Canales [5] and Greenberg et al. [35]). The use of matrix products in the (max, +) algebra is discussed in section 3.1.

Define the following column vector, V_n, containing the service completion times of packet n at nodes 1 and 2, together with the node 2 service completion times of the previous $W - 1$ packets:

$$V_n = \begin{bmatrix} B_n \\ D_n \\ D_{n-1} \\ \vdots \\ D_{n+1-W} \\ 0 \end{bmatrix} .$$

Define matrix $M_{1,n}$, of size $W + 2$:

$$M_{1,n} = \begin{bmatrix} s_{1,n} & -\infty & \dots & \dots & -\infty & t_{2,n-W} + s_{1,n} & A_n + s_{1,n} \\ -\infty & 0 & -\infty & \dots & -\infty & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty & \dots & \dots & -\infty \\ -\infty & \dots & -\infty & 0 & -\infty & \dots & -\infty \\ \vdots & \vdots & & & \ddots & \vdots & \vdots \\ -\infty & \dots & \dots & \dots & -\infty & 0 & -\infty \\ -\infty & \dots & \dots & \dots & \dots & -\infty & 0 \end{bmatrix}, \tag{6.10}$$

as the effect of node 1 on the vector V_n. The effect of node 2 on V_n is the matrix $M_{2,n}$ (size $W + 2$):

$$M_{2,n} = \begin{bmatrix} 0 & -\infty & \dots & \dots & \dots & \dots & -\infty \\ t_{1,n} + s_{2,n} & s_{2,n} & -\infty & \dots & \dots & \dots & -\infty \\ -\infty & 0 & -\infty & \dots & \dots & \dots & -\infty \\ -\infty & -\infty & 0 & -\infty & \dots & \dots & -\infty \\ \vdots & \vdots & & \ddots & & \vdots & \vdots \\ -\infty & \dots & \dots & -\infty & 0 & -\infty & -\infty \\ -\infty & \dots & \dots & \dots & \dots & -\infty & 0 \end{bmatrix}. \tag{6.11}$$

Thus:

$$V_n = M_{2,n} \odot M_{1,n} \odot V_{n-1}, \tag{6.12}$$

where matrix-matrix and matrix-vector multiplication is in the $(\max, +)$ algebra described in section 3.1, and V_0 is the initial vector of service completion instants for the current batch.

The matrices $M_{1,n}$ and $M_{2,n}$ can be combined in the $(\max, +)$ algebra to give matrix

M_n:

$$M_n = M_{2,n} \odot M_{1,n} = \begin{bmatrix} s_{1,n} & -\infty & \dots & \dots & -\infty & t_{2,n-W} + s_{1,n} & A_n + s_{1,n} \\ x_n & s_{2,n} & -\infty & \dots & -\infty & y_n & z_n \\ -\infty & 0 & -\infty & \dots & \dots & \dots & -\infty \\ -\infty & -\infty & 0 & -\infty & \dots & \dots & -\infty \\ \vdots & \vdots & & \ddots & & \vdots & \vdots \\ -\infty & \dots & \dots & -\infty & 0 & -\infty & -\infty \\ -\infty & \dots & \dots & \dots & \dots & -\infty & 0 \end{bmatrix}, \tag{6.13}$$

where x_n, y_n and z_n, are given by:

$$x_n = s_{2,n} + t_{1,n} + s_{1,n} \ ,$$

$$y_n = s_{2,n} + t_{1,n} + t_{2,n-W} + s_{1,n} \ ,$$

$$z_n = A_n + s_{1,n} + s_{2,n} + t_{1,n} \ .$$

With these definitions, it is not difficult to verify that equations (6.8) and (6.9) are equivalent to:

$$V_n = M_n \odot V_{n-1} \ ; \ \ n = 1, 2, \dots \ . \tag{6.14}$$

The solution of (6.14) is:

$$V_n = M_n \odot M_{n-1} \odot \dots \odot M_1 \odot V_0 \ ; \ \ n = 1, 2, \dots \ , \tag{6.15}$$

with the above definition of V_0.

Thus, the parallel computation of the sample path is reduced to that of the partial matrix products in (6.15). Since matrix multiplication in the $(\max, +)$ algebra is associative, this is again a special case of the parallel prefix problem. It can therefore be

solved by means of the existing parallel prefix algorithm. This means that the full simulation can achieve linear speed-up: a sample path following the progress of B consecutive packets can be simulated on P processors in time on the order of $O(B/P)$.

It should be pointed out that, although the size of M_n can be large, only its first two rows need to be considered when performing the $(\max, +)$ matrix multiplications. The remaining rows are simply shifted down, except the last row, which remains unchanged. To improve the efficiency of the matrix multiplications, those row shifts can be performed by copying rows from one matrix to the next. Alternatively, to save memory and improve efficiency further, only the first two rows need to be stored; all other rows can be manipulated by pointers.

6.5 Experimental Results

Experiments were carried out on a cluster of eight Pentium II 233Mhz workstations, connected over fast Ethernet. The two implementations were written using the LAM [66], implementation of MPI [19], running under Linux. The cluster was extended by the use of a shared memory quad processor system running at 450Mhz. This allowed the number of processors to be increased to 12.

Varying numbers of processors were used to produce results for two sliding window simulations. The window size was fixed in each run; it was equal to 4 in the first set of experiments and 8 in the second. In both cases results were produced for algorithm 1 and algorithm 2. In each simulation run 10^7 packets were generated. The implementation of algorithm 1 included the improvement mentioned in the last paragraph of subsection 6.4.1.

In both simulations, queue 2 was assumed to be unbounded. Its size was monitored, and the packets which found it larger than a given threshold were marked. However, since the object of these simulations is to study the efficiency of the two parallel algorithms and the speed-up that they achieved, the only metric plotted is the speed-up ratio T_0/T_p.

T_0 is the execution time of the best sequential simulation on a single processor and T_p is the execution time of the parallel simulation run on p processors.

Figures 6.2 and 6.3 illustrate the speed-up achieved for the cases of window sizes 4 and 8 respectively. Each figure shows the effect of running the simulation with different batch sizes B.

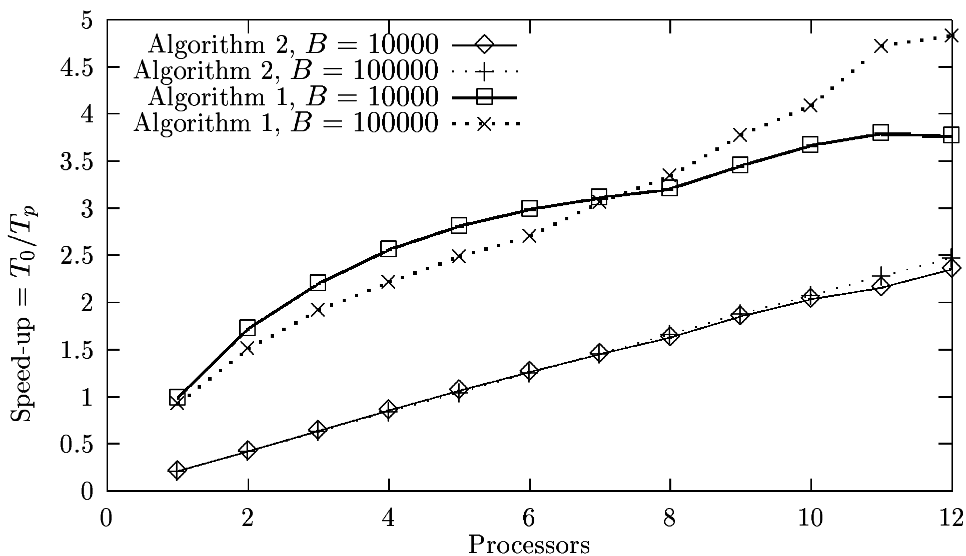

Figure 6.2: Simulation speed-ups for $W = 4$

In both figures, algorithm 1 shows a considerably larger speed-up than algorithm 2, for all batch sizes. However, this improvement appears to tail off as the processor count increases. For algorithm 2, the results start off much poorer than the sequential case; parity with the sequential implementation is achieved at 5 and 8 processors respectively. On the other hand, the graphs confirm that the linear speed-up is maintained as the number of processors is increased. It is a reasonable prediction that, if the experiment is extended to larger numbers of processors, the performance of algorithm 2 would eventually overtake that of algorithm 1.

The examples presented here indicate that both algorithms perform better with larger batch sizes. This is intuitive, since processors need to communicate with each other at

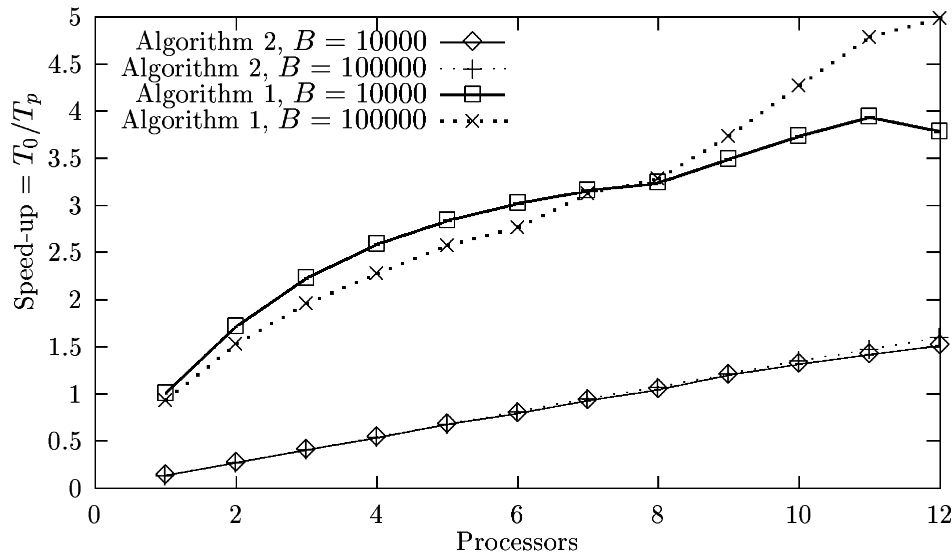

Figure 6.3: Simulation speed-ups for $W = 8$

the end of each portion and each batch, and therefore the fewer the number of batches and portions, the lower the communication overhead. On the other hand, the larger the batches, the more information needs to be stored by each processor. When the memory requirements become too large, performance begins to be affected adversely by phenomena such as page faults.

6.6 Conclusion

It has been shown that parallel simulation techniques can be applied to the performance evaluation of sliding window communication protocols. The first of the two algorithms presented is more general, and also more efficient when the numbers of processors is not large. The second is more restrictive, but has an unbounded degree of parallelism and therefore displays linear speed-up with any number of processors.

It would be nice to generalise algorithm 2, so that it can handle buffer overflows at node 2, and changes in the window size, W. A possible way of achieving this is by introducing relaxations. Roughly, these work as follows: A first version of the sample

path for the current batch is produced as described in section 3, by assuming that W is fixed and the queue at node 2 is unbounded. That sample path is monitored, and packets which would have found a full buffer at node 2 are marked as lost; also, instants when W would have changed value are noted. The relevant portions of the sample path, and those that depend on them, are then re-computed, using new initial conditions and new matrices M_n. The process is repeated until two consecutive versions of the sample path are the same.

If buffer overflows and changes of the window size are relatively rare events, then a small number of relaxations usually suffice. Hence, the benefits of parallelism can be maintained.

In algorithm 2 the generation of a matrix as a product of the matrices describing each node can be applied to other networks. This however may lead to large matrices which will require large amounts of processing power on which to perform the parallel prefix method.

Chapter 7

Conclusion

In this chapter a summary of the achieved results is presented along with an indication of possible areas of future research.

7.1 Summary

The overall aim of this thesis was to develop efficient parallel algorithms for performing simulations, achieving as close to linear speed-up as possible. Two systems were selected that had potential for time-parallel decomposition. The first was a single ATM switch, with multiple bursty sources and cell losses due to buffer overflow. The second system was a sliding window protocol with feedback and re-transmissions.

In all presented algorithms near linear speed-up was achieved, with respect to the number of processors used. However, as with all other examples of time-parallel simulation, these algorithms take advantage of properties of the system under consideration. The algorithms were, however, developed to be as general as possible. The nature of the parallel computer was also a factor in developing the algorithms. Distributed algorithms required lower interprocess communications than the shared memory algorithms.

The algorithms rely on the operations of parallel prefix, parallel merging and relaxation. The first two of these can be implemented efficiently, achieving almost linear

speed-up. However, relaxation in the worst case will require p iterations to complete, when p is the number of processors. In the case of the algorithms used here the number of iterations required were normally much less than p. Also the proportion of the overall execution time spent performing the relaxation was small. Thus even in the worst case overall speed-up was still achievable.

The use of these parallel simulation algorithms may be combined with hybrid simulation techniques. In these simulations analytical approaches are used to reduce the complexity of the system before simulation is used to solve the overall system. In general the use of parallel simulation algorithms can be used alongside Hierarchical Modelling techniques [7]. Sequential simulation stages can be replaced by parallel stages for which there is an appropriate parallel algorithm.

7.1.1 ATM Switch

Algorithms for simulating an ATM switch are efficient and offer near linear speed-up. Previous work in this area produced either approximate solutions and / or did not handle the bursty arrival sources.

The algorithm presented in chapter 5, provides an efficient technique to generate and merge a number of (bursty) arrival sources on a distributed network of workstations.

Alterations to the parameters for these simulations will affect the proportion of cells lost; this affects the efficiency of the algorithm. In the case of the first algorithm, increasing the proportion of cells lost will quickly increase the execution time. This is due to the increase in sequential work "filling up" batches due to lost cells and the increase in the number of iterations required to converge. The second algorithm is far less sensitive to increases in cell losses, as only a small amount of its efficiency comes from performing few iterations. Both algorithms showed small reductions in execution times as the cell loss ratio decreased. The time the program spends dealing with possibly lost cells is now very small, thus decreasing the cell loss ratio has little effect on the execution time. The

sequential simulation algorithm was not affected by the increase or decrease in cell losses, as the loss of a cell can be determined simply by checking the current queue size.

The techniques used for generating bursty arrival sources can be used for sources with multiple arrival rates, that change at random intervals. The acceptance algorithms are also of more general use than ATM switches. These algorithms may be applied to systems where the acceptance and loss of cells need to be determined. In the case of the second algorithm this can even be in the cases of high cell loss.

7.1.2 Sliding Window

The algorithms presented for simulating a sliding window protocol deal with feedback from the destination node. This illustrates the use of the main technique of parallel prefix. One of the algorithms presented here shows how parallel prefix can be used to handle dependencies between packets in service. This may be applied to any simulation problem in which a given event is directly affected by an event prior to the preceding event.

The size of the sliding window will have an effect on the overall performance. In the case of the first algorithm the increase should be small as the window size increases. However, for the second algorithm increasing the window size will significantly increase the execution time. The sequential algorithm should be affected in a similar manner to that of the first algorithm.

7.2 Future Work

The algorithms for simulating an ATM switch determine cell losses, however, no attempt is made to retransmit these lost cells. Thus the combination of the sliding window protocol and an ATM switch would provide insight into the effect that these lost cells have on the overall network.

Both systems that are presented here are of single entities, either a switch or a single

sliding window link. It would be of interest to develop algorithms for simulating a network of these to determine if the speed-up obtained in a single node can be maintained.

A significant proportion of the simulation time for the algorithms presented here is required for generating the arrival traffic. The algorithms that use these arrival streams do not depend in any manner on these arrival models. It would be of interest to try other arrival types, these could be trace driven sources or other arrival models such as correlated traffic. The efficiency of these approaches would depend largely on being able to generate the data in time $O(N/P)$.

It would be of interest to add in the ability to deal with ATM traffic of different priorities. Two different cases can be considered here. In the first case if a cell arrives to a full queue, then a cell already in the queue of lower priority is discarded so that the new cell may be accepted. The second case will discard cell of lower priority before the queue becomes full in an attempt to reserve space for higher priority cells. The former case is potentially the most difficult to deal with. The acceptance of a cell now also depends on there not being a cell of higher priority arriving to a full queue before the initial cell receives service. One approach to dealing with this is to allow the simulation to run as normal, once a loss is detected a correction is made by marking a previously accepted cell as lost. This may cause problems with cells in one batch needing to re-mark cells from previous batches as lost.

In the second case cells of lower priority are lost once the queue size reaches a certain level. This could be easily added into the second acceptance algorithm.

The sliding window protocol could be extended to become a full TCP protocol with losses effecting the window size. This could be used to compare the different TCP protocols such as RENO, SACK and Vegas. In the case of the first algorithm this could easily be implemented by replacing the existing sequential sliding window algorithm with a full TCP/IP simulation algorithm.

To extend the second sliding window algorithm into a full TCP/IP simulation would

require more effort. One approach would be to use relaxation. The current simulation could be run with a fixed window size and unbounded buffer at the receiving node. The points in the simulation where the window size would change and points where the buffer at the receiving node would lose packets can be recorded. The portions of the sample path that would be affected by these changes can now be re-computed. This process is repeated until two consecutive versions of the sample path are the same.

The technique of relaxation is often shunned due to the fact that in the worst case it is no better than sequential execution. This has also reduced the acceptance of parallel prefix approaches, as relaxation is often required with parallel prefix. It would be of great benefit if an alternative algorithm could be developed that does not have such a worst case scenario.

It would be of interest to use the algorithms developed here in the parallel simulation of other systems such as systems with multiple servers to determine if the same performance can be achieved.

The simulation of larger systems by the techniques of time-parallel simulation would appear to be best tackled by the decomposition of the system into smaller sub-systems. Each of these sub-systems could then be solved one at a time as in the example system in section 3.6. This has the advantage of reducing the problem to smaller problems for which known solutions already exist. For example a TCP/IP network sending traffic over an ATM network. The initial departures from the source server could be fed into the simulation of the ATM switch, the result times from the switch (and rest of ATM network) can be fed into the sink server. Acknowledgments could be returned via a similar process.

Bibliography

[1] B. Abali, F. Ozguner, and A. Bataineh. Balanced parallel sort on hypercube multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):572–581, May 1993.

[2] S. G. Akl. *The design and ananlysis of parallel algorithms.* Prentice Hall, Englewood Cliffs, N.J., 1989.

[3] S. Andradóttir and T. J. Ott. Time-segmentation parallel simulation of networks of queues with loss or communication blocking. *ACM Transactions on Modelling and Computer Simulation*, 5(4):269–305, October 1995.

[4] R. Ayani and H. Rajaei. Parallel simulation based on conservative time windows: A performance study. *Concurrency: Practice and Experience*, 6(2):119–142, April 1994.

[5] F. Baccelli and M. Canales. Parallel simulation of stochastic petri nets using recurrence equations. *ACM Transactions on Modeling and Computer Simulation*, 3(1):20–41, January 1993.

[6] R. Bagrodia, K. M. Chandy, and W. T. Liao. A unifying framework for distributed simulation. *ACM Transactions on Modelling and Computer Simulation*, 1(4):348–385, October 1991.

[7] S. Balsamo, L. Donatiello, and R. Mirandola. Hierarchical modelling for performance evaluation of computer and communication systems. In *Proceedings of the Workshop on Performance Modeling and Evaluation of ATM Networks*, pages 11/1–11/9, University of Bradford, UK, June 1993.

[8] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS 1968 Spring Joint Computer Conference*, pages 130–145, Washington, D.C. USA, May 1968.

[9] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 1, pages 35–60. Morgan Kaufmann, 1993.

[10] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical Report MIT/LCS/TR-188, Massachusetts Institute of Technology, November 1977.

[11] C. D. Carothers and K. S. Perumalla. Efficent optimistic parallel simulations using reverse computation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 126–135, Atlanta, Georgia USA, May 1999.

[12] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engeneering*, 5(5):440–452, September 1979.

[13] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[14] K. M. Chandy and R. Sherman. Space-time and simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 53–57, Tampa, Florida USA, March - June 1989.

[15] L. Chen. Parallel simulation by multi-instruction, longest-path algorithms. *Queueing Systems*, 27(1-2):37–54, 1997.

[16] R. Cole. Parallel merge sort. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 10, pages 453–495. Morgan Kaufmann, 1993.

[17] S. G. Eich, A. G. Greenberg, B. D. Lubachevsky, and A.Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. *ACM Transactions on Modeling and Computer Simulation*, 3(4):287–314, October 1993.

[18] K. Fall and S. Floyd. Simulation–based comparison of tahoe, reno and sack tcp. *Computer Communication Review*, 26(3):5–21, 1996.

[19] Message Passing Interface Forum. http://www.mpi-forum.org/, April 2000.

[20] W. R. Franta and K. Maly. An efficient data structure for the simulation event set. *Communications of the ACM*, 20(8):596–602, August 1977.

[21] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[22] R. M. Fujimoto. Parallel discrete event simulation: Will the field survive? *OSRA Journal on Computing*, 5(3):213–230, Summer 1993.

[23] R. M. Fujimoto. Parallel and distributed simulation. In *Proceedings of the 1995 Winter Simulation Conference*, pages 118–125, Arlington, VA USA, December 1995.

[24] R. M. Fujimoto. Parallel and distributed simulation. In *Proceedings of the 1999 Winter Simulation Conference*, pages 122–131, Phonix, Arizona USA, December 1999.

[25] R. M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley-Interscience Publication, New York, 2000.

[26] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, October 1997.

[27] R. M. Fujimoto and D. Nicol. State of the art in parallel simulation. In *Proceedings of the 1992 Winter Simulation Conference*, pages 246–254, Arlington, VA USA, December 1992.

[28] R. M. Fujimoto, I. Nikolaids, and C. A. Cooper. Parallel simulation of statistical multiplexers. *Discrete Event Dynamic Systems: Theory and Applications*, 5(2–3):115–140, April–July 1995.

[29] B. Gaujal, A. G. Greenberg, and D. M. Nicol. A sweep algorithm for massively parallel simulation of circuit-switched networks. Technical Report 189680, NASA, July 1992.

[30] B. Gaujal, A. G. Greenberg, and D. M. Nicol. A sweep algorithm for massively parallel simulation of circuit-switched networks. *Journal of Parallel and Distributed Computing*, 18(4):484–500, August 1993.

[31] A. I. Geist. *PVM–parallel virtual machine : a users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, Mass., 1994.

[32] P. W. Glynn and P. Heidelberger. Analysis of parallel replicated simulations under a complearion time constraint. *ACM Transactions on Modelling and Computer Simulation*, 1(1):3–23, January 1991.

[33] G. H. Gonnet. Heaps applied to event driven mechanisms. *Communications of the ACM*, 19(7):417–418, July 1976.

[34] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. Unboundedly parallel simulations via recurrence relations. In *Proceedings of the ACM conference on Measurement and modeling of computer systems*, pages 1–12, Bolder, CO USA, May 1990.

[35] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. Algorithms for unboundedly parallel simulations. *ACM Transactions on Computer Systems*, 9(3):201–221, August 1991.

[36] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. Superfast parallel discrete event simulations. *ACM Transactions Modeling and Computer Simulation*, 6(2):107–136, April 1996.

[37] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[38] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[39] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. In *Proceedings of the 1990 Winter Simulation Conference*, pages 734–737, New Orleans, LA USA, 1990.

[40] D. Jefferson. Virtual time ii: Storage managment in distributed simulation. In *Proceedings of the Ninth Annual ACM Symposium on Principles of distributed computing*, pages 75–89, Quebec City, PQ Canada, August 1990.

[41] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[42] W. D. Kelton and A. M. Law. An analytical evaluation of alternative strategies in steady-state simulation. *Operations Reasearch*, 32(1):169–184, January–February 1984.

[43] D. E. Knuth. *The Art of Computer Programming Vol III: Sorting and Searching 2nd Ed.* Addison Wesley Publishing, 1998.

[44] P. M. Kogge and H. S. Stone. A parallel algorithm for efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–792, August 1973.

[45] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, c-34(10):965–968, October 1985.

[46] K. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, C–32(10):942–946, October 1983.

[47] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association of Computing Machinery*, 27(4):831–838, October 1980.

[48] Y. B. Lin. Parallel trace-driven simulation for packet loss in finite-buffered voice multiplexers. *Parallel Computing*, 19(2):219–228, February 1993.

[49] Y. B. Lin. Parallel trace-driven simulation of packet-switched multiplexer under priority scheduling policy. *Information Processing Letters*, 47:197–201, September 1993.

[50] Y-B. Lin and E. D. Lazowska. A time division algorithm for parallel simulation. *ACM Transactions on Modelling and Computer Simulation*, 1(1):73–83, January 1991.

[51] Y-B. Lin and B. R. Preiss. Optimal memory managment for time warp parallel simulation. *ACM Transactions on Modelling and Computer Simulation*, 1(4):283–307, October 1991.

[52] Y-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in time warp simulation. In *Proceedings of the 1993 workshop on Parallel and distributed simulation*, pages 16–19, San Diego, CA USA, May 1993.

[53] B. D. Lubachevsky. Efficent distributed event–driven simulations of multiple–loop networks. *Communications of the ACM*, 32(1):111–131, January 1989.

[54] B. D. Lubachevsky and A. G. Greenberg. Simple, efficient asynchroneous parallel prefix algorithms. In *Proceedings of the 1987 Conference on Parallel Processing*, pages 66–69, St. Charles, Illinois USA, August 1987.

[55] F. Mattern. Efficent algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, August 1993.

[56] A. S. McGough. Parallel simulation. Master's thesis, Computing Science, University of Newcastle Upon Tyne, 1996.

[57] A. S. McGough and I. Mitrani. Parallel simulation of atm switches. In *Third Conference of the United Kingdom Simulation Society*, pages 72–79, Keswick, U.K., April 1997.

[58] A. S. McGough and I. Mitrani. Parallel simulation of atm switches using relaxation. In *Sixth IFIP workshop on performance modelling and evaluation of ATM networks*, pages 54/1–54/10, Ilkley, U.K., July 1998.

[59] A. S. McGough and I. Mitrani. Efficent distributed simulation of a communication switch with bursty sources and losses. In *14th Workshop on Parallel and Distributed Simulation*, pages 85–92, Bolognia, Italy, May 2000.

[60] A. S. McGough and I. Mitrani. Efficent parallel simulation of a sliding window protocol. In *Eighth IFIP workshop on performance modelling and evaluation of ATM & IP networks*, pages 54/1–54/9, Ilkley, U.K., July 2000.

[61] A. S. McGough and I. Mitrani. Parallel simulation of atm switches using relaxation. *Journal of Performance Evaluation*, 41(2–3):149–164, July 2000.

[62] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[63] D. M. Nicol, A. G. Greenberg, and B. D. Lubachevsky. Massively parallel algorithms for trace–driven cache simulation. In *Proceedings of the 6th workshop on Parallel and Distributed Simulation*, pages 3–11, Newport Beach, CA USA, January 1992.

[64] A. Nicolau and H. Wang. Optimal schedules for parallel prefix computation with bounded resources. In *Proceedings of the third ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10, Williamsburg, VA USA, April 1991.

[65] I. Nikolaidis, R. Fujimoto, and C. A. Cooper. Time-parallel simulation of cascade statistical multiplexers. In *Proceedings of the 1994 conference on Measurement and modeling of computer systems*, pages 231–240, Nashville, TN USA, May 1994.

[66] University of Notre Dame. Lam / mpi parallel computing. http://www.mpi.nd.edu/lam/, April 2000.

[67] Jr. P. F. Reynolds. A spectrum of options for parallel simulation. In *Proceedings of the 1988 Winter Simulation Conference*, pages 325–332, San Diego, CA USA, December 1988.

[68] J. Padhye, V. Firoiu, and D. Towsley. A stochastic model of tcp reno congestion avoidance and control. Paper TR 99-02, University of Massachusetts, February 1999.

[69] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modelling tcp throughput: A simple model and its empirical validation. Paper TR 98-008, University of Massachusetts, February 1998.

[70] C. G. Plaxton. Load balancing, selection and sorting on the hypercube. In *Proceedings of the 1989 ACM Symposium on Parallel algorithms and architectures*, pages 64–73, Santa Fe, NM USA, 1989.

[71] F. Quaglia. Combining periodic and probabilistic checkpoints in optimistic simulation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 109–116, Atlanta, Georgia USA, May 1999.

[72] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, March 1981.

[73] L. S. Sokol, J. B. Weissman, and P. A. Mutchler. Mtw: An empirical performance study. In *Proceedings of the 1991 Winter Simulation Conference*, pages 557–563, Phoenix, AZ USA, December 1991.

[74] D.B. Wagner and E. D. Lazowska. Parallel simulation of queueing networks: Limitations and potentials. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 146–155, Oakland, CA USA, May 1989.

[75] J. J. Wang and M. Abrams. Approximate time-parallel simulation of queueing systems with losses. In *Proceedings of the 1992 Winter Simulation Conference*, pages 700–708, Arlington, VA USA, December 1992.

[76] W. Whitt. The efficiency of one long run versus independant replications in steady-state simulations. *Managemnet Science*, 37(6):645–666, June 1991.

[77] C. Williamson, B. Unger, and X. Zhonge. Parallel simulation of atm networks: Case study and lessons learned. In *Proceedings of the 2nd Canadian Conference on Broadband Research*, pages 78–88, Ottawa Canada, June 1998.

[78] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary. Scheduling critical channels in conservative parallel discrete event simulation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 20–28, Atlanta, Georgia USA, May 1999.