Durham E-Theses

Parallel simulation techniques for telecommunication network modelling

Hind, Alan

How to cite:

Hind, Alan (1994) *Parallel simulation techniques for telecommunication network modelling*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/5520/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

Parallel Simulation Techniques for Telecommunication Network Modelling

Alan Hind
B.Sc. (Salford)

School of Engineering and Computer Science
University of Durham

A thesis submitted in partial fulfilment of the requirements of the Council of the University of Durham for the Degree of Doctor of Philosophy (Ph.D.).

January 1994

- 7 JUN 1994

Abstract

In this thesis, we consider the application of parallel simulation to the performance modelling of telecommunication networks.

A largely automated approach was first explored using a parallelizing compiler to speed-up the simulation of simple models of circuit-switched networks. This yielded reasonable results for relatively little effort compared with other approaches. However, more complex simulation models of packet- and cell-based telecommunication networks, requiring the use of discrete event techniques, need an alternative approach.

A critical review of parallel discrete event simulation indicated that a distributed model components approach using conservative or optimistic synchronization would be worth exploring. Experiments were therefore conducted using simulation models of queueing networks and Asynchronous Transfer Mode (ATM) networks to explore the potential speed-up possible using this approach. Specifically, it is shown that these techniques can be used successfully to speed-up the execution of useful telecommunication network simulations.

A detailed investigation has demonstrated that conservative synchronization performs very well for applications with good lookahead properties and sufficient message traffic density and, given such properties, will significantly outperform optimistic synchronization. Optimistic synchronization, however, gives reasonable speed-up for models with a wider range of such properties and can be optimized for speed-up and memory usage at run time. Thus, it is confirmed as being more generally applicable particularly as model development is somewhat easier than for conservative synchronization. This has to be balanced against the more difficult task of developing and debugging an optimistic synchronization kernel and the application models.

DEO GRATIA

Acknowledgments

The following people have been vital to the production of this work; either in their direct advice and input or just in putting up with the fact that I was busy doing this.

- To my wife, Angie, and my children, Ben and Heather – for their love.
- To my parents – for their encouragement.
- To my supervisor, Professor Phil Mars of the University of Durham – for his direction and advice.
- To Neil Macfadyen of BT. Laboratories, Core and Global Networks Division – for discussions without number and enthusiasm without bounds.
- To Richard Earnshaw of the University of Twente, The Netherlands – for friendship, working partnership and the ATM. Simulator.
- To Bruno Preiss and Wayne Loucks at the University of Waterloo, Ontario, Canada – for their kind hospitality when visiting their country and for YADDES.
- To Ron Kerr and Kevin Conner of the Computing Laboratory of the University of Newcastle-upon-Tyne – for allowing me access to NEWTON, and for their time and advice.
- To Steve Turner at the University of Exeter and Brian Roberts, Chris Booth and David Bruce at DRA. Malvern – for their friendship, lively discussion and TWSIM.
- To John, Raghu, Matt, Phillip, Jeremy, Chen and David in the lab – for a good laugh when in vital need.
- To Sylvia – for all the arrangements.

The following trademarks are acknowledged: BT. and British Telecom are trademarks of British Telecommunications plc.; IMS. and occam are trademarks of Inmos Limited; IBM., SNA. and PC/AT. are a trademarks of International Business Machines Corp.; 3L is a trademark of 3L Limited; VAX is a trademark of Digital Equipment Corp.; Sun is a trademark of Sun Microsystems Corp.; Ethernet is a trademark of Xerox Corp.; Multimax is a trademark of Encore Computer Corp.; and Sim++ and Jade is a trademark of Jade Simulations International Corp.; Comnet II.5 and ModSim are trademarks of CACI Products Company Inc.

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree, and that all sources of information have been duly acknowledged.

© Copyright 1994, Alan Hind

The copyright of this thesis rests with the author. No quotation from it should be published without his written consent, and information derived from it should be acknowledged.

Contents

1 Introduction 1

1.1 Performance Evaluation 1

1.2 Simulation Objectives 3

1.3 Simulation Speed-up 5

1.3.1 Amdahl's Law 9

1.4 Parallel Hardware Architectures 11

1.5 Outline of Thesis 16

2 A Review of Parallel Discrete Event Simulation 19

2.1 Introduction 19

2.2 Parallelizing Compilers 21

2.3 Distributed Simulation Experiments 24

2.4 Distributed Simulation Functions 25

2.5 Distributed Simulation Events 27

2.6 Distributed Simulation Model Components 28

2.7 Combined Approaches 29

2.8 Time Parallelism 30

2.9 Summary 32

3 Synchronization Approaches for Distributed Model Components 34

3.1 Introduction 34

3.2 Synchronization in Action 35

3.3 Conservative Synchronization Approaches 38

3.3.1 Performance of Conservative Synchronization Approaches 43

3.3.2 Critique of Conservative Synchronization Approaches 44

3.4 Synchronous Approaches 45

3.4.1 Performance of Synchronous Approaches 48
3.4.2 Critique of Synchronous Approaches 48
3.5 Optimistic Synchronization Approaches 49
3.5.1 Enhancements to Optimistic Synchronization Approaches 50
3.5.2 Performance of Optimistic Synchronization Approaches 57
3.5.3 Critique of Optimistic Synchronization Approaches 59
3.6 Summary 61

4 Parallel Simulation of Circuit-Switched Networks using a Parallelizing Compiler 63

4.1 Introduction 63
4.2 The Testbed Architecture 65
4.2.1 Hardware Architecture 65
4.2.2 Software Architecture 66
4.2.3 Execution Model 67
4.3 Discussion of Results 68
4.3.1 Introduction 68
4.3.2 Uniprocessor Simulation Results 71
4.3.3 Multiprocessor Simulation Results for the Five-node Model 72
4.3.4 Multiprocessor Simulation Results for the Ten-node Model 73
4.3.5 Multiprocessor Simulation Results for the Twenty-node Model . . . 74
4.4 Conclusions 75

5 Parallel Simulation of Queueing Networks 85

5.1 Introduction 85
5.2 YADDES — Yet Another Distributed Discrete Event Simulator 86
5.2.1 Sequential Event-list Synchronization 87
5.2.2 Distributed Multiple Event-list Synchronization 88
5.2.3 Conservative Distributed Event-list Synchronization 88
5.2.4 Optimistic Distributed Simulation Synchronization 89
5.3 The Simulation Models 90
5.4 Discussion of Results 93
5.4.1 Initial Results for the Closed Stochastic Queueing Network 93
5.4.2 Main Results for the Closed Stochastic Queueing Network 94

5.4.3 Virtual Time (VT) Memory Management Results for the Closed Stochastic Queueing Network . . . 98
5.4.4 Discussion of Results for the Tandem Queueing Network . . . 100
5.5 Conclusions . . . 102

6 Parallel Simulation of Asynchronous Transfer Mode Networks **135**
6.1 Introduction . . . 135
6.2 Broadband Networks . . . 137
6.3 Simulator Architecture . . . 139
6.3.1 The Multiprocessor Testbed . . . 139
6.3.2 The Software Architecture . . . 139
6.4 The Synchronization Mechanism . . . 140
6.5 The Simulator Results . . . 142
6.6 Performance Analysis of the Simulator . . . 142
6.6.1 Performance of Production Runs . . . 142
6.6.2 Variations in Lookahead . . . 145
6.6.3 Asymmetric Traffic . . . 147
6.7 Conclusions . . . 147

7 Conclusions and Further Work **159**
7.1 Conclusions . . . 159
7.2 Further Work . . . 162

A Simulation Model Files **183**

B Published Papers **184**

List of Figures

1.1 Amdahl's law 18

3.1 Example network of processes at initialisation — all link-times are at zero. . 37
3.2 Example network of processes at time 2—the link-times represent the time-stamp of the last message to cross the link. 38
3.3 Example network of processes—simulated using time windows 46
3.4 Example network of processes—simulated using time windows 47
3.5 Time warp rollback using different state-saving schemes 54

4.1 Shared memory multiprocessor architecture with a single bus and local caches. 65
4.2 Execution trace of a parallelized program on the Multimax shared memory multiprocessor. 68
4.3 The ten-node sparsely-connected network. The numbers in the square brackets indicate the capacity in circuits of each link. 70
4.4 Speed-up results for the ten-node fully-connected network comparing the alternative reference times 77
4.5 Speed-up results for the five-node fully-connected network model 78
4.6 Comparison of percentage run-times used by the two most expensive processes using the automatically parallelized simulator 79
4.7 Speed-up results for the ten-node fully- and sparsely-connected network models 80
4.8 Speed-up results for the ten-node fully-connected network model showing means and standard deviations from ten simulation runs 81
4.9 Speed-up results for the twenty-node fully-connected network model 82
4.10 Speed-up results comparing the performance of the hand optimized simulator for all network models 83

4.11 Speed-up results showing the performance of the hand optimized simulator for the fully-connected network models with Amdahl's law 84

5.1 A 4-dimensional hypercube. 91

5.2 YADDES process model of one switch of the tandem queueing network. . . 92

5.3 Speed-up using multiple distributed event-list (ML) synchronization for the hypercube of queues. 104

5.4 Speed-up using conservative Chandy-Misra synchronization (CM) without NULL-message cancellation for the hypercube of queues. 105

5.5 Speed-up using conservative Chandy-Misra synchronization (CM) with NULL-message cancellation (NMC) for the hypercube of queues. 106

5.6 Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of one for the hypercube of queues. 107

5.7 Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of one for the hypercube of queues. 108

5.8 Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of two for the hypercube of queues. 109

5.9 Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of two for the hypercube of queues. 110

5.10 Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of four for the hypercube of queues. 111

5.11 Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of four for the hypercube of queues. 112

5.12 Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of eight for the hypercube of queues. 113

5.13 Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of eight for the hypercube of queues. 114

5.14 Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of sixteen for the hypercube of queues. 115

5.15 Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of sixteen for the hypercube of queues. 116

5.16 Speed-up against CPI for virtual time synchronization (VT) on two processors for the hypercube of queues. 117

5.17 Speed-up against CPI for virtual time synchronization (VT) on four processors for the hypercube of queues. 118
5.18 Speed-up against CPI for virtual time synchronization (VT) on eight processors for the hypercube of queues. 119
5.19 Comparison of synchronization mechanisms for a load of one customer per queue for the hypercube of queues. 120
5.20 Comparison of synchronization mechanisms for a load of four customers per queue for the hypercube of queues. 121
5.21 Comparison of synchronization mechanisms for a load of eight customers per queue for the hypercube of queues. 122
5.22 State memory usage for virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues. 123
5.23 Message memory usage for virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues. . 124
5.24 Total memory usage for virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues. 125
5.25 Total memory usage against simulation time for a CPI of 1, 2, 4, 8 and 16 using virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues. 126
5.26 Total memory usage against speed-up for a CPI of 1, 2, 4, 8 and 16 using virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues. 127
5.27 Speed-up comparison of conservative synchronization methods for the tandem queueing network. 128
5.28 Speed-up using virtual time synchronization (VT) with lazy cancellation for the tandem queueing network. 129
5.29 Speed-up using virtual time synchronization (VT) with aggressive cancellation for the tandem queueing network. 130
5.30 Speed-up against CPI for virtual time synchronization (VT) with two processors for the tandem queueing network. 131
5.31 Speed-up against CPI for virtual time synchronization (VT) with four processors for the tandem queueing network. 132

5.32 Speed-up against CPI for virtual time synchronization (VT) with eight processors for the tandem queueing network. 133
5.33 Speed-up comparison of all synchronization mechanisms for the tandem queueing network. 134

6.1 High-speed transputer-based telecommunication network simulator — hardware configuration. 149
6.2 The overall hierarchy of the simulation model. The Event scheduler is a control-plane for all of the upper layers. 150
6.3 Basic network topology used for the simulator performance analysis runs. The processor assignments are also shown. 151
6.4 Parallel simulation run times as a function of traffic load for the twelve-node networks on twelve transputers. The 150 Mbits/s times are scaled to take account of the difference in simulation length. 152
6.5 Speed-up curves as a function of traffic load. Speed-up is calculated relative to the optimized or unoptimized uniprocessor simulations. 153
6.6 Speed-up (optimized) as a function of NULL-message ratio. The difference between the two curves represents the extra parallelism that can be extracted from the higher speed rings. 154
6.7 NULL-message ratio (NMR) as a function of load. As might be expected, the ratio is largely independent of the ring speed. 155
6.8 Speed-up (optimized) as a function of load for a range of lookahead values and symmetric and asymmetric traffic patterns. 156
6.9 NULL-message ratio (NMR) as a function of load for a range of lookahead values and symmetric and asymmetric traffic patterns. 157
6.10 Number of NULL-messages normalised against the number at the lowest load (0.05 calls/source/s) plotted against link load for a selection of source-switch links. 158

List of Abbreviations

ATM	Asynchronous Transfer Mode
B-ISDN	Broadband Integrated Services Digital Network
CM	Chandy-Misra
CMB	Chandy-Misra-Bryant
CPI	Checkpoint Interval
FCFS	First-Come-First-Served
GVT	Global Virtual Time
IBM	International Business Machines
ISDN	Integrated Services Digital Network
I/O	Input and Output
LAN	Local Area Network
LVT	Local Virtual Time
LP	Logical Process
MIMD	Multiple Instruction stream Multiple Data stream
MISD	Multiple Instruction stream Single Data stream
MMT	Minimum Message Time
MVT	Minimum Virtual Time
NMC	NULL-Message Cancellation
NMR	NULL-Message Ratio
PDES	Parallel Discrete Event Simulation
PP	Physical Process
RR	Round-Robin
SIMD	Single Instruction stream Multiple Data stream
SISD	Single Instruction stream Single Data stream
VT	Virtual Time
WAN	Wide Area Network
YADDES	Yet Another Distributed Discrete Event Simulator

Chapter 1

Introduction

1.1 Performance Evaluation

Telecommunication networks have evolved rapidly over the last decade both in their size and complexity. As a direct result of this evolution, a large number of computer aided engineering techniques have been developed to help the engineer in the design, development and management of these networks. One of the main issues at stake, particularly in the design stage, is that of performance evaluation.

So that we can understand the aims and objectives of performance evaluation, we must first examine the structure of the telecommunication network itself. Its basic function is to transmit information through a common transmission facility. The network consists of switching nodes, multiplexers and demultiplexers connected by various transmission media. Network users generate messages which must be transmitted from sources to destinations within the network. The network itself will also generate its own internal signalling messages to control network operation. The engineer must therefore consider several issues; message formats, multiplexing and demultiplexing techniques, switching techniques, storage, error control, routing strategies, flow control, network operation, management functions: to name but a few. The method used to resolve each of these issues will have some impact on the overall network performance.

The performance measures of interest to the engineer will depend on the type of network under study, its application, and whether the network uses circuit- or packet-switching. For a circuit-switched network the important performance measures are the probability of blocked calls and the associated grade of service. For a packet-switched network, through-

put, average message delay, link utilisation and buffer use statistics are more useful. Evaluating network architectures with respect to such performance measures is the job of the performance engineer.

There are three general approaches that can be used to obtain estimates of such performance measures. We can perform measurements on a *real* system; we can use analytical techniques in order to calculate performance estimates from a mathematical model; or we can use computer simulation techniques.

The most direct approach is obviously to measure the performance of an existing network. This is usually difficult due to the size, complexity and accessibility of such networks. Access can be difficult as the systems are in constant use and to build a duplicate system just for performance evaluation purposes is often prohibitive. However, small model[1] systems are still often built, particularly to aid the design and development stage of a new system. Such a system is often termed a testbed or system emulation. If experimentation is required on an existing network, it may be economically impracticable or even dangerous to suspend the normal operation of the network in order to perform experiments. Nevertheless some measurements are always required from existing networks in order to validate performance estimates obtained by other means. Indeed, there is a growing trend to incorporate measurement and monitoring tools as an integral part of the network management software. The data from such tools can be used to feed capacity planning tools to answer "what-if" type questions about changes to the network capacity and architecture. BEST/1-SNA is such a capacity planning tool for IBM SNA (System Network Architecture) networks which uses measurement data [1].

Another method of obtaining performance estimates is to use mathematical analysis. Analytic models generally require a high degree of abstraction in order to ensure that the model is tractable. This means that the development of an analytic model which accurately reflects the network under study is difficult and requires considerable effort and skill. Analytic models have great difficulty with transient conditions, such as traffic fluctuations or component failures, as these are difficult to express mathematically. Also, in making the model tractable, a model may be distorted due to the omission or approximation of system characteristics which are difficult to solve analytically. However, once the model is developed, its solution is generally very fast.

[1]The word model is used here in the sense of a small imitation of the real thing.

Simulation techniques allow a network to be modelled at an arbitrary level of detail. This means that less abstraction is required and also the process of model construction can be more straightforward. In practice, however, the more secondary detail that can be abstracted out the faster the simulation will be. Generally speaking, the execution of a simulation model will use substantially more computing time than an analytical model. However, in many cases simulation is the only option as an analytic model is inaccurate, intractable or unavailable. There are interesting statistics concerning the use of RESQ [2], the research queueing package developed at the IBM Thomas J. Watson Research Centre. This package allows the user to develop simulation models of communication systems using either analytic or simulation models based on queueing networks and it was found that over 99% of all the models developed used simulation [3].

These three approaches to network performance evaluation should be used in a complementary manner in order to validate results and obtain the fastest results to the required accuracy. Reviews of simulation approaches can be found in Frost et. al. [4] and in Kurose and Mouftah [3].

1.2 Simulation Objectives

The simulation of a telecommunication network may have one or more objectives: understanding the behaviour of a system, obtaining estimates of average or worst/best case performance parameters, guiding the selection of design parameters and/or operating strategies and verifying the simulation results against measurements taken from a real system.

The simulation objective can have a profound impact on how the simulation should be designed and how it should be run. If we are only interested in understanding the behaviour of the system, then we would perhaps like an animated graphical output presentation, adjustable simulation speed and system parameters: all of this controlled via a good user interface. If we are interested in obtaining estimates of performance parameters then we need the simulation to give us results to the required accuracy with confidence intervals as fast as possible. For the selection of design parameters, the simulation becomes part of an optimization loop. Thus the simulation needs to be interactive and fast with controllable accuracy. The information obtained from the simulation can consist of a number of estimates of a performance measure corresponding to the random variation of particular parameters. The information is then used to calculate gradient estimates to predict how to optimize

design parameters. If we are fitting a simulation model to measurements taken from a real system a number of simulations will have to be performed with different parameter settings that correspond to the measured result conditions. In this case the statistical properties of the results are very important so as to be able to predict the accuracy of the simulated results.

It is well established that simulations of telecommunication networks are generally slow to develop and slow to run. As we have already mentioned, the telecommunication networks we are interested in simulating are large and complex consisting of a great variety of component parts. This means that our simulation models will reflect this. Existing networks are also constantly changing with time as new equipment is installed, old equipment is replaced and software up-graded. Thus the simulation model of an existing system must keep pace with the current and predicted changes. Networks always seem to grow and never shrink, so ever larger simulations are required. A frustrating re-occurrent factor for the performance engineer is that when larger computing resources become available the network which needs to be simulated has also grown.

Simulations are also slow for statistical reasons as, for any result to be statistically significant (ie. to a given accuracy), a sufficient run length, or a number of runs, has to be performed. The generation of events by a simulation program is usually driven by an input stream of pseudo-random numbers. These are used to generate lengths of time, lengths of messages, probabilities etc. As the event generation depends on random numbers then the performance measures output by the simulation model are themselves stochastic in nature. Therefore a simulation represents a statistical experiment and the results should be subjected to careful statistical analysis.

In some cases we may be interested in the behaviour of a system as it progresses from some initial condition. This is known as the transient response. This usually lasts for a well-defined period of (simulated) time. If we are interested in long term average results (steady state), the transient response must be detected and discarded, or the simulation must be run for long enough so that the effects of the initial conditions are negligible. The problem of identifying the transient phase of a simulation is addressed by Schruben [5], Welch [6] and Heidelberger and Welch [7].

If steady state average results of performance measures are required then we find that every time we run the simulation with different random number streams we get variations in our results. If we were to run a simulation ten times and get results within one per cent

(say) then we may be confident of our result. If the variation were much greater, then our confidence would be less. The generation of confidence intervals based on the variance of the performance measure over the simulation(s) is therefore very important. These can be generated by dividing up a single long simulation, without the transient period, into n equal length time periods and constructing the confidence interval from values of the performance measure in each interval. Otherwise we can do n separate simulation runs with different random input streams, again without the transient period. How large n should be is a difficult but crucial problem, as also is the length of the time periods. These problems are dealt with by Welch [6] and by Law [8].

For the single run case, if n is too small the samples will be correlated, if n is too large, excessive simulation time will be used. Nevertheless, the single run case has the advantage of only having to discard a single transient period. Also, the longer the simulation, the smaller the variance in the result will be. Where to stop is very much governed by the law of diminishing returns.

Apart from the two more obvious methods described above, there are other methods of obtaining confidence intervals from simulation results. The regenerative method is also based on partitioning a single simulation into independent runs which have the same regeneration state. This is a state such that the future progress of the simulation is statistically the same each time the state is entered. The problem is in recognising a regeneration state for the particular system of interest. This method is described by Crane and Lemoine [9]. A fourth method, known as the spectral method which also uses a single run, takes into account the correlation between data gathered by the simulation in successive time periods. This is described by Heidelberger [10].

Reviews of simulation objectives and their impact on simulation studies have been written by Kurose [3] and Righter and Walrand [11].

1.3 Simulation Speed-up

So far we have discussed reasons for the complexity, size and slowness of telecommunication network simulations; we now need to consider methods of speeding them up. Simulation run times can be reduced by three methods which may also be used together. The use of parallel multiprocessor systems, the integration of analytical models into the simulation where possible (as their solution is faster) known as hybrid simulation, and the use of

statistical methods known as variance reduction techniques.

In parallel simulation, the ultimate goal is to obtain a simulator that runs as quickly as possible; if the speed of the parallel simulator is less than that of a conventional simulator then there is no reason for using it (and many good reasons for not doing so). However, it is normally impossible to compare parallel and sequential simulators directly since the two are written in an entirely different manner and the programmer rarely wants to write both. Simulation speed-up is defined as the time it takes a single processor system to perform a simulation divided by the time it takes a multiprocessor system to perform the same simulation. This definition is incomplete however, as we have to specify what we mean by performing the simulation on a single or a multiprocessor system. We might think that a safe definition would be to specify the use of the same type of processor for the single and multiprocessor systems. However, memory requirements would dictate that the single processor system would need a much larger memory than any individual processor in the multiprocessor system: we must therefore allow for this. Another method of estimating speed-up is to measure the total busy time of each processor during the parallel simulation. This would give a greater speed-up measure than is actually available in distributing the simulation over multiple processors.

A good indication of the possible behaviour of the conventional simulator can sometimes be obtained by running a version of the parallel simulator optimized to run on a uniprocessor. The time taken for the uniprocessor version can be compared with that of the multiprocessor version. The *speed-up* of the simulator is then the ratio of the time for the multiprocessor version to that for the uniprocessor. Most processors will perform badly though due to the overhead of multiple task scheduling necessary.

The speed-up should normally lie in the range between one and n when the multiprocessor version is run on n processors. A speed-up of n is said to be *unitary*, and the success of the parallel simulator is measured by how close we can approach to unitary speed-up. A speed-up of less than one, *sub-linear*, would indicate that the uniprocessor simulation is faster; which is clearly unacceptable.

Helmbold and McDowell [12] have defined a family of useful qualitative terms for speed-up (see table 1.1) which includes terms for speed-up greater than n. *Super-linear* speed-up, where the measured speed-up grows without bound as the number of processors is increased, has not been observed practically, or even predicted theoretically, for any real application. *Linear super-unitary* speed-up however, has been observed; but not for parallel

simulation [12]. Such super-unitary applications are characterised by one of two factors. Firstly, the sequential algorithm used by the uniprocessor version may be constrained to an inferior method compared with that of the parallel algorithm. Thus, when the uniprocessor version uses the "same" algorithm as the multiprocessor, the super-unitary speed-up becomes unattainable. Secondly, the problem may be NP-complete[2] and the best known algorithm is some form of randomized search whose run time is highly dependent on the initial search point. When the search is done in parallel with multiple initial search points the expected run time will drop rapidly as the number of processors is increased.

Term	*Speed-up, $S(n)$*
sub-linear	$\lim_{n\to\infty} S(n)/n = 0$
linear sub-unitary	$0 < \lim_{n\to\infty} S(n)/n < 1$
unitary	$\lim_{n\to\infty} S(n)/n = 1$
linear super-unitary	$1 < \lim_{n\to\infty} S(n)/n < \infty$
super-linear	$\lim_{n\to\infty} S(n)/n = \infty$

Table 1.1: Qualitative terms for speed-up figures.

A practical argument against using parallel simulation is the well-known difficulty of debugging parallel programs. In some cases this is tackled by debugging a sequential version of the parallel program on a single processor using standard debugging tools. For instance, this technique can be used on a program before using a parallelizing compiler which produces parallel code automatically from a sequential program (see chapters 2 and 4). Certain microprocessors used in parallel multiprocessor machines, such as the Inmos transputer, can effectively support the running of an actual parallel program on a single processor (given sufficient memory) by task switching. The debugging process is then one step closer to true parallel debugging. In practice, the problem of debugging a parallel program whilst running on a multiprocessor can only be addressed by either writing a specialised parallel debugging tool for the particular multiprocessor, or, by embedding large amounts of tracing code within the program which prints out information on all the relevant details of the programs progress. Such detailed tracing will obviously affect the programs run time performance and will only be required during development and actual debugging. Therefore it is usually configured so that it can be switched on (or off) at run time or, preferably, at compile time. Indeed, this is the technique used by three parallel simulation tools[3] discussed in

[2]Non-deterministic Polynomial or NP-complete problems are not solvable in polynomial time by any known deterministic algorithm and the only guarantee of a globally optimum solution is by exhaustive search.

[3]YADDES (Yet Another Distributed Discrete Event Simulator) described in chapter 5, TWOS (the Time

later chapters.

In hybrid simulation, a simulation model is combined in some way with an analytical model. As the solution of analytical models is generally much faster than that of simulation models we will get a net speed-up. Hybrid simulation can be approached by decomposition of the simulation model or by replacing the simulation model with an analytic model, known as conditional expectation. The simulation model can be decomposed into sub-models, often in a hierarchical manner, and those parts which have known analytical models are solved analytically. A little intelligence is also usually applied to the decision of which simulation sub-models to replace by considering which sub-models have most bearing on the performance measure(s) of interest. Those sub-models requiring modelling in greater depth, or to a greater accuracy, are not replaced. Conditional expectation is useful when an analytic model of a complete system is available but when the vital parameters are not known but can be simulated. In this case the simulation model feeds the analytic model in a subordinate manner.

The speed-up obtained by using hybrid simulation techniques is offset somewhat by its disadvantages. The application of these techniques is highly dependent on the model. If analytic models are not known for a given situation then a significant amount of expert knowledge will be required in order to explore whether one is possible and to generate it if it is. In moving to analytic models there is also a consequent loss of the detail compared with that in the simulation model. This may, or may not, be significant; again expert knowledge is required to assess this. The loss of detail can also mean that the variance of the results obtained from a hybrid simulation is significantly increased. For instance, each simulated traffic source in a packet-switched network model may be replaced by a simple random number generator and an appropriately chosen shaping distribution. Thus, some accuracy is lost in return for faster model execution.

Hybrid simulation is a subject in its own right, but does not depend on the use of parallel or distributed architectures, though it may be used in such an environment. General discussions of hybrid simulation have been written by Shanthikumar and Sargent [13], Frost et. al. [4].

Variance reduction techniques exploit the statistical nature of simulation models to reduce the variance, or uncertainty, in the output results. Thus, for a given length of

Warp Operating System) described in chapter 3 and the University of Durham ATM simulator described in Chapter 6.

simulation run-time a more accurate result is obtained, or, for a given accuracy a shorter simulation run is possible. This is usually described as statistical speed-up. There are many well known techniques that fall into the category of variance reduction techniques such as, antithetic sampling, common random number streams, control variates, importance sampling and stratified sampling. These techniques have been studied extensively and are well documented but have not often been used in real simulation applications. This may be due to ignorance or, more possibly, to the added complication of their use. Also, most experiments using variance reduction techniques have reported only modest speed-ups. An excellent survey of variance reduction techniques has been written recently by McGeoch [14].

Statistical speed-up can be exploited in multiple processor systems. In this case, we define speed-up as the time taken for a single processor to obtain the estimated performance measure(s) of interest to a given accuracy divided by the time taken for the multiple processor system to obtain the same estimate to the same accuracy. We can now consider using n processors to do n short simulation runs instead of one long one on a single processor. Heidelberger [15] discusses these statistical speed-up issues. His, and other work in the area, will be discussed in chapter 2.

1.3.1 Amdahl's Law

The speed-up, $S(n)$, of a multiprocessor with n processors is defined to be the ratio of the total execution time on a uniprocessor to the total execution time on the multiprocessor. Let us first define a simple model for the execution of a parallel program on a multiprocessor. We define the total number of operations performed by the complete program as W. Various types of operation are involved, so we define W_i as the number of times that operations of type i are performed. Each operation has a cost in processor time, so we can define $C_i(n)$ as the time taken by all the processors to perform one operation of type i on an n processor system. Let W_s represent the number of sequential operations and $C_s(n)$ the cost of performing a single sequential operation on an n processor system. Likewise, let W_p and $C_p(n)$ be the amount and cost of work which exhibits n-fold parallelism. Therefore, speed-up is defined as;

$$S(n) = \frac{W_s C_s(1) + W_p C_p(1)}{\frac{W_s C_s(n)}{n} + \frac{W_p C_p(n)}{n}} \tag{1.1}$$

If we define the cost of one sequential operation to be one unit of processor time, then

$C_s(1) = 1$. Also we can assume that when performing a single sequential operation on an n processor system, one processor is working and the other $n-1$ are idle; hence $C_s(n) = nC_s(1) = n$. Further, if we assume that the cost of one operation is the same for sequential and parallel operation, $C_p(1) = 1$ and $C_p(n) = 1$, then equation (1.1) reduces to Amdahl's law [16].

$$S(n) = \frac{W_s + W_p}{W_s + \frac{W_p}{n}} \tag{1.2}$$

Amdahl's law for various values of W_p, where W_p and W_s are expressed as a fraction of the total number of operations (ie. $W_p + W_s = 1$), is shown graphically in figure 1.1. Comparing this with our qualitative terms for speed-up in table 1.1, we can see that the curve for $W_p = 1$ is effectively the unitary speed-up curve. Above this, is the region of super-unitary speed-up and below it, sub-unitary. Also, notice that for any $W_p > 0$, the speed-up is never sub-linear.

Amdahl's law also leads to a theoretical upper-bound for speed-up for this model of;

$$\lim_{n\to\infty} S(n) = \frac{W_s + W_p}{W_s} = \frac{1}{1 - W_p} \tag{1.3}$$

These can be observed in figure 1.1 (particularly for values of $W_p \le 0.95$) as the curves move asymptotically to the upper bounds given in table 1.2.

W_p	*Speed-up* - $S(\infty)$
0.25	1.33
0.50	2
0.75	4
0.90	10
0.95	20
0.99	100

Table 1.2: Upper bounds on speed-up predicted by Amdahl's law

There are several consequences of Amdahl's law and the predicted upper bounds. First of all, this model does not take any account of any overheads. In the case of a shared memory multiprocessor, for instance, there will be overheads in synchronization, communication and memory reference overheads due to local cache misses. Also, whatever the architecture of the multiprocessor, there is invariably a small period of time at the beginning and end of any program which is unavoidably sequential; even if it only consists of reading code and data files from mass storage, downloading them to each processor and writing the final

results back. These two factors on their own mean that the curves in figure 1.1 and the "upper bounds" in table 1.2 are in reality upper bounds on the speed-up we can expect to achieve. Other, more practical, limits on the speed-up achievable are the number of processors available and the number of processes into which we can decompose the problem; in this case the simulation model. It may often be possible to decompose a model into more processes (ostensibly to exploit more parallelism) but this may result in greater overheads in communication between processes/processors, synchronization and load balancing problems. Such problems are addressed in chapters 2 and 3.

1.4 Parallel Hardware Architectures

The last decade has seen the advent of a huge variety of new computer architectures for parallel processing. This variety can be bewildering to the non-specialist in computer architecture who needs to know which architecture is the most suitable for his application. In order to make an informed choice we need to be able to classify the different types of architecture which are available and which are possible. This is usually addressed by a taxonomy. A taxonomy is a classification scheme based on the salient features of the things being classified: in this case the things are parallel computers and the salient feature is the type of parallelism employed. In any computer system design the parallelism can be achieved at various levels. These are defined as the task level, process level, instruction level and arithmetic or bit level.

The task level is the highest level at which parallelism can occur. It occurs in multiprogrammed systems where multiple tasks are executed concurrently using time-sharing. True parallelism occurs when we have multiple processors and each task is allocated to a separate one. A task is usually defined as a complete program, though it should be obvious that it is possible to define a task as a complete phase of a program. Thus there is an overlap with the next lower level of parallelism. Here we are clearly describing multiprocessor systems.

The process level is usually taken to mean parallel processing at the sub-program, procedure or sub-routine level. Also included are separate instances of loops and even iterations of loops, if there is no data dependency. At this level we are describing vector, array and multi-function processors. Instruction level parallelism is where individual instructions are executed in parallel. This is normally described as pipeline processing.

The final, and lowest, level is the arithmetic or bit level. A good example is the widely

used parallel adder. This level is more the domain of the integrated circuit logic designer rather than a primary topic in a discussion of parallel processing. So we will restrict ourselves to the three higher levels of parallelism.

Duncan [17] believes we should go further and "exclude architectures incorporating only low-level parallel mechanisms that have become common place features of modern computers". In this category he places instruction pipelining (for overlapping decode, fetch, execute and store operations) and co-processors (for mathematics and logic or input/output control).

There are several classification schemes available, the most popular of which is the taxonomy proposed by Flynn [18]. The problem with any formal classification scheme is that several well established architectures do not fit into them very neatly. This has led to several suggested modifications to Flynn's taxonomy and alternative approaches based on descriptive notations. The taxonomy used in this survey is that of Duncan [17]. This is an informal high-level taxonomy based on Flynn's work which distinguishes between the principle parallel computer architectures which are currently being explored.

Let us first briefly introduce Flynn's taxonomy. Flynn [18] considered the sequential (often called the von Neumann) model as a single stream of instructions controlling a single stream of data (hence SISD). One step towards parallelism can be made be adding multiple data steams (SIMD), and a second by adding multiple instruction streams (MIMD). The SIMD machine consists of a single control unit and several processors carrying out the same instructions on many different items of data. These are usually array or vector processors. The MIMD machine is a *true* multiprocessor machine with each processor executing its own instructions on its own data. The MISD machine, multiple instructions acting on the same data item, is sometimes deemed impractical, though it seems to describe a multiprocessor pipeline. The taxonomy is summarised in table 1.3 below.

	Single data stream	*Multiple data stream*
Single instruction stream	SISD (von Neumann)	SIMD (Array/Vector processor)
Multiple instruction stream	MISD (Pipeline)	MIMD (*true* multiprocessor)

Table 1.3: Flynn's Taxonomy.

Flynn's taxonomy has the attraction of simplicity, providing a useful shorthand, but is insufficient for classifying many modern architectures of which almost all would be classed

as MIMD. Therefore many modifications to it have been suggested. Duncan summarises much of the work on extending Flynn's taxonomy in his paper [17]. He also introduces an informal high-level taxonomy, based around Flynn's taxonomy, to distinguish between principle approaches to parallel computer architectures. This is shown in figure 1.4.

<table>
<tr><td rowspan="4">Synchronous</td><td rowspan="2">SIMD</td><td>Processor array</td></tr>
<tr><td>Associative memory</td></tr>
<tr><td colspan="2">Vector</td></tr>
<tr><td colspan="2">Systolic</td></tr>
<tr><td rowspan="2">MIMD</td><td colspan="2">Distributed memory</td></tr>
<tr><td colspan="2">Shared memory</td></tr>
<tr><td rowspan="4">MIMD-based paradigm</td><td colspan="2">MIMD/SIMD</td></tr>
<tr><td colspan="2">Dataflow</td></tr>
<tr><td colspan="2">Reduction</td></tr>
<tr><td colspan="2">Wavefront</td></tr>
</table>

Table 1.4: Duncan's Taxonomy.

Duncan begins at the highest level with three classes, Synchronous, MIMD and MIMD-based architectural paradigms. Synchronous parallel architectures co-ordinate concurrent operations in lock-step through the use of global clocks, central control units, or vector unit controllers. His MIMD class matches Flynn's MIMD class, in that multiple processors can execute independent instruction streams using local data. His MIMD-based paradigm class embraces hybrid architectures that do not comfortably fall into either Flynn's MIMD or SIMD classes. Duncan ignores SISD as he is only interested in parallel architectures. The following is a short summary of Duncan's classifications.

Vector processors are characterised by multiple, pipelined functional units which implement arithmetic and Boolean operators for both vector and scalar quantities and can operate concurrently. Thus parallel vector processing is provided by streaming many vector elements sequentially through a number of functional pipeline units in parallel. Because of the existence of the pipelines there is always a significant start-up overhead. So efficient operation is only achieved if the pipelines are continuously full. Also it has been found that the vector operand lengths need to be multiples of the vector register size (ie. the number of pipelines). If not then some of the pipelines will be idle while smaller vectors are processed and vectors which are too large will have to be processed in batches.

SIMD architectures normally consist of interconnected processors with a central control unit. The control unit broadcasts instructions to the processors which execute them in lock-step on local data. The interconnections allow communication between processors and

also between processors and local memory.

Processor array architectures are a type of SIMD architecture which are structured for numerical SIMD execution. These have been used extensively for large-scale scientific calculations such as image processing and nuclear energy modelling. The interconnection networks used have usually been nearest-neighbour meshes or crossbar approaches, though recently hypercube topologies have been explored. A variant on this architecture is the use of large numbers of 1 bit processors in processor grid arrangement. These are very much geared to image processing applications by mapping pixels to the grid processing elements.

Associative memory processor architectures are another a type of SIMD architecture based on the idea of data stored in an associative memory which can be addressed by its contents. The major advantage over random access memory (RAM) is its capability of performing parallel search and comparison operations. These facilities are frequently needed in many applications such as dynamic databases, image processing and artificial intelligence. The major disadvantage is the increased hardware cost.

Systolic architectures (arrays) are pipelined multiprocessors in which data is pulsed in a rhythmic fashion from memory through a network of processors before returning to memory. This idea was formulated to balance intensive computations with demanding input/output bandwidths. A global clock is used to synchronize the data flow through the interconnected processors. Each processor performs a specific invariant sequence of instructions on the data. A high degree of parallelism is obtained by pipelining the data through multiple processors typically arranged as two-dimensional arrays. Systolic architectures are best suited for algorithm specific applications, particularly in signal processing. Some machines have been constructed which are programmable, and so are not limited to implementing a single algorithm.

MIMD architectures, as defined by Flynn, use multiple processors to execute independent instruction streams using local data. Therefore MIMD machines are highly suited to applications that require processors to operate in a substantially autonomous manner. The software processes executing on each processor are synchronized by passing messages via an interconnection network or via shared memory. Thus there is no central control and MIMD architectures are asynchronous. They therefore support higher level parallelism at the task and process level. Another advantage is that they can be extended much more easily than SIMD machines. MIMD architectures can be further distinguished by their memory organisation.

Distributed memory MIMD architectures need the processing nodes (processor plus local memory, or cache) to be connected using some interconnection network. Nodes can then share data by explicitly passing messages through the interconnection network. Various static interconnection network topologies have been explored to support various applications, such as pipelines, meshes, trees, rings, chordal rings, cubes, hypercubes etc. Each of these are particularly suited to certain applications. For instance, a mesh topology is suited to matrix oriented algorithms, tree topologies are suited to searching and sorting algorithms. Dynamic, or reconfigurable topologies, are also possible by using a programmable switching matrix. These can be single stage, multi-stage or a crossbar. Interconnection networks are examined in detail by Feng [19], Siegel et. al. [20] and Almasi and Gottlieb [21]. A disadvantage is that the communication overhead associated with this architecture, particularly where data has to be queued and forwarded by intermediate nodes, can significantly reduce the performance.

Shared memory MIMD architectures allow communication between processors via a common shared memory which each processor can access. Shared memory architectures thus replace message sending problems with data access synchronization and cache coherency problems. To coordinate processors with shared variables a synchronization mechanism is required to prevent one process accessing a piece of data before another finishes updating it. Also, each processor in a shared memory architecture often has a local cache memory. Therefore we can have multiple copies of the same shared memory data in the caches of various processors. Maintaining consistent versions of the shared data is the cache coherency problem. As in the case of distributed memory MIMD architectures, there are several alternatives for the interconnection of the multiple processors to the shared memory. Some major examples are time-shared bus interconnections, crossbar interconnections and various forms of multi-stage interconnection network.

Duncan's category of MIMD-based architectural paradigms is meant to cover a variety of hybrid architectures which don't fit neatly into most orderly taxonomies of parallel architectures, including Flynn's. Though all the types included are based on the MIMD principle of multiple instructions and data streams and asynchronous operation, each has a distinctive fundamental operating principle.

A MIMD/SIMD architecture describes a machine which is a MIMD architecture but can be controlled in an SIMD manner. This is usually implemented using a master/slave approach. For instance, let us consider a tree-structured machine as an example. The

root processor of a sub-tree can act as an SIMD controller broadcasting instructions to descendant nodes which then execute the instructions on local data. This is a very flexible approach and one which is receiving a great deal of attention.

Dataflow architectures depend on an execution paradigm where instructions are enabled for execution as soon as all their data becomes available. This is known as a data-driven architecture, in contrast with the normal instruction-driven architecture with which we are most familiar. This means that the sequence in which instructions are executed is data dependent. This allows dataflow architectures to exploit parallelism at the task, process and instruction levels. Dataflow machines are a direct result of research effort into new computational models and languages to effectively exploit large-scale parallelism.

Reduction architectures are often referred to as demand-driven architectures and, like dataflow architectures, have their own execution paradigm. In this case, an instruction is enabled for execution when its results are required as data for another instruction which has been enabled for execution. Reduction architecture research has the aim of producing parallel architectures which will support functional programming languages.

Wavefront array architectures combine systolic data pipelining with an asynchronous dataflow execution paradigm. Wavefront and systolic architectures are both characterised by modular processors and local interconnection networks. However, the wavefront array architecture co-ordinates inter-processor data movement using asynchronous handshaking. Therefore, when a processor has finished its processing and is ready to pass on its data, it informs the next processor, and sends the data when it indicates that it is ready. Once the processor receives an acknowledgement that the transmission was successful, it can proceed to its next task. Thus computational wavefronts pass through the array in correct sequence.

More detailed discussions of parallel computer architectures can be found in Almasi and Gottlieb [21], Patterson and Hennessey [22] and Trew and Wilson [23].

1.5 Outline of Thesis

The main body of the thesis following this introduction is divided into five parts which are covered in the following chapters. Chapter 2 and 3 review parallel discrete event simulation in some detail with particular emphasis on work relevant to the modelling of telecommunication networks. Chapter 2 explores the various methods used to decompose a simulation model into a set of communicating parallel processes; highlighting their strengths and

weaknesses. Chapter 3 concentrates on a particular decomposition approach called here the distributed model components approach and the fundamental problem of synchronization. This involves decomposing the system model into loosely-coupled components (sub-models) and simulating each with a logical process. One or more processes are then allocated to each processor. The distributed model components approach is often referred to in the literature as parallel discrete event simulation (PDES) as it is by far the most popular approach. Chapters 4, 5 and 6 describe the results of the original work performed.

Chapter 4 describes the parallel simulation of circuit-switched telecommunication networks using a parallelizing compiler on a shared memory multiprocessor computer. Work has been conducted previously using this technique to attempt to speed-up parallel discrete event simulations of queueing networks with little success. Here, a simple model of a circuit-switched network, which is not a discrete event model, is used to explore the use of a parallelizing compiler. The results showed good speed-up figures with reasonably large networks of ten nodes and greater.

Chapter 5 describes the parallel simulation of closed and tandem queueing network models. Such models are often used as the basis for modelling packet-switched telecommunication networks. This work was conducted on a distributed memory multiprocessor computer using the YADDES[4] tool from the University of Waterloo, Ontario, Canada. This simulator enabled the comparison of the performance of several different parallel simulation synchronization approaches using common model specifications.

Chapter 6 describes the parallel simulation of asynchronous transfer mode (ATM) networks. This work was also conducted on a distributed memory multiprocessor computer using a simulator written at the University of Durham. Simulation studies of ATM systems have thus far largely centred on the behaviour of single traffic sources, multiplexors or switching nodes. Here, the parallel simulation of a complete network has been implemented allowing realistic network performance to be studied with reasonable simulation run times.

Chapter 7 draws all of the conclusions together and attempts to highlight areas which appear worthy of further study. There are also two appendices. Appendix A contains information on the availability of a floppy disk formatted for an IBM. PC. compatible computer. This contains examples of the simulation model files used in this study. Appendix B contains the journal and conference papers published as a result of this work.

[4]YADDES is an acronym for Yet Another Distributed Discrete Event Simulator.

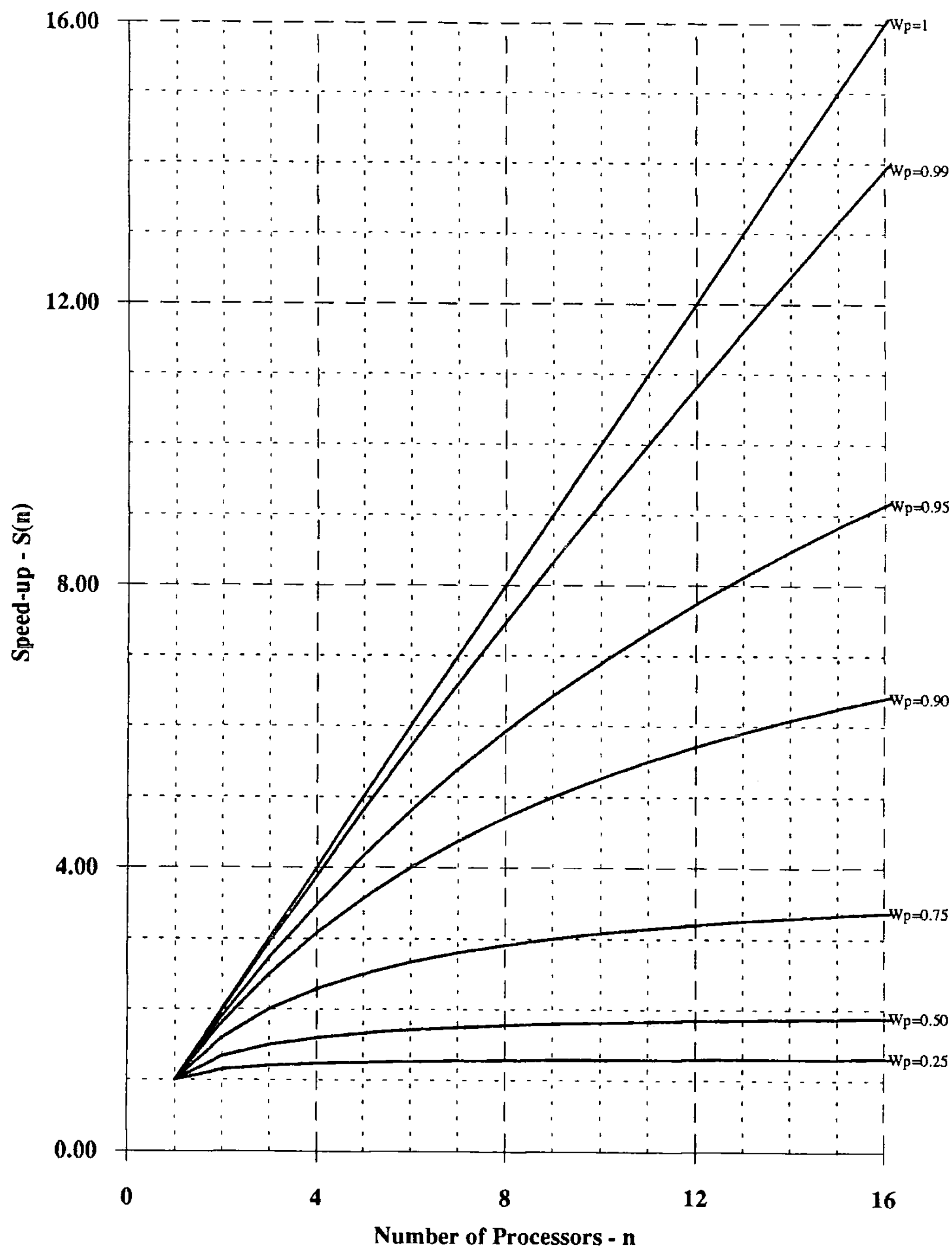


Figure 1.1: Amdahl's law

Chapter 2

A Review of Parallel Discrete Event Simulation

2.1 Introduction

A discrete event simulation is a computer program model for a system where changes in the state of the system occur at discrete points in (simulated) time. Thus, the simulation model only makes the transition from one state to another when an *event* occurs. The system state is represented by a set of state variables which each event may modify (thus changing the system state) and/or schedule new events in the future. An *event-list* is maintained containing all the pending events that have been scheduled but not yet processed. Each event has its own unique time-stamp and the simulation progresses by removing the event from the event-list with the lowest time-stamp. A global *clock* variable is used to keep track of how far the simulation has progressed.

This type of model is suitable for modelling many types of communication network. For a packet-switched network model, say, state variables may represent the amount of traffic carried on links, lengths of packet queues, processor status etc. Typical events may include, new call arrivals, call clear downs, packet arrivals at a switching node, packet departures after processing and routeing and, even, equipment failures. Such a system model is termed an *asynchronous* system in that events occur at irregular intervals which can be modelled by stochastic processes; that is, they are not synchronized by a global system-wide clock. For such systems, the occurrence of events with the same time-stamp is very small so that parallel processing techniques based on lock-step execution under the control of a global

simulation clock tend to perform poorly.

Parallel discrete event simulation (PDES) refers to the execution of a single discrete event simulation program on a parallel multiprocessor computer. For a given simulation, five techniques for decomposing it into parallel processes for execution on a multiprocessor machine have been identified [11, 24].

- A **Parallelizing Compiler** can be used to compile a sequential simulation, written in a conventional sequential language to run on a sequential uniprocessor, so that it will run on a particular multiprocessor machine.
- **Distributed Simulation Experiments** may be conducted by running separate simulations on separate processors in parallel. This is often called replicated trials.
- **Distributed Simulation Functions** involves different subroutines or tasks of a simulation being placed on separate processors. For instance processors may be dedicated to random number generation, event-list processing, statistics collection etc.
- **Distributed Simulation Events** uses a global event-list, as in sequential simulation, to schedule available processors to process the next event on the list.
- **Distributed Simulation Model Components** involves decomposing our system model into loosely coupled components (sub-models) and simulating each with a process. One or more processes are then allocated to each processor.
- **Time Parallelism** involves partitioning a simulation model into a set of time periods. Each time period is simulated in parallel and then the results from each period are joined, or merged, together. This joining process is performed so that the final set of results is the same as that which would be produced by a single long simulation run. This is radically different from all of the above approaches which may be said to exploit space parallelism.

Where the term PDES is found in the literature, it almost invariably refers to distributed model components, as it is by far the most popular. The term distributed model components approach will be used here to (hopefully) avoid any confusion. The good and bad points of all of the above decomposition methods are discussed in the following sections giving examples of work done in each area particularly emphasising work directly (or indirectly) related to the simulation of telecommunication networks. The section on the distributed

model components approach merely introduces the subject as it is dealt with in much more detail in chapter 3. Obviously some combined approaches are also possible which are some combination of the above techniques and these are also discussed.

2.2 Parallelizing Compilers

We are very familiar with the concept of a compiler which takes a high-level language source code and compiles it to produce the machine-level object code which runs directly on a sequential uniprocessor system. One approach to tackling the production of code for a multiprocessor computer is to apply a compiler which takes, as its input, a conventional sequential high-level language and produces, as its output, the object code to run on each of the multiprocessors. This is termed a parallelizing compiler. The compiler thus has the responsibility to recognise sequences in the source code which can be executed in parallel and scheduled to run on separate processors. This definition thus distinguishes a parallelizing compiler from a compiler which takes a high-level parallel language and compiles it to run on a multiprocessor computer. Though such a compiler could be made responsible for automatically assigning processes to processors. An excellent survey of such compilers and automatic parallelization generally has been produced recently by Bannerjee et. al. [25].

The overwhelming advantage is that the approach is largely transparent to the user. A new parallel language does not have to be learned, the multiprocessor architecture should not impact the original program structure and existing sequential software, so-called *dusty decks*, may be ported. The disadvantage often found is that the problem has been coded in sequential form, thus largely ignoring any parallelism in the structure of the problem. This results in relatively small portions of the available parallelism in the problem being exploited and, hence, the speed-up in moving to the multiprocessor architecture is generally disappointing.

Parallelizing compilers are usually constructed such that they are effective in translating a wide range of sequential programs for use on the particular multiprocessor. Indeed, many of the algorithms used in the program analysis and transformation may well have been developed with many classes of scalable parallel multiprocessor in mind. This means in practice that a parallelizing compiler written for and running on a particular architecture is unlikely to give the best performance in terms of speed-up for *one* particular application. The best performance will always be obtained by writing the application from scratch.

There are two basic approaches to converting sequential code to run on multiprocessor architectures. A parallelizing compiler takes sequential code directly and produces parallel code to run on the target multiprocessor system. Alternatively, intelligent run-time support and parallel routine libraries can be provided with a programming environment which allows the conversion of sequential into parallel code.

The former approach is exemplified by the work of Chandak and Brown [26] and by Reed [27]. The work done by Chandak and Brown, showed that discrete event simulations cannot generally be parallelized using this approach and, more specifically, that the discrete event simulation of any network of queues containing feedback loops cannot be parallelized. As most simulation models of telecommunication networks involving queues are almost bound to contain feedback loops this was a disappointing result. On a positive note though, they showed that careful optimization of event-list processing could produce a speed-up of two on a CDC Cyber 205 (a synchronous vector architecture, see Duncan [17]) even for non-parallelized code.

This result was borne out by Reed. He used a Cray X-MP (also a synchronous vector architecture) and Cray's Fortran 77 vectorizing compiler to investigate the simulation of queueing networks. The results were compared with the simulations performance on a Vax 11/780 using the same sequential code. The results showed a speed-up of about a hundred which is almost the same given by the two computers rated performance on sequential code. It was suspected that the amount of vectorization was small and using an execution monitor it was found to be between 1 and 5% for various simulation models.

Nevertheless, Reed did hold out some hope for this approach as he believed the current limitations were with the Cray Fortran vectorizing compiler. There were many DO loops in the simulation which potentially could have been parallelized but weren't because the compiler couldn't parallelize any loops containing IF statements, function calls or data dependencies; even by automatic rearrangement of the code. Indeed there have been some significant advances in parallelizing and vectorizing compiler technology (eg. the Cray Fortran 77 compiler [28], PTRAN at IBM Research [29] and SUIF at Stanford University [30]) and associated performance monitoring tools since then [31–35]. A more recent trend is towards integrated development environments; a good example of which is the ParaScope system [36]. All this points to the approach becoming increasingly more attractive; particularly as many shared memory multiprocessor machines are becoming available which can function as general purpose Unix machines as well as parallel processing platforms [23].

The second approach, of providing intelligent run-time support and parallel routine libraries has been taken by several commercial products such as Express, Linda and Helios. Express [37,38] is based on work done originally at the California Institute of Technology. Code development is done by first achieving a working sequential C or Fortran program. Then, either system calls to Express library routines can be added by hand, or ASPAR, an Automatic Symbolic PARallelizer, can be used to generate parallel C or Fortran code with the built-in calls to Express library routines. ASPAR also provides decomposition, processor mapping and load balancing facilities. It is recognised in [38], that ASPAR works extremely well on applications which use regular meshes, but can fail on communication intensive applications. However, this disadvantage is slightly offset by diagnostics which are issued on failure to suggest corrections to the problem to allow parallelization. Debugging and testing of the parallel code is aided by the tool NDB. There is also a performance monitoring tool, PM, which gives graphical output displaying processor utilisation and communication.

The basic concept used in Linda [39] is the idea of a "tuple-space", which is effectively a database for software objects of various kinds. Processing nodes communicate by dropping objects into the database which can then be extracted by other nodes. This concept has a very simple and elegant implementation but it suffers from quite severe performance problems. This is particularly so on distributed memory MIMD architectures where the database searching required to find an object can involve intensive inter-node communication within the operating system. More recent versions of Linda [40] have extended the original concept by adding additional "tuple-spaces" and allowing the user to specify to which space an object should be sent and from which it should be retrieved. This new style is similar to a "mailbox" approach.

Helios [41] is a distributed memory MIMD architecture operating system designed for Inmos transputer networks. It offers Unix-like utilities such as compilers, editors and printers all available from the nodes of the transputer system. Interprocessor communication is achieved using "pipes" though no collective communication support is provided. No automated programming or parallelizing tools are provided.

A research project at Texas Christian University at Fort Worth has used a pre-processor approach similar to that of Express. A Pascal based event oriented simulation language SIMPAS was used and the sequential code was passed through a pre-processor to produce Pascal code which could be compiled to run on a Texas Instruments 990/12 multitask-

ing minicomputer or a tightly coupled network of eight Motorola 68000 based processors connected using a VME bus [42]. See also the section on distributed simulation functions concerning this project.

Another interesting approach is the development of a hardware architecture which specifically supports a high-level language. This has been done for Lisp, Prolog and Simula amongst others. Both a Parallel Simula machine [43] and a Prolog co-processor machine [44] have been used for simulation purposes. Performance figures are not quoted for either machine.

2.3 Distributed Simulation Experiments

If we have n multiple processors available, an obvious approach is to do n experiments (simulation runs) simultaneously. This is particularly efficient for stochastic simulations, as results can be averaged at the end of the run, and also for doing several "what-if" simulations simultaneously with slightly different parameters. This approach seems extremely efficient as no co-ordination is required between processors except for result averaging or presentation. Hence, for n processors we may approach an ideal speed-up of n. The only other overhead is loading the model into each processor which is often negligible for production runs.

Let us first consider the case of stochastic simulation. In some cases it is better to do a single long run, in terms of simulated time, than to do several short runs and average the results. Heidelberger [15] considered the statistical properties of estimates obtained from discrete event simulations run on parallel processing computers. He showed that, generally speaking, if the run length required for a particular accuracy is long or the initial transient is weak, then distributed simulation experiments will be statistically more efficient than distributed simulation for estimating steady state quantities. If a reasonable speed-up factor can be attained using the distributed simulation, it will be statistically more efficient than distributed experiments for short runs, for simulations with a strong transient or if a large number of processors are available. In between these two extremes (ie. moderate transient relative to the required run length), the optimal policy, he suggests, is a combination of a small number of processors per experiment and a large number of experiments.

Heidelberger's principles have been explored by Rajagopal and Comfort [45] simulating an M/M/c queueing network using occam on Inmos transputers. A combination of

distributed simulation functions and distributed simulation experiments was used. The speed-up obtained using the distributed experiments was near perfect ignoring initial loading of the processors and data collection at the end of the simulation. In general, the steady state was reached more quickly but, surprisingly, the mean squared errors were not significantly different. It was concluded, however, that this may have been due to the small number of processors employed (two and three sets of three).

Independent simulation runs with different parameters have been explored by Biles, Daniels and O'Donnell [46]. They used a hierarchical tree network of microcomputers, in which lower levels performed the same simulation with different parameters. Again, this is very efficient, but does not allow interactive decisions to be made. This is because decisions about all the parameters for all the runs to be executed in parallel must be made before they take place. Thus, optimization of a simulation, by examining results of previous runs, is more difficult. Also many of the simulations in a particular batch may be wasted because of this hindrance to sequential decision making. To summarise, a few short fast single runs are often better for optimization and decision making than making many slow ones in parallel. In terms of the hardware required, distributed experiments may not be possible. Distributed experiments require that we have multiple identical processors each with sufficient memory to contain the whole simulation program as well as enough memory to run it. This suggests an MIMD architecture that is loosely coupled, as communication between processors is not generally required during simulation. In most distributed memory systems each processor does not have large amounts of local memory. Even in a shared memory system there may not be enough memory capacity for many copies of the simulation program. This leads us back to considering, as did Biles, Daniels and O'Donnell, a network of uniprocessors. Nevertheless, if these memory deficiencies do not apply to the particular simulation application, the distributed experiments approach can be very efficient and can also use existing sequential simulation programs adapted for this purpose.

2.4 Distributed Simulation Functions

For any simulation there are always a certain number of key activities which must be supported. Functions such as random number generation, event processing, result collection, statistical calculations, results presentation, file manipulation etc. Also other functions may be desired such as animated graphics during simulation or intelligent supervision of

the simulation process. Each of these functions may be supported by distributing them to individual processors. The processors may be identical, or may be tailored to each individual function. This is very much like the approach taken in many personal computers; a general purpose processor, an arithmetic co-processor for floating-point calculations, a graphics co-processor, other processors for controlling input/output functions with keyboards, printers or communications links.

The advantages of this decomposition method is its freedom from the possibility of deadlock (a cycle of blocked processes) and its potential scalability. The architecture may also be made transparent to the user as each function's code can be divided up and placed with each processor fairly easily. It could even be made an automatic process at compilation. This would obviously be much easier if identical processors were used.

Its disadvantages are the communication overhead between functional processors, which may become the limiting factor in performance, and the failure to exploit any parallelism in the system being modelled.

At Texas A & M University work has proceeded in this area for several years [42,47–52]. Initially, the GASP IV simulation language was used and the simulation support processes were separated to run on eight separate processors. A tightly coupled network of eight Motorola 68000 based processors, connected using a VME bus was implemented. Unfortunately problems were found between the operating system and the multitasking Fortran 78, on which the GASP IV depended. The idea was abandoned. However, the idea was also emulated on a TI 990/12 minicomputer to determine the feasibility of the approach which was successful.

Comfort has also worked on this approach, see [53–55]. In [53] he describes a comparison experiment using a PDP-11/44 alone and using the PDP-11/44 with a Motorola 68000 microprocessor for event set processing, for the simulation of an M/M/k queueing network. He obtained a 40% improvement. In [54] he describes a three processor system achieving a 45% improvement over a single processor. In [55] a master/slave approach is described in which event set processing is done by slave processors controlled by a master host processor which does everything else. Speed-ups as high as 1.3 were obtained with two or three slave processors, but no incremental benefit was found beyond three.

Rajagopal and Comfort used this technique in the simulation of an M/M/c queueing network using occam running on Inmos transputers [56] and [45]. The speed-up factors obtained were 0.96 to 1.3 for two processor systems, 0.96 to 1.6 for three processor systems

and 1.2 to 1.6 for four processor systems. Also it was found that speed-up improved with event set size. Individual processor efficiencies were between 0.31 and 0.64. These results are encouraging though only a small number of processors were used. This was partly due to the transputer only having four physical links which limited the complexity of the model that could be simulated.

Rajagopal and Comfort's work, particularly, seem to point to this approach being fruitful for a small number of processors. Unfortunately, the benefits only increase marginally with the number of processors and may even fall away. More recently, this level of limited speed-up on a small number of processors was also reported by Zhang, Zeigler and Concepcion [57, 58]. They used a hierarchical decomposition of the simulation model in order to produce events, each consisting of several sub-events, to be processed in parallel.

2.5 Distributed Simulation Events

In sequential simulation, a global simulation event-list is usually maintained. The next event is the one at the top of the list. When this is processed, any events generated by it are inserted into the list at the correct time slot. In the distributed simulation events approach, a global event-list is maintained in the same manner, but as each processor becomes available it processes the next scheduled event. The difficulty is maintaining consistency in the simulation as the next event available on the list may be affected, or pre-empted, by another event currently being processed by other processors. The need for global simulation control points very much towards the use of a shared memory MIMD architecture so that all processors can have access to the global event-list.

Jones describes the distributed simulation events approach as concurrent simulation using temporal decomposition [59]. He develops the concept of "limit times", which determine whether the next event on the list can be safely executed. However there are no results for his algorithm.

Cota and Sargent have developed an algorithm for parallel discrete event simulation using a shared memory architecture machine [60]. It has been implemented on an Encore Multimax machine though no performance results are available as it was still under evaluation.

The Texas A & M University has put a great deal of work into developing a concurrent simulation environment using Ada on a Sequent Balance 8000 shared memory multiproces-

sor (twelve) machine [51,61]. Generally it was found that if tasks communicated very little, significant speed-ups over sequential simulations could be achieved. However, if the tasks had many rendezvous then the simulation essentially became sequential.

The results for this approach seem to indicate that it is reasonable if there are only a small number of processes required and a large amount of global information used by the components of the system.

2.6 Distributed Simulation Model Components

The final, and most popular, method of decomposing a simulation is to decompose the simulation model into a number of components, or sub-models, and assign the simulation of each component to a process. One, or many, processes can then be assigned to execute on each processor. Each process must be able to communicate with other processes so that either global shared variables or a message passing scheme between processors is required. Model decomposition often follows the logical structure of the real system being simulated. For instance, each exchange in a packet-switched network may be modelled by an individual process.

This approach can take advantage of any parallelism inherent in the system to be modelled, so it seems to promise significant speed-up on a multiprocessor system. However, this only holds if the simulation does not require a significant amount of global information and control. The major overhead will be communication between processes executing on different processors. This can be handled by message passing on a distributed memory MIMD architecture or shared variables or message passing on a shared memory MIMD architecture. The other major problem is the synchronization of events during the simulation. Generally speaking, the more loosely-coupled the processes can be (ie. asynchronous requiring little communication), the more likely the simulation is to be processor bound. That is, the performance of the simulator will be limited by processor performance. On the other hand, the more tightly-coupled the processes are, the more likely the simulation is to be communication bound. That is, limited by the performance of the communication mechanism between processors.

For a network which has a fixed topology, such as the public circuit-switched network, the most natural way to decompose the model is to assign a process to each exchange, or node, and represent network traffic by messages passing between the nodes. These messages

could be implemented in a shared or distributed memory machine. Each message would contain a time-stamp representing its simulated time of generation. An alternative approach is to have processes representing the traffic status as well. In this case messages passed will represent a change in traffic status.

In a dynamic topology system, such as a mobile telephone network, processes can represent interacting components; the users, the fixed transmitting/receiving stations and the normal telephone network exchanges. Messages between processes will then represent interactions between the components. Another method, originally developed by battlefield simulators [62] is to divide the simulation geographically into sectors. Each sector is then simulated by a process and then the processes are mapped onto the processors. At this point it has been found that there are trade-offs between placing adjacent sectors onto the same processor and placing non-adjacent sectors on the same processor. In battlefield simulations, the former minimises communications, but the latter produces better load balancing between processors. For communication systems, it is suspected that there will be little or nothing to choose between these sector placement schemes.

The two major tasks with distributed model components is the development of the model processes themselves and the synchronization of the processes during simulation. Model building is purely a software problem; synchronization is a problem of both simulation and software. As we shall see the method employed to synchronize the distributed model impacts the way the model is developed, the facilities required of the software and the hardware, and most importantly the performance of the simulation. The performance is affected as it is the synchronization message passing overhead which prevents ideal speed-up. A detailed discussion of synchronization approaches for distributed model components and its impact on performance can be found in chapter 3.

2.7 Combined Approaches

The ideal decomposition approach for a particular application may well be a combined approach integrating two or more of the above. Several scenarios are possible. Heidelberger [15], for instance, discusses the relative merits of combining distributed simulation experiments with other approaches using multiple processors to execute single simulations. The use of a parallelizing compiler could actually lead to a distributed event approach depending on how the compiler divided up and scheduled the processes. However, it is difficult

to imagine how these two approaches could interact with the other approaches. But let us now consider what approaches might work effectively together.

We could begin by decomposing our simulation model into our loosely-coupled components and modelling each with a process. Then, instead of placing each component in a single processor, we could decompose each process into its simulation functions and place each function in a processor. Each component process will thus be executed in a cluster of processors. This seems a useful approach as the research on distributed simulation functions seems to be most efficient using a small number of processors. Also we are exploiting the parallelism of the system at a fine-grain size. The disadvantages will be with code generation, loading and lack of flexibility and scalability. The distributed simulation functions approach could also be exploited alongside distributed model components by using extra processors to handle global results collection, statistical calculations and animation. This particular approach has been used at Durham University in the simulation of wide area telecommunication networks [63–66]. Here each node of the communication network is simulated on an Inmos transputer and the system set-up, graphics, results and user interface are handled by other processors.

The distributed experiments approach may be combined with the distributed functions or distributed model component approaches, or both. This simulator could then run several different simulations in parallel as well as exploiting the parallelism in each simulation. The approach using distributed experiments and distributed functions has been explored by Rajagopal and Comfort [45], and described above in the sections on distributed simulation experiments and distributed simulation functions.

2.8 Time Parallelism

The most obvious form of parallelism in physical systems, exploited as distributed model components, is due to concurrent activity among spatially separated objects. This is sometimes referred to as space parallelism. If we model a hundred physical objects with a hundred logical processes then Amdahl's law predicts that our maximum speed-up is also a hundred. However, our realistic speed-up, due to some inherently sequential sections and synchronization and communication overheads will be considerably less.

It has recently been recognised that parallelism in some simulation models can also be exploited in time. This was first discussed by Chandy and Sherman [67], where the

authors observe that simulations are fixed-point computations, and as such can be executed as asynchronous-update computations. Practical exploitation of time parallelism was first reported by Greenberg et. al. [68], where they showed how certain queueing systems (eg. a single FCFS G/G/1 queue and networks of feed-forward queues) can be expressed as systems of recurrence relations (in the time domain) which can be solved using standard parallel prefix methods on massively parallel machines. The most impressive thing about this approach is that the degree of parallelism which can be exploited is only limited by the size of the parallel machine and its memory.

A more direct approach to time parallelism is to partition the time domain, assigning different processors to different regions of time. A process, LP_n, assumes some initial state for the system at the beginning of its assigned interval, t_n say, and simulates it. The processor LP_{n-1} whose interval terminates at t_n may have a different final state at t_n than the one assumed by LP_n. In this case a fix-up operation must be performed. This method will work if the cost of a fix-up is much less than the cost of re-simulating the interval. Variations on this idea are described by Ammar and Deng [69] and by Lin and Lazowska [70]. This technique is easier for trace-driven simulations, where an input file is available which contains a trace of all the inputs to the simulation. Often the trace is one obtained from a real system. This technique has already been explored for cache simulations [71, 72], queueing networks [69] and packet-switched multiplexers [73]. It also has potential for digital logic circuits.

These techniques are still in their infancy, though they have already been applied to a variety of systems as shown below in table 2.1.

Author(s)	*Ref.*	*Application*
Ammar and Deng	[69]	Queueing networks
Baccelli and Canales	[74]	Stochastic petri nets
Gaujal et. al.	[75]	Circuit-switched telecommunication networks
Heidelberger and Stone	[71]	Trace-driven cache simulations
Nicol et. al.	[72]	Trace-driven cache simulations

Table 2.1: Applications explored using Time Parallel approaches.

Time offers another dimension in which speed-up may be obtained. However, as yet there are few implementations and any speed-up will rely heavily on the specifics of the application. This is not surprising, given the diverse ways in which simulation models evolve in simulation time. Therefore, it seems unlikely that a general purpose protocol will be developed that can be consistently effective in exploiting time parallelism for a variety

of applications. Nevertheless, as seen above, there are already some non-trivial examples of important applications that have been shown to benefit from time parallelism. Future efforts will be directed towards expanding the class of applications where time-parallelism works. The applications which do benefit need to be characterised so that the techniques used can be generalised. Much work also needs to be done in the performance analysis of such approaches.

2.9 Summary

The speed-up possible in using parallel simulation is clearly dependent on many interrelated factors. The application, the multiprocessor architecture, the decomposition approach, the synchronization approach (if applicable) all have a direct bearing on the simulators performance and also interact with each other in a complex manner. As has been discussed in chapter 1, some of the decisions concerning these factors will be made on what is required from the simulation. For instance, if real-time decisions need to be made by observing the behaviour of the simulator using graphical animation of the system, time driven synchronization will be needed and, for instance, the distributed experiments approach will never be appropriate. However, ignoring these complex interrelated problems for the moment, let us summarise the performance of the parallel simulation approaches.

The use of a parallelizing compiler tends to ignore the potential parallelism in the structure of the problem if sequential simulation code is translated into parallel code automatically. This results in relatively small portions of the available parallelism in the problem being exploited and, hence, the speed-up in moving to a multiple processor architecture is generally disappointing. Most compilers in this category, to be fair, have been aimed at synchronous SIMD and vector machines. Thus, they have been tailored to work with vectors, matrices and generally fine-grain problems inappropriate for communication network simulations. However, the work in this area has given rise to multiprocessor operating systems and development environments which should make life easier for the performance engineer wishing to use a multiprocessor to speed-up the execution of some simulation models.

The distributed simulation experiments approach is a simple but effective technique which, if coupled with variance reduction techniques, can give impressive performance. The relative unpopularity of the approach is perhaps because the most common need from a simulation is fast accurate results. As we are not exploiting any parallelism in the problem,

speed-up is more in terms of statistical efficiency and simulation throughput (ie. more simulation can be done in the same time). For comparing many similar designs at once it is probably the ideal approach. Also the performance engineer will probably be able to use existing sequential simulations as a basis.

The work on the distributed simulation functions approach seems to indicate that this is a fruitful approach if the number of processes that the simulation is decomposed into is small. The law of diminishing returns sets in at an early stage above more than a handful of processors.

The distributed simulation events approach depends very much on the use of a shared memory MIMD multiprocessor. The results for this approach seem to indicate that it is reasonable if there are only a small number of processes required and a large amount of global information used by the components of the system. Otherwise the results seem somewhat disappointing.

The distributed simulation model components approach offers the greatest potential speed-up in terms of a reducing the execution time for a single simulation. Also, the decomposition of the simulation model can follow the structure of the problem making it easier to understand and develop the models. In the case of the simulation of communication networks, we may be producing something akin to a software emulation. The synchronization problem complicates the approach, but guidelines are beginning to appear as to which synchronization method will be the best for a particular type of application.

The ultimate speed-up for any particular application is more than likely only available using a combined approach. The parallelizing compiler and distributed simulation events approaches do not lend themselves to combination with any of the other techniques though they may be interrelated themselves. Distributed simulation functions seem to give good results particularly when combined in order to relieve the rest of the simulation of the ancillary functions of statistical calculations, graphics control and results processing. An ideal mixture would seem to be multiple simulation experiments, each executing on multiple processors, feeding output results to other global processors for processing and display.

Time parallelism is a relatively new area potentially offering substantial speed-up on massively parallel multiprocessor machines. It seems unlikely that it will be a generally applicable technique but certain problems which lend themselves to this type of approach may yield good results.

Chapter 3

Synchronization Approaches for Distributed Model Components

3.1 Introduction

A significant amount of work has been done to ascertain the most efficient method of synchronizing the simulation of systems using distributed model components. Reviews of the plethora of possible schemes, in chronological order, can be found in [24, 76–79].

Before we discuss various synchronization schemes, it is important to review why it is such a difficult problem. In a sequential simulation, the synchronization of the simulation is maintained by manipulation of the event-list. This contains the pending events in the system usually (but not necessarily) in time-stamped order. The simulation progresses by removing the event with the earliest time-stamp from the list and processing it. If a new event is generated, it is inserted into the event-list at its time-stamp position. Thus the simulator processes the events in synchronized chronological order. If we now distribute the simulation over several processes, it becomes possible for a processor to process an event which is not the earliest. Also, in processing this event we may affect conditions for earlier, as yet un-simulated events. Thus the future is affecting the past, which is clearly unacceptable, and is known as a *causality error.*

Thus, synchronization schemes generally fall into one of three categories; *conservative*, *optimistic* and *synchronous* approaches. Conservative approaches avoid causality errors ever occurring by relying on some strategy of determining events which are "safe" to process. That is, they must determine when all events that could affect the event in question have

been processed. Optimistic approaches allow causality errors to occur, but when they are detected, a *rollback* mechanism is employed to recover. More recently, so-called *synchronous* approaches have been developed which determine iteratively which events are "safe" to process within a bounded time period. Each iteration is ended by some form of barrier synchronization in order to keep all the processes in synchrony.

Optimistic approaches detect and recover from causality errors rather than avoid them. Therefore optimistic approaches don't need to determine whether or not it is "safe" to proceed; they only need to detect the error and recover. The original work was done by Jefferson on the mechanism called *time warp*, based on the concept of *virtual time* [80]. In this case, virtual time is synonymous with simulated time. In the time warp mechanism, a causality error is detected whenever an event message is received by a process that contains a time-stamp earlier than the processes' local clock (ie. the time of the last processed message). This is known as a *straggler*. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler. This is known as *rollback*. Two things are affected by rollback. The process state may be modified; this is accomplished by returning to the correct old state which is taken from a store of previous states. Also, previously sent messages must be effectively un-sent; this is achieved by sending *antimessages* that cancel the effect of the original. If the original message has already been processed then that process in turn must also rollback. This process continues until the effects of the causality error are cancelled.

3.2 Synchronization in Action

Before discussing each of these approaches in more detail let us briefly discuss the synchronisation problem and the two classical approaches to it using a simple example; based on that of Fujimoto and Nicol [79]. Consider the network of three processes shown in figure 3.1. Each process sends jobs in the form of messages to each of the two other processes. Let each process execute on its own processor and let the minimum service time of any job be 0.1. Therefore, each process must maintain its own event-list; for example, A:2 indicates a job arrival event scheduled for time 2. The values on the communication links, called link-times, indicate the time-stamp of the last message sent over that link. When the simulation is initiated, a process knows its first job arrival and associated time-stamp (defined as part of the initiation), and the link-times are all initialised to zero. In conservative protocols,

no process can simulate its first event until it is certain that it will not receive a job from another process with a time-stamp less than its first arrival time. Therefore, we have a problem; as the arrival times are all strictly greater than the initial link-times. However, this can be resolved as each process "knows" that even if a job were received at time 0, that job would require at least 0.1 service time. Therefore, the process can "promise", to all the processes connected to it, that it will not send a job until at least time 0.1. Thus, the process can send a (so-called) NULL-message with time-stamp 0.1 to the processes to which it sends jobs. Since every process can do this, every link-time eventually increases to 0.1. Note that a NULL-message does not represent any entity moving between processes and hence has no counterpart in the physical system being modelled, it is used simply to inform another simulation object of the time elsewhere in the simulation. Hence, it can be considered to be a promise to the receiving process that it can carry on accepting messages on a link safe in the knowledge that no new message or job will arrive on that link with a time-stamp earlier than the last NULL-message received.

Using a conservative protocol, a process may execute the message associated with the smallest link-time. In the example a process eventually receives two NULL-messages, with the same time-stamp, and these may be processed. As a result, each process sends two new NULL-messages, now with time-stamp 0.2. This increment in NULL-message time-stamps continues until the link-times reach the time of the Q1 arrival, time 2. At this point, "real" simulation activity begins.

Twenty rounds of NULL-message increments were required to reach this point. Now, let us suppose that when the Q1 arrival goes into service it cannot be pre-empted, and will not depart until time 3, event D:3. Knowing this, Q3 can now send NULL-messages with time-stamp 3 to both Q2 and Q3, effectively "looking ahead" in simulation time to the job's completion. This leads to the situation shown in figure 3.2. Further increments in NULL-message time-stamps are then needed to raise the link-times to the point where the Q1 departure at time 3 can be simulated.

The problem with the above scheme is clearly the high volume of NULL-messages. An optimistic approach such as time warp [80] avoids these. In time warp, every process checkpoints (stores) its state, then optimistically executes its first event. Taking our example network once more, figure 3.2. The Q1 arrival at time 2 departs at time 3 and may then be sent to Q3. Unfortunately, Q3 has already simulated an arrival at time 4, which must now be undone along with all the messages which may have been sent prior to time 3. Therefore,

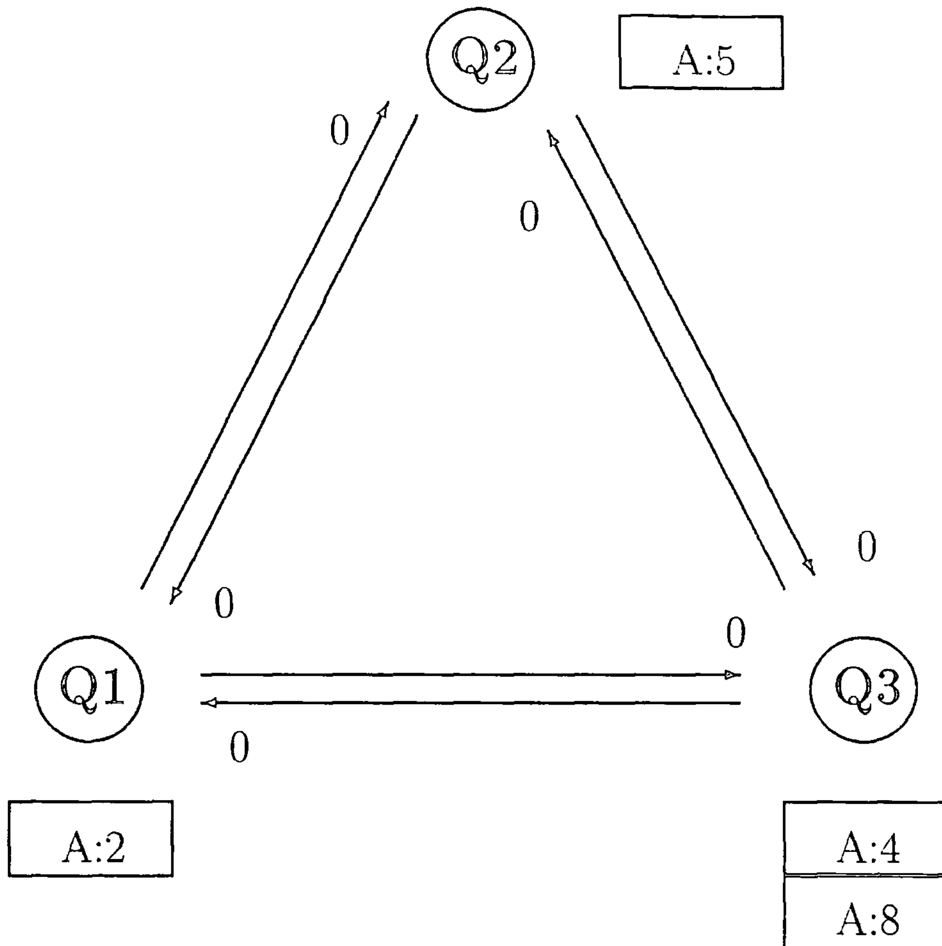


Figure 3.1: Example network of processes at initialisation — all link-times are at zero.

Q3 must recover its state at time 3 (its initial state in this case) and simulate the new arrival. Thus Q3 is said to roll-back to time 3. Let us now suppose that the allocated service time is 1 unit and that the job is then routed to Q2 at time 4. Since Q2 has already simulated an arrival at time 5, it too must roll-back, send anti-messages after messages it erroneously sent, recover its initial state, and simulate the new arrival. These descriptions are intended to suggest that synchronization protocols may typically impose severe overheads depending on the behaviour of the simulation model. The goal of synchronization protocol research is to reduce those overheads. Let us now discuss these synchronization approaches in more detail.

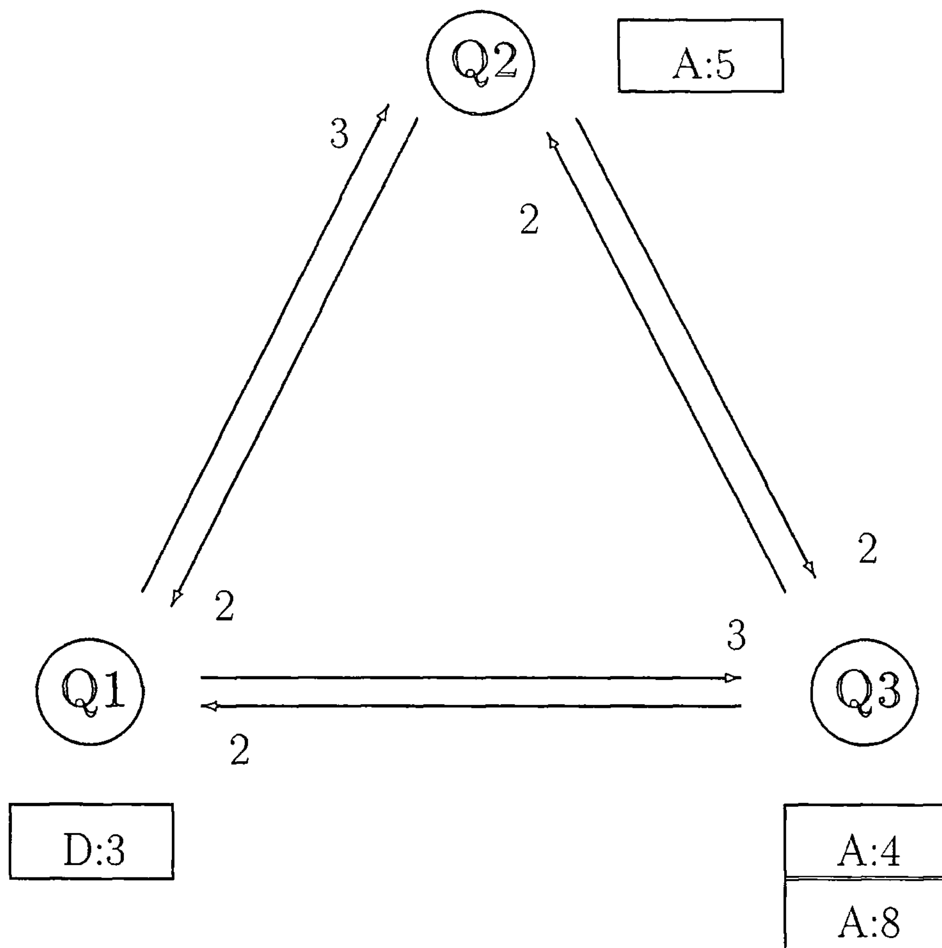


Figure 3.2: Example network of processes at time 2—the link-times represent the time-stamp of the last message to cross the link.

3.3 Conservative Synchronization Approaches

Conservative approaches were the first to appear historically. An added problem which categorises various conservative approaches is that of *deadlock*. If processes do not have a "safe" event which they can process then they are *blocked* and cannot progress. If a cycle of blocked processes occurs then we have deadlock and the simulation will grind to a halt unless the deadlock can be broken. Generally it has been found that, if there are relatively small numbers of pending events compared with the number of communication links between processes in the simulation model, or, if such a scenario occurs in a localised part of the model, then deadlock may occur frequently. Early work by Chandy, Misra and Holmes [81–83], Bryant [84], and Peacock, Wong and Manning [85, 86] investigated various solutions to this problem.

Chandy and Misra's first approach was that of deadlock avoidance using NULL-messages [81–83]. The same approach was investigated independently by Bryant [84], hence conservative protocols using NULL-messages are often called Chandy-Misra (CM) or Chandy-Misra-Bryant (CMB) protocols. The NULL-messages are purely for synchronization purposes and do not correspond to any simulation activity. The clock value of each input to a process (the link-times in the example) give a lower bound on the time-stamp of the next unprocessed event which will appear on that input. These clock values can thus be used to calculate the lower bound of the time of the next output messages from the process on each output. Therefore, when a process blocks, it sends a NULL-message out on each output indicating these lower bounds. Processes receiving these NULL-messages can thus calculate lower bounds for their own outputs and so on. It can be shown that this scheme avoids deadlock as long as no cycles of processes occur in which the collective time-stamp around the cycle is zero (ie. all the processes in the cycle have the same clock time). This approach has been used successfully in many simulation studies such as; [27,87–89]. The main problem with this approach is the NULL-message overhead which can reduce simulation performance significantly, see [90].

An alternative NULL-message approach is to only send them at the request of a process [49,76,85,86,91,92]. The query, or probe, is sent when a process is blocked and needs an improved clock time. Nicol and Reynolds used a variation on this approach in their SRADS (Shared Resource Algorithm for Distributed Simulation) simulation protocol [93,94]. Here, when a process is about to become blocked, due to the incoming link with the smallest link-time having no further messages waiting to be processed, it sends a request to the process on the other end of the link for the next message. Execution can then continue when this message is received. Thus, these approaches reduce NULL-message traffic at the expense of query and query-reply traffic.

Chandy and Misra also developed an alternative approach based on the idea of deadlock detection and recovery [76,83,95]. Various deadlock detection algorithms are also described by Dijkstra and Scholten [96] and by Groselj and Tropper [97]. Deadlock can be broken using the basic fact that the message (or messages) having the smallest time-stamp can always be processed safely. Another method is to compute lower bounds on the link-times at each process, not unlike the NULL-message computation. Such mechanisms are aimed at global deadlock situations. Misra discusses a modification to this to deal with local deadlock involving only a small cycle of the processes [76]. This uses a pre-processor to identify

all local cycles prone to deadlock within a complete simulation model and apply these techniques only on those. However, the overheads on implementing such an approach would seem to be large especially on large models where the network of processes could contain many cycles. Liu and Tropper discuss similar techniques for detecting cycles of processes prone to deadlock [98]. In certain cases, the topology of the network to be simulated can be exploited to simplify the synchronization protocol: Kumar [99] has shown this for acyclic networks. This fact has been exploited by Lin, Lazowska and Baer [100]. They eliminate cycles from a network by combining cyclic sub-networks into single logical processes. The resulting acyclic network is then simulated using the standard CMB protocols.

Significant improvement in the performance of the simulation can be achieved if, instead of holding a message in a process until the clock times on all of its inputs reach the time of departure, the message is processed and passed on to the next object as soon as the causality of the simulation can be guaranteed. In doing this the simulation model can lose some information such as queue length and additional calculations would be required if such information needs to be collected [82]. It should also be noted that such a technique cannot be used with priority queueing structures or pre-emptive messages since the message processing order cannot be determined in advance. The ability to do this was formalised as the *lookahead* capability of the simulation by Fujimoto [101, 102]. This refers to the ability to predict what will (or will not) happen in the simulated time future based on application specific knowledge. The lookahead of the simulation is defined as the interval of time during which the output events can be determined solely from information already received. Thus, the lookahead for a logical process can be known from the minimum time of the next new scheduled event (eg. minimum service time in a queue) and/or from knowing the minimum time-stamp of events received from other logical processes.

A physical system which is well suited to simulation in this manner will have a high degree of lookahead. Work by Reed, Malony and McCredie [103] and by Reed and Malony [90] has suggested that central server queueing systems were unsuited to this type of simulation algorithm, but Fujimoto [101, 104] has since shown that simulations of this form are possible provided that the lookahead is exploited effectively by pre-computing service times.

Nicol [105] first proposed a method for improving the lookahead ability of processes by pre-computing part of the computations for future events. The example put forward is the pre-computing of service times for a network of FCFS queues with no pre-emption.

Pre-computation itself depends on lookahead and is only possible if it can be done without knowledge of the future event message which causes that computation and without knowledge of the state of the process when it occurs. For example, if the service time for a message in a packet-switched communication network simulation depended on the message length, and this were variable and not known until it was generated, then pre-computation would not be possible. However, where pre-computation is possible, it has proved to be a useful technique. For example, in a queueing network simulation, we may take advantage of a non-pre-emptive queueing discipline, and state independent service times and routeing decisions by pre-sending job completions at the point the job enters service, and by pre-sampling a job's service time upon recognising the message reporting its arrival.

Chandy and Sherman [106] introduced the idea of conditional events instead of NULL-messages for improving simulation performance. In a sequential simulation, a *conditional* event is defined as one that will be executed in the future, provided that there are no other events scheduled for earlier execution which will modify or de-schedule it. A *definite* event is therefore one which will be executed unconditionally. For a sequential simulator there is always at least one definite-event (with the minimum time-stamp) and many conditional-events. In a parallel simulation, each logical process does not necessarily have a definite event at the head of its event-list since messages may still arrive from other processes that will pre-empt them. However, there will be at least one, and probably more, definite events somewhere in the simulation. Therefore, it is the task of the simulation protocol to promote as many conditional to definite events as possible by sharing timing information between the processes. Good speed-up figures have been obtained for scenarios that usually showed poor results using other conservative algorithms.

One of the reasons that the example network requires so many NULL-messages using a CMB protocol is that the NULL-messages carry so little information. Consider the initial case in figure 3.1. If Q1 had some way of learning that it was only waiting for itself before proceeding, it could clearly simulate the first arrival at time 2; A:2. If, for the case in figure 3.2, it could then learn that no other process will send a job prior to time 3, it could also simulate the departure; D:3. This observation is explored by Cai and Turner [107] using the "Carrier NULL-Message" approach.

In standard CMB algorithms NULL-messages propagate through the system; the result of receiving a NULL-message is usually to send a number of others. In the carrier NULL-message approach one appends a list of visited processes and pending event times to the

NULL-messages. This information allows a process to learn if it is free to execute an event, potentially, well before it would have done using ordinary NULL-messages. Consider our example network in figure 3.1 once more. Q1 initially sends out NULL-messages with time-stamp 0.1, but appends its own identity and first event time (Q1,A:2). One copy of the message is received by Q2, which appends (Q2,A:5) and sends it back to Q1 and also forwards it to Q3. Q3 appends (Q3,A:4) and also sends a copy back to Q1. The feedback on both incoming links allows Q1 work out that it may proceed in processing the job arrival at time 2. Thus carrier NULL-messages trade some extra computation and longer NULL-messages against a reduction in their number.

Even with carrier NULL-messages, CMB algorithms can still generate large numbers of NULL-messages. Another optimisation, explored by Preiss et. al. in [108], attempts to reduce NULL-message propagation by recognising when a NULL-message becomes *stale*. In our example network, Q1 sends a stream of NULL-messages to Q2 and Q3, successively increasing in time-stamp by 0.1. Now, suppose a NULL-message with time-stamp t_1 arrives from Q1 at Q2's message queue, where it finds an unreceived NULL-message from Q1 at time $t_2 < t_1$. There is no point in having Q2 process the earlier NULL-message; it may be discarded. Indeed, any message (NULL or otherwise) from a process that discovers a NULL-message with smaller time-stamp from the same process may discard it. This is called by the authors NULL-message cancellation.

Still another set of optimizations arise when considering the high cost of message-passing in distributed memory multiprocessors. The cost of sending an m byte message is very well modelled as $a + bm$, where a is a large fixed start-up cost owing (usually) to software overheads, and b is the transfer cost per byte. This provides a strong incentive to concatenate logical messages together into a single long message. CMB variations doing this are explored in [92] where a number of issues are examined, including receiver or sender initiated transfer as well as *lazy* (demand-driven) or *eager* (the original technique) transmission. A more specific instance of this effect is explored by Gould et. al. [109] for communications in an array of Inmos transputers; they show that the throughput of the communication channels increase dramatically as the size of the messages sent is increased and the number decreased by concatenation.

3.3.1 Performance of Conservative Synchronization Approaches

A substantial amount of work has been done on evaluating the performance of various CMB protocols. Reed, Malony and McCredie [103] did extensive measurements using deadlock avoidance and the deadlock detection and recovery approaches for the simulation of various types of queueing networks. Generally they report poor performance except for the case of feed-forward networks with no cycles. However, they made no attempt to exploit lookahead. Fujimoto [102] reproduced these results but was able to improve the performance dramatically by exploiting lookahead.

Su and Seitz [92] report some success simulating digital logic circuits using variations of the deadlock avoidance algorithm using NULL-messages. The simulations were performed on a distributed memory multiprocessor (Intel iPSC) and they argue that superior performance should be possible on a shared memory multiprocessor using this approach due to the reduced overhead in messaging. This assertion was borne out by Wagner, Lazowska and Bershad [110] who implemented the techniques on a shared memory architecture and improved their performance still further. Wagner and Lazowska [111] and Lin and Lazowska [112] looked at lookahead analytically to derive expressions for the lookahead in various types of queueing network simulations. Loucks and Preiss [113] went further with this work on queueing networks and verified experimentally that lookahead has a very significant impact on performance. They also showed that the effective exploitation of lookahead achieves better performance by reducing communication overheads without introducing corresponding computation overheads.

Table 3.1 below summarises some of the best performances achieved using CMB approaches all of which are sub-unitary as expected.

Author(s)	*Ref.*	*Machine*	*Application*	$S(n)$	n
Su and Seitz	[92]	Intel iPSC	Logic circuits	8	64
				10–20	128
Ayani	[114]	Sequent Balance	Queueing networks	5	9
Chandy and Sherman	[106]	Intel iPSC	Queueing Networks	7	12
				9	24
Merrifield et. al.	[115]	Inmos transputers	Road traffic	19	33
Preiss et. al.	[108]	Inmos transputers	Queueing Networks	6.5	8

Table 3.1: Speed-up figures for CMB approaches.

3.3.2 Critique of Conservative Synchronization Approaches

A great deal has been learned about the performance of conservative approaches. The most fruitful approach has been deadlock avoidance using NULL-messages often employing extra optimizations such as pre-computing event times [105], NULL-message cancellation [108] or carrying extra information in each NULL-message [107].

The reliance on lookahead for good performance often limits the usefulness of conservative approaches. Simulation models which have pre-emptive behaviour, possible zero (or very small) time-stamp increments or dependence of output message parameters on the process state at the time of transmission all have poor lookahead properties. Such models will therefore not attain good speed-up using conservative approaches even though there may be significant amounts of parallelism available. We may say that, particularly in contrast to the optimistic approaches to be discussed later, that conservative approaches are *pessimistic* in that they force sequential execution of two events if it is possible that the earlier may directly or indirectly affect the later event. In practice, earlier will only affect later events part of the time.

Simulation models using conservative synchronization approaches (with the exception of deadlock detection and recovery) have to be constructed with knowledge of the logical processes behaviour built in explicitly. Also, the model must be constructed with detailed knowledge of the synchronization approach employed. This leads to the models being more difficult to develop in the first instance and less robust in terms of later changes. The configuration of logical processes must be static. Therefore dynamic processes are not possible and changes to the topology of the processes can affect the performance. Later additions to a model, not anticipated when the model was first developed, can have marked performance effects. For instance, the addition of high priority messages which pre-empt the processing of other messages added to a computer network simulation model.

The criticism that the simulation developer must have detailed knowledge of the synchronization mechanism largely comes from the traditional sequential and the optimistic synchronization communities. They argue that the simulationist should not have to be concerned with such details; users of sequential simulators and optimistic parallel simulators largely don't. This problem is being addressed though the development of simulation languages which allow the synchronization mechanism to extract the necessary information from the model code; an example of which is Maisie [116,117].

3.4 Synchronous Approaches

Several researchers have used synchronous algorithms which determine iteratively which events are "safe" to process [106, 114, 118–120]. Each iteration, or window, consists of "safe" event determination, event processing and ends with some form of barrier synchronization in order to keep the events in each iteration separate. As the "safe" event determination and the barrier synchronization need to be computed globally, these algorithms are more suited to shared memory machines as the message passing overhead of global synchronization on a distributed memory architecture is prohibitive.

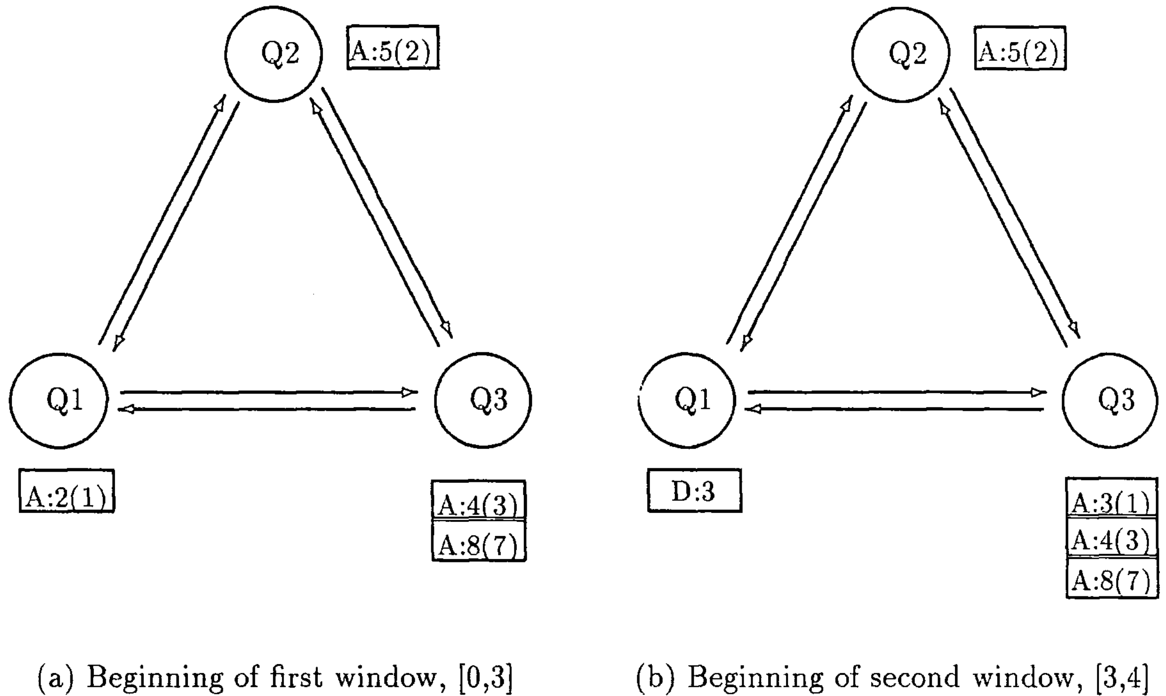
There is more than a passing resemblance between synchronous approaches and the deadlock detection and recovery approach discussed earlier. Both approaches involve a global computation to determine which events are safe to process in the next phase. The difference is in the computation. Ideally, a deadlock detection and recovery approach will never (or very seldom) deadlock, whereas a synchronous approach will continually block and restart at the end of each iteration. This would seem to indicate that deadlock detection and recovery should outperform a synchronous approach. In practice it has been found that in the period leading up to a deadlock, execution is largely sequential as there are increasingly fewer events to process. This effectively throttles any speed-up as per Amdahl's law. Synchronous approaches control potential deadlock by controlling the amount of computation in each iteration. In addition, they do not require a deadlock detection algorithm.

Synchronous approaches are distinguished by the mechanism used to determine which events are "safe" to process within an iteration. Lubachevsky [118] introduced the idea of a moving (simulated) time window. The lower edge of the window is defined as the minimum time-stamp of any unprocessed event; the upper edge is usually set using application specific knowledge based on the same concept of lookahead discussed earlier.

The algorithm studied by Nicol [119] can be applied to a queueing network as follows. We assume that the queue knows all there is to know about a job's departure at the time it enters service. Therefore, we can immediately report the job's arrival at the next queue; this is known as pre-sending. Using knowledge of the queueing discipline, and the assumption that no further job will arrive, the queue can compute the time of the next message it will send. This time must be the departure time of the next job to enter service. Let us suppose that all the processors have simulated up to time t and have synchronised globally. Each processor p must compute the time $\delta_p(t)$ of the next message it will send (in the

absence of receiving further messages), and the processors together compute the minimum $\delta(t) = min_p\{\delta_p(t)\}$. The window $[t, \delta(t)]$ is now defined, and every processor is now free to simulate all events with time-stamps within it.

Now let us see how this mechanism would be applied to our example network of processes, again based on that of Fujimoto and Nicol [79]. Initially, all the processors are synchronised at time 0. This is the situation shown in figure 3.3 (a). The computations for the next messages from each process are as follows;



(a) Beginning of first window, [0,3]

(b) Beginning of second window, [3,4]

Figure 3.3: Example network of processes—simulated using time windows

$$\delta_1(0) = 3,\ \delta_2(0) = 7,\ \delta_3(0) = 7\ and\ min_p\{\delta_p(0)\} = 3$$

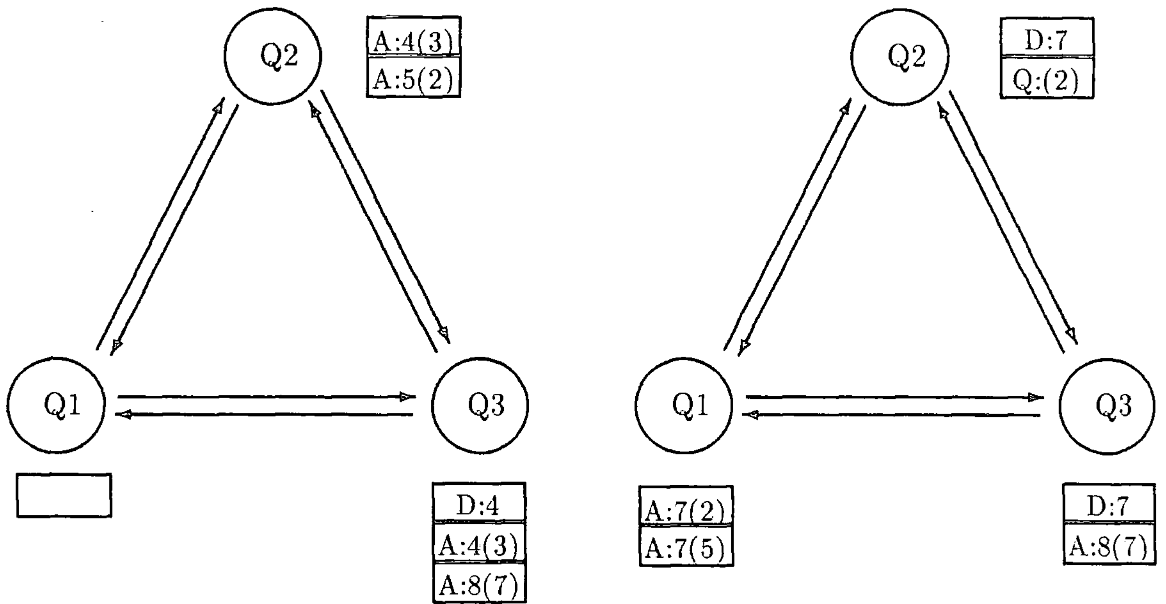
Each process identifies the completion time of the next job to receive service, a calculation made possible by pre-sampling service times as shown in figure 3.3 (a); eg. A:2(1), is a job arrival at time 2 which receives a service time of 1 unit.

Only one event occurs in the first window [0,3]; the arrival at process Q1. Q1 puts the job in service and decides that Q3 will receive it next and sends a message informing it of the arrival. Q3 must pre-sample a service time for the new arrival, A:3(1). Q1 must also generate a departure event (D:3) and place it in its event-list. So, at the end of the first

window, we have the situation in figure 3.3 (b) and we can compute the size of the next window.

$$\delta_1(3) = \infty,\ \delta_2(3) = 7,\ \delta_3(3) = 4\ and\ min_p\{\delta_p(3)\} = 4$$

Thus the second window is [3,4]. In this window the departure at Q1 is simulated, the corresponding arrival at Q3 is simulated, and notification of a new arrival at time 4 is given to Q2, A:4(3). The situation at the end of the second window is shown in figure 3.4 (a).



(a) Beginning of third window, [4,7]

(b) Beginning of fourth window, [7,9]

Figure 3.4: Example network of processes—simulated using time windows

$$\delta_1(4) = \infty,\ \delta_2(4) = 7,\ \delta_3(4) = 7\ and\ min_p\{\delta_p(4)\} = 7$$

For the third window, [4,7], Q3 simulates a departure and an arrival at time 4; pre-sending notification of the arriving job's departure (at time 7) to Q1, which chooses a service time of 2. Q2 simulates an arrival at time 4 (pre-sending to Q1), and simulates the job arrival at time 5 by marking the job as being in the queue as the server is busy. Upon receiving the arrival at time 7, Q1 pre-samples a service time of 5 units and places the new arrival event in its event-list. The situation at the beginning of the fourth window is shown in figure 3.4 (b). The fourth window will be [7,9] as shown below.

$$\delta_1(7) = 9,\ \delta_2(7) = 9,\ \delta_3(7) = 15\ and\ min_p\{\delta_p(7)\} = 9$$

3.4.1 Performance of Synchronous Approaches

The performance of the bounded lag approach which uses synchronous execution, lookahead and time windows has been examined by Lubachevsky [118, 121], see table 3.2. Again, as for the CMB approach, the speed-up found was sub-unitary but it did scale well as predicted.

Author(s)	*Ref.*	*Machine*	*Application*	$S(n)$	n
Lubachevsky	[118]	Sequent Balance	Queueing networks	16	25
	[121]	Connection machine	Ising spin model	1900	16,384

Table 3.2: Speed-up figures for Synchronous approaches.

3.4.2 Critique of Synchronous Approaches

The issue which needs to be addressed for window algorithms is whether enough parallel events are processed in each window for it to be effective. This is discussed for the window algorithm described above, as well as for the Bounded Lag algorithm [121]. Both algorithms should be scalable, which means that their performance characteristics do not degrade as the size of the problem (and the machine architecture) increases.

Some insight into the scalability of the window algorithm is gained if we suppose that a job's service time, t_s, is always greater than zero. Since the $\delta_p(t)$ value computed by a processor is the completion time of a job that has not yet entered service, one infers that $\delta_p(t) - t \geq t_s$ for all p, so that the width of the window is at least t_s. The average number of events processed in a window is at least $\mathcal{E}t_s$, where $\mathcal{E}$ is the event density (events/unit simulation time) for the entire simulation model. Increasing the problem size increases the event density; the number of events in a window increases proportionally with $\mathcal{E}$. Assuming the simulation load is evenly balanced (or that the imbalance does not grow with the number of processors), the number of events a processor executes per window does not decrease if the number of processors and event density simultaneously increase in fixed proportion. Another advantage of window-based protocols is that they are relatively easy to use on SIMD architectures and this has been done successfully for a switching network by Berkman [122].

3.5 Optimistic Synchronization Approaches

Optimistic approaches detect and recover from causality errors rather than avoid them. Therefore optimistic approaches don't need to determine whether or not it is "safe" to proceed; they only need to detect the error and recover. The advantage of this is that the simulator can exploit parallelism fully in applications which may produce causality errors but in reality rarely do. Obviously, the greater the amount of causality errors that a simulation produces, the greater the synchronization overhead with this approach.

The original work was done by Jefferson on the mechanism called *time warp*, based on a concept of *virtual time* [80]. In this case, virtual time is synonymous with simulated time. In the time warp mechanism, a causality error is detected whenever an event message is received by a process that contains a time-stamp earlier than the process' local clock (ie. the time of the last processed message). This is known as a *straggler*. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler. This is known as *roll-back*. Two things are affected by roll-back. The process state may be modified; this is accomplished by returning to the correct old state which is taken from a store of previous states. Also, previously sent messages must be unsent; this is achieved by sending *anti-messages* which cancel the effect of the original. If the original message has already been processed then that process in turn must also roll-back. This process continues until the effects of the causality error are cancelled.

For even a moderate size of simulation this seems to imply a large amount of memory to save the states for each process. However, as the earliest time-stamped event is always "safe" to process, this is designated *global virtual time* (GVT) and is used to discard all states before this time. This process of reclaiming memory, which is irrevocable, is known as *fossil collection*. GVT has another function which is to commit irrevocable operations such as I/O. Many algorithms have been suggested for computing GVT, most of which are based on including the "local" GVT of a logical process with all, or selected, messages sent to other processes. Discussions of various GVT algorithms can be found in the work of Bellenot [123], Lin and Lazowska [124] and Priess [125]. GVT is discussed in more detail in the next section.

A variation on the above cancellation approach, which is said to use *aggressive* cancellation using anti-messages, is an approach which seeks to repair the "damage". This is known as *lazy* cancellation, discussed by Gafni [126]. In this case, instead of immediately

sending out anti-messages, the process waits to see which messages that the re-execution of the process produces are different to those produced before. If the same message is produced, there is no need to send out an anti-message. It has been found that, depending on the application, lazy cancellation may improve or degrade the simulation performance. Improvement, noted by Berry [127] and by Som et. al. [128], is usually due to processes with incorrect input still producing correct output. Degradation can be due to the additional message checking overheads and the fact that incorrect computations have longer to spread out, causing more processes to roll-back.

A variation on lazy cancellation is that of *lazy re-evaluation*, or *jump forward* proposed by West [129]. If, after rolling back and processing a straggler, it is found that the state vector is unaltered and no new messages have arrived, then re-execution of the rolled back events will be the same as before. Therefore, re-execution is unnecessary and the simulation may *jump forward* over them. This sounds useful but was found to significantly complicate the time warp kernel. An implementation was tried in the Jet Propulsion Laboratory (JPL) time warp kernel but was later removed due to such problems. This was reported by Jefferson et. al. [130].

Fujimoto [131] proposed an enhancement called *direct cancellation* which works with any of the above cancellation approaches if implemented on a shared memory multiprocessor. Any event scheduling a new event keeps a pointer between the two, thus allowing faster cancellation compared with conventional time warp systems which must search for the messages to cancel. This has proved to be a useful optimization, if limited in scope.

3.5.1 Enhancements to Optimistic Synchronization Approaches

There have been several variations on the basic approaches seeking to optimize the time warp mechanism. Optimizations have been sought in two basic areas, rollback stability and memory management.

The rollback stability problem stems from the possibility that a "fast" processor (or processors) may simulate too far ahead of the others. This is most likely to occur when communication between processors is low and processor loads not equal. A form of *thrashing* may then occur where the simulation processes are caught in an increasing cycle of roll-backs and cannot progress termed by Lubachevsky et. al. *wildfire cascade* rollbacks [132]. Lubachevsky et. al. also introduced the terms *gushing cascade* rollback, for the situation where rollbacks propagate rapidly through a simulation temporarily halting progress, and

echo rollback where a small cycle of processes cause each other to rollback. The latter classes of rollback have been observed by simulationists, though rarely, and the cause has often been found to be an error in the simulation model rather than a flaw within time warp. Wildfire cascade has not been observed as yet to the knowledge of this author and Lubachevsky et. al. though this does not mean necessarily that it cannot occur. Indeed, Lubachevsky et. al. [132] and Fujimoto [24] argue analytically and from experience that such thrashing behaviour is exacerbated if the cost of rollback is too high.

One idea presented for preventing such unwanted rollbacks is to cause controlled pre-emptive rollbacks or to freeze the computation of all adjacent processors. For example, when a processor needs to rollback it may immediately issue rollback instructions to other processors who will probably have to rollback anyway. This effectively causes the rollbacks to occur in parallel rather than serially. An algorithm to achieve this has been developed by Madisetti et. al. called *Wolf calls* [133]. An alternative way of implementing this idea, also by Madisetti et. al. [134], is to build periodic (or even random) pre-emptive rollbacks that occur in the simulation model independently of any activity. This keeps all the processors loosely synchronized in the same period of simulation time.

An alternative to these ideas is simply to restrain processors from getting too far ahead of the rest, effectively curbing the optimism. The simplest method is to use a window, $[t, t + \delta t]$. Events with a time-stamp less than $t + \delta t$ are simulated as normal; those with a time-stamp greater than or equal to $t + \delta t$ are left until all of the processors have simulated up to time t. This is achieved using a barrier synchronization as described by Nicol [135]. A new window $[t + \delta t, t + 2\delta t]$ is then defined and simulated. This idea has been investigated by Sokol and and Stucky [136], Turner and Xu [137] and by Ball and Hoyt [138]. A similar proposal to extend constrained optimism to the Bounded-Lag protocol has been proposed by Lubachevsky [139].

Effective memory management has been required for optimistic synchronization approaches like time warp since their first development. In time warp, three types of mechanism have been used to restrain the amount of memory that is required to perform the simulation. Firstly, and fundamentally, fossil collection using GVT calculation. Secondly, some form of infrequent or reduced state-saving, often termed checkpointing. Thirdly, a rollback-based recovery mechanism often achieved by limiting the optimism as described above.

Before discussing fossil collection and GVT calculation let us establish what time warp

does in each logical process. In order to be able to recover (rollback) from a causality error, each logical process must maintain past state vectors (records) in its state queue, previously processed events in its input queue, and records of previously sent messages (saved as anti-messages) in its output queue. Fossil collection, the recovery of memory for re-use by discarding contents of the above queues which are no longer needed by the simulation, is made possible by the distributed calculation of the global virtual time (GVT). All memory occupied by state vectors and messages whose time-stamps are older than GVT can be recovered.

GVT is the time-stamp of the earliest event in the simulation, but also represents a lower bound on the time-stamp of any future rollback. In time warp, as originally defined by Jefferson [80], rollbacks only occur when receiving messages whose time-stamp is less than the local clock time of the logical process, often called local virtual time (LVT). This means that, strictly, GVT has to be defined as the minimum of the time-stamp of all messages in transit (sent but not received) as well as the LVT of all logical processes. If a logical process has no unprocessed messages in its input queue, then the LVT is set to infinity. If there are no unprocessed messages, or messages in transit, in the entire system then GVT will be set to infinity and the simulation will terminate.

If barrier synchronization (eg. that of Nicol [135]) is possible (and not prohibitively expensive) in the multiprocessor, then GVT calculation is simple. If this is not possible then GVT calculation is more difficult as it must be distributed. The difficulty arises as inaccurate values of GVT may be calculated due to messages still in transit and race conditions. The former problem can be addressed with acknowledgements or by a logical process only proceeding with the calculation if it has information on LVT from all the adjacent processes. These algorithms are discussed by Lin and Lazowska [124]. The effect of race conditions can only be solved completely using a barrier synchronisation to ensure that all simulation computations halt before the GVT calculation is begun. The processes agree to synchronize at some barrier simulation time t_b. A process enters the barrier when it has no events left to process with time-stamps less than t_b, but rolls back out of the barrier if it subsequently receives a message with a time-stamp less than t_b. This ensures that a process does not leave the barrier until all processors have simulated all events at times less than or equal to t_b. Emerging from this barrier, a process knows that GVT is t_b. Thus it can perform fossil collection and simulate up to the next barrier synchronization time.

Other approaches to GVT calculation have been suggested in order to avoid using barrier

synchronization. Preiss [125] uses a token passing scheme where the processes are organised in a logical ring, and continually compute GVT as the token containing LVT values for each process visited is passed around the ring. Similarly, Bellenot uses a logical tree structure to initiate, compute, and distribute GVT values [123]. Reynolds [140] also uses a tree structure to compute GVT in his hardware synchronization network.

Even with fossil collection time warp still uses prodigious amounts of memory compared with sequential and parallel simulation using a conservative or synchronous synchronization approach. Memory may also be saved by cutting down on the overheads of state-saving either by doing it incrementally or infrequently. These schemes are illustrated in figure 3.5. Where the state vector is large and only a small portion is modified by each event, incremental state-saving may be useful. Here, only the incremental changes to the state are saved rather than the entire state vector reducing both memory space and access time. The disadvantage is that rollbacks become more expensive as the state vector must be reconstructed from the incremental changes. This is a problem because, as noted above, time warp is more prone to thrashing if rollback costs are high. Briner [141] used incremental state-saving in an implementation of time warp for digital logic simulation, and found that the state-saving overheads were significantly reduced for transistor and gate level simulations.

An alternative approach is to reduce the frequency of state-saving, or conversely, to increase the checkpoint interval. As illustrated in figure 3.5, rollback must be to the last checkpointed event older than the straggler's time-stamp and the relevant state recreated. This phase, between the last checkpoint time and the straggler arrival time, is known as *coasting forward* as message sending is turned off. As before, with incremental state-saving, the cost of rollback is increased but the state-saving overhead in time and space is reduced. As the checkpoint interval may be varied there is an obvious trade-off between memory usage and the performance of the simulation. Lin and Lazowska [142] investigated this trade-off analytically and derived expressions for the range in which the optimum checkpoint interval should be set. They also showed that it is better to err on the side of a longer checkpoint interval as storage space is reduced but speed-up is not greatly affected. Preiss, MacIntyre, and Loucks [143] and Bellenot [144] validated Lin and Lazowska's results experimentally. Further collaborative work by Lin, Preiss, Loucks and Lazowska [145] has resulted in an improved version of Lin and Lazowska's model. This has been used as the basis for a heuristic algorithm for automatically selecting the optimum checkpoint interval in terms of reducing the execution time. Palaniswamy and Wilsey [146] compare checkpointing with

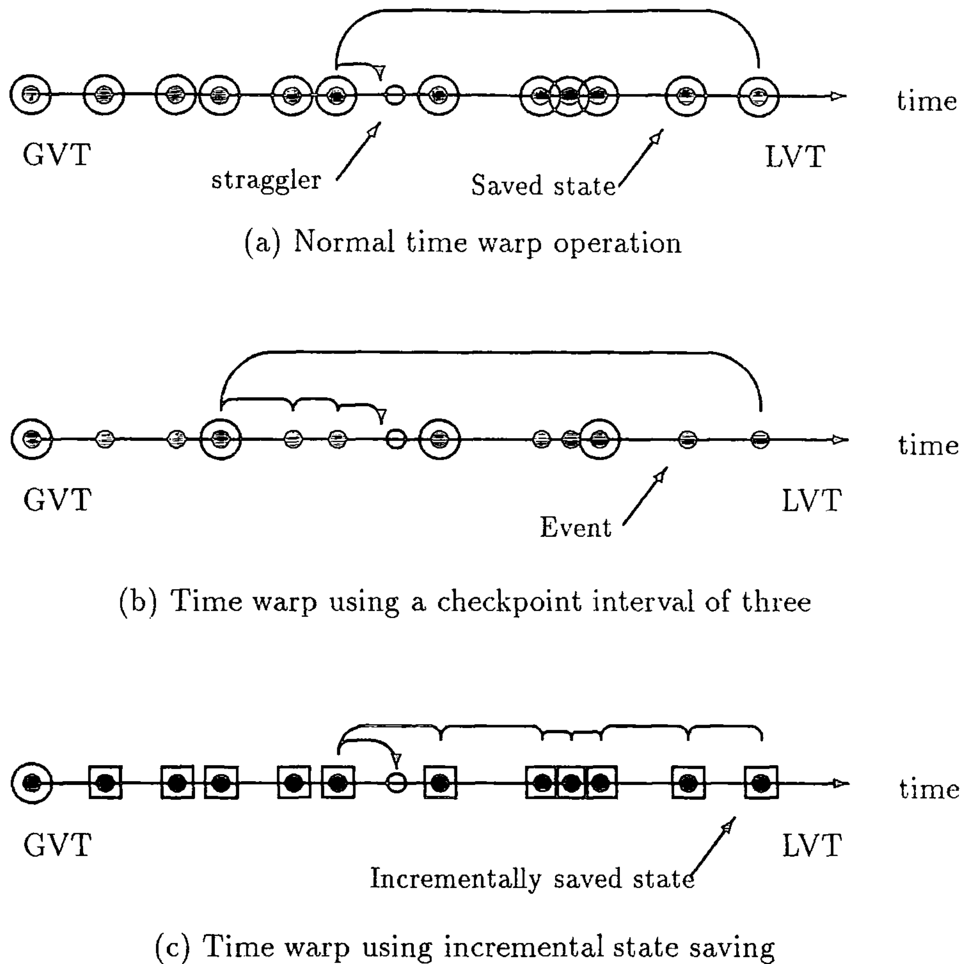


Figure 3.5: Time warp rollback using different state-saving schemes

incremental state-saving analytically using Lin and Lazowska's model as a starting point. They conclude that incremental state-saving should generally outperform checkpointing for applications where there are many small rollbacks and and a long execution time compared with the total state-saving overhead. This conclusion would seem, in practical terms, to favour checkpointing with an optimum interval.

All of the above strategies have the drawback that if the system runs out of memory then the simulation must terminate. The problem may lie with the time warp approach itself (too optimistic) or the application. Whichever is the cause, and due to the large amounts of memory used by time warp, much attention has been paid to allowing time warp to work within limited memory. The basic idea underlying these enhancements is to rollback overly optimistic computations and reclaim memory.

Jefferson, when time warp was first proposed, described a mechanism called *message sendback* to achieve this effect [80]. Here, the time warp kernel may return a message to

the sending process without processing it, and reclaim the memory. Upon receiving the returned message, the sending process will usually have to rollback to the send time-stamp of the message as it will be a straggler, and re-generate it. Assuming aggressive cancellation (which Jefferson did), this rollback will cause anti-messages to be sent and the subsequent annihilations, and possibly other rollbacks, release additional memory resources in the system. Obviously, only messages with send time-stamps greater than GVT can be returned as rollbacks beyond GVT would result causing an unrecoverable error. Message sendback is triggered when a process receives a message, but has no memory available to store it. The message with the largest send time-stamp is then returned. Gafni [126] proposed a mechanism which uses message sendback along with other mechanisms to reclaim storage used by state vectors and messages stored in the output queue. More recently, Jefferson has proposed an alternative approach called *cancelback* [147]. While Gafni's algorithm will only discard states in the process which ran out of memory, cancelback also allows states in any process to be reclaimed. Messages containing high send time-stamps are sent back to reclaim storage allocated to messages. This is intended to rollback processes that are ahead of others in the simulation.

Message sendback and cancelback require us to redefine GVT slightly. Messages returned to the sending process may now cause rollbacks, so the send time-stamps of messages must now be considered in addition to receive time-stamps in the definition. For cancelback, Lin [148] has defined GVT as the minimum of the LVT of all the logical processes in the simulation and the send time-stamp of all messages in transit. Lin also proposed a mechanism called *artificial rollback* which uses the same definition of GVT. If memory storage becomes exhausted and fossil collection cannot reclaim additional memory, processes are rolled back. The process that is the furthest ahead in simulation time is rolled back to the time of the second most advanced process. This is repeated until the supply of free memory reaches a certain threshold termed the *salvage* parameter, which is a control that can be used for tuning performance and is essentially the amount of memory reclaimed when the system runs out. Artificial rollback is similar to cancelback in the sense that cancelback returns messages which cause the sender to rollback, and artificial rollback rolls back the processes directly. The principal advantage of artificial rollback over cancelback is that it has been found to be simpler to implement. Lin has shown that artificial rollback and cancelback have the property of being storage optimal. That is, they are able to execute the simulation program using no more than a constant times the amount of memory re-

quired by the sequential simulation that uses an event-list. Also, in the case where there is insufficient memory to run the time warp simulation without using artificial rollback or cancelback; then the simulation will still run.

It has been found that while time warp with cancelback or artificial rollback is storage optimal, certain conservative simulation protocols are not. Lin, Lazowska and Baer [100] and Jefferson [147] show that the CMB algorithms may require $O(nk)$ space for parallel simulations on n processors where the sequential simulation requires only $O(n+k)$ space. Lin and Preiss [149] report the existence of simulations where CMB algorithms have exponential space complexity, and so use far more storage than even the sequential simulation. Conversely, they also indicate that the same algorithm may sometimes use less storage than that required by the sequential simulator. They show that time warp with cancelback or artificial rollback always requires at least this much.

The main drawback of time warp with cancelback or artificial rollback is that it will run very slowly if it is only provided with the minimum of memory. Akilidiz et. al. [150] explore the performance of time warp with varying amounts of memory both analytically and using an implementation of time warp using cancelback. They found that time warp needs relatively little memory in order to perform well, compared with execution using unlimited memory (ie. more than enough). However, they only explored the case of homogeneous processing loads. An experimental study on the same implementation by Das and Fujimoto [151] has examined the performance/memory trade-off using various non-homogeneous processing loads. Also, a stress case of a processing load containing a number of overly optimistic processes that advance unthrottled into the simulated future was investigated. Again, they found that time warp with cancelback, even with asymmetric processing loads, only needed a modest amount of extra memory above that needed for sequential simulation and still performed well. However, it did not perform as well as the case with symmetric processing loads, as expected. This was achieved by optimizing the setting of the salvage parameter. It was found that setting it too low causes poor performance especially if the system is memory bound; setting it too high (the maximum setting essentially causes the simulation to delete everything except that required for sequential execution) also degrades performance because correct computations are unnecessarily rolled back. Between these two extremes, however, performance appeared to be relatively insensitive to the salvage parameter setting. Further, it was discovered that an inefficient implementation of the event-list (ie. the input queue) in each logical process can seriously degrade the

performance of the system in limited memory situations.

3.5.2 Performance of Optimistic Synchronization Approaches

Many successes have been reported in using the time warp approach to speed-up simulations. Impressive results have been published in areas as diverse as battlefield simulations, biological systems (eg. an ant foraging model, shark's world, health care systems), petri nets, colliding pucks, queueing networks and computer communication networks. A selection of the most successful results are shown in table 3.3

Author(s)	*Ref.*	*Machine*	*Application*	$S(n)$	n
Wieland et. al.	[152]	Caltech/JPL Mk. III Hypercube BBN Butterfly	Battlefield simulation	28.6 36.8	60 100
Sokol and Stucky	[136]	BBN Butterfly	Battlefield simulation	8.59	9
Baezner et. al.	[153]	Inmos transputers	Battlefield simulation	6–10	32
Ebling et. al.	[154]	Caltech/JPL Mk. III Hypercube	Biological systems	12.7	32
Hontalas et. al.	[155]	Caltech/JPL Mk. III Hypercube	Colliding pucks	11.5	32
Presley et. al.	[156]	Caltech/JPL Mk. III Hypercube	Computer networks	11 16.2	16 32
Briner	[141]	BBN Butterfly	Digital logic circuits	23	32
Fujimoto	[131] [157]	BBN Butterfly	Queueing networks Synthetic (Phold) model	57 32-54	64 64

Table 3.3: Speed-up figures for Optimistic approaches.

It has been found that significant performance improvements can also be made to time warp by using application specific knowledge in the form of lookahead, but the impact is far less than for the CMB approach. This has been noted by Loucks and Preiss [113], Baezner et. al. [158] and Fujimoto [131, 157]. Fujimoto showed, using a queueing network simulation, that time warp could obtain significant speed-up for models with poor lookahead where conservative approaches performed badly. This was confirmed in a further investigation using a synthetic workload model known as Phold, a parallel hold model[1]. Variations in lookahead, time-stamp increment distribution, topology and granularity of computation

[1]Phold is an extension of the synthetic hold model used in evaluating sequential simulation event-list implementations.

were investigated. It also was found that time warp was able to achieve speed-up in proportion to the amount of parallelism available in the workload. That is, speed-up continued to increase as the processes were allocated to more processors; this often called scaling. A less encouraging result was that time warp was found to be very sensitive to the cost of rollback and also to the size of state vectors; the performance is severely affected by increases in both. This result has fuelled the interest in hardware support for time warp, infrequent and incremental state-saving and analytical studies into time warp performance.

Hardware support for time warp has been investigated in two ways; hardware support for state-saving and hardware support for dissemination of global information. These both involve "add-on" hardware implementing certain time consuming operations used in the simulation. They are intended to be attached to existing parallel or distributed computer architectures so that they may be more "technology proof" and, therefore, still be useful as faster microprocessors and denser memory chips become available.

The Virtual Time Machine, developed by Fujimoto and Ghosh [159,160], is envisioned to be a general purpose parallel processor based on optimistic synchronization. The machine is essentially a shared memory multiprocessor with a special type of memory system called space-time memory, and a hardware implemented rollback mechanism. Fujimoto et. al. [161] have also designed a component called the rollback chip that provides hardware support for state-saving and rollback in time warp. This component was the forerunner to the space-time memory system mentioned above and can be viewed as a special memory management unit. A process may issue a special "mark" operation to indicate that a state vector must be preserved in case a rollback later occurs. The rollback chip hardware then modifies the addresses of subsequent memory writes to preserve it, thus minimising the amount of copying that is required and reducing the cost of restoring a state if rollback occurs. This is similar to the *direct cancellation* also proposed by Fujimoto [131]. Simulations indicate that the state-saving overhead can be considerably reduced.

Reynolds et. al. have proposed a hardware mechanism to rapidly collect, operate on, and disseminate synchronization information throughout a parallel simulation system [140,162–164]. The hardware is configured as a binary tree, with a processor assigned to each node. To compute GVT for instance, each processor indicates a local minimum among the processes assigned to it, and the tree automatically computes the global minimum in a distributed fashion[2] and distributes the computed value to all processors in the system by broadcasting

[2]Each node (processor) computes the minimum of its own LVT and that of its neighbours, and propagates

values down the tree. Simulations indicate that the time required to compute GVT is reduced by orders of magnitude over software based approaches. A prototype system is currently under construction.

Experimental investigations into the benefits of infrequent state saving, or checkpointing, have been done by Preiss, MacIntyre, and Loucks [143] and by Bellenot [144]. Preiss et. al. simulating queueing networks, found that the "time-optimal and space-optimal checkpoint intervals are not the same". They also found that "a checkpoint interval that is too small adversely affects space more than time; whereas a checkpoint interval which is too large adversely affects time more than space", thus providing a useful trade-off. Bellenot confirmed these conclusions with simulations of colliding pucks, a battlefield scenario and a computer network. However, he did find that "the positive effects of state skipping[3] generally decreases as the number of processors increase".

3.5.3 Critique of Optimistic Synchronization Approaches

The major fear with optimistic approaches such as time warp is that they may exhibit thrashing behaviour due to multiple rollbacks. However, the experience of researchers at a number of universities and research centres has been that such behaviour is seldom encountered in practice. Also, when it is observed, it is almost always attributable to some flaw in the simulation model or the implementation of the time warp mechanism. As mentioned previously, this conclusion is largely supported by the results of analytical studies of the time warp mechanism, assuming that the cost of rollback is sufficiently small.

There is an intuitive explanation as to why the behaviour of the time warp mechanism tends to be stable in this regard. First of all, erroneous computations can only occur when a correct event is processed prematurely. This premature execution, and all subsequent erroneous computations, must necessarily be in the simulated time future of the (correct) straggler event. Also, the further the incorrect computation spreads, the further it moves into the simulated time future. This lowers its priority for execution as scheduling preference is always given to events having smaller time-stamps. Therefore, the time warp mechanism tends to automatically slow the propagation of erroneous computations allowing the causality error detection and rollback mechanism to correct things before too much damage has been done. A potentially dangerous case has been suggested by Fujimoto [24]

the new minimum up the tree.

[3]State skipping is the term Bellenot uses for checkpointing or infrequent state-saving.

where the erroneous computation propagates with smaller time-stamp increments than the correct one. It remains to be seen, however, to what extent this behaviour can degrade performance, or if such pathological situations arise in practice with any frequency.

A more serious practical problem with the time warp mechanism is the need to periodically save the state of each logical process. As previously mentioned, the state-saving overhead can seriously degrade the performance of time warp programs, even if the state vector is relatively modest in size. The state-saving problem is further exacerbated by applications requiring dynamic memory allocation because one may have to traverse complex data structures to save the process's state. State-saving overhead limits the effectiveness of time warp to applications where the amount of computation required to process an event can be made significantly larger than the cost of saving a state vector[4]. This may be difficult to achieve for certain applications. A more general solution is to use hardware support for state-saving [126, 159]. Supporters of optimism concede that hardware support will probably be required to exploit fine-grain parallelism effectively.

Much has been learned with respect to techniques to control memory usage in optimistic protocols. However, some important questions still remain unanswered. Although, experimental data has provided some useful insights as how controls such as the checkpoint interval and the salvage parameter should be set; analytic models (as yet) do not exist to definitively answer such questions. Nevertheless, controls such as these do at least allow a practical trade-off between performance and memory usage. Furthermore, the performance/memory properties of conservative protocols have not been studied in any depth. It follows that mechanisms to ensure storage optimal execution for conservative protocols have yet to be developed.

In time warp, fossil collection and GVT computations are used to commit any irrevocable operations, eg. screen and disk IØ. Thus far, most of the work in PDES has been done with simulators which have used relatively little IØ. When PDES is used in interactive and real-time simulations, rapid commitment of events, and particularly GVT computations, becomes critical. The usefulness of optimistic mechanisms for such simulation applications is just beginning to be investigated by researchers; see Ghosh et. al. [166].

Unlike conservative approaches, optimistic mechanisms need to be able to recover from arbitrary run-time errors; particularly as such errors may be erased by a subsequent rollback.

[4] "A high granularity compared with the event overhead" is recommended for all applications using the time warp operating system (TWOS) [165].

Erroneous computations may enter infinite loops, requiring the time warp kernel to interact with the hardware interrupt system. In certain languages (eg. C and Pascal), pointers may be manipulated in arbitrary ways; time warp must be able to trap illegal pointer usages that result in run-time errors, and prevent incorrect computations from overwriting critical areas of memory. Although such problems are, in principal, not insurmountable, they may be difficult to circumvent in certain systems without appropriate hardware support. The alternative taken by most existing time warp systems is to leave the task of analyzing incorrect execution sequences to the user by providing copious information in the form of redundant statistics and run-time traces[5]. Practically, this means that errors in time warp simulation models can be extremely difficult to debug. Indeed, the user manual for JPL's time warp operating system (TWOS) [165] recommends the use of TWSim, a sequential simulator which runs TWOS applications, for debugging.

Finally, supporters of conservative approaches point out that the time warp mechanism is far more complex to implement than conservative approaches; particularly if one attempts to catch errors such as those described above. Although the actual time warp code is not very complex[6], if one ignores the error handling aspects, inexperienced implementors may make seemingly minor design mistakes that lead to extremely poor performance. For example, the use of an inappropriate process scheduling policy can be catastrophic. Further, debugging time warp implementations is time consuming because it may require detailed analysis of complex rollback scenarios. A certain amount of design experience (or pure luck) is often required to obtain a good, robust implementation of time warp. On the other hand, champions of optimism counter this by pointing out that this development cost need only be paid once when developing the time warp kernel.

3.6 Summary

This chapter has attempted to provide an overview of the distributed model components approach to parallel discrete event simulation and the fundamental problem of synchronizing the execution of multiple processes. Optimistic methods such as time warp seem to offer the greatest hope as a general purpose simulation mechanism assuming that the state-saving

[5]See the article by Reiher et. al. [167] for a practical insight into debugging time warp applications.

[6]The entire time warp kernel, described by Fujimoto [131] for a shared-memory multiprocessor, is only a few hundred lines of code. The rollback, message cancellation and event handling code in JPL's time warp operating system (TWOS) kernel for distributed memory multiprocessors is fewer than 1000 lines [165].

overhead can be kept to a manageable level. Significant success has been achieved across a very wide range of applications.

Conservative methods offer good potential for certain classes of problems. Significant successes have also been obtained particularly when application specific knowledge, in the form of lookahead, has been applied to maximise the efficiency of the simulation mechanism. Conservative methods may well find success in packaged simulation systems (eg. digital logic simulators) in which the simulation code is optimized for the synchronization algorithm and users only configure the provided simulation modules into specific systems.

Which strategy should then be used for a particular simulation problem? If state-saving overheads do not dominate, then time warp has a good chance of success assuming (of course) that the problem contains a reasonable degree of parallelism. If the application has good lookahead properties, conservative mechanisms may also perform well If the application has both poor lookahead and large state-saving overheads, all existing PDES approaches will have trouble obtaining good performance even if the application contains copious amounts of parallelism. However, time warp, aided with hardware support for state-saving, should provide a viable solution in this situation. Also, hybrid approaches of time warp with limited optimism (eg. breathing time warp) may well yet produce a more general parallel simulation mechanism.

Finally, perhaps the most challenging problem remaining to be explored is application of these techniques beyond the realm of discrete event simulation, in the world of general purpose parallel computation. A parallel simulator executes events in parallel, yet guarantees that the same results are obtained as would be if the events were processed sequentially in increasing time-stamp order. Consider any computation that is broken up into tasks, and the tasks are assigned time-stamps to reflect a valid sequential execution. For example, each task might represent a single iteration of a FOR-loop with the time-stamp indicating the iteration number. Parallelization of this FOR-loop is essentially the same problem as parallelizing a discrete event simulation: one must execute the iterations in parallel but obtain the same results they would compute if the events were executed sequentially in increasing time-stamp order. The degree to which the techniques described here can be applied to parallelizing arbitrary computations is yet to be explored.

Chapter 4

Parallel Simulation of Circuit-Switched Networks using a Parallelizing Compiler

4.1 Introduction

A parallelizing compiler is defined as a program which takes an application written in a conventional sequential programming language, determines which parts of the program can be executed in parallel, and produces the machine level code to run on multiple processors. The major motivation for such a compiler is that existing sequential programs can be ported to a multiprocessor architecture without changing the source code, but with a subsequent decrease in execution time. Thus, time and money is saved as a consequence of not having to re-write the program, learn a new language or tailor the program to the particular multiprocessor architecture. The experiments described in this chapter were intended to explore the utility of such an approach for speeding-up the execution of a circuit-switched telecommunication network simulator.

The approach taken was to try to discover the "path-of-least-resistance" to achieving significant speed-up. That is, to find out how to get the most speed-up from the simulator with the minimum of effort. With any multiprocessor architecture, the best way of fully exploiting the parallelism available is to re-code the application in the machine's own parallel language using inherently parallel algorithms wherever possible. In this case, we are trying to exploit the automated facilities and expend as little development effort as possible. The

nearest relevant research to this exercise done previously was by Chandak and Browne [26] and by Reed [27]. Both worked on parallelizing discrete event simulations of queueing network models and achieved relatively poor results (see chapter 2).

The original simulation program used was written by Nadereh Eshragh at the University of Durham as part of her Ph.D. thesis looking into dynamic routeing in circuit-switched telecommunication networks [168]. The program was written in Fortran 77 for execution on an Amdahl mainframe. The simulator was written to work with networks of up to ten fully-connected exchanges or nodes; though the program could easily be extended to larger numbers of nodes and also to networks which are not fully-connected. The routeing strategies available are fixed routeing, random routeing, automatic alternative routeing, least busy alternative routeing, dynamic alternative routeing and stochastic learning automata using linear reward inaction or linear reward penalty. The strategies are described in detail in her thesis [168].

The program, henceforth called NAD, requires two files; an input file, always called datsimt, consisting of the number of nodes, random number seeds, a capacity matrix and a traffic matrix; and an output file, which is usually called fort.8 (often a Fortran constraint), initially containing a simulation run title, the routeing information, an overload factor (scaling factor for network traffic intensity) and a trunk reservation parameter. Results are written into the output file along with the simulation input information[1].

The simulation consists of a number of time units, each time unit consisting of a cycle of the following actions. Firstly, the inter-arrival time, and the origin-destination pair of the next call arrival are generated. Next, the calls which have been completed during this inter-arrival time are cleared down. Then, an attempt to route the new call is made according to the specified routeing algorithm. Finally, the acceptance or rejection (blocking) of the call, along with the origin-destination pair information, is recorded into the appropriate arrays. Results are written into the output file at the end of each time unit. The operation of the program is discussed in much greater depth in the thesis.

Thus, we are dealing with very simple, highly aggregated, model of a circuit-switched network. Hence, many operational details of the network are ignored in order to focus on the performance of the routeing algorithms in relation to the overall blocking probability. In particular, it is not a true discrete event simulation; accurately modelling the contention

[1] All of the files used in this work are contained on a separately available floppy disk (see Appendix A for information on availability).

for trunk capacity between call arrivals and departures. As this model involves a great deal of matrix manipulation using DO loops there seemed a good possibility that some speed-up would be possible using a parallelizing compiler.

4.2 The Testbed Architecture

4.2.1 Hardware Architecture

The shared memory multiprocessor used in these experiments was an Encore Multimax. This was first designed in 1983 and is produced in several versions. The version available at the University of Newcastle's Computing Laboratory is a 520, known locally as NEWTON, with fourteen main processors and 160 Mbytes of shared memory. In computing terms the Multimax is a MIMD machine but is more commonly known as a shared memory multiprocessor (see section 1.4). The basic architecture is illustrated in figure 4.1.

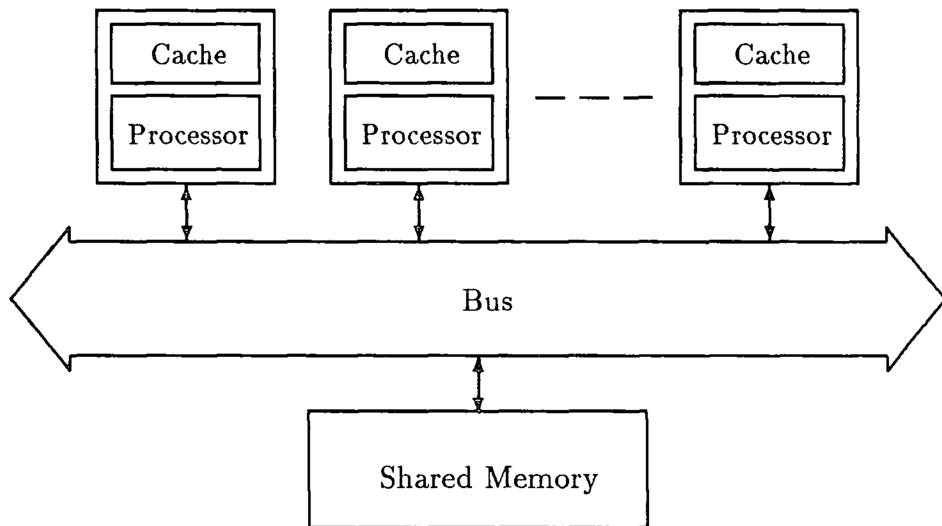


Figure 4.1: Shared memory multiprocessor architecture with a single bus and local caches.

The Multimax is controlled by a central processor which provides general monitoring and diagnostic facilities. This is based on a National Semiconductor NS32016 running at 10 MHz. The main processors are based on a processor/co-processor pair, the NS32532 and NS32381, running at 30 MHz. Encore claim a peak performance of about 8 MFlops for a main processor. Each main processor has a 256 kbyte cache memory controlled from the memory management circuitry in the NSC32532.

The complete system is built around a bus designed by Encore called the nanobus. It is made up of three individual buses; a 32 bit address bus, a 64 bit data bus and a 14

bit vector bus. In addition, each bus has an extra byte wide channel for parity checking. The Multimax 520 is so called as it has space for 20 card slots on the nanobus. The clock speed of the bus is nominally 12.5 Mhz giving a total memory bandwidth of 100 Mbytes per second. Inevitably, the bus is the bottleneck for this type of machine partially alleviated by the local cache memories.

4.2.2 Software Architecture

For the Multimax, Encore developed two varieties of UNIX : UMAX 4.3 and UMAX V. The former is based on Berkeley UNIX 4.3 BSD. and the latter is an implementation of the System V UNIX from AT&T. UMAX 4.3 is the system available on NEWTON. The languages available are C, UMAX Fortran and Encore Parallel Fortran (EPF) [169].

The UMAX Fortran compiler can produce standard sequential code for running on a single processor, or parallel code for running on multiple processors using EPF. The sequential Fortran meets the ANSI Fortran 77 standard and also includes most of the VAX/VMS extensions. In order to support the parallelism of the machine, a number of Fortran 90 features have also been included. Parallelization of programs may either be done manually or automatically by the parallel optimizer. The optimizer is implemented as a pre-processor to the compiler. If required, the programmer may take the output from the pre-processor to perform further parallelization by hand. The compiler also has an option to generate execution profiling code. Thus, when the program is executed a trace file is produced which can be analysed using the utility `gprof` to produce a report. The report will contain exact call counts, call graph arcs and statistically approximate timing data for a process as well as other data to aid in optimizing the program. Conventional Fortran 77 debugging tools can be used, such as `debug` and `dbx`, as well as the optional enhanced `fdb` debugging tool.

EPF analyses the source code to determine which program segments can be executed concurrently and converts them into parallel Fortran constructs. EPF uses explicit synchronization statements and local variables to improve concurrency automatically. The most fundamental optimization performed is loop spreading. This consists of converting `DO` loops into `DOALL` (parallel) loops; where all the statements in the loop can be executed in parallel. Loop spreading is enabled by EPF by splitting existing loops, re-ordering statements, introducing new variables and introducing explicit synchronization and ordering controls.

The extent to which any program can be parallelized using loop splitting and the asso-

ciated techniques described above is limited by four primary forms of data dependency. A flow dependency occurs where an assignment modifies a variable used in a later statement. Anti-dependence occurs where an assignment is made after a variable is altered. Output dependencies occur where an assignment must complete before another is made and control dependencies are caused by conditional statements which are dependent on prior statements.

The parallel programming tools provided with the Encore Multimax facilitate task creation and synchronization. A UMAX process contains the program, as a set of machine instructions and reserved address space, and operating system data (the necessary environment support structure). Multiple tasks must be created for parallel processing, with each task able to carry out a portion of the program on different processors. While the operating system treats each task as an independent process, there is a significant difference between a task and a UMAX process: tasks can share memory with the parent process and all other tasks spawned by that process; thus the group of tasks spawned by a process form a task set. Tasks are timeshared in the conventional manner regardless of whether or not they share memory. UMAX handles timesharing by context switching, idling one task and letting another use the processor while storing registers and program counters in order to restore the original task in its turn.

The Encore Parallel Fortran extensions exist within a task-based run-time model, the Encore Parallel Runtime library (EPR). On program start-up, an EPF program is initialised to use the EPR. A task shares the code of the main program, but has an automatic data stack, registers and a program counter of its own. The number of tasks that are available for use in parallel constructs can be specified by setting the environment variable EPR_PROCS. If not specified, the default is to use one task, and hence execute on a single processor. Built-in EPF functions track the number of tasks and their identities. In deference to other users; `nice` must be used to run the program at a low priority if there are more than four processors specified to be used at run time using EPR_PROCS.

4.2.3 Execution Model

The execution trace of the parallel simulation is relatively simple and is illustrated in figure 4.2 for four processors. This follows from the description above of the run-time model. As can be seen, Amdahl's law will effectively be an upper-bound on the possible speed-up which may be obtained on the Multimax. The execution trace also indicates how the timing of the execution of programs on the Multimax is achieved. A version of the Unix `time` com-

mand uses a hardware clock to keep track of effective execution periods and ignores that time when execution is suspended. This clock is also used by the system for general time-keeping and time-stamping outgoing messages to other computer systems. A time-stamp is taken from this clock at the time that the program is initiated and terminated and the difference is used to calculate the execution time. The clock is also read at other points to time system commands and also to avoid including the time that the system spends waiting for processors to become available. This was borne out by running the same simulations at various priority levels (levels of `nice`) which resulted in the same execution times to the nearest second.

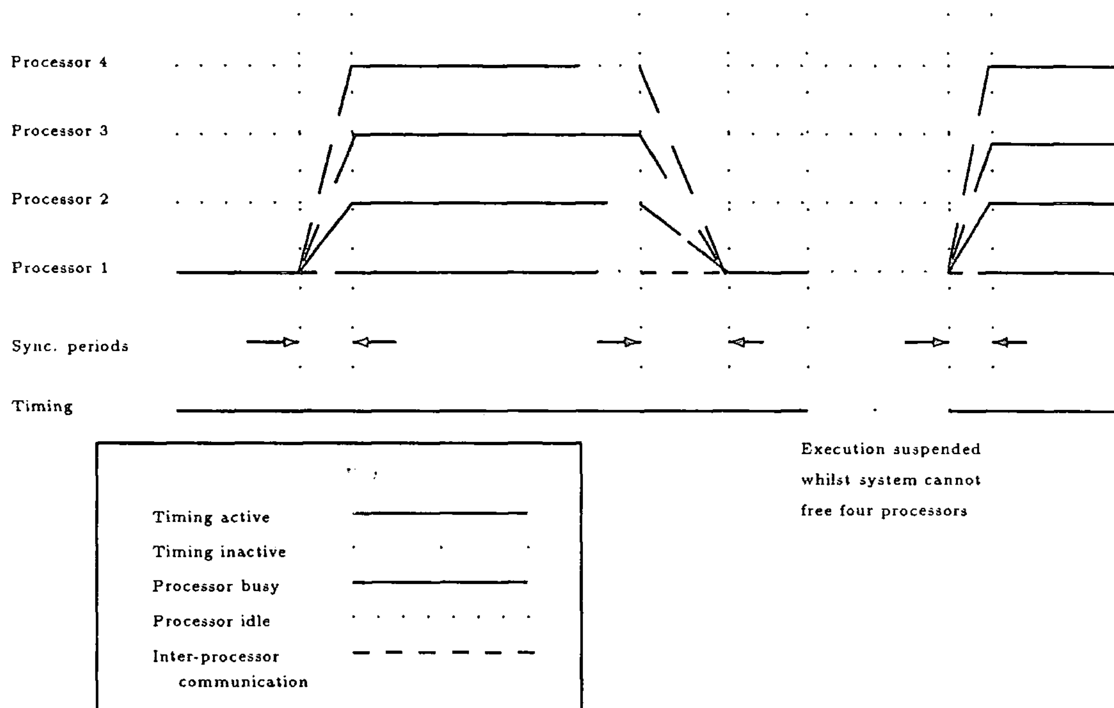


Figure 4.2: Execution trace of a parallelized program on the Multimax shared memory multiprocessor.

4.3 Discussion of Results

4.3.1 Introduction

The performance measure we are interested in is speed-up. But, as we have already seen, this is a dimensionless and relative measure. For these experiments it was decided to try to get as complete a comparison as possible. This was done by performing simulations on a single Multimax processor using a sequential compiler, simulations on a single processor

using the parallelizing compiler, as well as the true multiprocessor simulations. In addition, some sequential simulations were performed on a more modern uniprocessor machine to put the parallel simulations in context with the possible performance available without going to parallel simulation.

The models of the five- and ten-node fully-connected networks were taken from Nadereh Eshragh's thesis [168]. The traffic and capacity figures were chosen to reflect reasonably busy trunks in a realistic network. The link capacities were obtained using fixed route dimensioning using Erlang's formula constrained by a modularity factor of thirty circuits and random effects due to periodic upgrading of trunks. The twenty-node fully-connected and the ten-node sparsely-connected networks were constructed by the author. These were designed in a similar manner but were less realistic as the link capacities and traffic volumes needed to be much smaller to give lower overall network traffic and hence reasonable execution times. The topology of the ten-node sparsely-connected network is shown in figure 4.3 including the capacity of the links. The mean call holding time for all of the models was one time unit (nominally one second) and the traffic figures are for the average number of calls generated per second between each node pair. For each experiment, the simulator was run for 200 time units.

It should be noted that, at each stage of the experimentation, the results were compared with those collected by Nadereh Eshragh to ensure that they were reasonable. This was done by doing multiple runs[2] and ensuring that the blocking probabilities were within a standard deviation of the means. There were no problems observed in this respect.

Chronologically, the experiments proceeded as outlined below. The following sections will discuss the results in more detail.

- Uniprocessor simulations performed.
- First set of multiprocessor simulations performed on the five- and ten-node fully-connected models using the automatically parallelized simulator. Little or no speed-up observed.
- Run-time profiles made of first multiprocessor simulations to investigate poor speed-up. Parallelizing compiler is found not to have exploited all the available parallelism in the simulator.

[2]Ten runs were performed for selected experiments using different random number seeds.

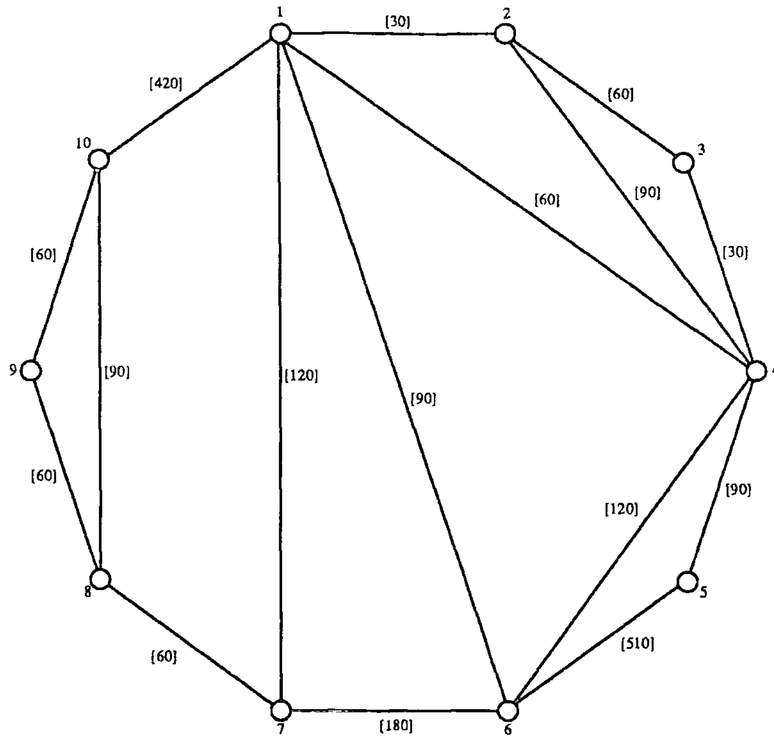


Figure 4.3: The ten-node sparsely-connected network. The numbers in the square brackets indicate the capacity in circuits of each link.

- Simulator is "hand optimized" in order to aid the compiler in recognising and exploiting parallelism.

- Second set of multiprocessor simulations performed on the five- and ten-node fully-connected models using the new "hand optimized" simulator. Reasonable speed-up observed for the ten-node model.

- Third set of multiprocessor simulations performed on the twenty-node fully-connected network model to investigate if the speed-up scales for larger models. This is shown to be the case.

- Fourth set of multiprocessor simulations performed on the ten-node sparsely connected network to investigate if reduced connectivity reduces performance. It does, but not significantly.

- Multiple simulation runs performed using the ten-node fully-connected model to investigate the variation in run-times (and hence speed-up) with different random number seeds. This is found to be not unreasonable as expected.

4.3.2 Uniprocessor Simulation Results

For these simulations the fixed routeing strategy was used and the same random number seeds. Initially, runs were done with various routeing options but there was little to choose between them in execution time and the fixed routeing strategy generally gave the shortest run-time. In each case, the simulation run length was 200 time units (seconds), the input was read from a file and the results written to a file and to the screen. The only change between implementations on different machines were the Fortran `READ` and `WRITE` statements which needed to be different depending on the Fortran implementation. In each case the results were checked against those obtained from the original implementation to ensure consistency. Execution times were obtained using the Unix `Time` command and the operating system (MTS) logging facilities on the Amdahl. The uniprocessor execution times are shown in Table 4.1.

Machine	*five-node model*	*ten-node model*	*twenty-node model*
Amdahl 470	607	782	3212
Encore Multimax 520	675	844	3308
Sun 4/460 SPARCstation	126	256	1173

Table 4.1: Execution times for the uniprocessor simulations in seconds.

A single processor on the Encore Multimax using the sequential option to compile the code yields roughly the same performance as the Amdahl mainframe. Thus the Encore Multimax times were used as the reference for the speed-up calculations. The difference in reference points for the speed-up are illustrated for the ten-node fully-connected network in figure 4.4. The more modern Sun SPARCstation is easily the fastest; as much as four times faster than the Amdahl or Encore Multimax.

For comparison purposes, an attempt was made to simulate the five-node fully-connected model on an IBM. PC. compatible using a commercial telecommunication network simulator Comnet II.5 produced by CACI. This is a flexible package allowing the simulation of many types of network with much greater depth of detail than in NAD. This extra detail, and hence more complex simulation models, led to a much longer run-time. In reality only a simulation run length of 4 seconds was possible due to insufficient hard disk space for the temporary files created by the simulation. Only 21 Mbytes were available! If this were scaled accordingly to reflect a run length of 200 seconds, the run-time would be 177,400 seconds (49 hours 16 minutes and 40 seconds!). It is interesting to note that the simulation model for Comnet can be entered as a flat text file or using a graphical user interface. Using a

flat text file is far quicker than the graphical interface once the user is familiar with the format. This is due to the increased model complexity requiring the user to traverse dozens of different menu and form screens many of which, for this application, weren't needed.

4.3.3 Multiprocessor Simulation Results for the Five-node Model

The multiprocessor simulations were begun by using the parallelizing compiler without any modifications to the source code. The results, in the form of a speed-up graph are shown in figure 4.5. Overall, the results are disappointing with no speed-up and with the trend being downwards as n increases.

These results were investigated by re-compiling the simulator with the profiler option and re-running some of the simulations. From careful examination of the profile reports the most expensive subroutines and functions could be noted, including the compiler generated functions which manage parallel execution. One of these, `mtask_fork`, which manages the re-synchronization of tasks before and after parallel execution phases, becomes more expensive as n increases. The overheads of this function effectively stifle any potential speed-up. This is illustrated in figure 4.6.

At this point, it was decided that work must be concentrated on reducing the cost of the most expensive tasks of the original program thereby reducing the impact of the `mtask_fork` function. Only two tasks were involved; the subroutines for clearing down calls and that for calculating link loads. These were by far the most expensive. This was achieved in both cases by hand coding. Data dependencies which couldn't be removed by the compiler were removed using local independent variables and coding some of the function calls in-line. This allowed the compiler to achieve greater parallelization in the tasks by loop spreading (converting `DO` to `DOALL` loops)[3].

This "hand optimized" version of the simulator yielded slightly better speed-up results as shown in figure 4.5. However, there is still no evidence of speed-up and the trend is still downwards as n increases.

The simulations were again profiled showing that the execution times of the hand-coded functions had decreased but the number of calls to `mtask_fork` were considerably increased due to the increased parallelism. Thus, the synchronization overheads still stifled the speed-up as n increased. Therefore it was decided that a larger simulation model might lead to

[3]The Fortran source files of the auto-parallelized program and the hand optimized program are contained on a separately available floppy disk (see Appendix A for information on availability).

more significant speed-up as the synchronization overheads would remain about the same but the amount of parallelism would increase.

4.3.4 Multiprocessor Simulation Results for the Ten-node Model

The results shown in figure 4.7 using the 'hand optimized" version of the simulator are much more encouraging with a net speed-up for all n. There is an intriguing dip in the speed-up graph for six, seven and eight processors, picking up again for nine, peaking at ten and dipping again above ten. This type of behaviour is not uncommon with machines employing parallelizing compilers. The greatest speed-up is most often observed when n coincides numerically with the granularity of the application's parallelism. Where it doesn't match more overheads are incurred as extra tasks are run on less than the full complement of processors. This decreases the effective utilisation of the processors available. This phenomenon is described for Cray Parallel Fortran by Almasi and Gottlieb [21]. The other possible explanation for such phenomena is statistical variation, as each point on the speed-up graph is only the result of two experiments. To allay this fear, ten runs of each experiment (on n processors) were performed with different random number seeds. The results can be seen in figure 4.8. It is clear from these that even though the "granularity" effect is present, it is not as marked as was first thought from figure 4.7.

The execution profiles bore out the assumption made after the five-node model experiments in that the compiler generated functions had much less impact on the execution time; see figure 4.6. That is, the processors spend much more time executing tasks in parallel than previously with the five-node model; thus making the synchronization overheads less significant. The fastest executions of the ten-node model were with four, five and ten processors which were equal to the nearest three seconds of execution time.

The ten-node sparsely-connected network is a modified version of the fully-connected network. This means that the results should logically have lower execution times, due to the reduction in overall traffic intensity, but lower speed-up figures, due to the extra overheads in routeing. The routeing for sparsely-connected networks is handled by the simulator with the addition of fixed routeing tables. The results for the ten-node sparsely-connected network are shown in figure 4.7. It shows that the logical speculations are largely confirmed; both lower execution times (actually the lowest as it had the lowest overall traffic intensity) and lower speed-up figures. The shapes of the speed-up graphs for the two networks are similar but there is a marked difference between the "hand optimized" simulator version

for eight processors and above. The sparsely-connected network results dip much further and do not recover anything like as well for ten processors. The extra work in routeing and clearing down calls is obviously allowing the parallel synchronization overheads to have a more significant impact than for the fully-connected network whose routes never involve more than two links.

4.3.5 Multiprocessor Simulation Results for the Twenty-node Model

The twenty-node model was created to investigate whether the encouraging results found with the ten-node model would scale for larger models. The network was created by hand with only two rules: the overall traffic intensity should not be so large as to cause excessive execution times, and that the link capacities should reflect the same modularity factor of thirty circuits as used for the previous models. The only change required to the simulator source was to increase the size of the data structures to accommodate the larger model.

The speed-up results are shown in figure 4.9. These indicate that speed-up does increase with model size. The "granularity" factor is not observable in the shape of the speed-up graph at all. Indeed, there are no peaks in the graph for the "hand optimized" version of the simulator, only a slight dip for eight processors. The speed-up is still increasing as we run out of available processors. This would indicate that the increase in the parallel synchronization overheads as the number of processors increases is much less significant.

The following observations are on a slightly more pessimistic note. The results for the original automatically parallelized simulator version are the best of all, see figure 4.9, but still no speed-up was observed. The twenty-node model results are the only ones which show the "hand optimized" simulator performing worse than the original for a single processor executing the parallel simulation. Indeed, it is the worst for all of the models. This may be due to cache misses as the model is now (obviously) larger. It should be noted that the code for the new simulator version is 127,063 bytes and the static data structures occupy 84,140 bytes compared with a processor cache size of 256 Kbytes. This can only be an educated guess as the profiling tool was not used to trace cache misses due to the much longer execution times involved; usually at least two or three times longer.

4.4 Conclusions

These experiments have highlighted the fact that there needs to be sufficient parallelism in a problem to make it possible to achieve reasonable speed-up when porting it to a multiprocessor architecture. The results for the five-node model were poor as the model did not contain enough parallelism to offset the cost of the parallel synchronization overheads. The threshold, or "break-even" point, was obviously passed with the ten-node model and the results for the twenty-node model indicate that this approach will scale to larger network models though more processors would obviously be needed to take full advantage of the available parallelism. It is known the effective break-even point is difficult to predict, even by professional parallel programmers; though some prediction can be made with the help of Amdahl's law (see Patterson and Hennessy [22] pp. 8–11).

Figure 4.11 shows a speed-up graph for the results of the hand optimized version of the simulator executing the fully-connected network models. Superimposed are the contours for Amdahl's law for various values of Wp (Wp is the fraction of the simulator run-time executed in parallel). As a measure of the success of the ten-node simulations, consider the following. The worst simulation was with seven processors giving a speed-up of 1.65. This indicates, using Amdahl's law (see section 1.3.1), that the percentage of the program parallelized was 45.6% giving a theoretical maximum speed-up of 1.84. The "best" simulation was with four processors yielding a speed-up of 1.99. This indicates a percentage parallelized of 66.3% and a theoretical maximum speed-up of 2.97. These comparisons make the results obtained look quite reasonable.

The complicating factor is of course due to the parallel synchronization overheads which are assumed to be negligible in the derivation of Amdahl's law. Interestingly, when the profile reports for the ten-node fully-connected model simulations were examined, between 22.6% ($n = 4$) and 44.3% ($n = 14$) of the execution times were expended on synchronization operations. Removing these overheads would yield speed-ups of 2.57 and 3.12 respectively. Furthermore, the percentage of the program parallelized indicated by these revised figures is closely grouped between 71.7% ($n = 6$) and 76.7% ($n = 10$); a theoretical maximum speed-up of 4.30. Thus, a revised graph of speed-up against n for the ten-node model would follow Amdahl's law for $Wp = 0.75$ quite closely. It must be noted, however, that overheads due to cache misses have not been removed though their effect must be much lower than the synchronization overheads.

The speed-up results for the twenty-node network model have not been investigated more closely due to the excessive execution times incurred when using the profiling tool as mentioned previously.

The Encore Parallel Fortran package would seem to be quite reasonable for this kind of exercise. The most difficult aspect, as always with parallel processing, is debugging. In this case, the parallel simulation can be run on one processor with the debugging tool `fdb`. This was never severely put to the test in this case. The other debugging alternative is to write the application as a sequential program initially and debug this with the standard tools before proceeding with the parallelization. The profiler `gprof` proved to be an essential tool in highlighting problem areas and assessing performance, echoing again the findings of Reed [27]. The hand coding performed as part of the exercise was not particularly arduous involving finding out how to do it, doing it, and debugging successfully. The most interesting point to note is that the key to success was knowledge, not of the machine's architecture, but of the compiler's "hooks".

The parallelizing compiler approach is obviously attractive if an acceptable level of speed-up for the simulation can be obtained. Unfortunately, discrete event simulations have been shown to achieve little or no speed-up using this approach [26,27]. We must therefore look to the distributed model components approach to give us more general solutions for successful parallel discrete event simulation.

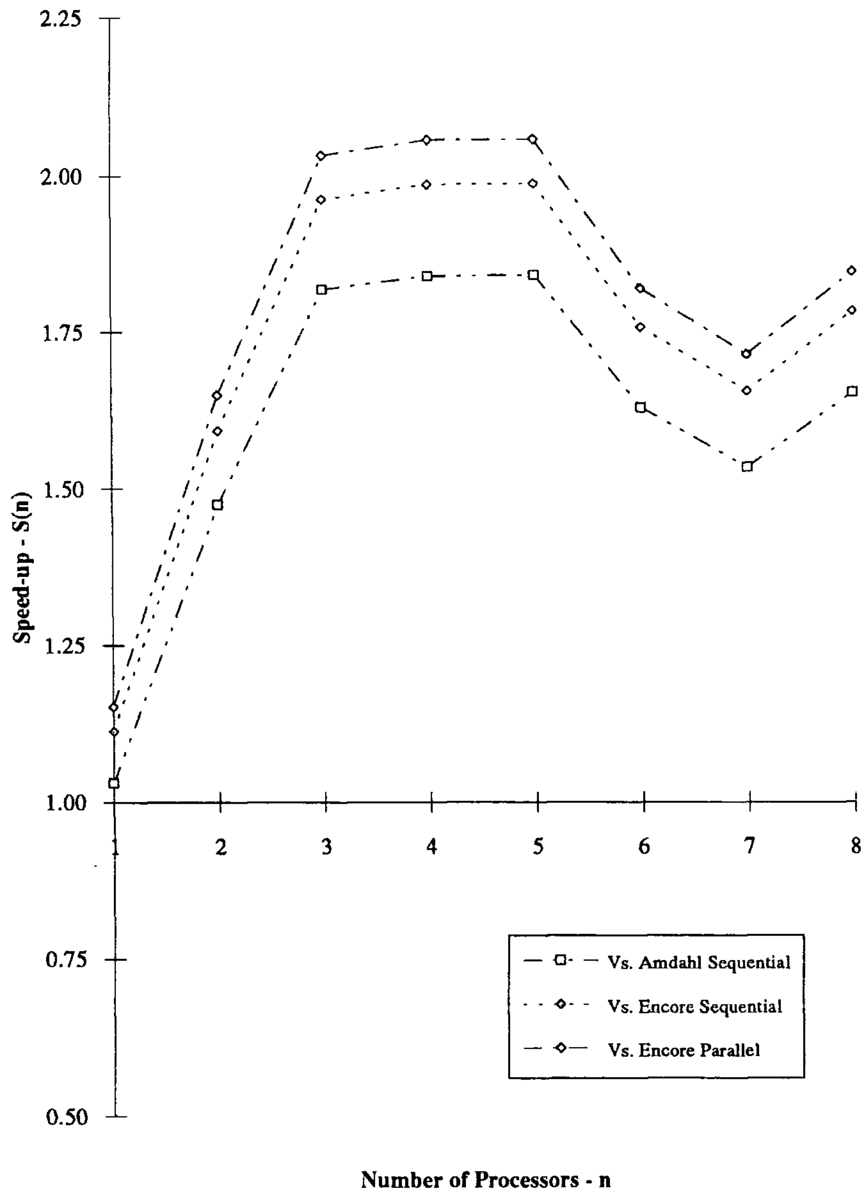


Figure 4.4: Speed-up results for the ten-node fully-connected network comparing the alternative reference times

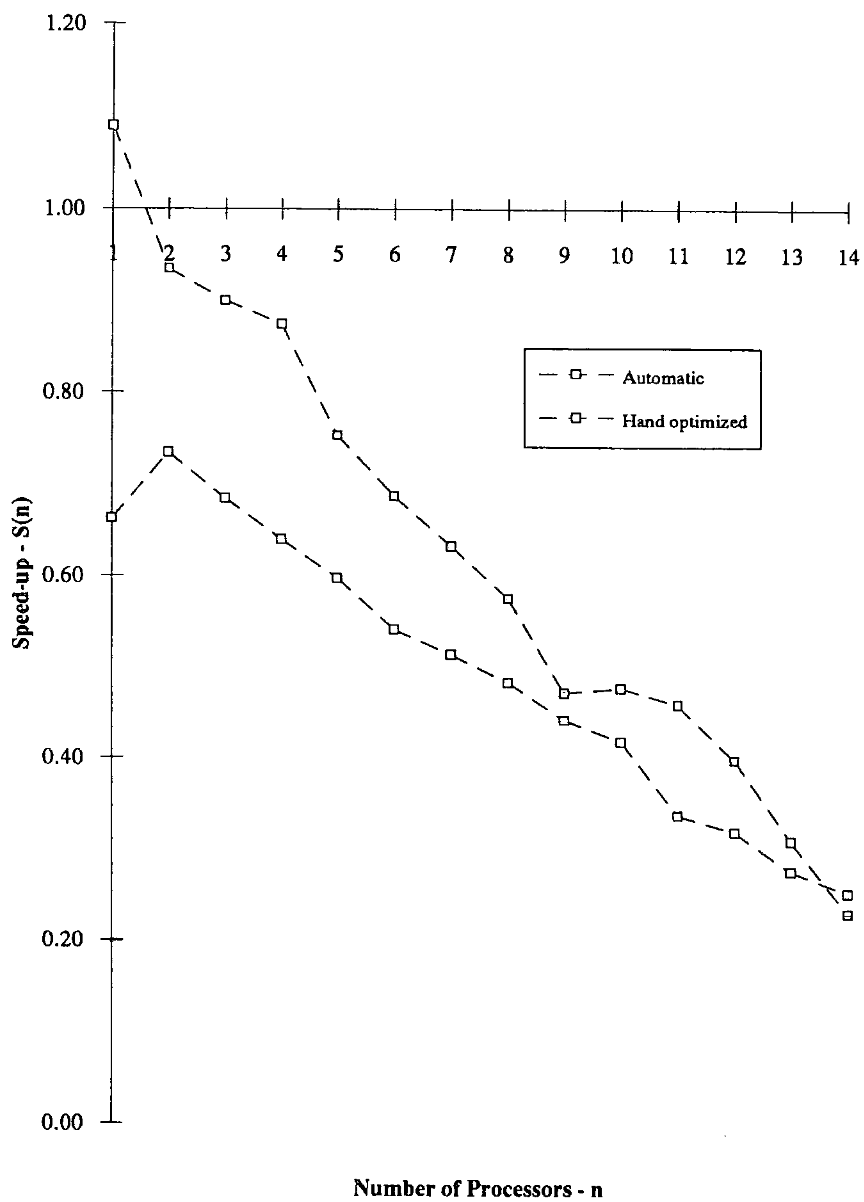


Figure 4.5: Speed-up results for the five-node fully-connected network model

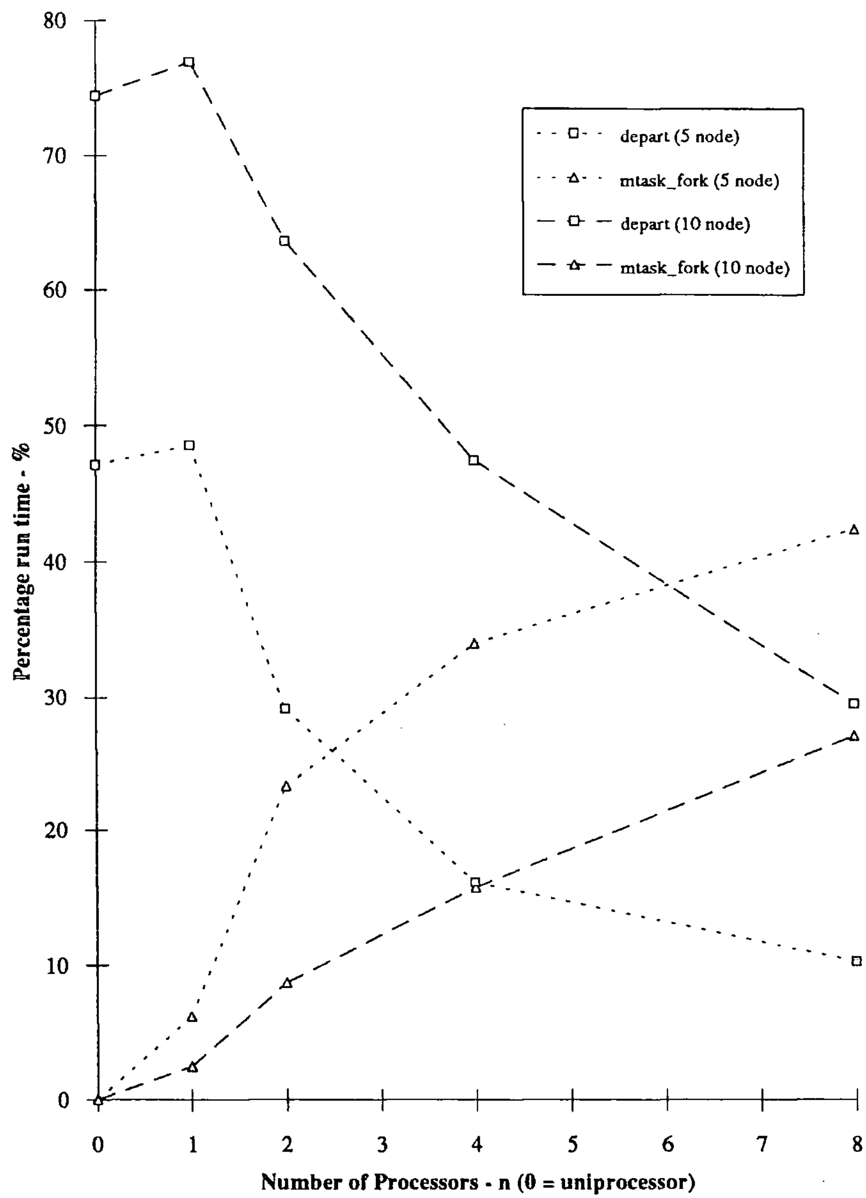


Figure 4.6: Comparison of percentage run-times used by the two most expensive processes using the automatically parallelized simulator

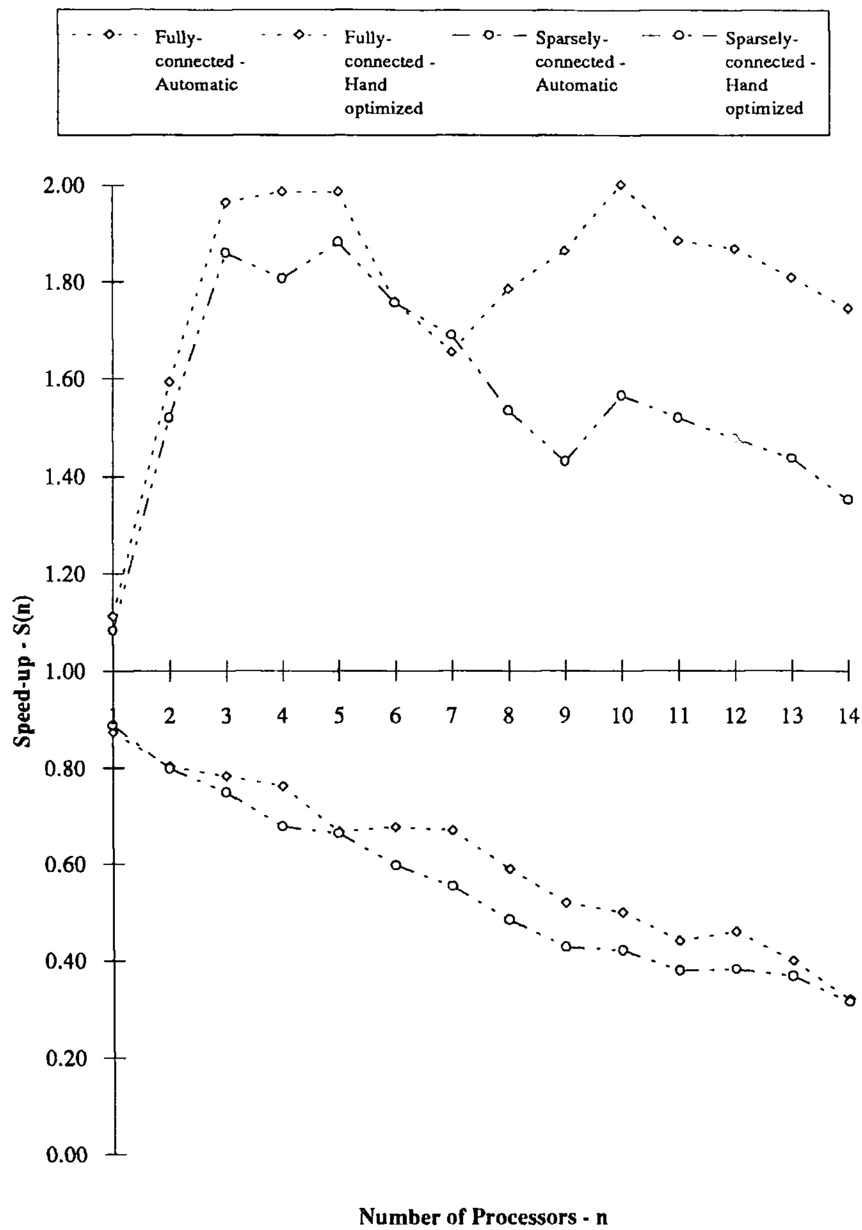


Figure 4.7: Speed-up results for the ten-node fully- and sparsely-connected network models

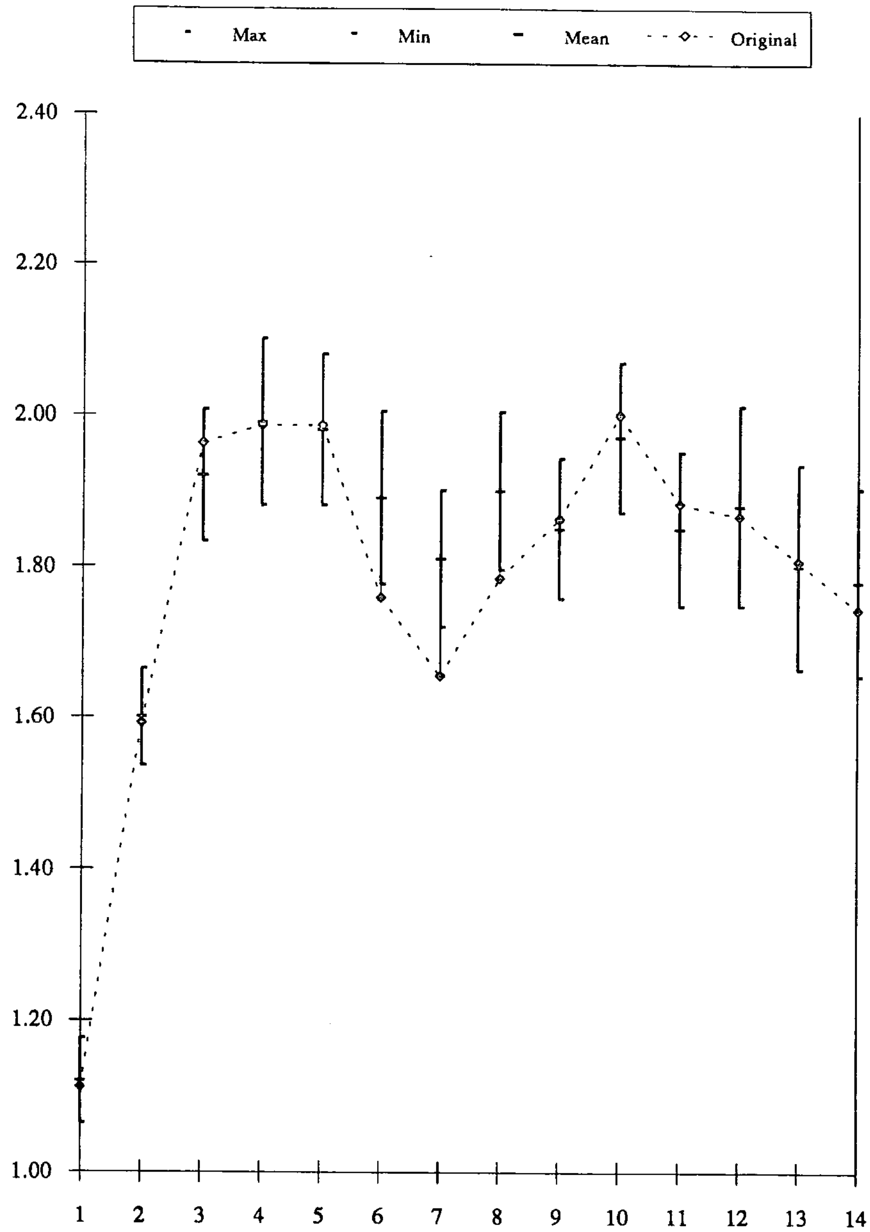


Figure 4.8: Speed-up results for the ten-node fully-connected network model showing means and standard deviations from ten simulation runs

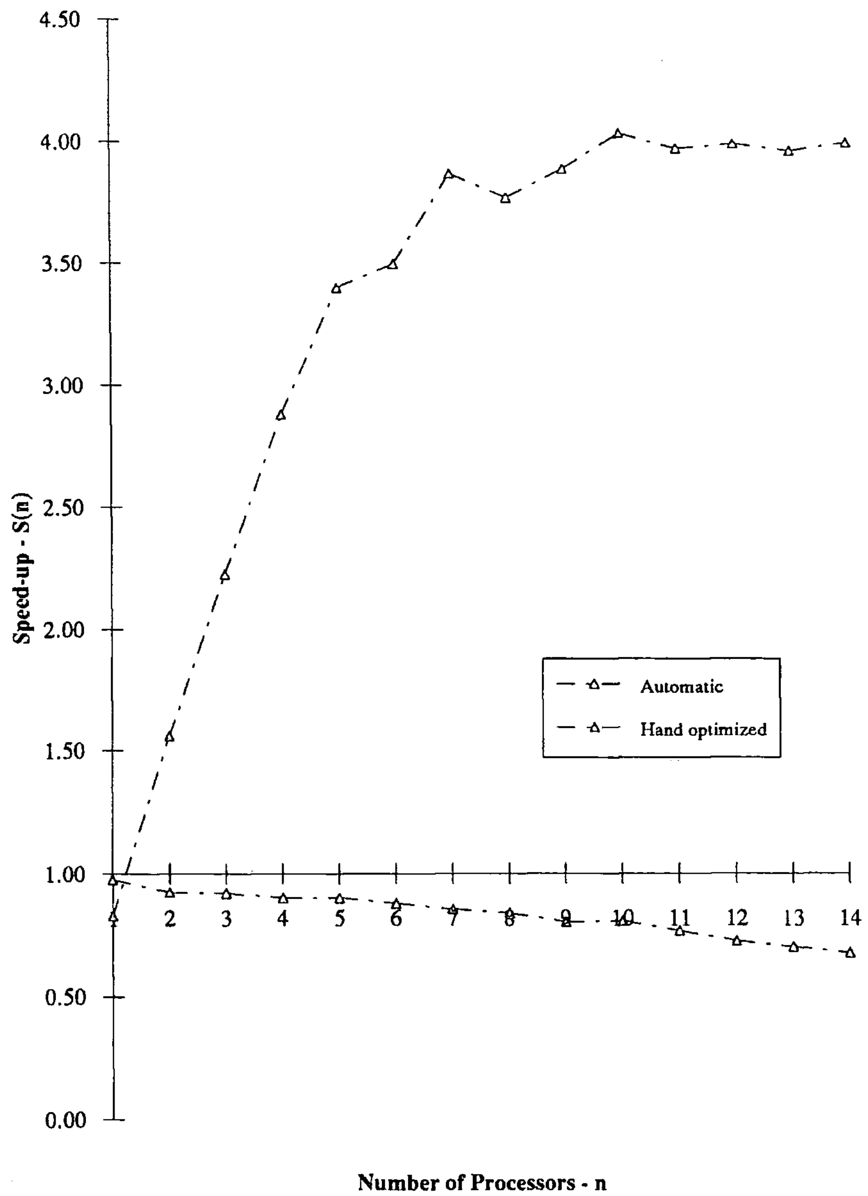


Figure 4.9: Speed-up results for the twenty-node fully-connected network model

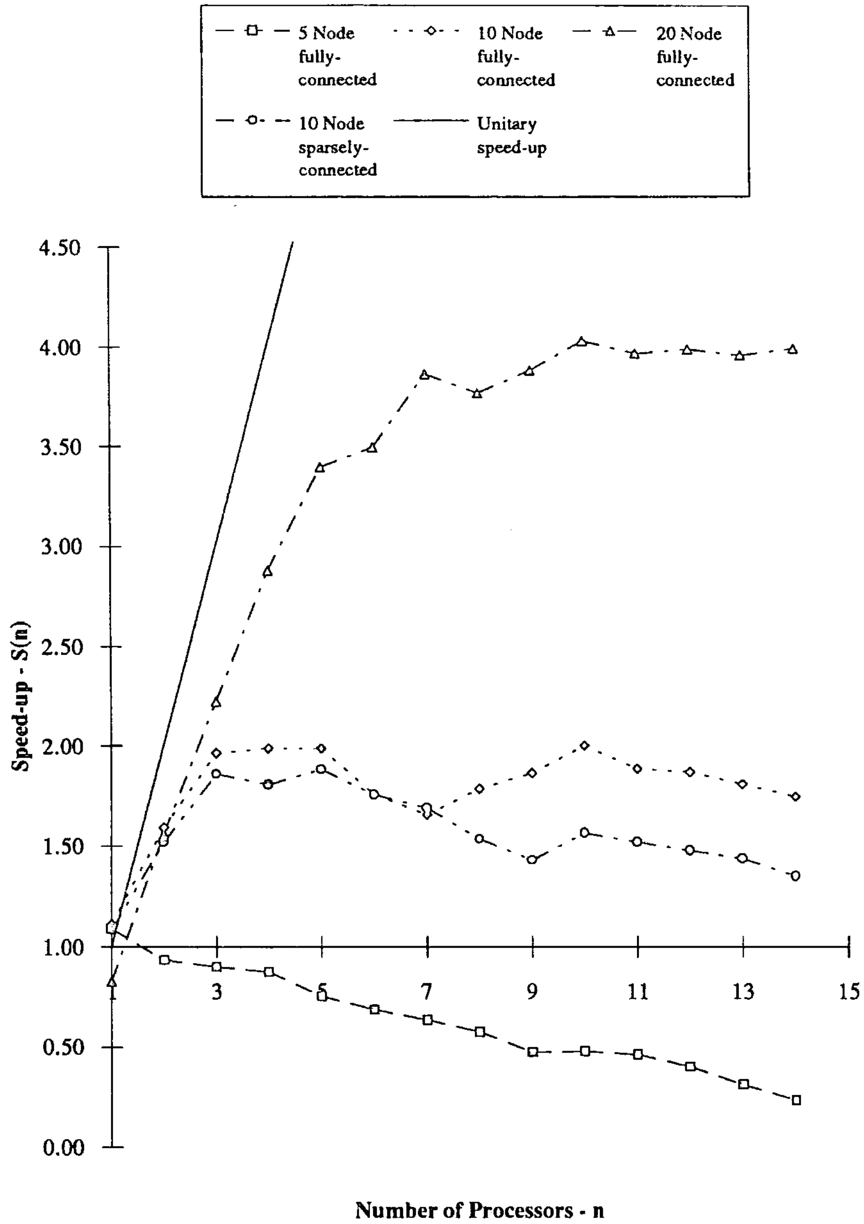


Figure 4.10: Speed-up results comparing the performance of the hand optimized simulator for all network models

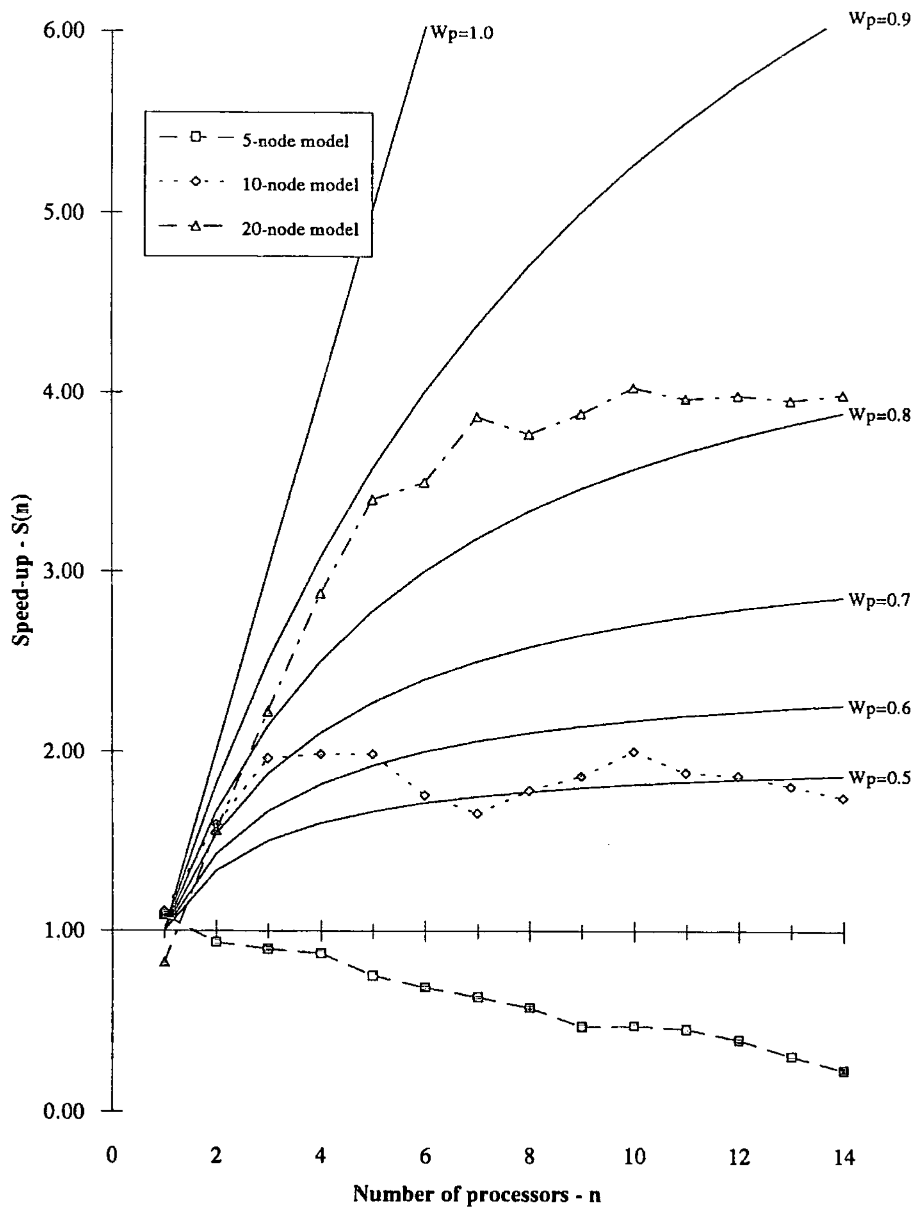


Figure 4.11: Speed-up results showing the performance of the hand optimized simulator for the fully-connected network models with Amdahl's law

Chapter 5

Parallel Simulation of Queueing Networks

5.1 Introduction

Queueing networks are often used as the basis for simulation models of packet- and cell-based telecommunication networks as well as many other engineering systems. As such they can be used as a useful generic benchmark for parallel simulation. Chapters 2 and 3 have described the work of various parallel simulation practitioners who have shown that discrete event simulations, particularly of queueing networks, cannot be parallelized. The most successful method reported for parallel simulation of queueing networks is that of distributed model components. This chapter describes a series of experiments conducted to evaluate the performance of various simulation synchronization algorithms using this approach.

The simulation models were constructed using YADDES (an acronym for Yet Another Distributed Discrete Event Simulator). The YADDES system is set of tools for constructing discrete event simulations which was developed in the Computer Communication Networks Group, Department of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada. The implementation used for these experiments runs on a single Inmos transputer or a hardwired cube of eight transputers. A manual is available for this implementation [170] which should be read in conjunction with the original manual [171]. There is also a discussion of it's implementation by the author [172].

The parallel discrete event simulation algorithms supported by YADDES are a sequen-

tial (uniprocessor) event-list (EL), a distributed multiple event-list (ML), a conservative synchronization algorithm based on that of Chandy and Misra (CM) and and optimistic synchronization algorithm using the concept of Virtual Time (VT). YADDES allows a common simulation model description to be compiled to use any of these algorithms thus enabling direct performance comparisons.

5.2 YADDES — Yet Another Distributed Discrete Event Simulator

The principle features of the YADDES system are;

- The YADDES simulation specification language and compiler.
- A set of libraries which support various simulation time synchronization methods with trace and debugging support.
- A pseudo-random number generator package which supports multiple, independent pseudo-random number streams.
- A distributed statistics collection and reporting package.

The user begins by writing a YADDES specification of the desired simulation model. YADDES then compiles the specification into a collection of C language functions. These functions are then compiled using the relevant C compiler and then linked to a simulation synchronization library to form a complete program which performs the simulation. The user need not be concerned with the synchronization method used. In fact, every YADDES specification can be executed using any method merely by linking to the appropriate library. Provided that the specifications are coded properly, the results of a simulation are independent of the synchronization method used.

The modelling approach used in the YADDES system is based on that of Chandy-Misra-Bryant (CMB) distributed discrete event simulation (see chapter 3). The real-world system is modelled by a collection of physical processes (PPs) that periodically exchange information. This exchange of information takes place at discrete points in time. Every instant at which one PP provides information to another is called an event. The computer simulation of the real-world system is obtained by constructing a computer program in which the behaviour of each PP is modelled by a logical process (LP). The exchange of information

by the PPs is modelled in the simulation by the exchange of messages by the LPs. Since the computer simulation does not execute in real time, each LP has its own notion of time and each message is time-stamped with the time of the corresponding real-world event.

The YADDES system provides a language and compiler that aids in the specification of the behaviour of LPs. An LP in the YADDES system is a general state machine. A general state machine has an arbitrary (finite) number of inputs and outputs and a (finite, albeit possibly large) set of states. The state machine is driven by the occurrence of event combinations. An event combination is a collection of one or more input events having the same time-stamp. In response to an event combination, an LP may change its state and produce zero, or more, output events on each of its outputs. The YADDES specification language is used to specify the state of an LP and to associate programs with event combinations. These programs being written out as sequences of C language statements.

In the YADDES system, the connections between LPs are static. Each input of every LP must be connected to the output of another LP; they require unity fan-in. The output of an LP may be connected to zero, or more, LPs; they may have arbitrary fan-out. The YADDES specification language also provides a means for describing the connections between LPs.

The YADDES system currently supports four different synchronization methods. However, the YADDES system hides the details of the underlying synchronization method so that the user need only be concerned with the specification, not the implementation of the simulation. There are three important advantages of the ability to support different synchronization methods. First, by executing the same specification using different synchronization methods, it is possible to directly and quantitatively compare the performance of the synchronization methods. Second, the user can change the synchronization method used without having to re-code the simulation specifications. In this way, the most efficient synchronization method can be chosen experimentally. Third, not only can new synchronization methods be added relatively easily, as they are entirely decoupled from the model description, but aspects of the existing methods can be also be altered for investigation.

5.2.1 Sequential Event-list Synchronization

The sequential simulation library uses the traditional discrete event simulation method. A single data structure, called the event-list, is used to hold future events. The event-list is sorted by time and by the LP identifiers. The basic execution cycle involves removing events from the head of the event-list, forming event combinations, and causing the appropriate

LPs to perform the action associated with the given input event combination. When an action causes output events, those events are inserted into the event-list.

5.2.2 Distributed Multiple Event-list Synchronization

This method is a simple extension of the sequential simulation method for execution on a multiprocessor. In this method, each processor has its own event-list. In addition, one processor (the scheduler) has special status and acts to coordinate the other processors. The basic execution cycle is somewhat more complex in order to guarantee correct execution on the multiprocessor. First, each processor sends a message to the scheduler indicating the simulation time of the next event on its local event-list. The scheduler selects the minimum next event time and broadcasts this value. Each processor having this minimum value removes events from its event-list, forms event combinations, and invokes the appropriate LP's actions. When an action causes an output event, that event is either inserted into the local event-list, or a message is sent to a remote processor requesting that it insert an event into its local event-list. When a processor is finished executing all the actions for a given value of simulation time, it sends a completion message to all its successors indicating that it is done. Finally, each processor waits until it receives a completion message from all its predecessors. At this point the execution cycle is complete and may begin again.

In YADDES, each LP is statically assigned to a processor. This assignment is specified in the YADDES source. Since the assignment is static, each LP knows a priori whether to insert output events into its own future event-list or to send a message to another processor.

5.2.3 Conservative Distributed Event-list Synchronization

In conservative (CMB) distributed simulation each LP runs as a separate task. Tasks may run on the same or on different processors and the number of tasks need not be the same as the number of processors. In this environment, the LPs consist of the YADDES model, state and output tables, all encapsulated in an envelope.

The basic execution cycle begins when an envelope receives a message. The envelope buffers messages until an event combination can be formed. An event combination with simulation time t can only be formed when an envelope has received an event message for each of its inputs having time t' greater than or equal to t. When an event combination is formed, the LP is activated. If a LP causes an output event, it sends an event message to the envelope of the appropriate LP.

This synchronization method has the potential for deadlock. The current implementation does not use any deadlock detection/recovery scheme and the user is required to explicitly avoid deadlock. Two approaches are provided in order to achieve this. NULL-messages are events inserted by the user which assert that no event has occurred up to the given time-stamp, t_{Null}. This allows an envelope to determine that it has received all the events up to time t_{Null} on a given model input. In certain cases, it is possible for the model itself to know that no event will occur on one or more of its inputs up to a certain time. In such cases, the envelope can be informed that no event will occur and the envelope may safely ignore that input. This is termed by Loucks et. al. to be "additional knowledge" [113].

5.2.4 Optimistic Distributed Simulation Synchronization

The implementation of optimistic, or virtual time based, distributed simulation is based on that of the Time Warp Operating System (TWOS) [80,130]. As in the CMB mechanism each LP runs as a separate task consisting of the YADDES model, state and output tables encapsulated in an envelope. However the virtual time based mechanism cannot deadlock.

Every envelope has a local clock as before. When a message is received by an envelope, there are three possibilities; it's time-stamp is either before, after, or equal to the current local value of simulation time, called the local virtual time (LVT). If its time-stamp is after the LVT, an input event combination is formed and the LP is activated to process it or place it in the event-list it to be processed. However, if its time-stamp is less than or equal to the LVT, the envelope backs up to the time immediately prior to the incoming message time-stamp. This is facilitated by an elaborate checkpointing mechanism which allows an earlier state of the LP to be recovered. Essentially, an earlier state is restored, input event combinations are re-scheduled, and output events are cancelled by sending antimessages. The envelope has buffers which save past inputs, past states and antimessages. This state-saving overhead, known as checkpointing, can be reduced by decreasing it's frequency.

An important component of the implementation of the virtual-time environment is the fossil collection algorithm. Fossils are said to be past inputs, states and antimessages that are no longer needed and so may be discarded releasing memory space for further use. Essentially, these are events which are older than the current global minimum simulation time, called global virtual time (GVT). The YADDES fossil collection algorithm uses a circulating token message which visits each LP in turn collecting the LVTs. Once all LPs have been visited, each can update it's own LVT in the message, calculate a value for GVT

(the minimum value of all the LVTs), invoke the fossil collection algorithm and pass on the token message.

When a rollback occurs, output messages need to be processed in some way. There are two types of cancellation strategy - aggressive and lazy. When the aggressive strategy is used, antimessages are sent immediately when the rollback occurs. When lazy cancellation is used, antimessages are not sent immediately. Instead, they are placed in a buffer of pending antimessages. When the LP resumes execution, it will obviously generate output messages. If an output message is the same as a message that would have been cancelled during the rollback under aggressive cancellation, then the pending antimessage and the new output message annihilate[1]. If a new output message is generated, that is, one that was not generated previously, then this is sent. If the LVT progresses past the time-stamp of a pending antimessage, then the antimessage is sent as the original "real" message is now shown to be erroneous.

Finally, the number of LPs required for a simulation will often not match the number of processors available to perform the simulation. If the number of LPs exceeds the number of processors then a scheduling algorithm is invoked to select the next LP for execution. There are three scheduling algorithms available; round-robin (RR) scheduling, scheduling the LP with the minimum LVT (MVT) and scheduling the LP with the minimum message time-stamp (MMT) in it's input queue. All of these scheduling algorithms are non-pre-emptive.

5.3 The Simulation Models

The simulation models used were of a closed stochastic queueing network and a tandem queueing network. The configuration of the closed stochastic queueing network is of a four-dimensional hypercube consisting of sixteen queues which can be visualised as a cube within a cube, with a queue at each corner as shown in figure 5.1. It is a closed system in that the population of customers in the system is constant throughout the simulation. Each queue is initiated with a number of customers in the queue (the load) and these are processed in first-come-first-served (FCFS) order. Each customer is served with a negative exponential service time and then routed randomly to one of it's four nearest neighbour queues. In not using a global routeing scheme (ie. any node routes to any other node via intermediates) greatly simplified the model specification as a distinction need not be drawn

[1] Annihilation implies that both the "real" message and the antimessage are discarded.

between customers. The mappings of queues to processors was made simply by cutting the hypercube in the x-plane for two processors, the x- and y-planes for four processors and the x-, y- and z-planes for eight processors.

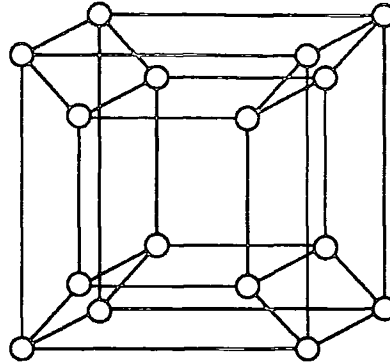


Figure 5.1: A 4-dimensional hypercube.

The second model, a tandem queueing network, was originally proposed for investigating policing in an asynchronous transfer mode (ATM) network[2]. Policing mechanisms are intended to control congestion and cell loss in the network; this subject is reviewed by Rathgeb [173]. In this case, the policing is done using a leaky bucket mechanism. The performance of the model is of interest in its own right, but it is quite large in simulation terms and would normally require long simulation times due mainly to the size and traffic intensity.

The performance measures of interest in the model are the cell delay variation collected at destinations and the cell loss which is collected at switch queues and leaky buckets. The cell delay variation can be examined by comparing the cell creation times with the time that they arrive at the destination. Cell delay variation was however not monitored in these simulations. Cell loss is typically very low in ATM networks and so very long simulations involving the generation and switching of more than 10^9 cells are often required to give results which are statistically significant. However, the simulation run length used in these experiments was deliberately short so as to allow reasonable run-times; and hence a reasonable exploration of the parallel simulation options.

The ATM switches were modelled by simple M/D/1 queues rather than an Orwell ring or other more complex switch. Each switch used five processes of three types, shown in figure 5.2; a queue, a server and a sink. The sink looks at the labels of the incoming cells

[2] ATM networks are introduced in the next chapter.

and removes from the stream all those which should terminate at that switch. The leaky bucket was also modelled as an M/D/1 queue. Thus, both the switch and the leaky bucket mechanism use the same queue and server models with different parameters. This was easily developed and at first was thought to be adequate if the queue length was set to an appropriate value. However, in a cursory exploration of the literature, leaky buckets are better implemented using a token pool so as not to introduce any delay as in Butto et. al. [174].

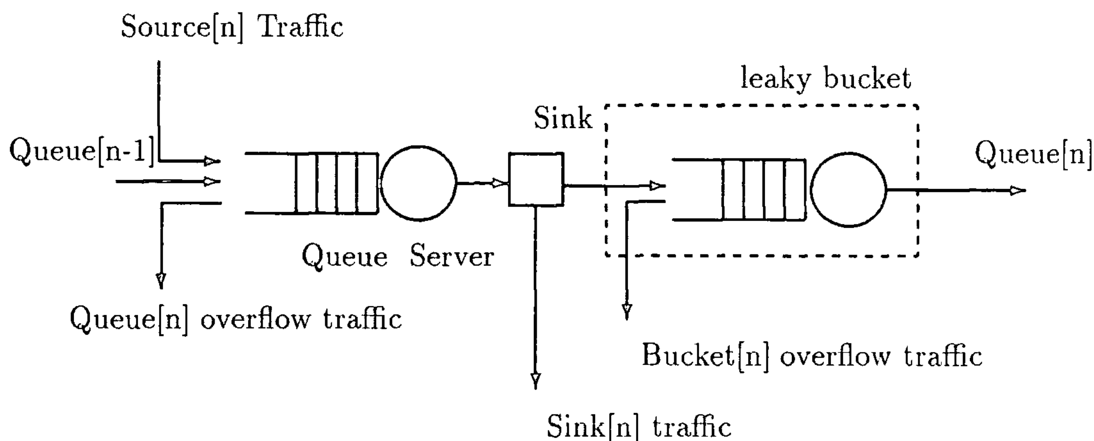


Figure 5.2: YADDES process model of one switch of the tandem queueing network.

The traffic sources were modelled with simple poisson sources. Ideally, for the ATM network modelling, an on/off source with a geometrically distributed burst length and silences with a negative exponential distribution needs to be developed, see Rathgeb [173].

The topology of the processors is a hardwired cube of eight Inmos transputers and so the process mapping is a straightforward "pipeline" arrangement for the eight switches. Each transputer carries a switch and a traffic source; except for the first which has an extra source feeding into the first switch, and the last which has no sink or leaky bucket. This arrangement was dictated by the way that some of the processes synchronize, which meant that breaks between queues and servers would not be practical as they would require two links. An alternative would be to move each leaky bucket to the next processor which may create a better load balance; this was however not tried. This indicates that there is a large amount of parallelism to be exploited and a large amount of scope for studying the effect of different mappings. Mappings to two and four processors used in these experiments are straightforward bearing in mind that a queue/server pair should not be split[3].

[3]The YADDES code for both of the models is available on a floppy disk (see appendix A for information on availability)

A major obstacle to realising the tandem queueing network model was the requirement for multiple instances of the same process with different parameters. This was done by introducing an extra "special" variable to YADDES called $label which could be used by a process to read an identification label and thus to be able to read the correct set of parameters from the configuration file. Some constants are also used to check that the correct number of parameters are read from the file.

5.4 Discussion of Results

5.4.1 Initial Results for the Closed Stochastic Queueing Network

The initial simulation experiments were begun whilst the transputer array only had 1 Mbyte of memory per processor. This was insufficient for the multiprocessor simulations using VT synchronization. So, attention was first made to the speed-up results of the CM simulations. These did not use NULL-message cancellation, as it was not enabled, and only used the lookahead afforded by assuming propagation delays over the modelled links between queues. This is known in YADDES as `epsilon` lookahead as it is the minimum available. The results are shown in table 5.1.

Synchronization mechanism	*No. of Processors*		
	1	2	4
ML	0.94	0.91	0.97
CM	0.36	0.73	0.89
VT	0.40	0.89	1.69

Table 5.1: Initial speed-up results using YADDES.

The speed-up figures, which are all calculated relative to the EL simulation, are disappointing. The speed-up figures for ML synchronization were not expected to be good but some speed-up should be possible and indeed was observed by Preiss [175] when simulating similar but somewhat larger systems. For CM, the results are very disappointing. However, it is well known that CM simulations are sensitive to the amount of lookahead. So, for the main simulation studies the lookahead was increased by making use of advanced simulation knowledge in terms of adding the next customer service time to the NULL-message time-stamp. This type of technique was described in chapter 3 and is discussed in detail for queueing networks by Loucks and Preiss [113].

The VT speed-up figures were completed once extra memory was available for the trans-

puters (4 Mbytes per processor). These again were disappointing but at least recorded some speed-up on eight processors. The two processor simulation would still not run within 4 Mbytes so this had to be done on two processors each with 16 Mbytes. This was true for all the two processor VT simulations performed. These simulations used aggressive cancellation, the round-robin process scheduling and a checkpoint interval (CPI) of one (ie. saving every state). It was known, from other work using YADDES by Preiss et. al. [143], that minimum virtual time scheduling and an increased CPI should give superior results. Lazy cancellation was also available which could possibly improve performance. These other options required extra coding and re-compilation to make them available, so they were held back until the modifications to the lookahead had been made.

One other change from the initial to the main simulation results was the method of timing the simulation runs. YADDES usually relies on the timing facilities offered by the operating system. For MS-DOS, the `time` command is not ideal as it was never intended for use in timing the execution of an application. Therefore, for the main simulation speed-up results a timing function was added to the YADDES synchronization libraries before compilation. This is always the first and last function called in a simulation run.

5.4.2 Main Results for the Closed Stochastic Queueing Network

Summary of Experiments

These were performed using the model described in the previous section modified to exploit more lookahead in terms of the service time for a customer plus the link propagation time. The full range of useful options available within YADDES was explored, listed below.

- Sequential event-list (EL) — for loads of 1, 4 and 8 customers per queue.
- Multiple distributed event-list (ML) — for loads of 1, 4 and 8 customers per queue, on 1, 2, 4 and 8 processors.
- Chandy-Misra (CM) — for loads of 1, 4 and 8 customers per queue, on 1, 2, 4 and 8 processors, with and without NULL-message cancellation (NMC).
- Virtual Time (VT) — for loads of 1, 4 and 8 customers per queue, on 1, 2, 4 and 8 processors, using aggressive and lazy cancellation, using checkpoint intervals (CPI) of 1, 2, 4, 8 and 16.

In addition, nine extra runs using different random number seeds were made of each of the following experiments to ascertain the mean and standard deviation of the run-times. These results are discussed in the relevant results sections.

- EL — load = 4.
- ML — load = 4, on 2 processors.
- CM — load = 4, without NMC, on 4 processors.
- VT — load = 4, aggressive cancellation, CPI = 1, on 8 processors.
- VT — load = 4, lazy cancellation, CPI = 16, on 8 processors.

The memory management performance of the VT simulations was investigated with a further ten runs. In order to produce results, the memory usage figures were coupled with the run-time and speed-up results from the equivalent simulation runs performed without the memory usage tracing code enabled. The run-times for the experiments with the tracing code enabled were inevitably much slower.

- VT — load = 4, on 8 processors, using aggressive and lazy cancellation and using checkpoint intervals (CPI) of 1, 2, 4, 8 and 16.

Multiple event-list (ML) Synchronization

The ML speed-up results, shown in figure 5.3 are still unimpressive, but some improvement is seen over the initial results, particularly at the higher load. This slight change in the result at a load of four over the initial results will be due to the change in the method of timing the runs; no other change introduced between the initial and main simulation runs would affect the ML runs. The ML results do show slight improvement with the increase in the load and with the number of processors but do not come near the speed-up results described by Preiss [175]. Here, a speed-up of just less than two was recorded at a load of eight. This is due to the model considered here being smaller and slightly less complex. The ML synchronization would obviously perform better with an application requiring more processing per event.

The multiple runs with different random number seeds for ML synchronization (two processors at a load of four) showed a standard deviation of just over 1% of the mean. The first time recorded, which was used for the speed-up graph, was actually the slowest. If the

mean were used for this point on the graph, this would give a speed-up of 1.04. The first figure was used as the random number seeds used in this run were the same as for the other simulation runs used in the graphs and thus affords a better comparison.

Conservative Chandy-Misra (CM) Synchronization

The CM speed-up results, shown in figures 5.4 and 5.5, are much improved over the initial simulation results. This is due primarily to the increase in lookahead in the model, though the change in timing method will also have a small impact. The other most noticeable feature is that NULL-message cancellation has a very marked affect on the speed-up; reducing it significantly at the lowest load (load = 1 customer per queue), improving slightly at the middle load (load = 4) and improving it greatly at the highest load (load = 8). Indeed, the speed-up figures with NULL-message cancellation at the highest load are the highest recorded of any of the experiments (4.02 on eight processors). This increase in effect with load is as expected; also, the overheads of the algorithm at lower loads obviously outweighs any benefits.

The multiple runs for CM synchronization exhibited the lowest standard deviation of all the multiple run experiments at just under 0.5%. Again, the first time recorded, which was used for the speed-up graph, was the slowest.

Optimistic Virtual Time (VT) synchronization

The VT speed-up results are shown in several graphs summarised in table 5.2 below. Each figure shows the speed-up against the number of processors for a certain cancellation strategy and CPI at all three loads.

Reference	*Cancellation Strategy*	*CPI*
5.6	Lazy	1
5.7	Aggressive	1
5.8	Lazy	2
5.9	Aggressive	2
5.10	Lazy	4
5.11	Aggressive	4
5.12	Lazy	8
5.13	Aggressive	8
5.14	Lazy	16
5.15	Aggressive	16

Table 5.2: Figure Numbers for the graphs of VT Speed-up results.

Figures 5.16, 5.17 and 5.18 show speed-up against CPI on two, four or eight processors respectively. The speed-up improvement from the initial results is largely due to the change from round-robin to minimum virtual time scheduling plus some small contribution due to the change in run-time measurement. For minimum virtual time scheduling, where there is more than one process per processor, the process with the lowest virtual time will be be scheduled next for execution. This tends to reduce the number of rollbacks created compared with round-robin scheduling.

The speed-up figures obtained are consistently good and relatively insensitive to load. There is some improvement in speed-up with load though this is not as marked as with CM synchronization. There seems little to choose between the lazy and aggressive cancellation strategies, this reinforces the findings by Preiss et. al. [143] that lazy and aggressive cancellation strategies perform similarly under minimum virtual time scheduling. However, lazy cancellation does slightly outperform aggressive cancellation at low settings of CPI. The opposite is true at higher settings; roughly equal at eight, aggressive better at sixteen. An optimum CPI for maximum speed-up is also discernible. This is discussed later in connection with the experiments on CPI and memory management.

The standard deviations for the multiple runs were again quite low at 2.1% for lazy cancellation on eight processors and a CPI of one and 1.2% for aggressive cancellation on eight processors and a CPI of sixteen.

Comparison of Speed-up Results for the Closed Stochastic Queueing Network

The different synchronization algorithms are compared at the three values of load in figures 5.19, 5.20 and 5.21. Only the best VT results are shown for lazy and aggressive cancellation at a CPI of two. Overall ML synchronization is the worst performer giving little or no speed-up. This proves the intuitive conclusion that any algorithm, such as ML synchronization, which tries to synchronize the parallel simulation using some form of global control will not result in significant speed-up.

CM synchronization exhibited both the best and worst speed-up results. At a load of one (figure 5.19) the algorithm performed much worse than the ML synchronization due to to the extra overheads of NULL-message calculation. With the addition of NULL-message cancellation this performance was exacerbated. At a load of four (figure 5.20) the performance is average, but at a load of eight the performance is the best of all with the addition of NULL-message cancellation. This shows that where there is good lookahead

properties in a model coupled with a significant message load CM synchronization can produce excellent speed-up results.

The VT synchronization speed-up results shown here are consistently good over all loads. Lazy cancellation is slightly better at low loads and we have also observed that the CPI should be optimized. In this case, the best interval is two[4].

One noticeable feature is that almost all the speed-up curves show some signs of saturation if the number of processors were increased much above eight, the limit used here. However, this is explained by the observation that the maximum amount of parallelism in the model is sixteen (the number of queues) so speed-up saturation is bound to start above eight processors. This will also be affected by the connectivity of the model and the mapping of this onto the processors; even numbers of processors will be most easily realisable. There would be an extra problem at sixteen processors due to the requirement for a connection back to the host as a transputer only has four physical links. This would have to be resolved using an extra, seventeenth, transputer.

5.4.3 Virtual Time (VT) Memory Management Results for the Closed Stochastic Queueing Network

The VT synchronization algorithm uses far more memory than the other algorithms. Indeed, for the initial simulation experiments, the multiprocessor VT experiments could not be run to completion within 1 Mbyte per processor and the two processor simulations could not be run to completion using 4 Mbytes. It was therefore decided that some investigation into it's memory usage would be useful. However, it is very difficult to obtain a time-average or a trace of the memory utilisation data for each process without significantly changing the behaviour of the simulation in terms of it's run-time and memory usage. However, it is possible is to keep track of the maximum memory utilisation for checkpointed state information and the message buffers for each LP as these are allocated separately. This monitoring lengthens the simulation run-time but barely changes the memory usage. The other possibility is to trace the execution of the simulation in terms of it's virtual time (VT) and global virtual time (GVT). The distance between the traces at any instant is proportional to the state memory usage. This technique has been used by a number of users to observe the progress of optimistic simulations.

[4]A CPI of three may have been better overall but trying one or two experiments yielded times slightly worse than that at two.

We have already observed that changing the CPI impacts the run-time (and hence the speed-up) of a simulation as the overhead of checkpointing is reduced but the penalty of a rollback is increased. Obviously, changing the CPI will also impact the amount of memory used to store checkpointed state information as well as the message buffers of each process. The experiments performed were on eight processors at a load of four. The memory usage figures are the maximum amounts used by one queue process on one of the eight processors to simplify analysis. To complete the picture of the memory map a single queue process occupies 598,107 bytes of memory and some memory will also be taken up by constants, variables, arrays etc. This is illustrated in table 5.3 taking the worst case scenarios with lazy cancellation and a CPI of one which requires 200,192 bytes total storage.

Number of Processors	*Number of Processes per processor*	*Total memory requirement*
1	16	12,772,784
2	8	6,386,392
4	4	3,193,196
8	2	1,596,598

Table 5.3: Estimated worst-case memory requirements in bytes for VT simulation.

This illustrates why the simulations using four and eight processors each with 1 Mbyte per processor and two processors with 4 Mbytes would not run to completion. Also, the four processor simulation had barely enough memory. However, these are "worst case" figures.

Figure 5.22 shows the maximum amount of memory needed for checkpointed state information against the CPI. It clearly shows that increasing the CPI substantially decreases the maximum state memory required though the impact tails off above eight. Figure 5.23 shows the maximum amount of memory needed for the message buffers. This increases with CPI as expected; the process state can be saved at the CPI but all incoming messages must be stored in order to be able to regenerate the unstored states on rollback. Also, as there is necessarily more processing to do during rollback the build-up of messages during such times must be greater.

The total memory usage is shown in figure 5.24. This shows that, as the CPI is increased, the maximum memory usage initially falls quickly to a minimum in the case of aggressive cancellation, at an interval of eight and above. For lazy cancellation, the usage continues to falls above this point at a lower rate. This would suggest, particularly in the case of lazy cancellation, that it is better to select a CPI that is too large rather than one which is too small.

Figures 5.25 and 5.26 show the maximum total memory usage against run-time and speed-up respectively for the range of CPI. The run-times used in this case are for the simulations without the memory tracing enabled. They clearly show that there is a space/time optimal CPI of two. The shape of the curves also suggest that a CPI of three would most probably be worse than "optimal" for aggressive cancellation and possibly better for lazy cancellation. As in the case of the total memory usage curve (figure 5.24) they would suggest that it is better to select a CPI that is a little too large. Memory usage is reduced but speed-up too is adversely affected. These figures compare well with those found by Preiss et. al. [143] for the case where checkpointing is not artificially penalised.

5.4.4 Discussion of Results for the Tandem Queueing Network

These experiments were performed using the tandem queueing network model. The wide range of useful options available within YADDES was again explored in the experiments listed below; though memory management was not examined in this instance. In addition to comparing the performance of the various synchronization methods within YADDES for this application, comparisons are also drawn in the following sections with previous experiments on the closed stochastic queueing network.

- Sequential event-list (EL) — on 1 processor.
- Multiple distributed event-list (ML) — on 1, 2, 4 and 8 processors.
- Chandy-Misra (CM) — on 1, 2, 4 and 8 processors, with and without NULL-message cancellation (NMC).
- Virtual Time (VT) — on 1, 2, 4 and 8 processors, using aggressive and lazy cancellation, using checkpoint intervals (CPI) of 1, 2, 4, 8 and 16.

Multiple Event-list (ML) Synchronization

The ML speed-up results, shown in figure 5.27 are unimpressive the highest being 1.39 on eight processors. However, these are an improvement over previous results for the closed stochastic queueing network and also come close to speed-up results recorded by Preiss [175] on large closed stochastic queueing networks.

Conservative Chandy-Misra (CM) Synchronization

The CM speed-up results, also shown in figure 5.27, are very impressive. The most noticeable feature is that NULL-message cancellation has a very marked affect on the speed-up for all n. Indeed, the speed-up figures with NULL-message cancellation at the highest load are the highest recorded of any of the experiments for the number of processors (1.87 on two processors, 3.01 on four processors, 5.65 on eight processors), see figure 5.33. Again, these results are an improvement on the previous experiments with the closed stochastic queuing network. It has been widely reported in parallel simulation literature (see Fujimoto [24]) that acyclic queuing networks are well suited to conservative synchronization approaches, particularly Chandy-Misra. This is because cycles of processes tend to slow the propagation of lookahead increments.

Optimistic Virtual Time (VT) Synchronization

The VT speed-up results are shown in several graphs as follows. Figures 5.28 and 5.29 show the speed-up against the number of processors and CPI for lazy and aggressive cancellation strategies. Figures 5.30, 5.31 and 5.32 show speed-up against CPI on two, four and eight processors respectively. The process scheduling strategy used was minimum virtual time scheduling as this has been shown to give the best performance of those available within YADDES — round-robin, minimum message time and minimum virtual time. For minimum virtual time scheduling, where there is more than one process per processor, the process with the lowest virtual time will be be scheduled next for execution.

The speed-up figures obtained are consistently good though not as good as CM synchronization. Only lazy cancellation with a CPI of four (the best VT result) out-performs CM without NULL-message cancellation. In the previous experiments, on closed stochastic queueing networks, there seemed little to choose between the lazy and aggressive cancellation strategies. However, here lazy cancellation easily out-performs aggressive cancellation at all settings of CPI. Again, this is most likely due to the acyclic nature of the model leading to rollbacks which seldom need the use of anti-messages to undo erroneous computations. An optimum CPI for maximum speed-up is also discernible from figures 5.30, 5.31 and 5.32 for both cancellation strategies: for lazy cancellation the optimum CPI is between four and eight (nearer eight) and for aggressive cancellation the optimum is around four. Again, this indicates that rollbacks are more expensive under aggressive cancellation.

Comparison of Speed-up Results

The different synchronization algorithms are compared in figure 5.33. Only the best VT results are shown for lazy (CPI = 8) and aggressive cancellation (CPI = 4). Overall ML synchronization is the worst performer giving little or no speed-up. This proves the intuitive conclusion that any algorithm, such as ML synchronization, which tries to synchronize a parallel simulation using global control will not result in significant speed-up. CM synchronization exhibited by far the best speed-up results. This shows that where there is good lookahead properties in a model coupled with a significant message load CM synchronization can produce excellent speed-up results.

The VT synchronization speed-up results shown here are consistently good with lazy cancellation consistently out-performing aggressive cancellation We have also observed that an optimum CPI improves speed-up as well as giving benefits in reduced memory usage.

An encouraging feature of figure 5.33 is that the better speed-up curves show less signs of saturation as the number of processors were increased than in previous experiments. This would seem to indicate that if more processors were added further speed-up would be observed. This is not surprising given the amount of parallelism in the model and the simple pipeline process topology.

5.5 Conclusions

This chapter has described the results of a programme of experiments to evaluate the performance of various parallel discrete event simulation algorithms. This has been possible using YADDES with it's common simulation model description language and the four parallel discrete event simulation synchronization algorithms supported. The two models used in the experiments were of a closed stochastic queueing network (a hypercube of queues) and a tandem queueing network of sixteen queues.

The speed-up results for distributed multiple event-list (ML) synchronization were poor as expected for both models. Any synchronization method which tries to control a parallel simulations progress globally from a single process is unlikely to yield significant speed-up.

The speed-up results for conservative Chandy-Misra (CM) synchronization show the importance of lookahead and, more particularly, the impact of NULL-message cancellation at higher model loads. Indeed, CM synchronization with NULL-message cancellation yielded the best speed-up result for both models. This shows generally that CM gives good speed-

up for models with good lookahead and high customer (and hence message) loads. However, at lower loads the speed-up is considerably reduced by NULL-message processing overheads. The speed-up results for the tandem queueing model were higher than those recorded for the closed queueing network. This result has been noted previously by various researchers (eg. Fujimoto [24]) and is due to the lack of cycles in the model. These tend to slow the propagation of lookahead increments and thus cause increased synchronization overheads.

The speed-up figures for optimistic virtual time (VT) synchronization were more modest than for CM, but were also more consistent, being relatively insensitive to the load. Optimistic synchronization was easily the best performer for lower loads only losing out to CM synchronization with NULL-message cancellation at the highest load.

The VT synchronization algorithm could use lazy or aggressive cancellation strategies. For the tandem queuing network model, lazy cancellation out-performed aggressive cancellation at all values of CPI. In the experiments on the closed queueing network model there seemed little to choose between the lazy and aggressive cancellation strategies except in the area of memory usage, where lazy cancellation used less, particularly at higher values of CPI. Again, this is most likely due to the acyclic nature of the model leading to rollbacks which seldom need the use of anti-messages to undo erroneous computations.

An optimum CPI for maximum speed-up is discernible for both models and cancellation strategies. For the tandem queueing network model, the optimum CPI is greater for lazy than for aggressive cancellation. Again, this seems to indicate that rollbacks are more expensive under aggressive cancellation. A larger CPI will also result in lower memory usage for the simulation though these were not measured.

The next chapter discusses the parallel simulation of asynchronous transfer mode (ATM) networks. The results of the experiments discussed here indicate that the conservative Chandy-Misra synchronization approach would be the best to use in simulating such networks using distributed memory multiprocessor computers.

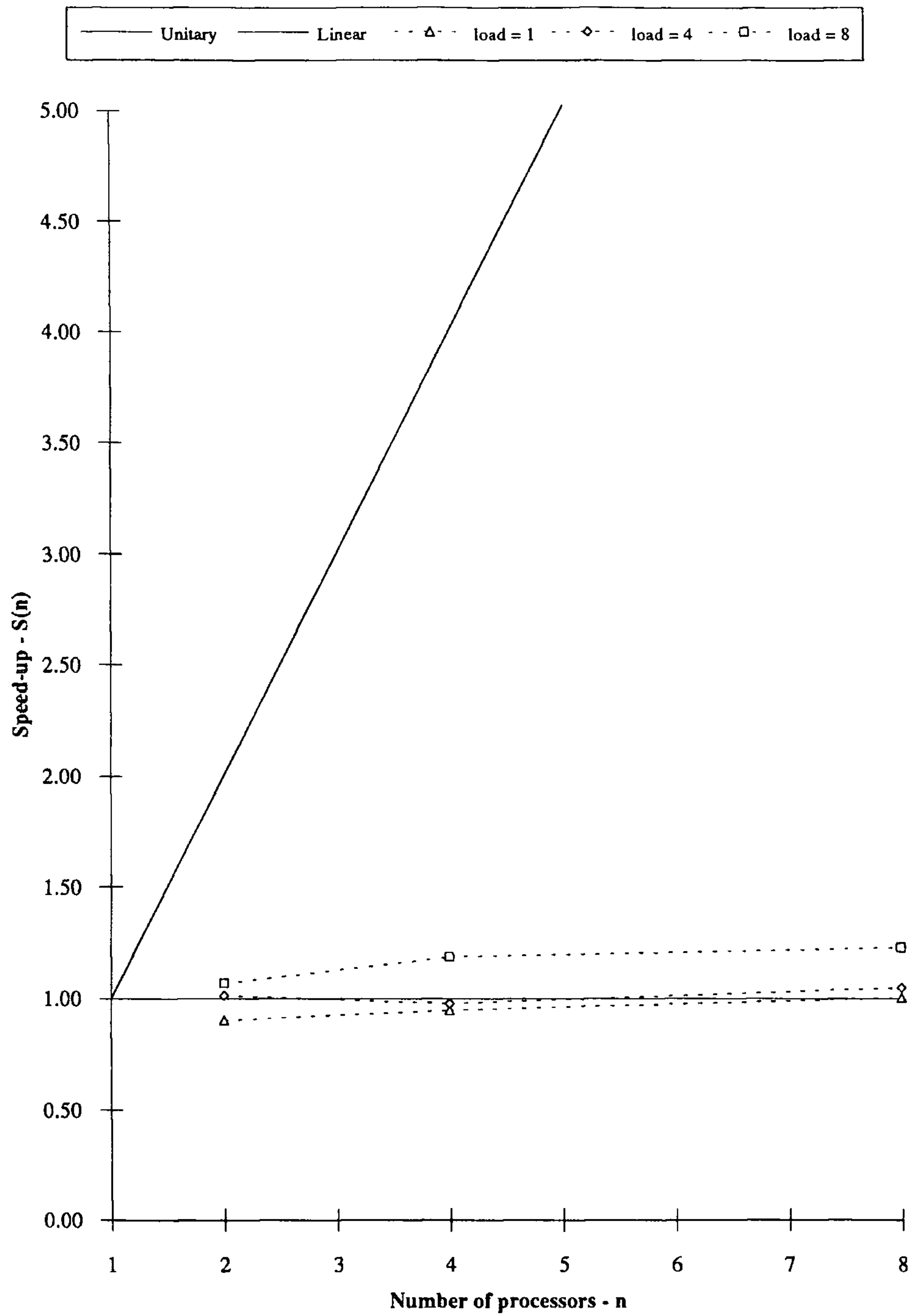


Figure 5.3: Speed-up using multiple distributed event-list (ML) synchronization for the hypercube of queues.

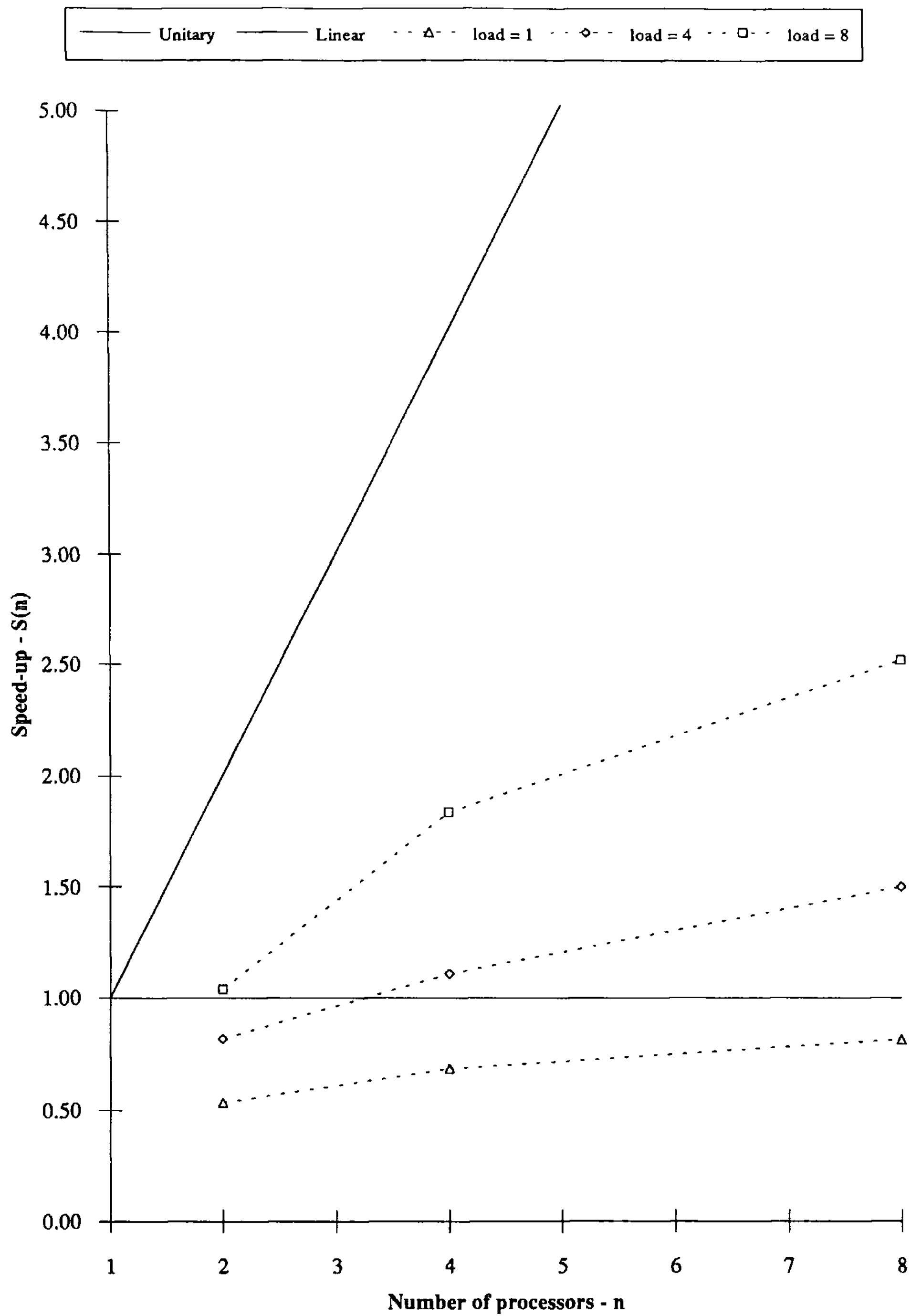


Figure 5.4: Speed-up using conservative Chandy-Misra synchronization (CM) without NULL-message cancellation for the hypercube of queues.

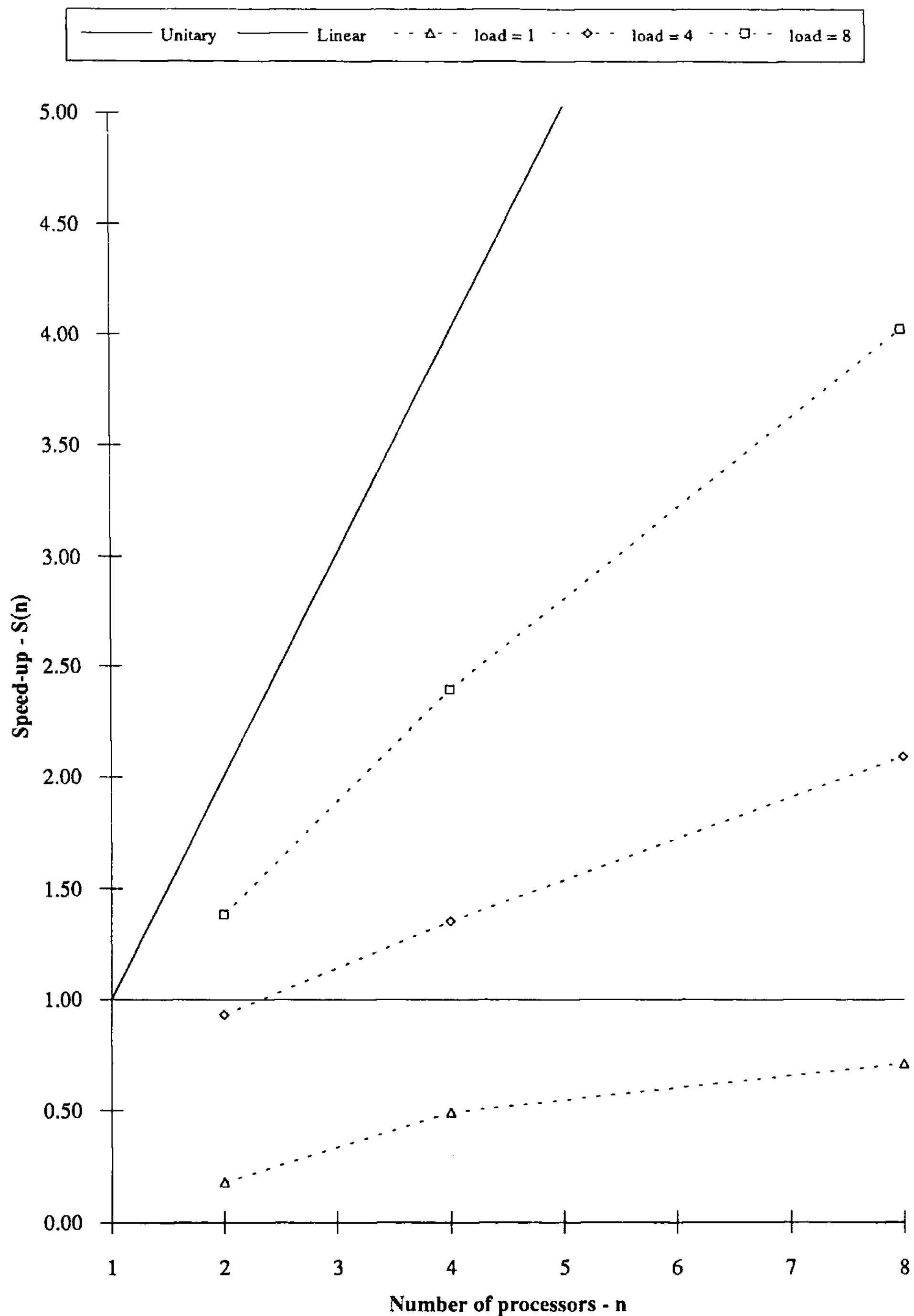


Figure 5.5: Speed-up using conservative Chandy-Misra synchronization (CM) with NULL-message cancellation (NMC) for the hypercube of queues.

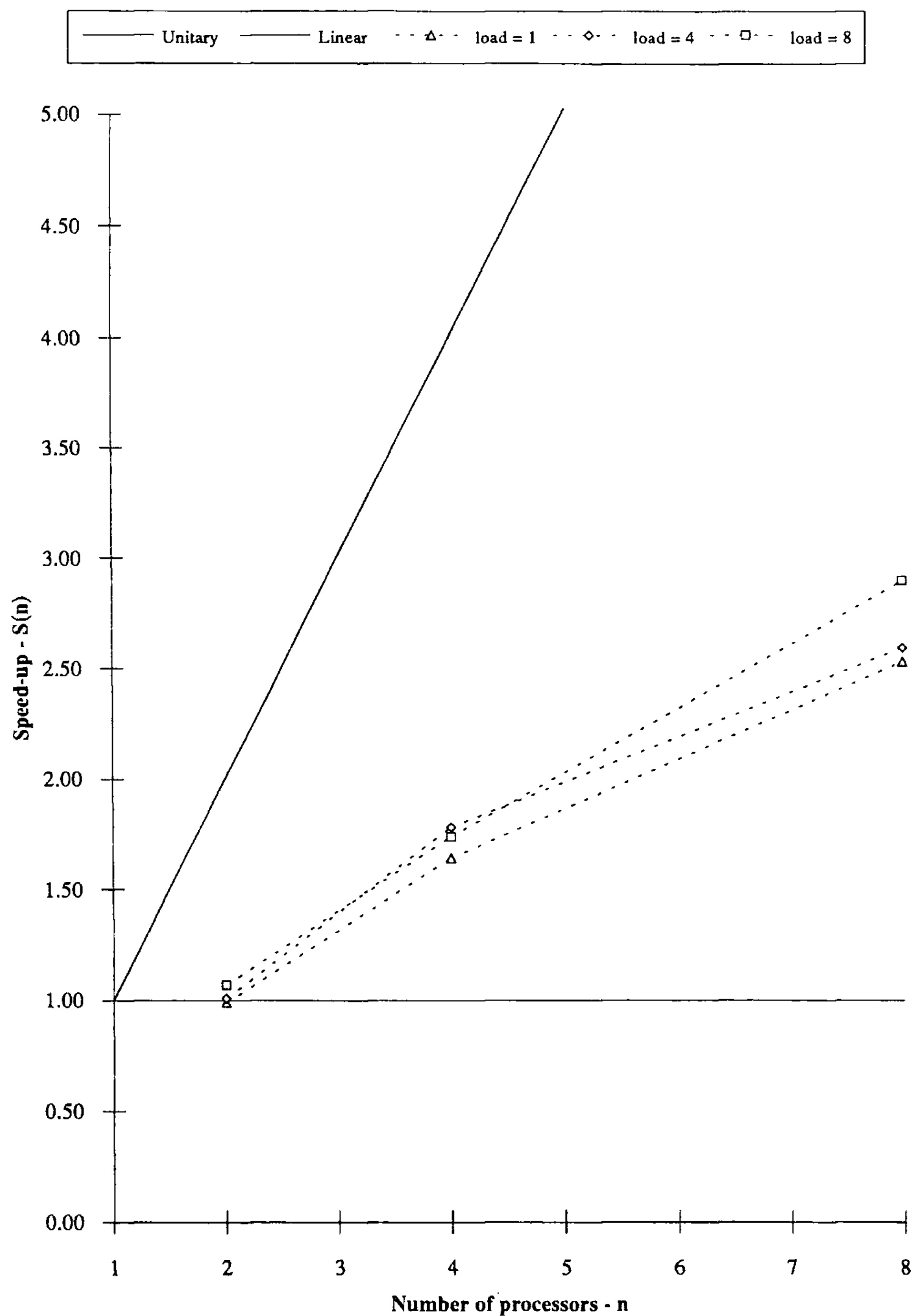


Figure 5.6: Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of one for the hypercube of queues.

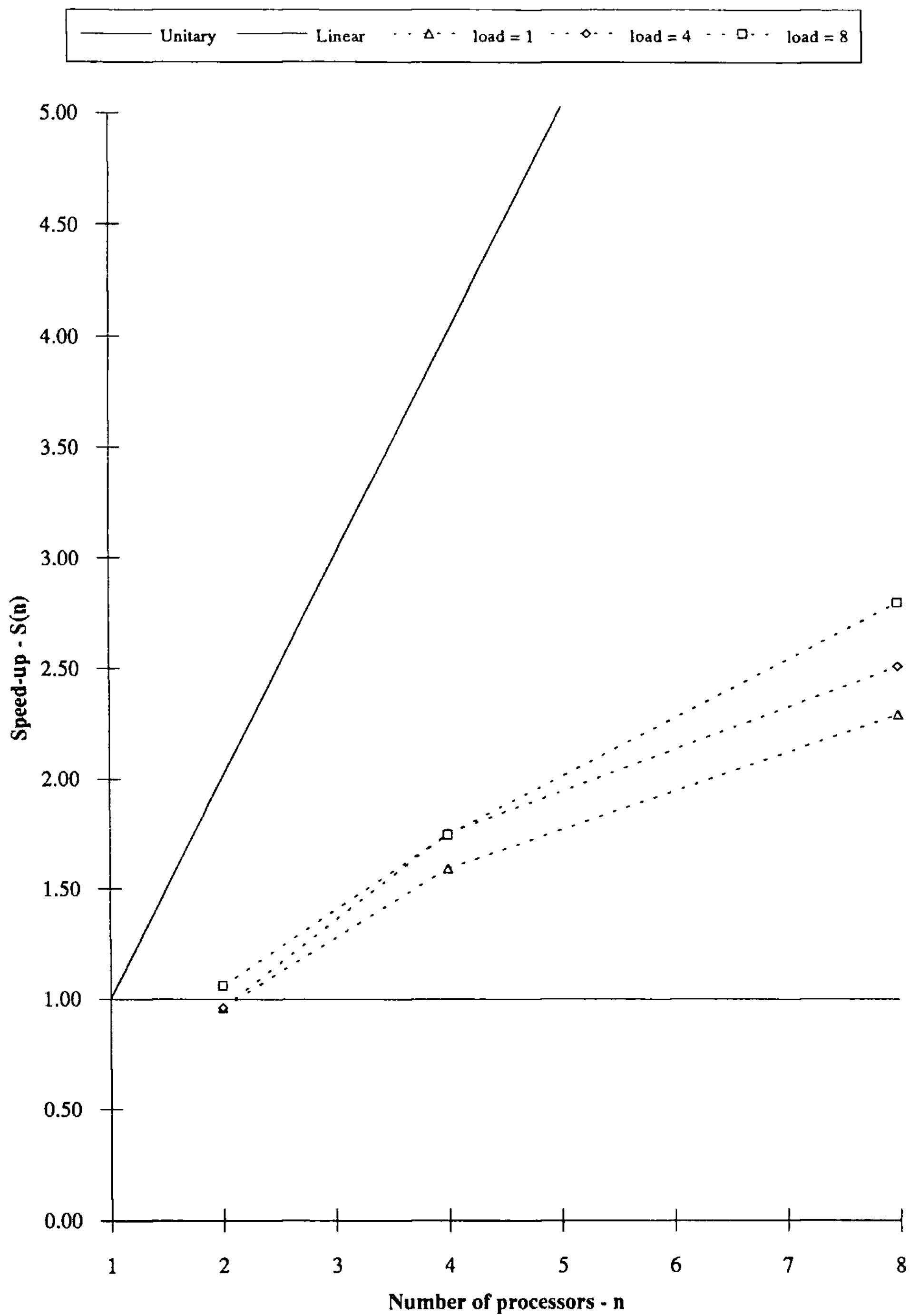


Figure 5.7: Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of one for the hypercube of queues.

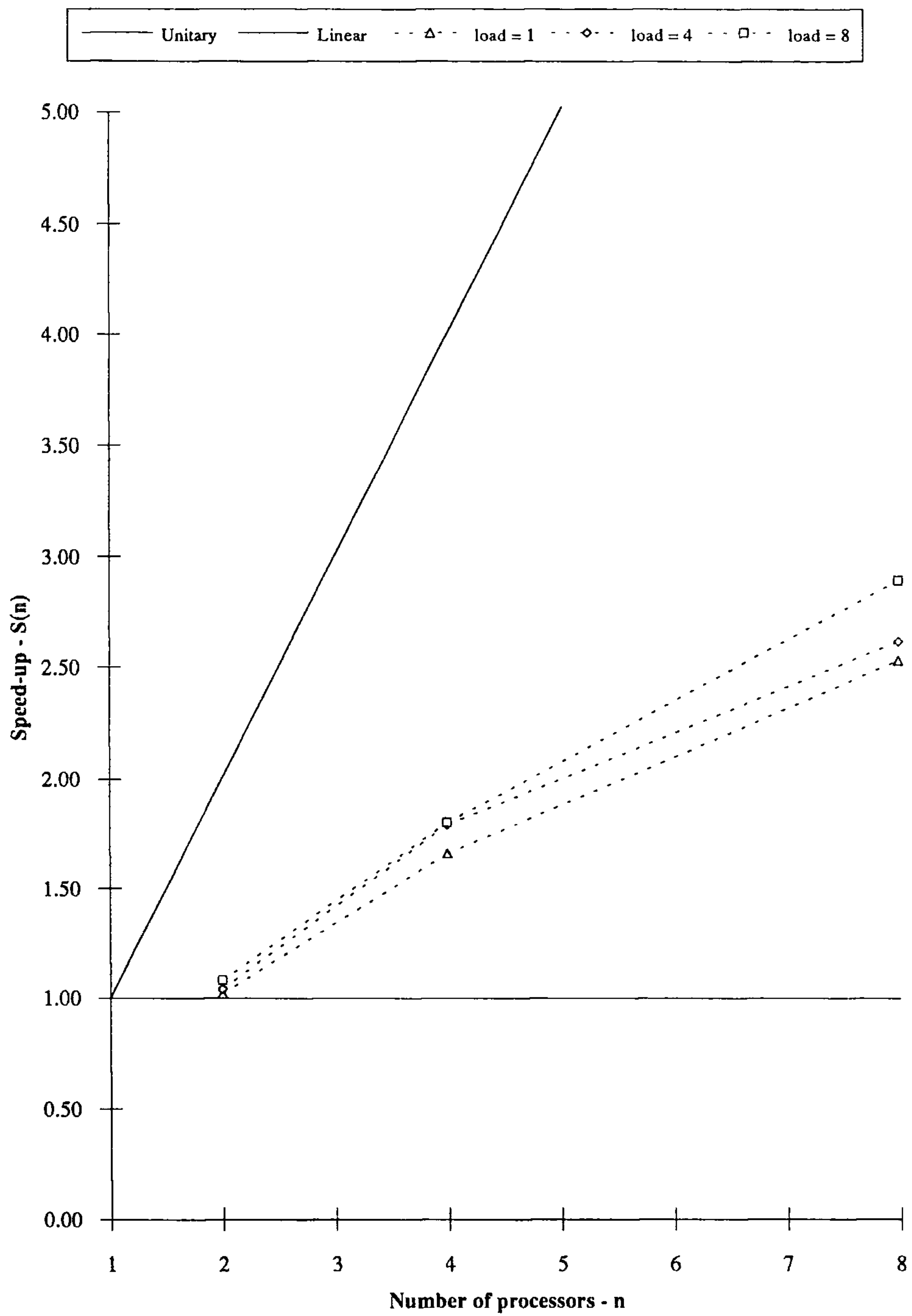


Figure 5.8: Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of two for the hypercube of queues.

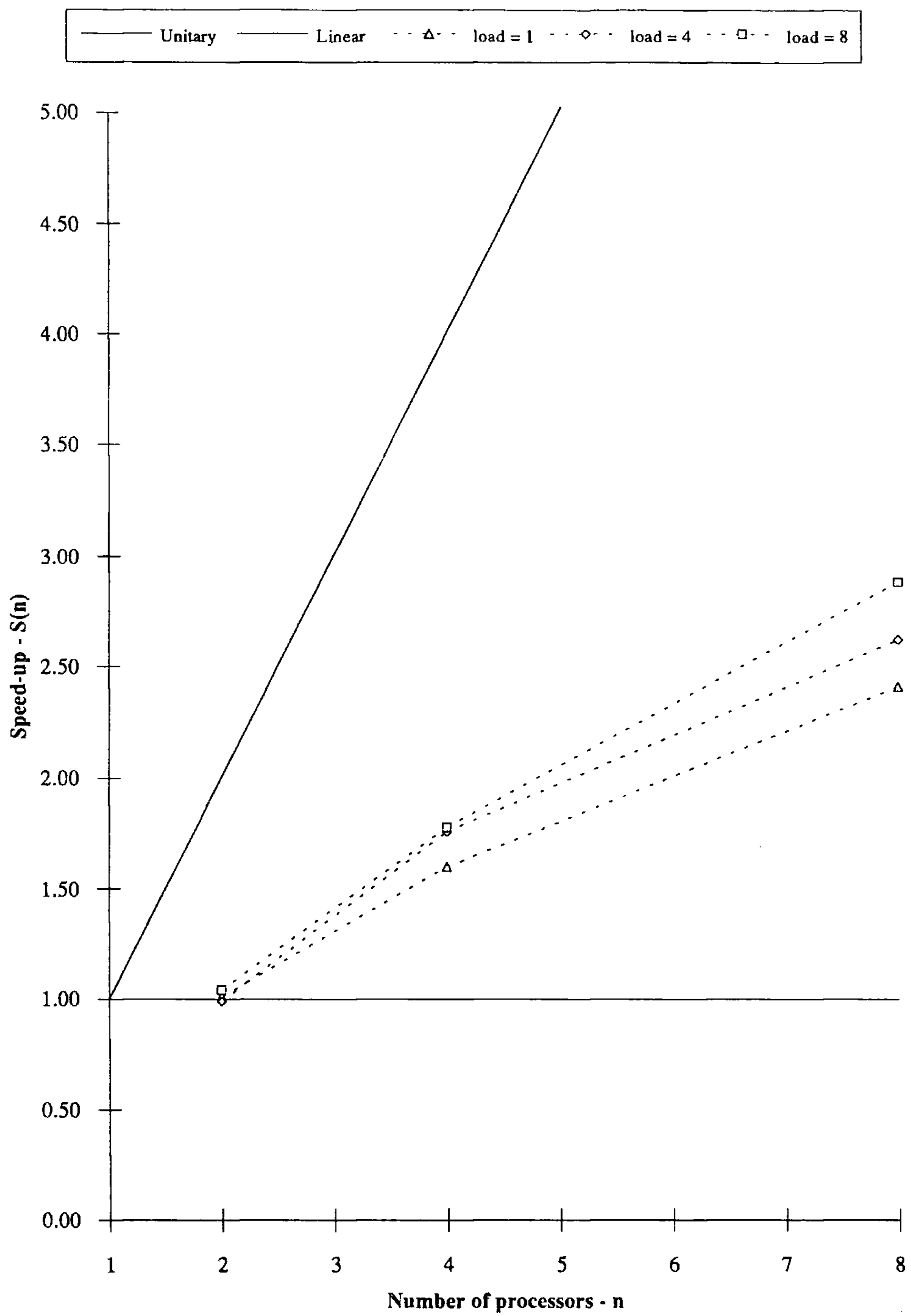


Figure 5.9: Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of two for the hypercube of queues.

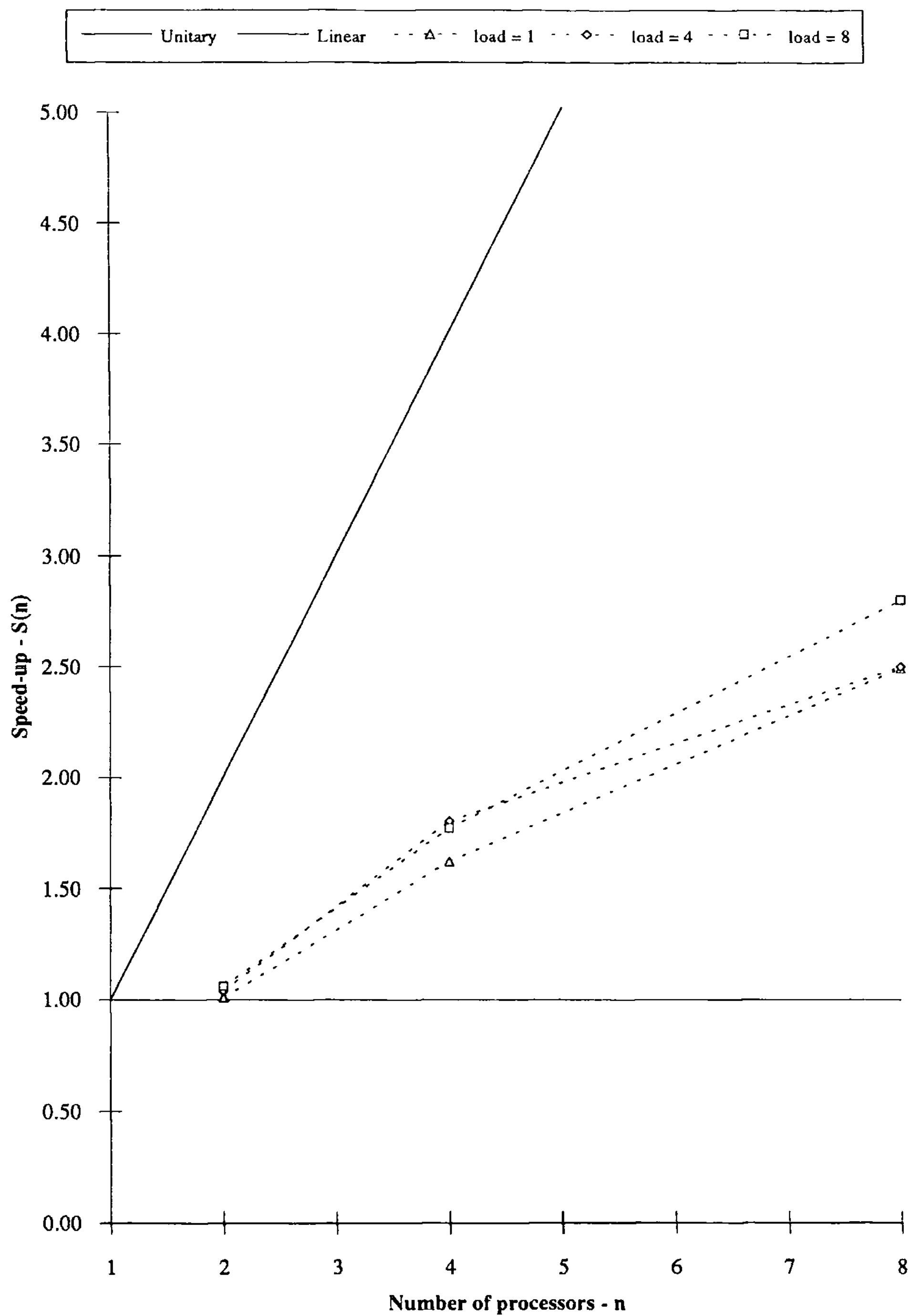


Figure 5.10: Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of four for the hypercube of queues.

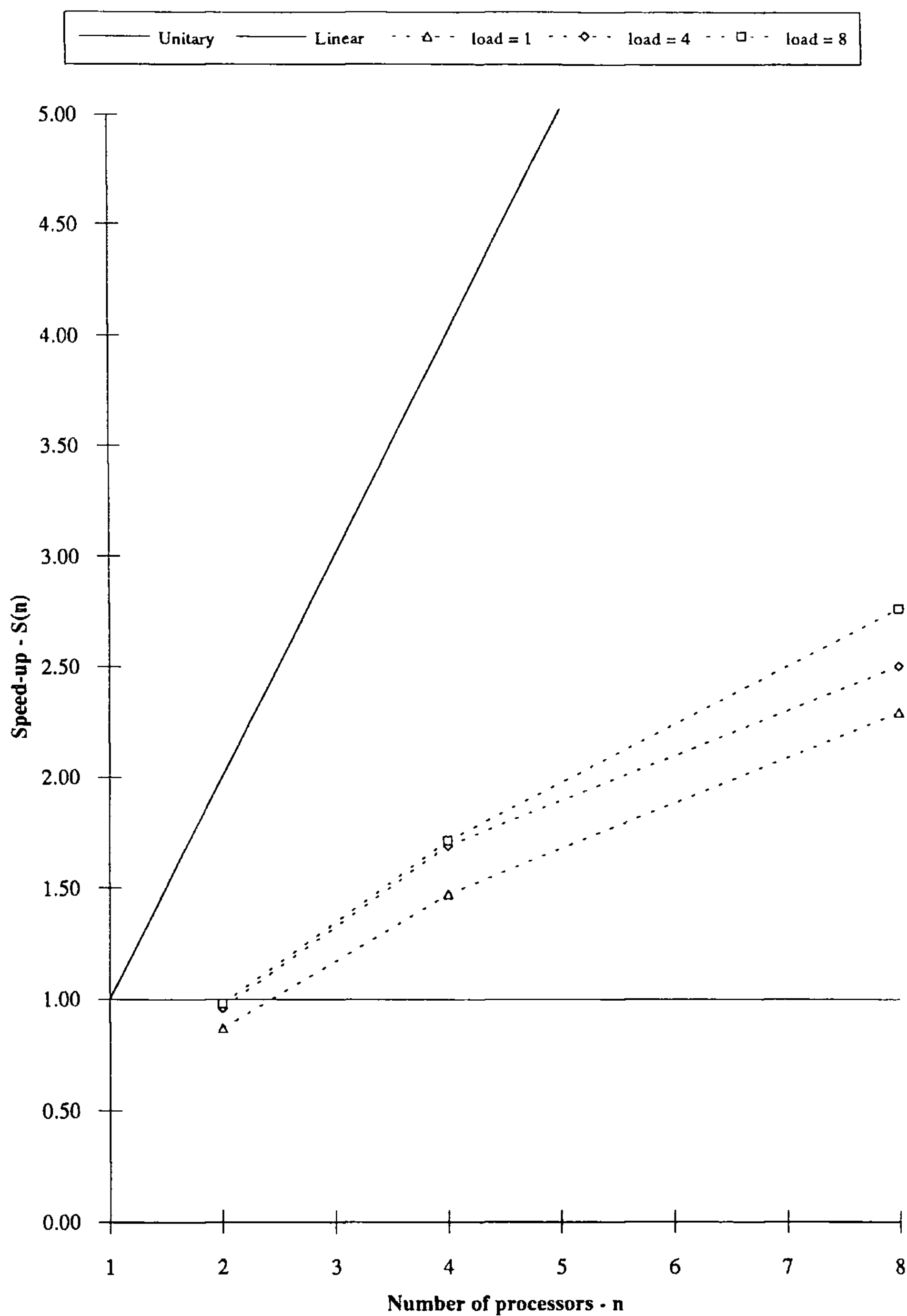


Figure 5.11: Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of four for the hypercube of queues.

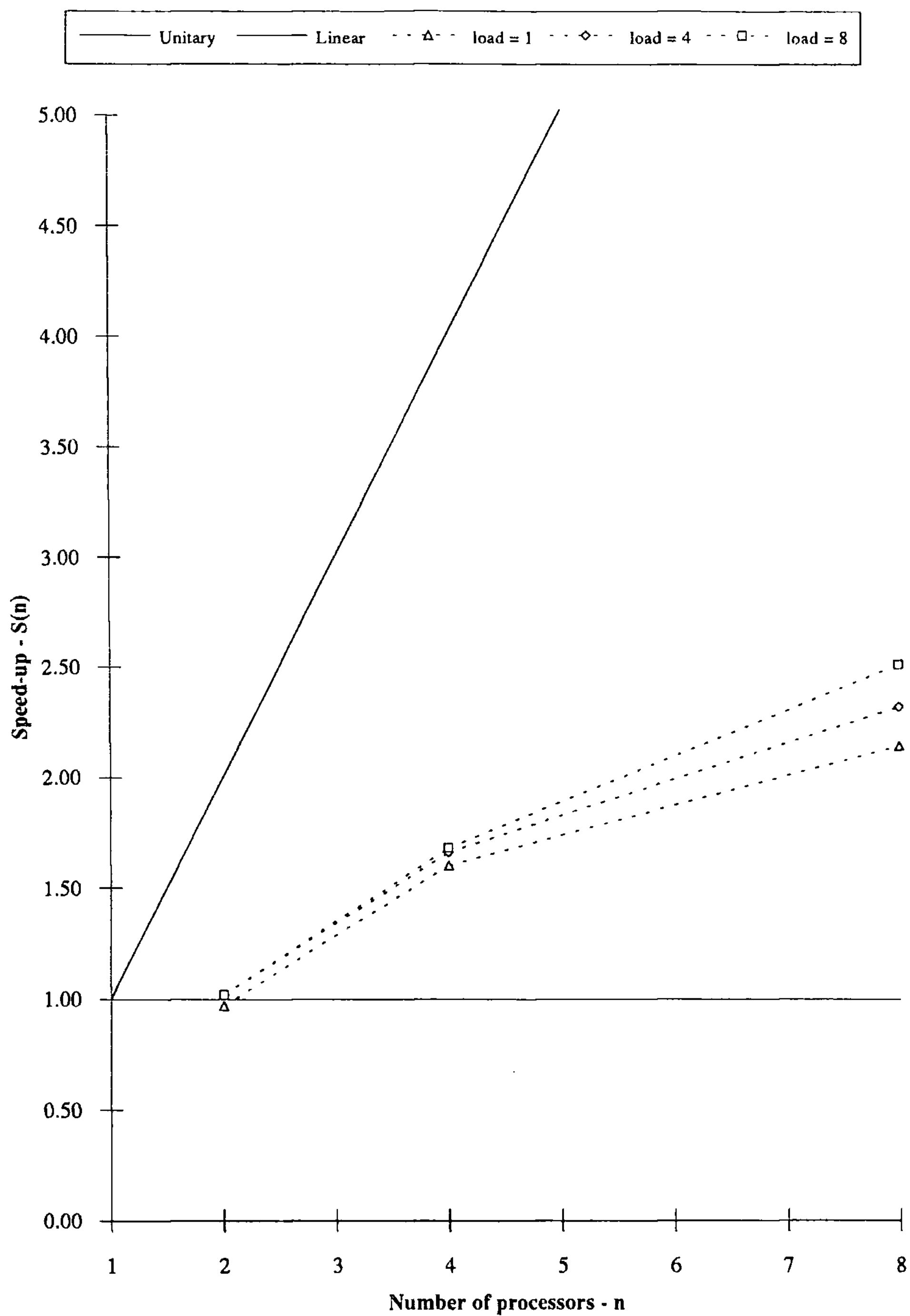


Figure 5.12: Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of eight for the hypercube of queues.

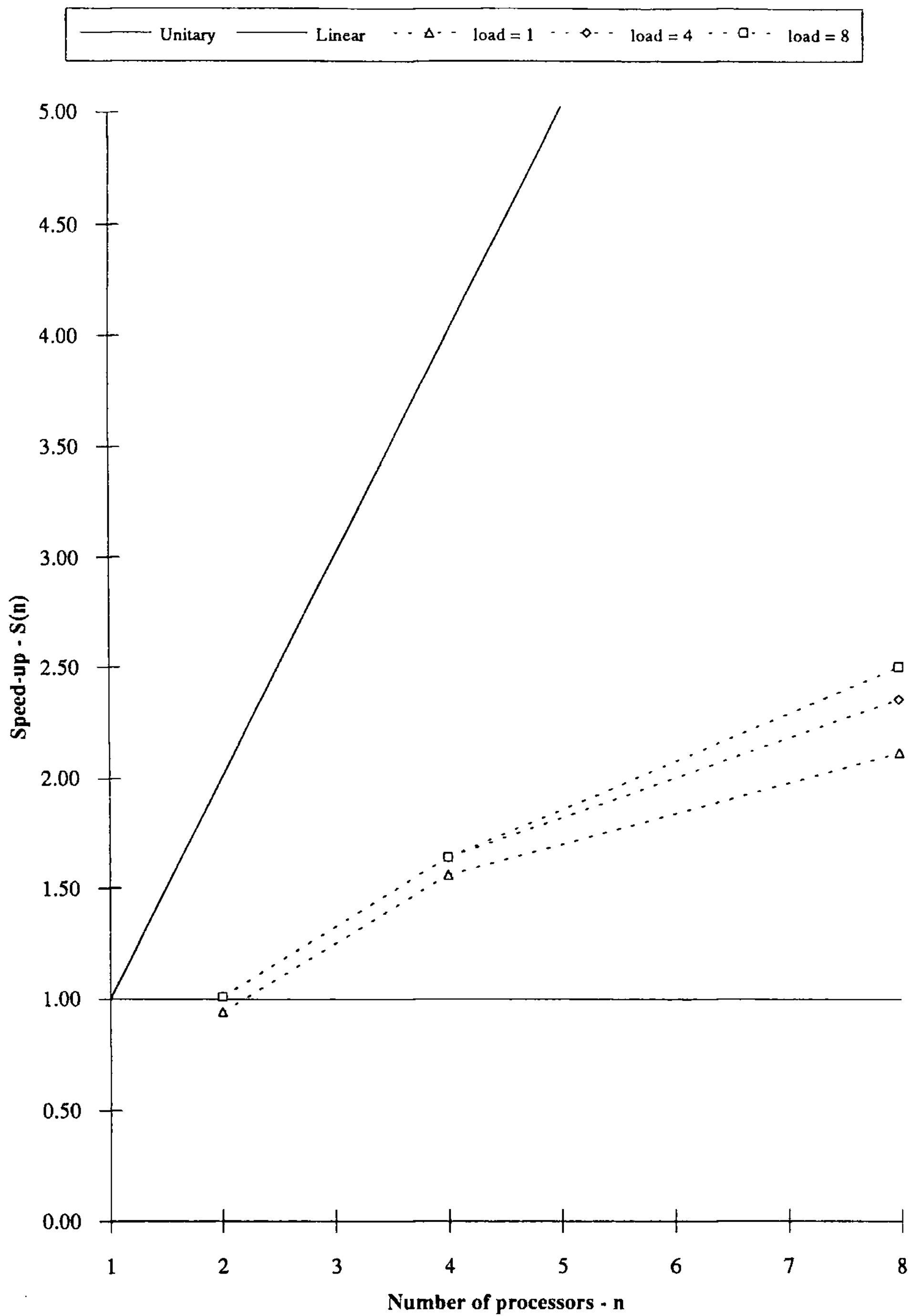


Figure 5.13: Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of eight for the hypercube of queues.

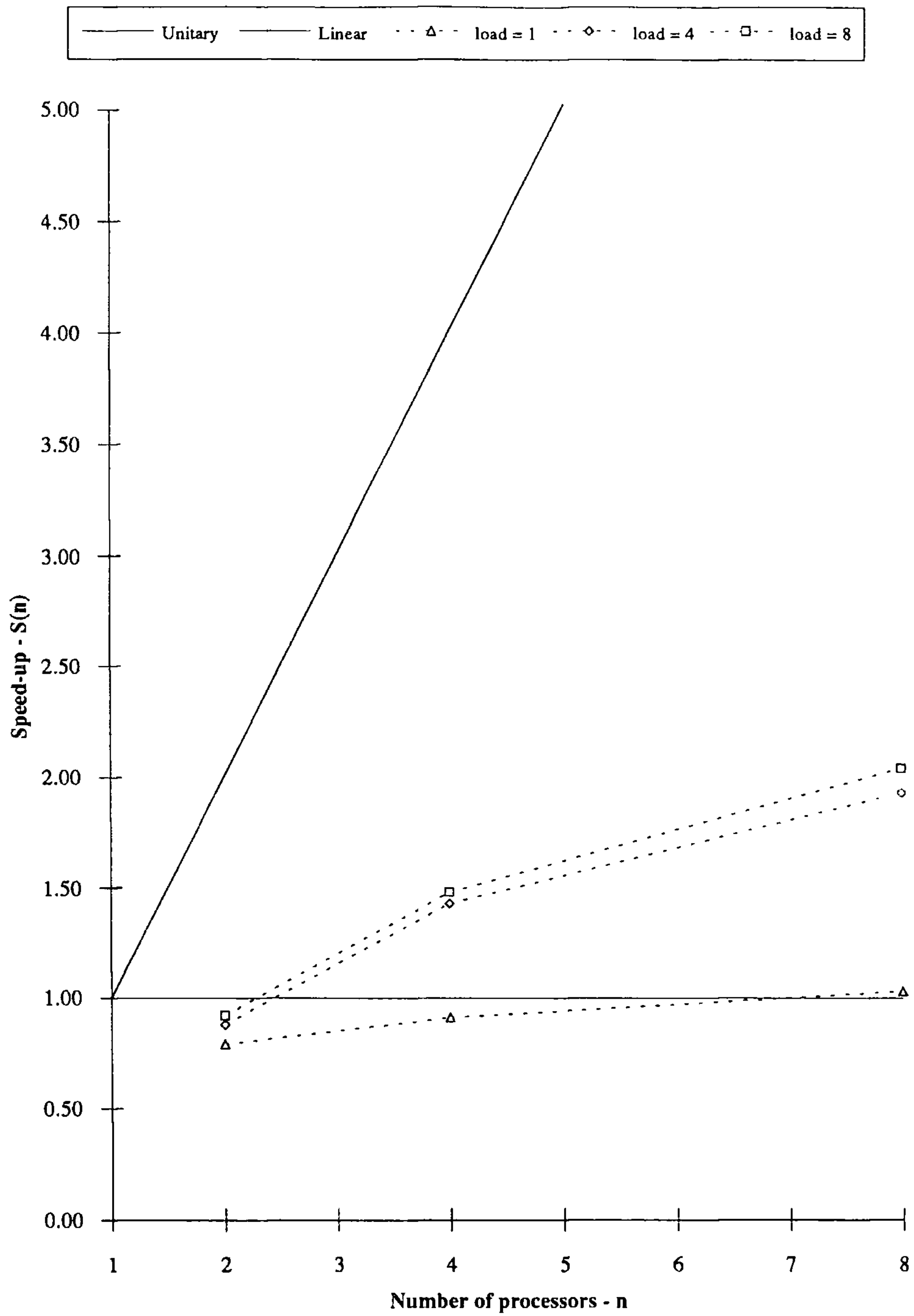


Figure 5.14: Speed-up using virtual time synchronization (VT) with lazy cancellation and a CPI of sixteen for the hypercube of queues.

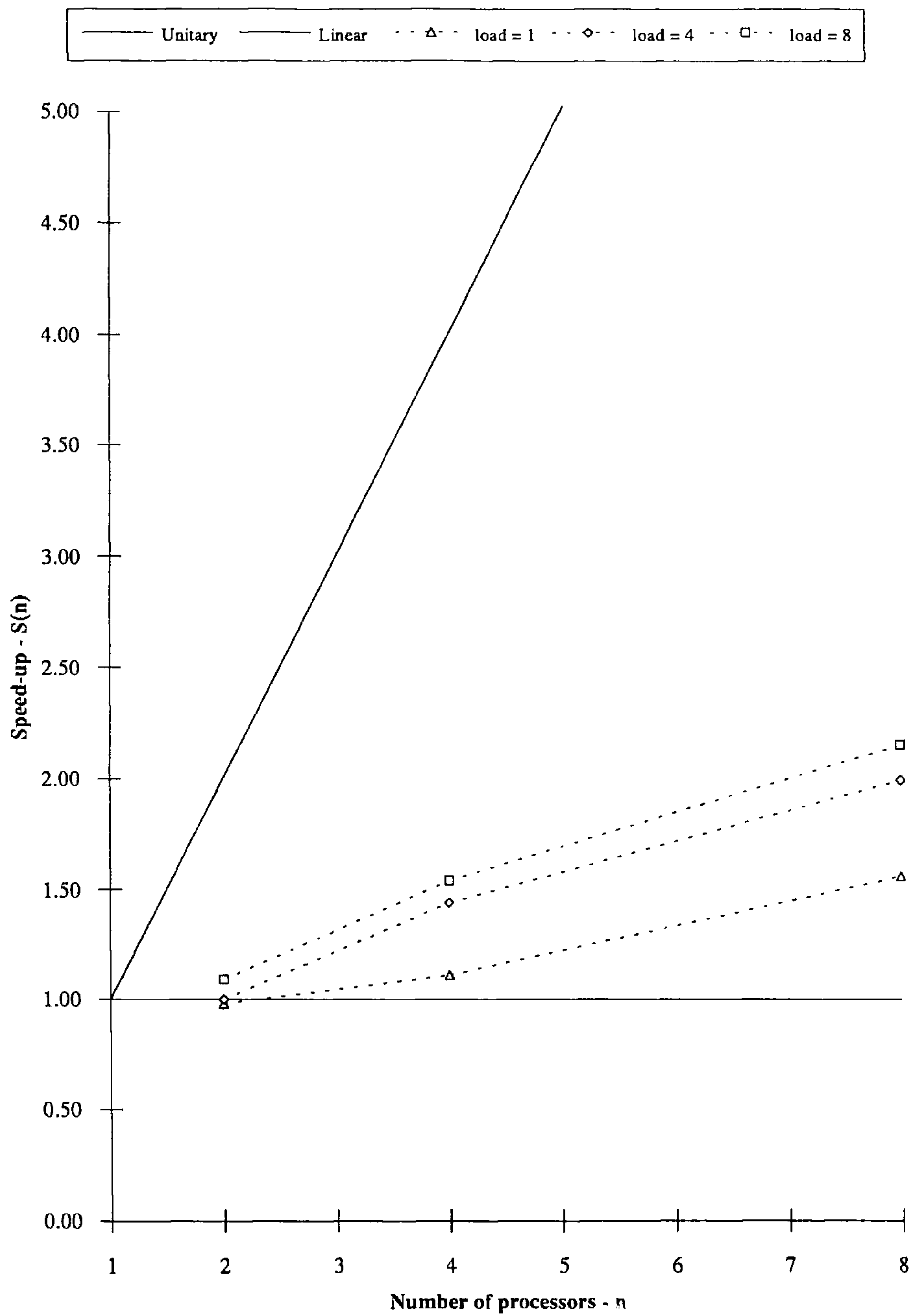


Figure 5.15: Speed-up using virtual time synchronization (VT) with aggressive cancellation and a CPI of sixteen for the hypercube of queues.

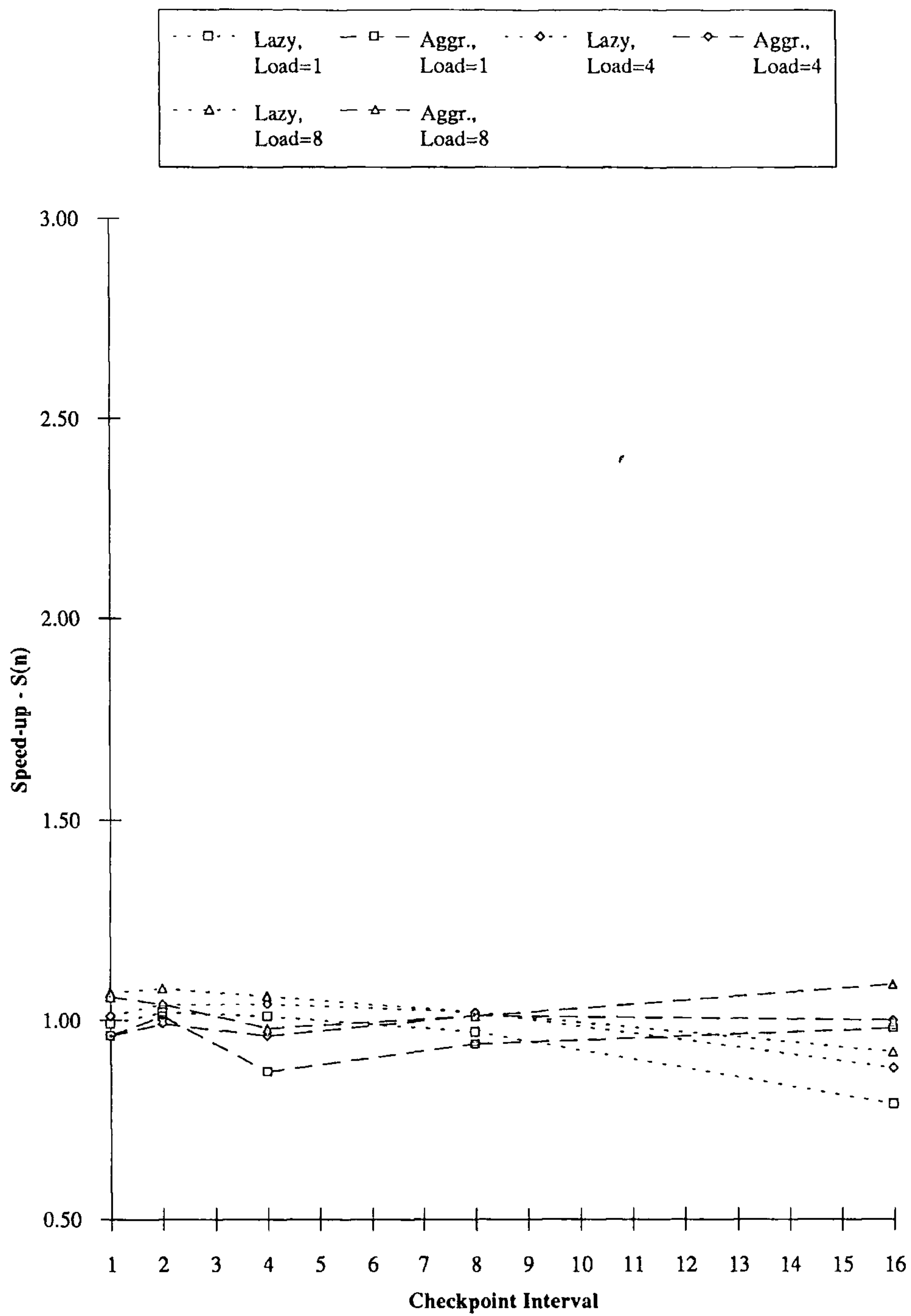


Figure 5.16: Speed-up against CPI for virtual time synchronization (VT) on two processors for the hypercube of queues.

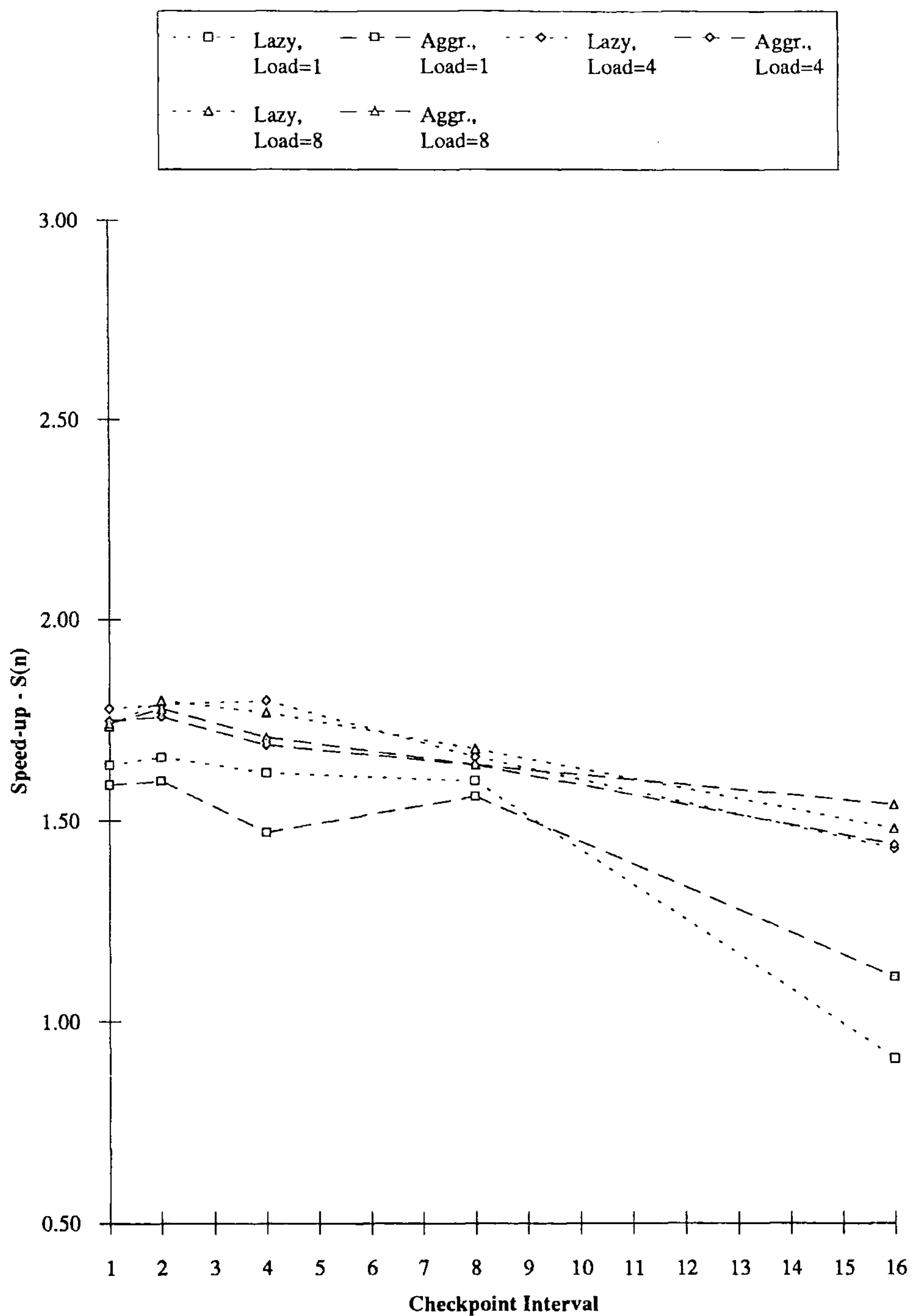


Figure 5.17: Speed-up against CPI for virtual time synchronization (VT) on four processors for the hypercube of queues.

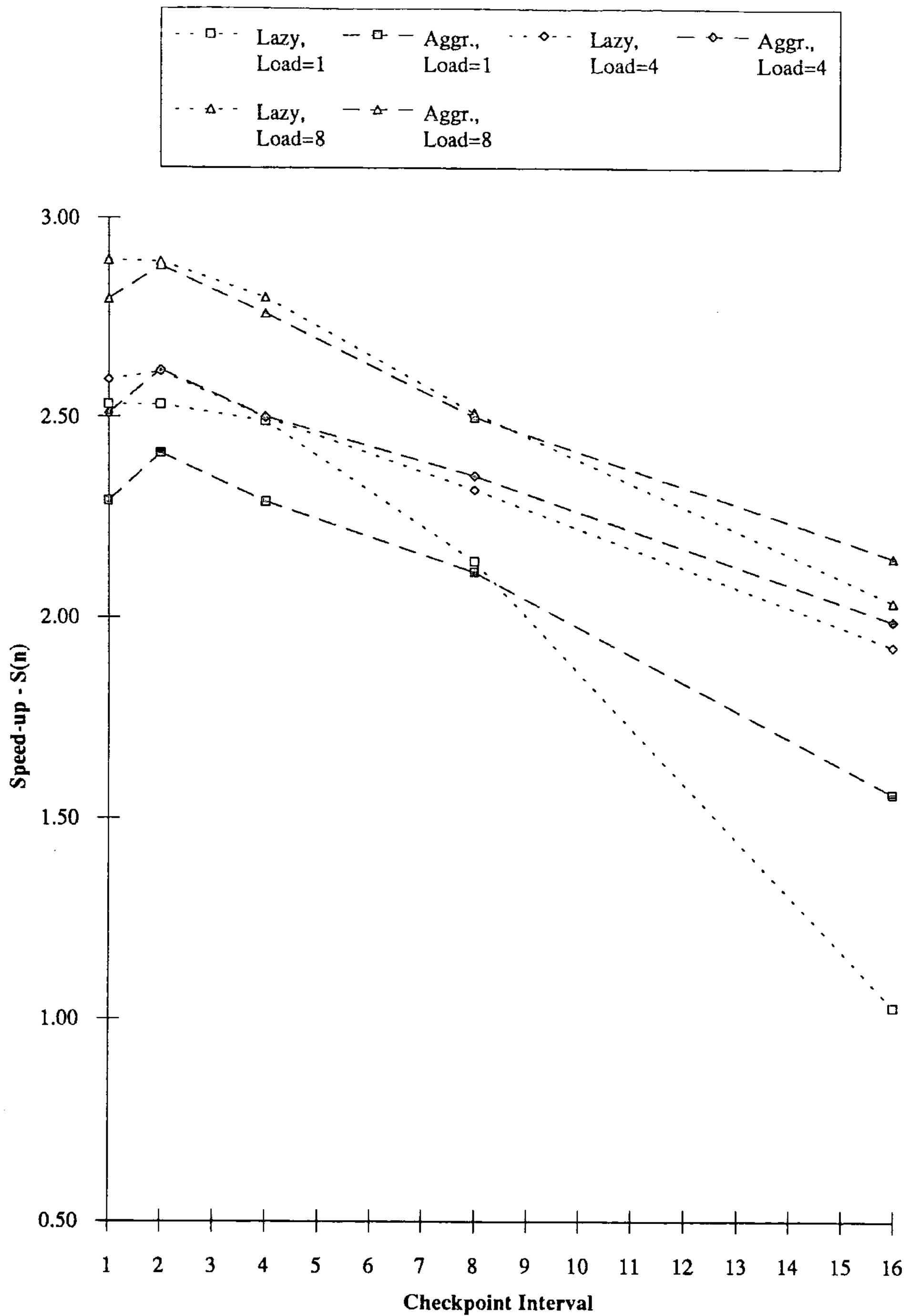


Figure 5.18: Speed-up against CPI for virtual time synchronization (VT) on eight processors for the hypercube of queues.

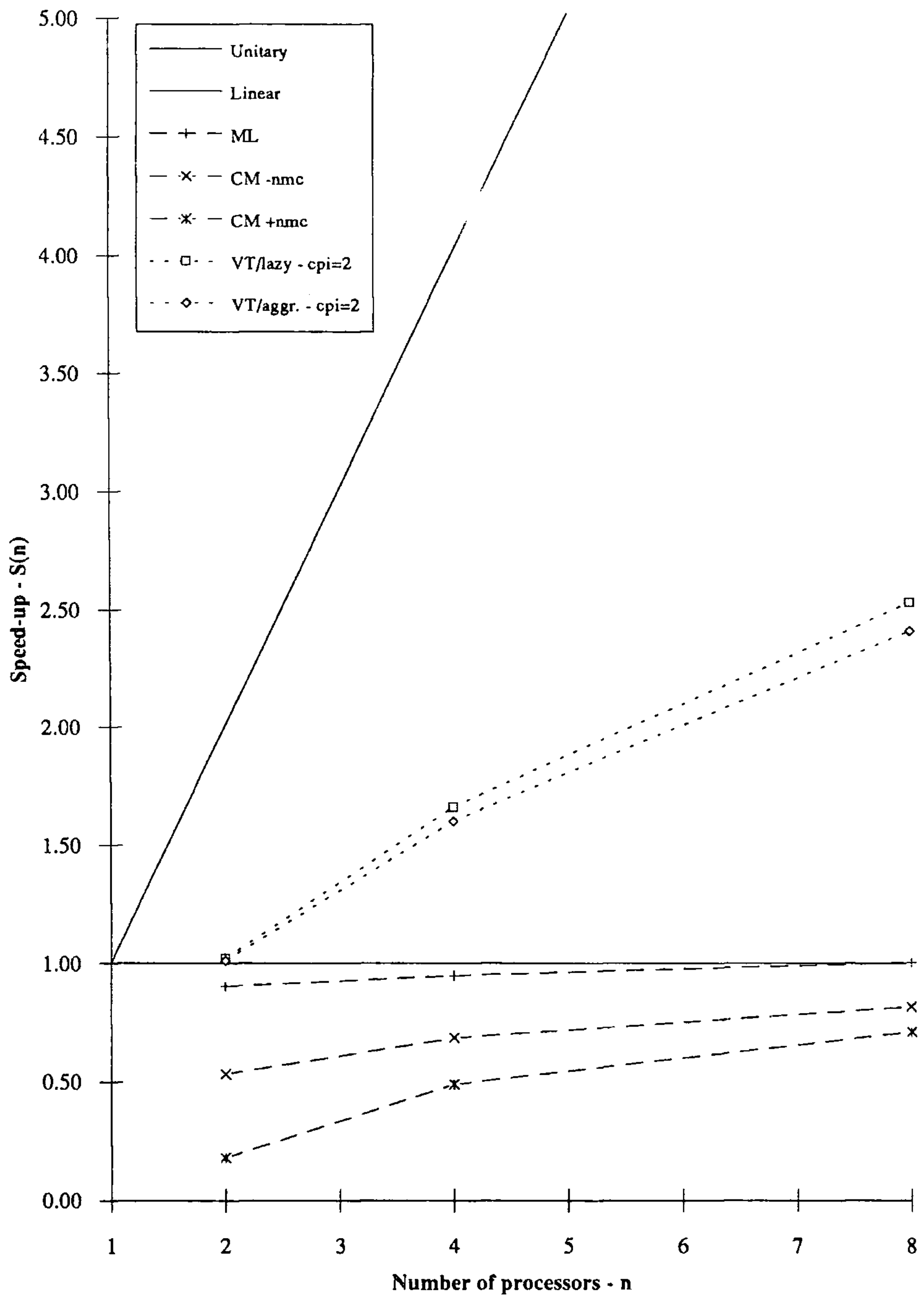


Figure 5.19: Comparison of synchronization mechanisms for a load of one customer per queue for the hypercube of queues.

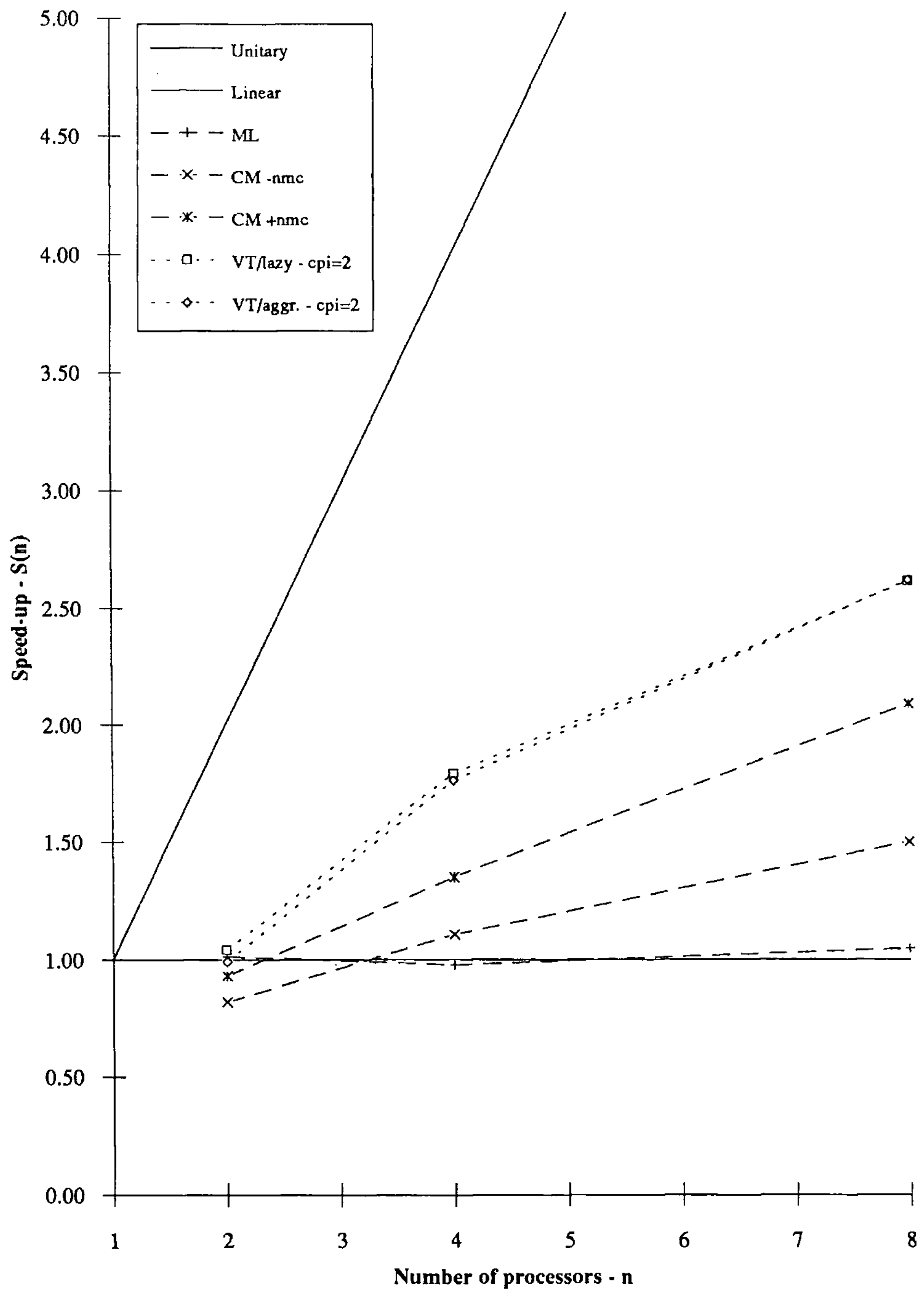


Figure 5.20: Comparison of synchronization mechanisms for a load of four customers per queue for the hypercube of queues.

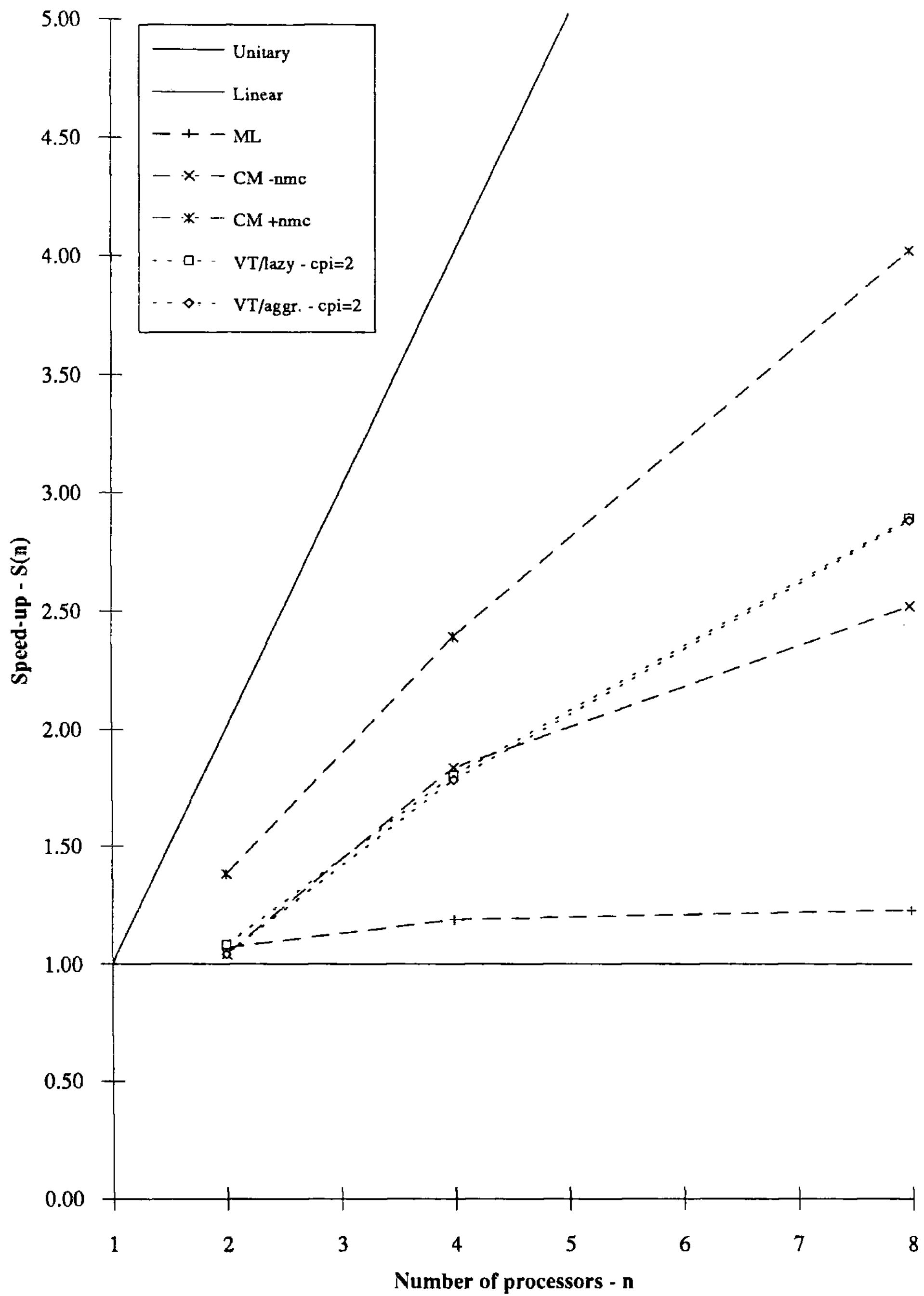


Figure 5.21: Comparison of synchronization mechanisms for a load of eight customers per queue for the hypercube of queues.

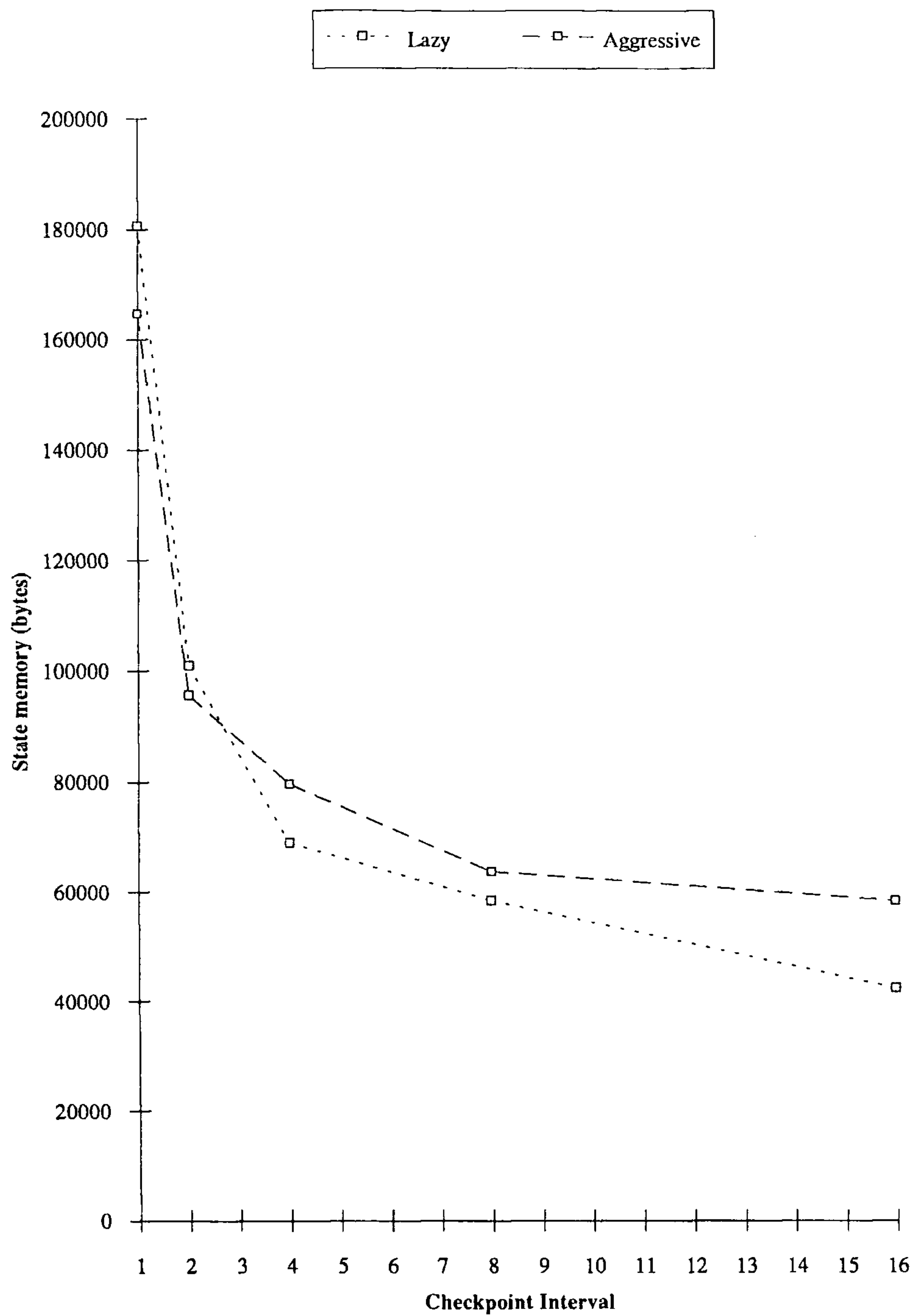


Figure 5.22: State memory usage for virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues.

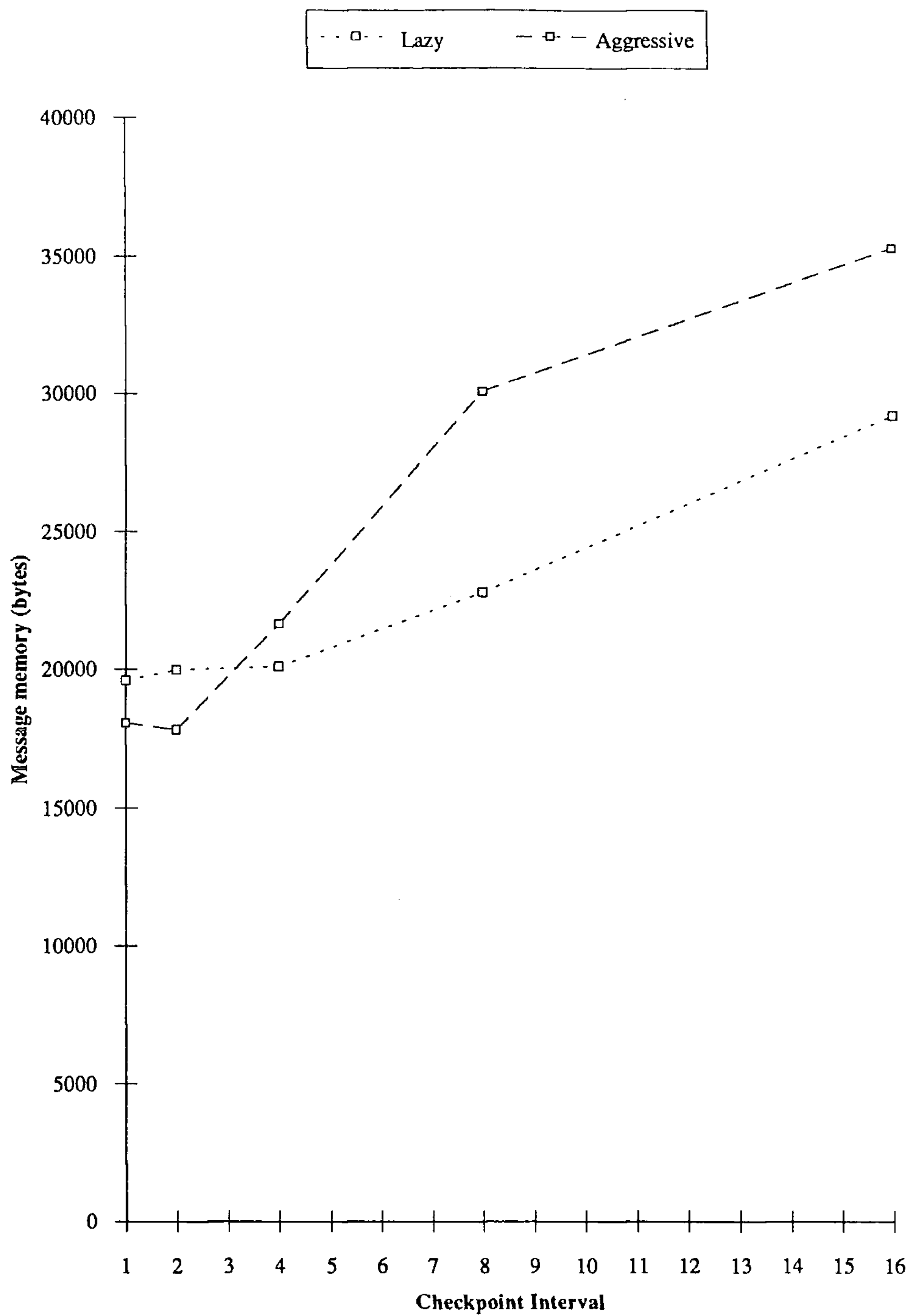


Figure 5.23: Message memory usage for virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues.

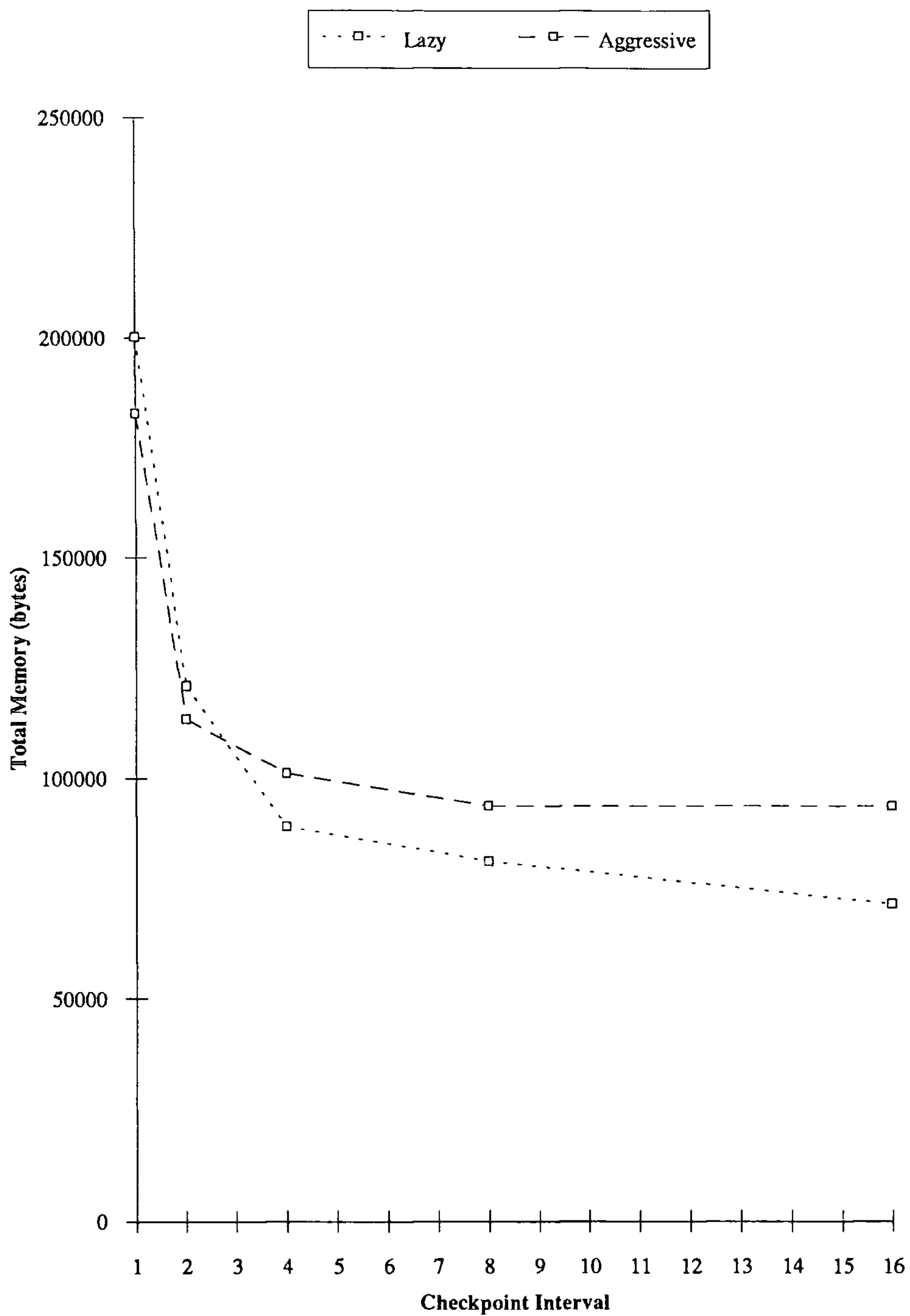


Figure 5.24: Total memory usage for virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues.

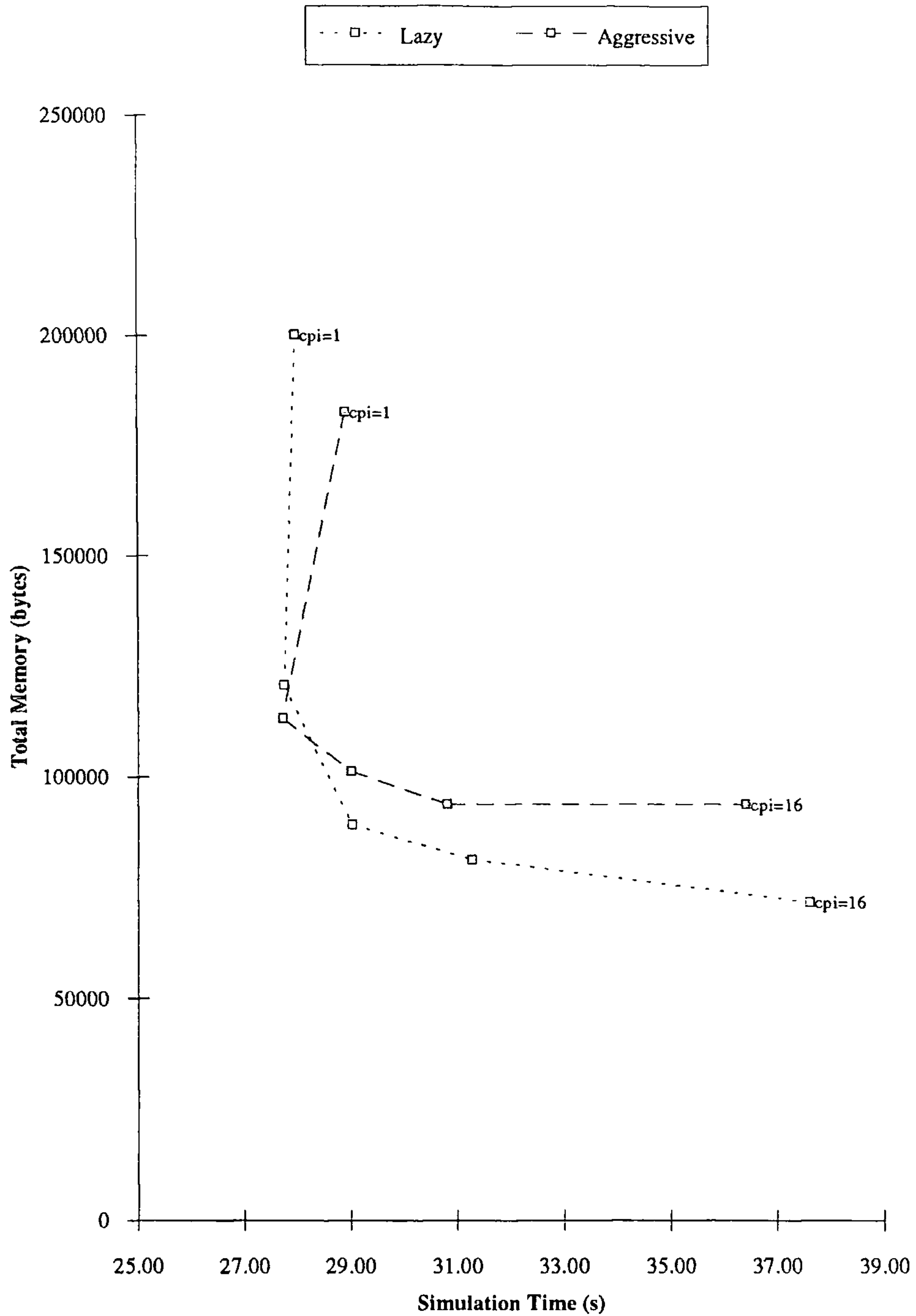


Figure 5.25: Total memory usage against simulation time for a CPI of 1, 2, 4, 8 and 16 using virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues.

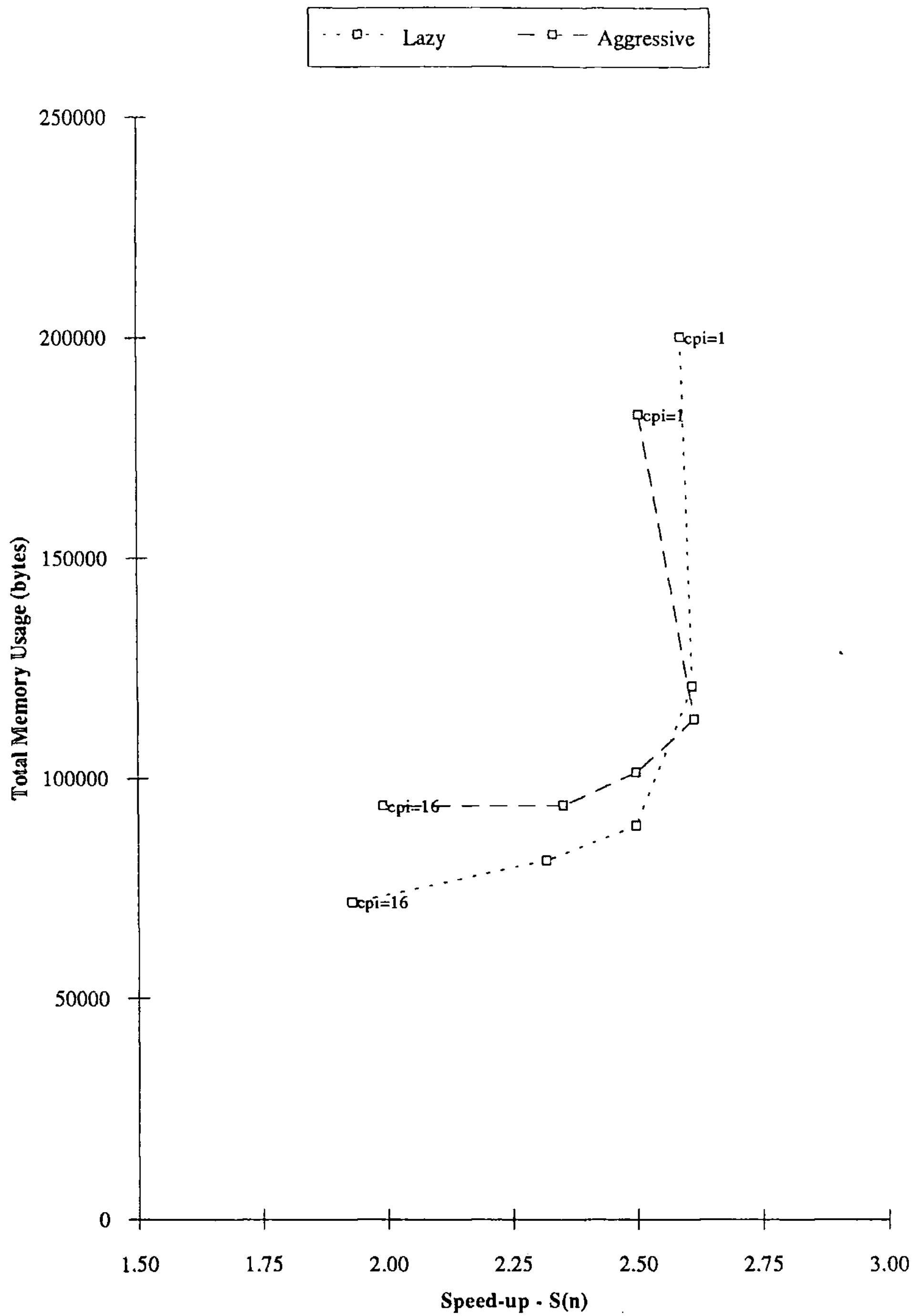


Figure 5.26: Total memory usage against speed-up for a CPI of 1, 2, 4, 8 and 16 using virtual time synchronization (VT) on eight processors at a load of four customers per queue for the hypercube of queues.

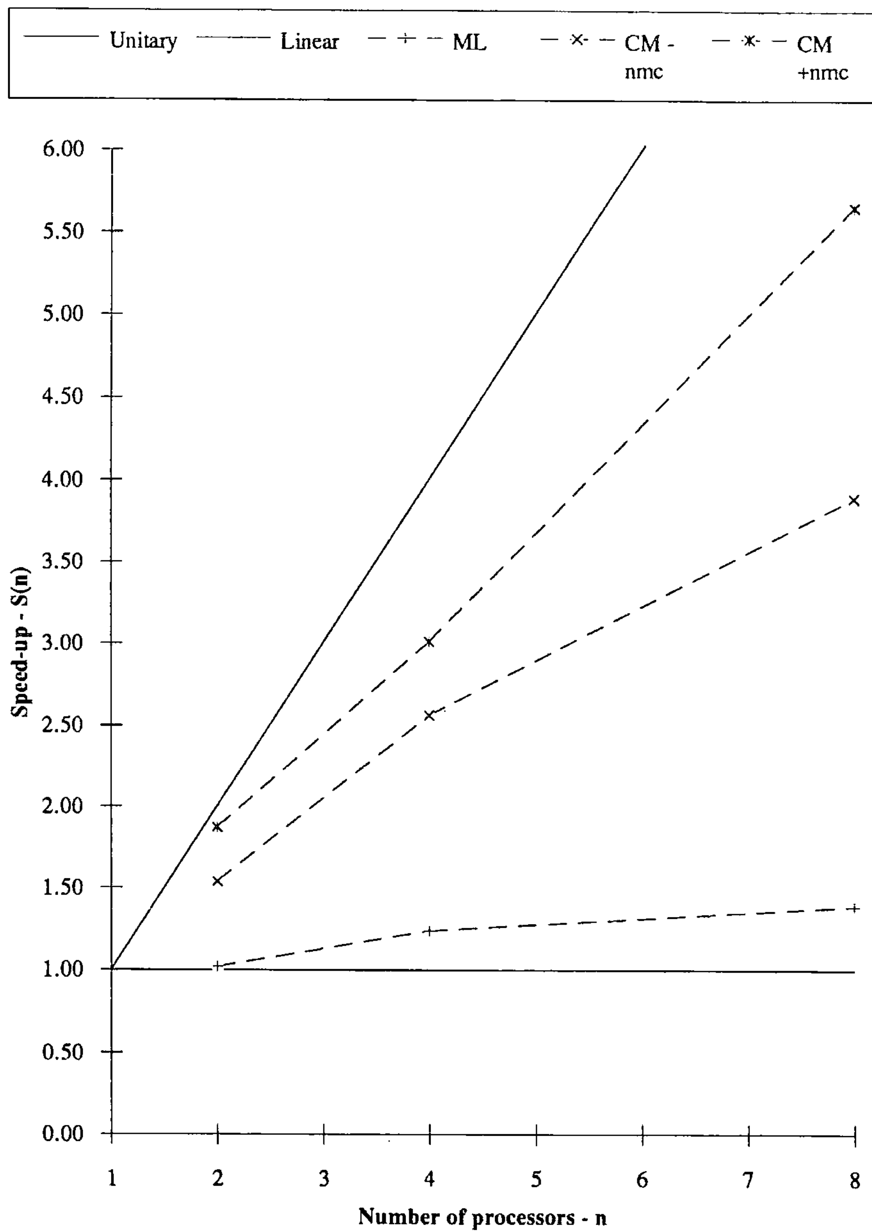


Figure 5.27: Speed-up comparison of conservative synchronization methods for the tandem queueing network.

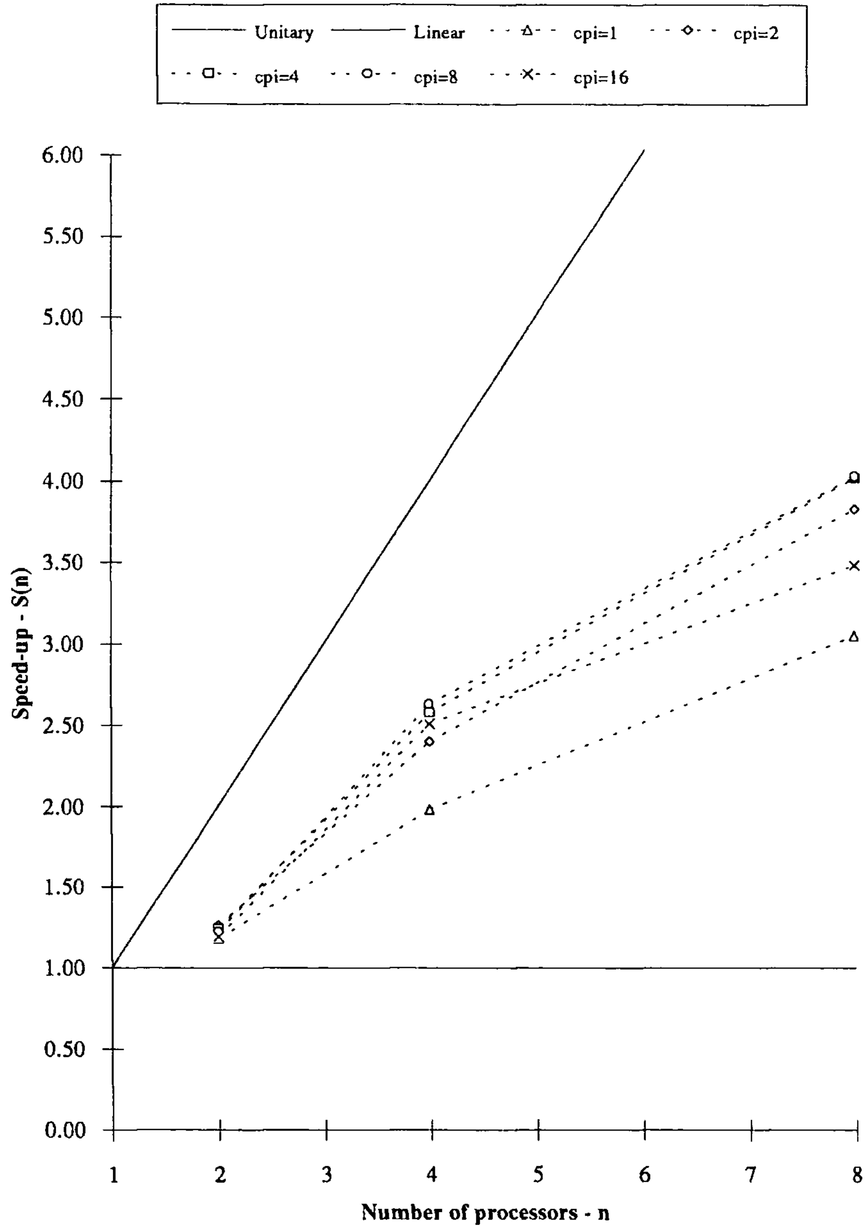


Figure 5.28: Speed-up using virtual time synchronization (VT) with lazy cancellation for the tandem queueing network.

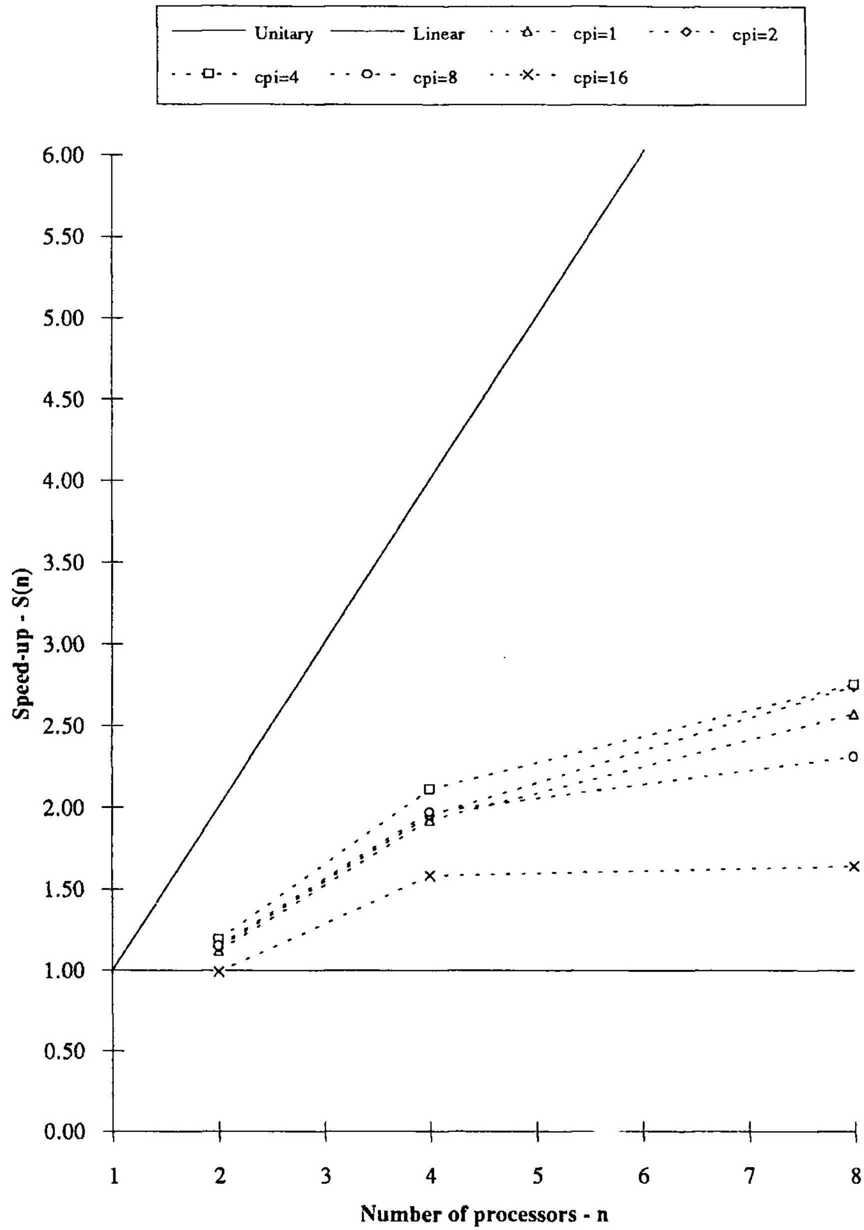


Figure 5.29: Speed-up using virtual time synchronization (VT) with aggressive cancellation for the tandem queueing network.

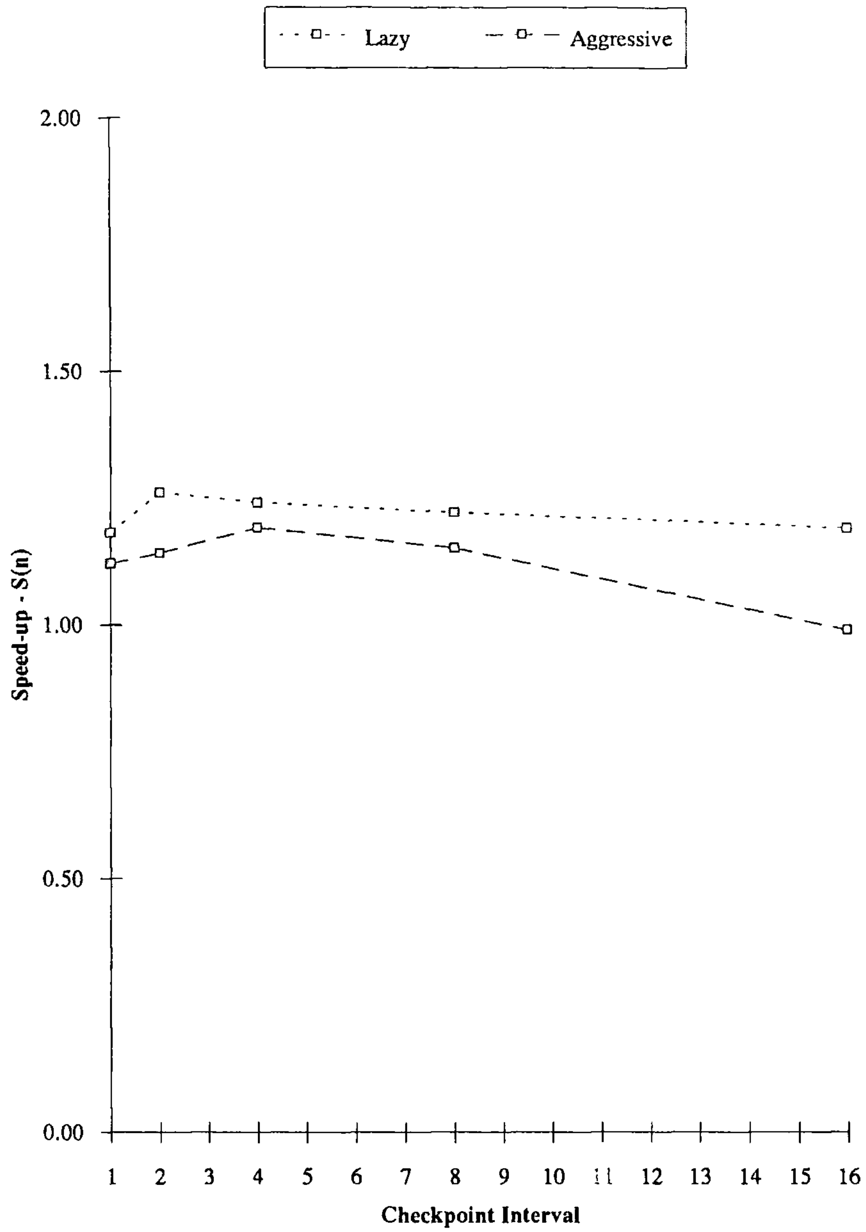


Figure 5.30: Speed-up against CPI for virtual time synchronization (VT) with two processors for the tandem queueing network.

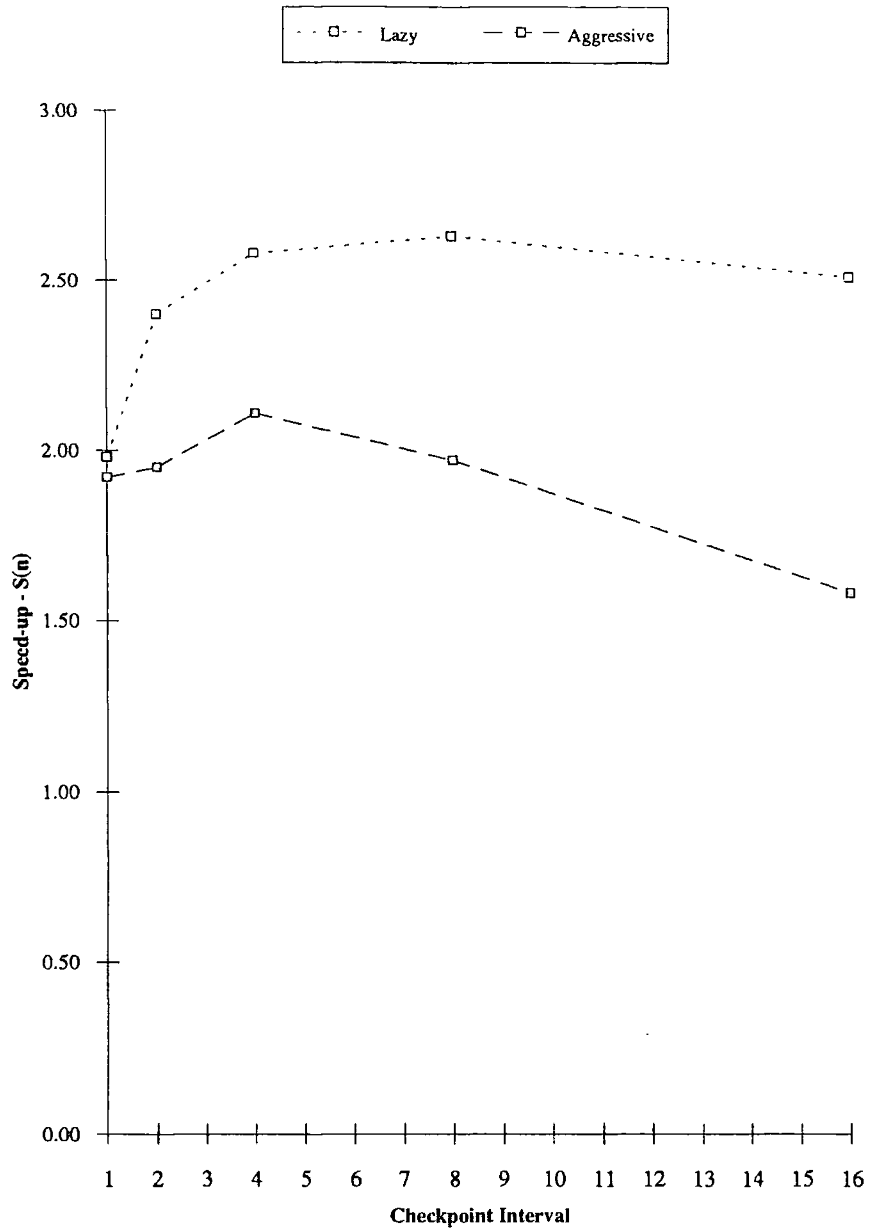


Figure 5.31: Speed-up against CPI for virtual time synchronization (VT) with four processors for the tandem queueing network.

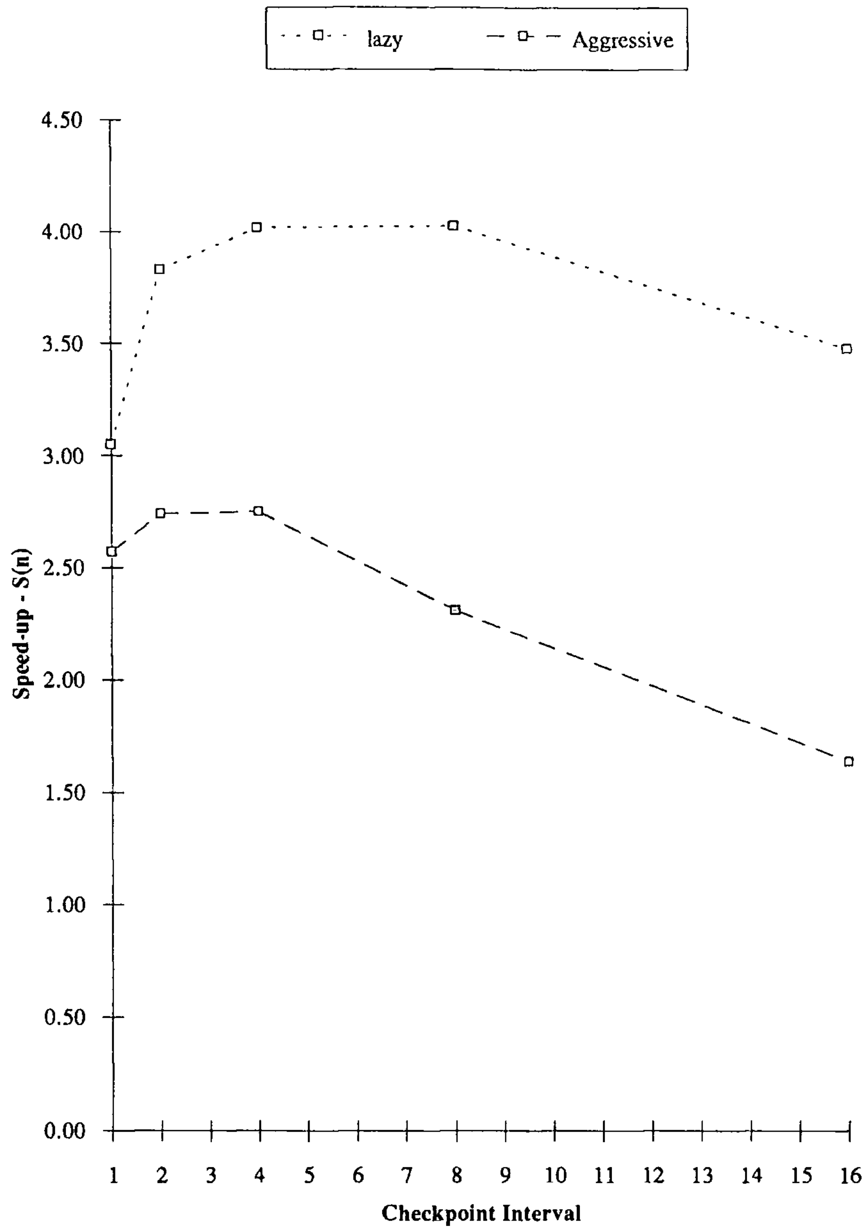


Figure 5.32: Speed-up against CPI for virtual time synchronization (VT) with eight processors for the tandem queueing network.

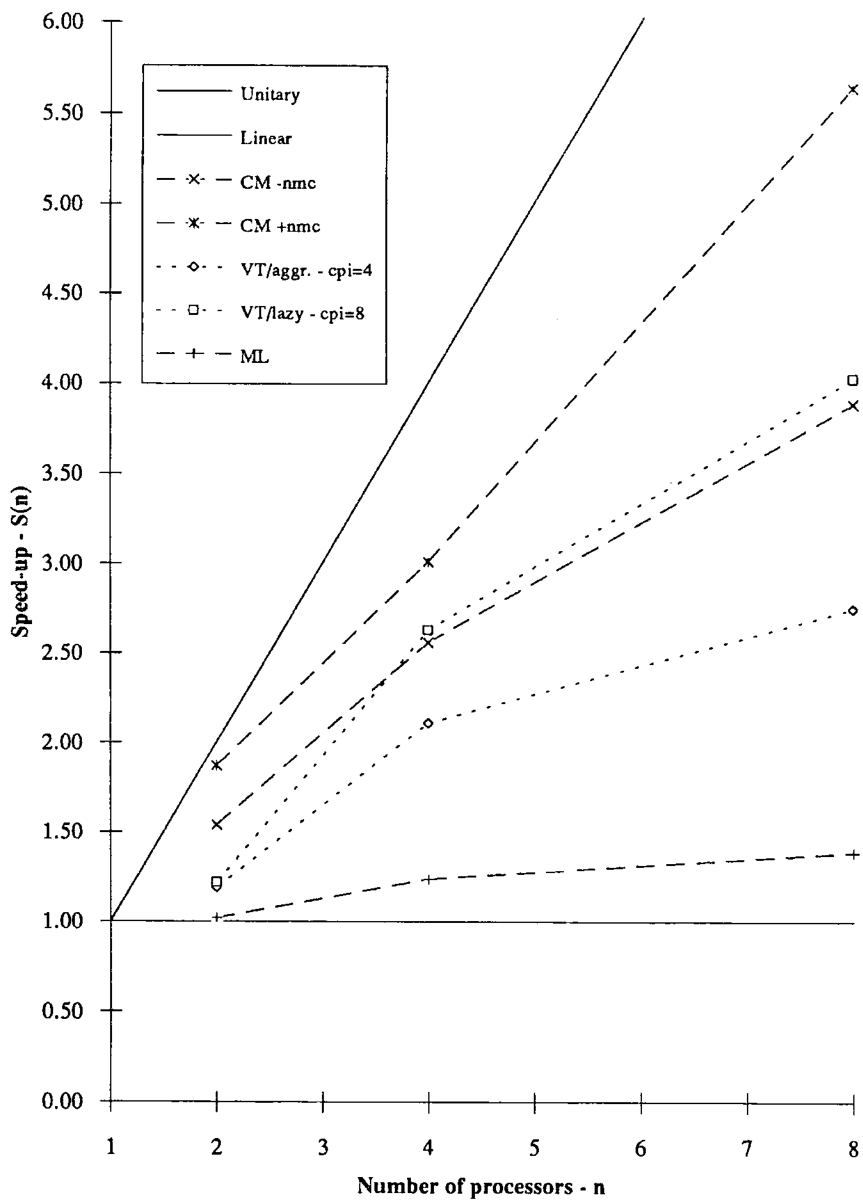


Figure 5.33: Speed-up comparison of all synchronization mechanisms for the tandem queueing network.

Chapter 6

Parallel Simulation of Asynchronous Transfer Mode Networks

6.1 Introduction

The arguments for parallel simulation of telecommunication networks become most pointed when considering the performance evaluation of proposed broadband networks for an integrated services digital network (ISDN). The complexity, traffic intensity and the potential size of such networks are all (at least potentially) very large. Simulation studies of systems of this nature have thus far largely centred on the behaviour of a single traffic source, multiplexor or switching node. When complete networks have been studied it has usually been at the call- or burst-level[1] since more detailed simulation involves levels of complexity (and hence processing time) which are orders of magnitude greater; the consequent loss of detailed information can sometimes also lead to misleading results. For many investigations into network behaviour detailed simulation is unavoidable and motivated the development of a multiprocessor ATM network Simulator at the University of Durham. This has been used for the study of switch behaviour, network behaviour and, particularly, for studying the integration of mobile communication protocols into the broadband environment [180]. A parallelizing compiler approach was not seriously considered for such a complex discrete

[1] A burst of cells is defined as a cell rate lasting for a particular period during which the inter-cell time is constant. Thus, in a burst-level simulation, an event marks a change in the cell rate. See the work of Pitts et. al. [176–179].

event simulation so the simulator used the distributed model components approach. Also, as the simulation model was itself complex to implement a conservative synchronization approach was used based on NULL-messages.

Other researchers have implemented parallel simulators to model ATM networks. Phillips and Cuthbert et. al. [88, 89] used a small network of Inmos transputers in the detailed cell-level modelling of ATM networks based on queueing network models. A typical model consisted of thirty queues and ten traffic sources. The speed-up results compare three synchronization strategies for up to three processors; a centralized event-list, a distributed event-list and a conservative approach using NULL-messages. The centralized event-list yielded the best results, but no speed-up. This was explained by the authors to be due to the high number of processes and the high level of communications; thus the simulations were communication bound. They went on to suggest that improvements in message handling and the use of lookahead (additional knowledge) should bring better results.

Mellor et. al. [181, 182] at the University of Durham have developed an ATM simulator also based on a small network of Inmos transputers using a conservative synchronization approach with NULL-messages. They model traffic at the burst-level and switching and other functions at the cell-level. They also report little or no speed-up using a small number (two to four) of processors. This result is again explained by high communication overheads. Also, they only implemented small models with relatively few processes and hence there was limited parallelism available for exploitation.

Chai and Ghosh [183] at Brown University, USA. in contrast, used a loosely-coupled network of industry standard SUN workstations using an Ethernet LAN. Thus, we should term this distributed rather than parallel simulation. The synchronization strategy used is an alternative form of the conservative NULL-message approach, where they are sent only at the request of a process. The query, called by the authors an alarm request, is sent when a process is blocked and needs an improved clock time. They have investigated several large detailed cell-level network models, between ten and fifty switching nodes and present a wide range of performance results for the networks. The speed-up obtained is difficult to assess as no uniprocessor simulations were performed; only runs on ten, thirty and fifty processors with one switch per processor in each case. Indeed, the size of the models and the way they were implemented made it impossible to achieve a uniprocessor simulation. What can be said is that larger network models can be simulated as the number of workstations available increases without significant time penalties; see table 6.1.

Number of switches (processors	*Run times (mins)*	*Cells sourced*	*Cells sourced per second*
10	31	724,000	23,354
30	87.75	2,265,000	25,812
50	90	3,440,000	38,222

Table 6.1: Brown and Chai's Distributed Simulation Results.

6.2 Broadband Networks

The CCITT define an integrated services digital network (ISDN) as one " ... that provides end-to-end digital connectivity to support a wide range of services, including voice and non-voice services, to which users have access by a limited set of standard multi-purpose user-network interfaces" [184]. This contrasts with most current networks which were designed for very specific services, such as voice telephony: using such networks for other services leads to problems and shortcomings. More and more service-dedicated networks are not the solution; ISDN hopefully is. Initially, *basic access* was centred around two 64 kbits/s data channels (B channels) and one 16 kbits/s signalling channel (D channel). In addition, *primary access* is defined with a gross bit rate of 1.5 or 2 Mbits/s. *Primary access* is more flexible offering a mixture of a 64 kbits/s signalling channel with a combination of basic, B, or high-speed, H, data channels at 384 to 1920 kbits/s. However, in the context of current local area network (LAN) technologies, these are relatively low-speed fixed bandwidth channels. The requirement for supporting even more advanced multi-media services within ISDN, particularly those with variable bit-rate traffic, has led to the development of broadband ISDN (B-ISDN).

Asynchronous transfer mode (ATM) is the target solution for B-ISDN defined by the CCITT. Such a network uses a fixed-size data packet, known as a *cell*, which consists of 48 octets of data and 5 octets of header. Cells are typically transmitted, within the network, using multi-megabit-per-second media, such as fibre-optic links; such links will typically be running at data rates in excess of 150 Mbit/s. Good background texts on B-ISDN, and ATM networks in particular, have recently been published by Händel and Huber [185] and by De Prycker [186].

The ATM switch used in this simulation study is based on the Orwell ring protocol [187] which is a slotted ring protocol. The ring is divided into slots which circulate around the ring; a node wishing to transmit a message waits until an unfilled slot is found, changes the slot header and transmits the message in the body of the slot. Slotted ring protocols have

been unpopular in the past for several reasons. A monitor node is required to ensure that slots that become corrupted can be identified and regenerated, thus correct behaviour of the ring is critically dependent on correct behaviour of the monitor. To get a reasonable number of slots onto the ring delays have to be inserted at each node and one node, normally the monitor, has to be able to adjust its delay so that there are an integral number of slots. In LAN terms, the efficiency of slotted rings is generally poor since the ratio of header to body is normally high. Its greatest advantage over token-based protocols, however, is that more that one node can be transmitting information at a time, using different slots on the ring. Acknowledgement of delivery is normally made by releasing the slot at the source (correct receipt there is taken to imply correct delivery at the destination); the node may not refill a slot that it has just released, ensuring that the slot is passed to the next node and thereby ensures fair access to all nodes on the ring. An earlier implementation of a slotted ring is the Cambridge Ring protocol (British Standard BS6531).

Examination of existing protocols has indicated that those based on a slotted ring are probably the best suited for carrying delay-sensitive traffic such as speech. However, simulation studies of high-bandwidth Cambridge Rings have indicated that there are still significant limitations when operated under high load [188] and, further, that load control is difficult since there is no relevant parameter that can easily be extracted from the ring. The Orwell protocol was developed after making a detailed study of the limitations of the Cambridge Ring protocol: it was found that by introducing destination release of slots, and by adding a novel, distributed, load control mechanism to bound access delays, a viable level of performance could be obtained [189, 190]. For higher capacity networks multiple, synchronized, rings can be used and such a network is known as an Orwell Torus. Thus, the Orwell ring shows considerable promise as a low capacity distributed ATM switch which can support multi-media traffic.

Whilst detailed simulations of a single Orwell ring have been made, under a variety of load and traffic services, there has, as yet, been very little investigation made into the behaviour of an Orwell torus, or ring behaviour in multi-ring systems. The reason for this, at least in part, is because of the large amount of simulation time required to investigate networks of Orwell rings. The original sequential simulation was written at BT. Laboratories and run on a VAX 11/750. The accuracy of the simulation model was validated against a testbed Orwell ring. The multiprocessor simulations described here were in turn validated against the results of the BT simulator.

6.3 Simulator Architecture

6.3.1 The Multiprocessor Testbed

The multiprocessor testbed used for the ATM simulator is based on a network of Inmos transputers. This was originally designed for use as a high-speed circuit-switched network simulator, with code written in occam; subsequently, a traditional packet-switched network simulator was also developed using the same language [63,65]. The testbed architecture consists of up to 31 simulation transputers, each with up to 16 Mbytes of memory (the implementation used in these experiments consisted of 13 T800 processors, one with 16 Mbytes and the others with 1 Mbyte of memory). The transputers are connected with a double layer of cross-point link switches which enables any link on each of the simulation processors to be connected to a link on any of the other processors; this flexibility enables the network to be configured in arbitrary topologies so that the system being simulated can be mapped closely onto the processor network, and enables the path length required when passing messages between processors to be kept to a minimum; there is no shared memory in the system. Finally, a layer of control processors are used to connect between the host transputer (so-called as it normally resides in a conventional workstation) and the link switches; one is connected to the link-switch programming interface, while both can be connected, via the switches, to any of the simulation transputers. Figure 6.1 shows a functional representation of the hardware used.

6.3.2 The Software Architecture

To isolate the simulation model, as far as possible, from the implementation details of the hardware, the simulator was structured in a hierarchical manner; each layer building on the abstraction of the layer below in a similar approach to that of the ISO. seven-layer model. At the lowest layer lie the transputer processors in a dynamically reconfigurable array. On top of this a multiplexor task on each processor provides the abstraction of virtual channels between each task in the simulation, regardless of where the tasks are mapped in the processor network. Thus, the multiplexor provides message buffering, routeing, and a virtual topology configuration which exactly matches the simulated network. A simple packetizer layer hides the fact that the channels in the multiplexor (and, indeed, the physical channels of the transputer itself) work most efficiently when presented with large packets as opposed to a series of very small ones. A synchronization layer uses the packet layer

processes; it ensures that each message is correctly marked with a time-stamp on dispatch and uses this at the receiver to maintain synchronization: the layer is optional, if there is no definable synchronization between two tasks (for example, diagnostic messages destined for the console) then the channel can be declared asynchronous and the packet layer accessed direct. Finally, in parallel with the simulation model and the synchronization layer, an event manager is responsible for scheduling components of the simulation model in the correct sequence. The overall hierarchy is shown in figure 6.2. The implementation is described by Earnshaw in more detail elsewhere [66,87,191].

6.4 The Synchronization Mechanism

Synchronization approaches for parallel simulation using the distributed model components approach have been discussed in chapter 3. The approach used here was the conservative Chandy-Misra-Bryant [81–84] approach using NULL-messages to avoid deadlock situations occurring. To recap, NULL-messages are only used for synchronization purposes and do not correspond to any activity in the physical system being simulated and, hence, have no message content only a time-stamp t_{Null}. Thus, they are essentially a promise that the sending process will not send a real message to the destination process with a time-stamp less than t_{Null}. NULL-messages are sent on each outgoing port whenever a process finishes simulating an event. Generally, conservative synchronization approaches can achieve good performance with sparsely-connected systems which have less opportunity for deadlock and an application which contains good lookahead properties. Lookahead refers to the ability to predict what will, or will not, happen in the simulated time future based on application specific knowledge.

For an ATM link between the switches in our simulation model it is possible to derive a simple formula describing the propagation delay and also the number of cells, N, that could be in transit across a link of length, L, at any instant:

$$N = \frac{LSn}{lc}$$

where, S is the speed of the link (adjusted to account for overheads such as framing), n is the refractive index of the transmission medium (typically, about 1.5 for a glass fibre), l is the cell size and c is the speed of light. Considering, for example, a modest 15 km link running at 150 Mbit/s, then there may be up to twenty-six cells in transit across the link

at any time; longer, or faster, links would have correspondingly larger numbers of cells in transit. This "pipeline" is used to advantage as a method of lookahead within the simulator. Effectively, a destination task can see a small amount of future behaviour for the link: this can then be exploited for two ends; the avoidance of deadlock with fewer NULL-messages and the improvement of concurrency between the processes.

In the Chandy-Misra-Bryant simulation, there is not normally an event processor in the classical sense. Instead, events are replaced exclusively by messages and the order of processing determined by selecting the message with the oldest time-stamp: there must be a message available from each incoming link in order to be able to do this; the absence of a message causes the node to block. In the ATM network simulator an event manager is used; consequently, in addition to adding dependence on the link mechanisms to the code of the event manager, monitoring for messages would be inefficient. To overcome this, the synchronization routines are implemented as normal events that run in the same manner as all other events in the simulator: two events are required between each pair of dependent processes; these are a NULL-message generating event and a process blocking event.

The NULL-message generating event runs on the output of a link: it compares the current simulation time with the time when a message was last sent to the remote process; if this is less than a link propagation delay it simply re-schedules itself to a time one propagation delay later than the time at which the last message was sent; otherwise, it must be exactly one propagation delay since a message was last sent, so a NULL-message is generated to the remote process and the generator re-schedules itself one propagation delay later. The process blocking event compares the simulation time against the time when a message was last received across a link from the remote process; if this is less than a propagation delay then it simply re-schedules itself for one propagation delay after the time the last message was received; otherwise it may not continue until a message is received and blocks until this occurs before re-scheduling itself accordingly. The process blocking event appears to the rest of the simulation as one that takes just sufficiently long to execute that the process remains in synchronization with its neighbours; however, while blocking, it consumes no processing time. It is possible to show that, provided that the lookahead is greater then zero, cyclic dependencies that could lead to deadlock cannot exist. This is shown by Earnshaw in his thesis [87].

6.5 The Simulator Results

The results produced by the simulator consist of sets of statistics for the traffic patterns and the switch activity. In addition, statistics are also gathered giving the various performance indices for the simulator itself: this can be assessed from the run-time, processor usage and link usages. The performance of the synchronization mechanism is monitored, along with several other aspects, by an event profiling process; this gives the number of instances and percentage processing time spent on various simulation events. Such profiling is made easier as the transputer has a hardware timer which allows the profiler to be run at fixed time intervals. Traffic patterns are reported as a set of histograms of the voice delay statistics for each source in the network. Switch activities are also reported as histograms of the input queue lengths to the Orwell rings, the ring reset and cell delay statistics.

6.6 Performance Analysis of the Simulator

6.6.1 Performance of Production Runs

The performance results given here are for the ATM Network Simulator configured as shown in figure 6.3: the network consists of four ATM switches in a fully-connected *trunk* network and eight *local* switches each of which is dual-parented onto two trunk switches. Each local switch has two traffic generators which can generate normal voice calls or a mixture of conventional and mobile voice calls. The switches were all running the Orwell ring protocol (see section 6.2). Two sets of results were taken initially with differing switch capacities and traffic mixes. In both cases the links were running at 150 Mbit/s and the propagation delay was set to 0.1 ms (equivalent to about 20 km of glass fibre, or about 35 cells). All inter-switch and source-to-switch distances were set to 20 km. The results for the lower traffic load were taken using 150 Mbit/s Orwell rings for all the switches with a mixture of voice and mobile traffic; the results for the higher loads used conventional voice traffic and a ring speed of 600 Mbit/s for the trunk switches. With the smaller capacity switches the maximum link loading was about 15% before the rings saturated, but this was increased to about 50% for the higher capacity rings. Two uniprocessor simulations were run for each load: one with identical code to the multiprocessor version, the unoptimized version; the other with the redundant multiplexors removed to speed message transfer, the optimized version. Ideally, of course, the speed-up figures should all be relative to a version of the

simulator running on a single processor (transputer) using efficient event-list processing. In the following graphs, when the load is shown it is expressed as the average percentage of the capacity of a link.

These experiments were production runs in that the results of the simulations were used to validate the simulator, examine the performance of the switches and verify the operation of the protocol developed to manage mobile voice traffic on the network. Thus, these runs had to be of sufficient length; 12.5 s for the 150 Mbits/s rings and 7.5 s for the 600 Mbits/s with the statistics reset after 2.5 s in both cases. The length of a production run is constrained by several criteria. First of all, the normal simulation constraints have to be observed. The statistics being collected should be reset (or collection begun) only after the *warm-up* period is complete and the system has reached equilibrium. Also, the simulation needs to be of sufficient length for the statistics gathered to be significant. To minimise execution times the call holding time can be reduced and the call arrival rate increased such that the average load remains the same (thus causing the simulation to reach equilibrium more rapidly): previous simulations have shown that this technique has no noticeable effect on the statistics collected due to the very large number of cells involved in even a short call. The technique is described by McGeeney [192].

Figure 6.4 shows the time taken to simulate the two models on an array of twelve processors. The times for the 150 Mbits/s rings has been scaled by a factor of 7.5/12.5 to take account of the difference in simulation length. Therefore, the simulation lengths are effectively the same. The fact that the two curves do not pass through the origin (ie. for zero traffic) has two causes: the NULL-message traffic for low loads and the overhead of simulating the ring slot-rotation action for the Orwell protocol. That it is the latter that represents the dominant factor can be inferred from the fact that the NULL message ratio for each of the two curves is almost identical for a given link loading, as can be seen in figure 6.7. If the NULL-messages were the cause, then the two curves would cut the axis at the same point[2]. As it is, the ring slot-rotation action is dominant as there are fewer slot-rotations (about two-thirds) in the 150 Mbits/s simulations. Figure 6.4 also shows that the simulation run time of the model using mixed conventional and mobile voice traffic is growing faster with load than for purely conventional voice traffic. This is not surprising due to the extra overheads incurred in managing the mobile protocol [180]. These times

[2]The number of NULL-messages in the two simulations of the same length with the same traffic load would be (approximately) the same as long the lookahead in both were the same.

should be borne in mind when looking at the speed-up results as the average run-time across all traffic loads for the 600 Mbits/s rings is just over three hours; thus a speed-up of ten implies a uniprocessor run-time in excess of thirty hours.

Figure 6.5 shows the speed-up of the simulator as a function of load for the 150 Mbit/s rings; it shows that, even for a load of just 15% of maximum capacity, the speed-up is approaching the ideal value of twelve for the unoptimized version, and is starting to level out at just over ten when compared with the optimized version. The difference between the two curves represents the proportion of the processing time that is taken up in switching the messages from one processor to another. The speed-up of the simulator relative to the unoptimized version can also be estimated from the processor activity monitoring of each of the transputers in the parallel simulation: the results from doing this agree well with the curve for the unoptimized version. Figure 6.5 shows, for the 600 Mbit/s in comparison with the unoptimized version, that the speed-up is greater than nine for all loads simulated, and for link loads greater than 30% it is almost unitary[3]. This result shows that the communication overheads in the parallel simulation are effectively hidden and that the synchronization method is very efficient for this simulation model. If the message passing code for the multiprocessor version could be made more efficient then it is possible that the speed-up, when compared against the optimized version, could be improved upon still further.

A useful performance indicator for the Chandy-Misra-Bryant synchronization method is the NULL-message ratio (NMR) which is defined as follows:

$$NMR = \frac{N_{null}}{N_{null} + N_{mess}}$$

where N_{null} is the number of NULL-messages generated in the simulation and N_{mess} is the number of real messages generated, in this case ATM cells. The numbers of messages reported here were averages across all twelve of the switches. This was felt to give better comparisons between different simulation runs (than the number for a single switch or the total for all switches) particularly in the case where asymmetric traffic patterns were used.

Figure 6.6 shows that the speed-up degrades gracefully with increasing NULL-message ratio; but, fortunately, as can be seen more clearly in figure 6.7, the NULL-message ratio

[3]Careful examination of figure 6.5 reveals that a speed-up slightly greater than twelve is achieved at a load of about 40%. This was due to the single and multiprocessor simulations being run with different random number seeds.

remains very low for a large range of the load. This reinforces the conclusion that the synchronization message overhead is very low for the parallel simulation with a realistic traffic load.

The results given in figure 6.6 for the 600 Mbit/s rings show that for a speed-up of approximately nine, the NULL-message ratio is almost one. As the NULL-message ratio decreases by a factor of around 5000, the speed-up only increases by about 20%. This seems to indicate that the NULL-message ratio has less impact than as first thought. However, this is explained by the overheads in NULL-message processing. As mentioned previously, a NULL-message generator event is periodically processed and re-scheduled for each output link of a process regardless of whether a NULL-message is eventually generated or not. It is important to note that there will at least be a NULL-message generator event scheduled at intervals equal to the lookahead; in this case, the propagation delay across a link. It is reasonable to assume that most of the processing overhead of sending a NULL-message is incurred by processing and scheduling the NULL-message generator event. As this overhead occurs whether the NULL-message is sent or not then this would explain why the reduction in the NULL-message ratio has relatively little impact on the speed-up

6.6.2 Variations in Lookahead

A series of simulations were then undertaken with the all of the inter-switch and source-to-switch distances set symmetrically to 2 km, 20 km or 200 km. Another series was done using a *UK. national* network[4]. All of these were performed using the usual range of symmetric conventional voice traffic loads. The national network has the same topology as before but with inter-switch distances between 51 km and 343 km (average 174 km) and source-to-switch distances between 0.5 km and 7.5 km (average 4.3 km). The overall average distance is 103 km. Thus, the national network can be said to have asymmetric lookahead.

The simulation length used for this series of experiments, 0.5 s, was much shorter than that used for the production runs. As we have already noted, the run times for the uniprocessor versions are prohibitive; particularly in terms of the series of simulations proposed above. The length used was a compromise: short enough to be practicable and long enough for the speed-up not to be biased by the sequential parts of the parallel simulation (pre-

[4]It is *national* in the sense that the trunk and local switches are placed at major cities in the UK. and the distances between them calculated accordingly. The source-to-switch distances were set randomly between 0.5 and 7.5 km. The distances normally found currently in the local loop.

processing data and booting the processor network at the beginning and writing results at the end of the simulation period). As the simulator results for the model are not of interest for these experiments, the previous assumptions concerning statistical significance may be ignored. It should be noted, however, that the speed-up curve for the 20 km symmetrical lookahead model in figure 6.8 is not the same as that in figure 6.5 as the shorter run times do not allow the simulations to reach equilibrium.

Figure 6.8 shows the speed-up against traffic load (with respect to the optimized version) for these experiments. They show that lookahead is the key factor in determining speed-up. There is a large increase in speed-up between the 2 km and the 20 km models, but less between the 20 km and the 200 km as there was less room here for improvement in the concurrency. This is borne out by examination of figure 6.9, where the NULL-message ratio against traffic load is quite close for the latter two models. The curve for the 2 km model is almost flat, reflecting the relatively small amount of lookahead in the model.

The speed-up curves of figure 6.8 also indicate that the introduction of asymmetry in terms of the lookahead for the national network has quite a profound effect on the speed-up. Indeed, the speed-up figures recorded are not consistent with the overall average (103 km) or the minimum lookahead (0.5 km) in the network as they are rather lower than those for the 20 km model but still greater than those for the 2 km model. They are most consistent with the average source-to-switch distance of 4.3 km. The answer lies partly in figure 6.9, where the NULL-message curve for the national network model lies only slightly below that for the 2 km model. When the number of NULL-messages was examined for each process the greatest number was recorded for the sources closest to the switches (i.e. those with the lowest lookahead). Indeed, the number of NULL-messages recorded at a source process is almost inversely proportional to the distance to the nearest switch. This is illustrated in figure 6.10 which shows the number of *Null*-messages normalised against the number at the lowest load (0.05 calls/source/s) plotted against link load for a selection of source-switch links.

Interestingly, the simulation run times for the national network are about the same as those for the 20 km model on a single processor, but are obviously longer for the multiprocessor. Examination of the processor activity revealed that the utilisations were lower than usual by about 10% on average. We deduce from this evidence that, due to the asymmetry in the lookahead, processes block much more often in the parallel simulation waiting for messages (real or NULL) on the shortest connected links. This scenario has much less

impact on the uniprocessor simulation as it is almost certain that there is another process waiting to be scheduled with work to do. The reduction in speed-up is of the order of 25 to 40% on the 20 km network depending on traffic load. Even so, a speed-up of around six on twelve processors is still a respectable result, particularly bearing in mind the length of the uniprocessor simulation.

6.6.3 Asymmetric Traffic

A short series of experiments similar to those described in the previous section was repeated but with an asymmetric traffic pattern. This was achieved by setting eight of the sources to transmit/receive calls to each other at 60 calls/s (42.4% of link load) and the rest to 20 call/s (14.13%). This gives a network average load of 40 calls/source/s (28.27%). The speed-up and NULL-message ratio results are plotted against traffic load in the figures 6.8 and 6.9 at this average load.

The speed-up figures for asymmetric traffics all show reductions from their symmetric traffic counterparts of between of 6 and 14% depending on the original lookahead. The higher the lookahead initially, the more impact the asymmetric traffic has. In fact it can be seen that the NULL-message ratio is biased towards that at the lower link load. On examining the processor activity it is plain that load imbalance is the primary cause due to a high concentration of traffic in one of the trunk switches.

6.7 Conclusions

The speed-up figures obtained were reasonable throughout the series of experiments, the speed-up for the production runs being particularly good. The production runs show that the synchronization mechanism, Chandy-Misra-Bryant message passing using NULL-messages and exploiting lookahead, is very effective at medium to high message loads for this kind of model. These runs have also indicated that the communication overheads in the simulator are effectively hidden. The most important result is that the lookahead in the simulation model is the key factor in determining good speed-up. It is not too surprising that the speed-up results relative to the unoptimized version of the simulator are nearly unitary as it has all the disadvantages of parallel execution (ie. the synchronization mechanism, the message passing and the process blocking) without any of the advantages. Also, the optimized version still has the overhead of the synchronization mechanism. Speed-up

comparisons with a true uniprocessor simulation using efficient event-list processing would obviously yield lower figures.

Both asymmetric traffic and asymmetric lookahead cause reductions in speed-up; asymmetric lookahead having the more profound impact as expected from the results on varying lookahead evenly. The reduction in speed-up for the national network is of the order of 25 to 40% over the 20 km network model for asymmetric lookahead depending on traffic load in line with average lookahead between sources and switches. This is thought to be due to the increased instance of process blocking in the parallel simulation while processes wait for messages on the shortest connected link. The reduction for asymmetric traffic is between 6 and 14% depending on the original lookahead in the model. This was found to be due to processor load imbalance brought about by the asymmetric traffic pattern.

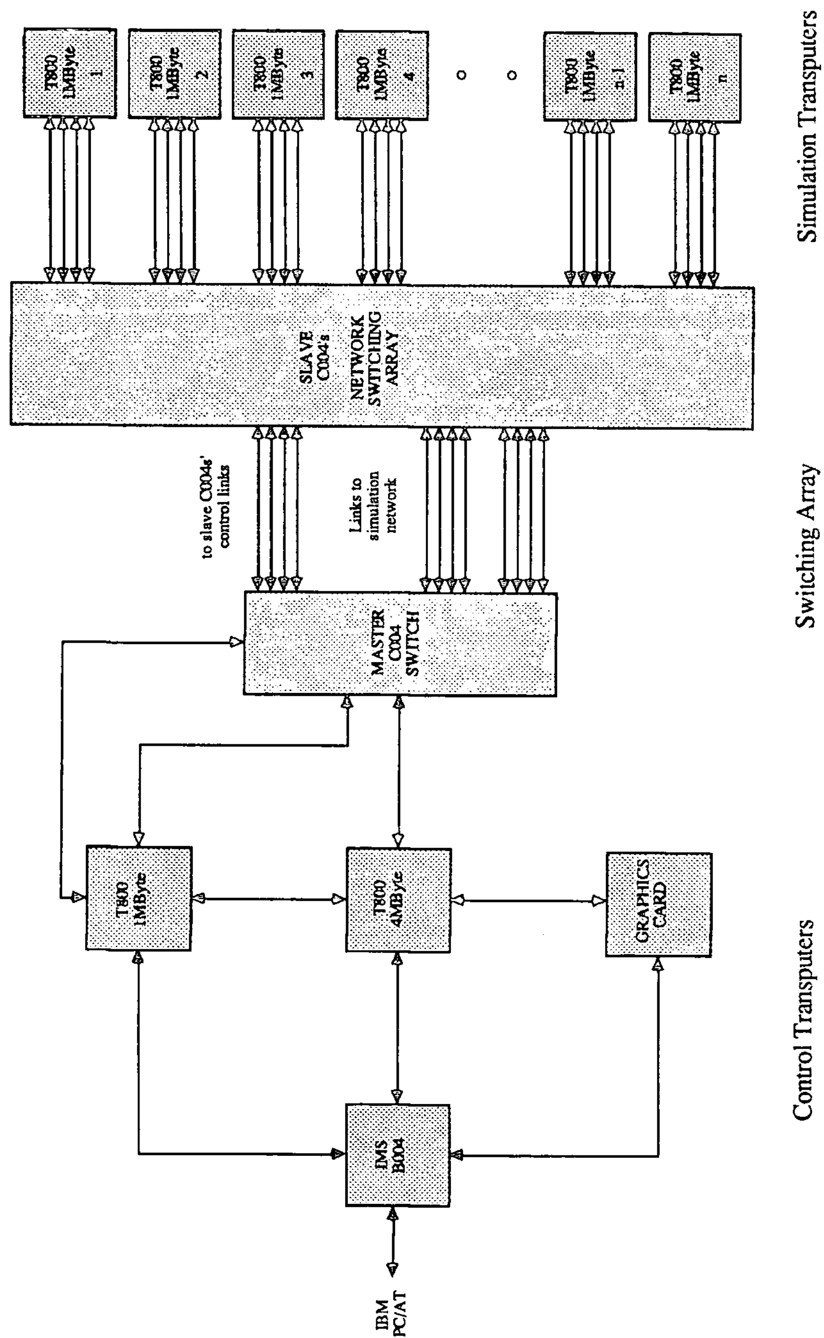


Figure 6.1: High-speed transputer-based telecommunication network simulator — hardware configuration.

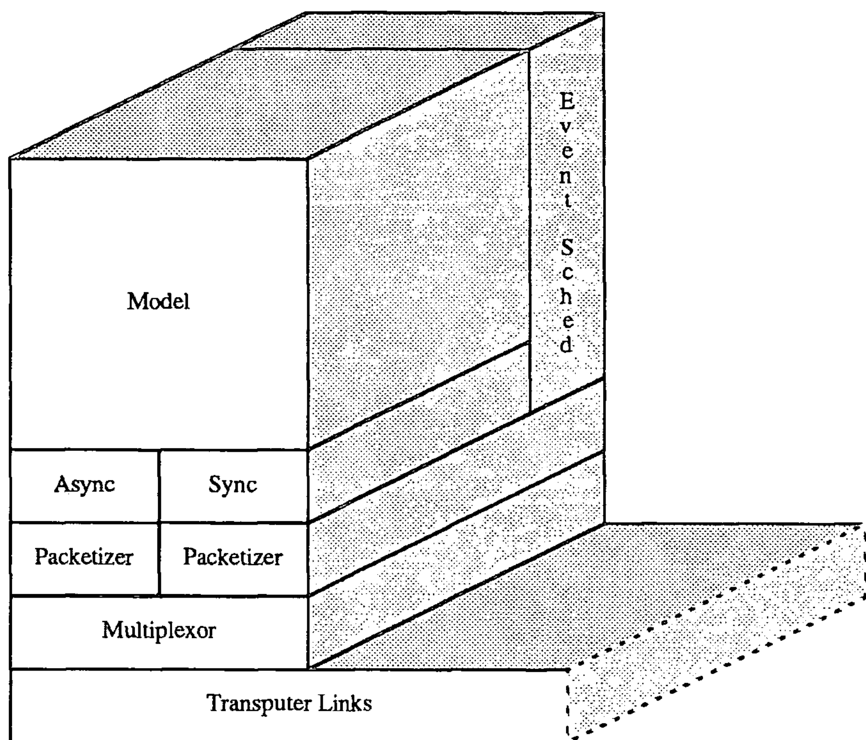


Figure 6.2: The overall hierarchy of the simulation model. The Event scheduler is a control-plane for all of the upper layers.

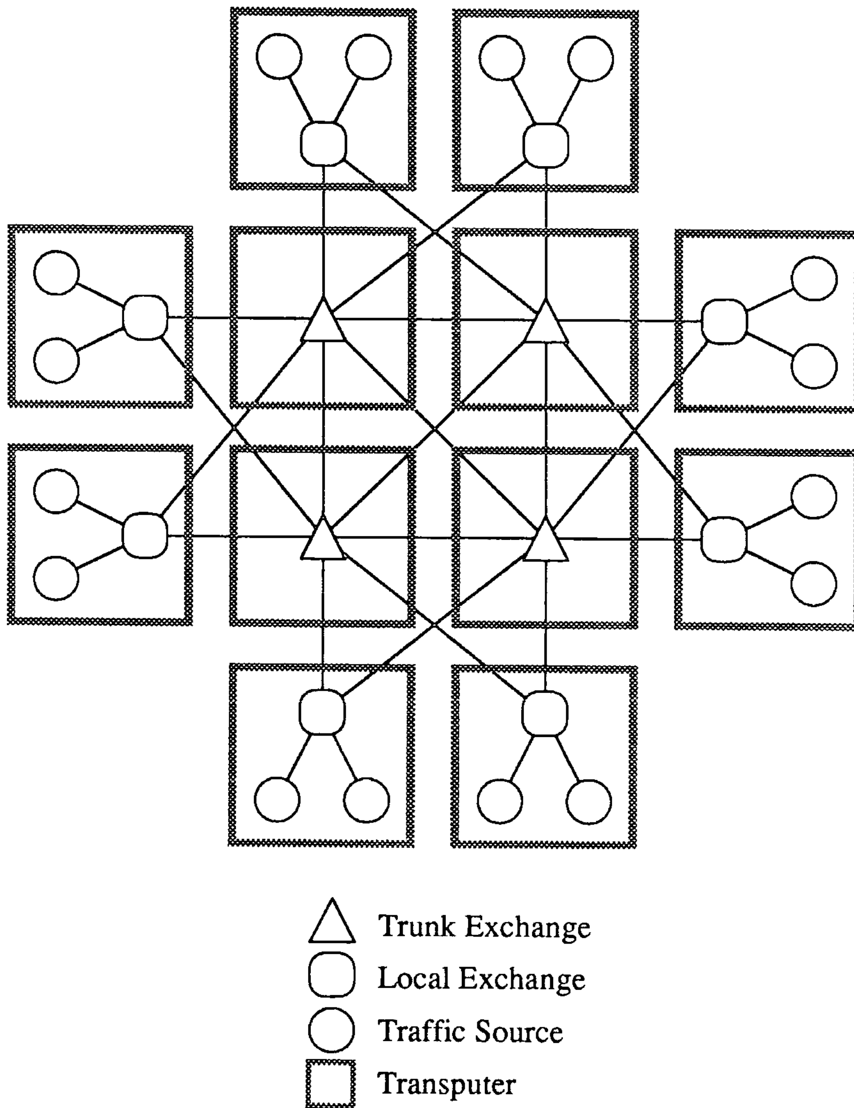


Figure 6.3: Basic network topology used for the simulator performance analysis runs. The processor assignments are also shown.

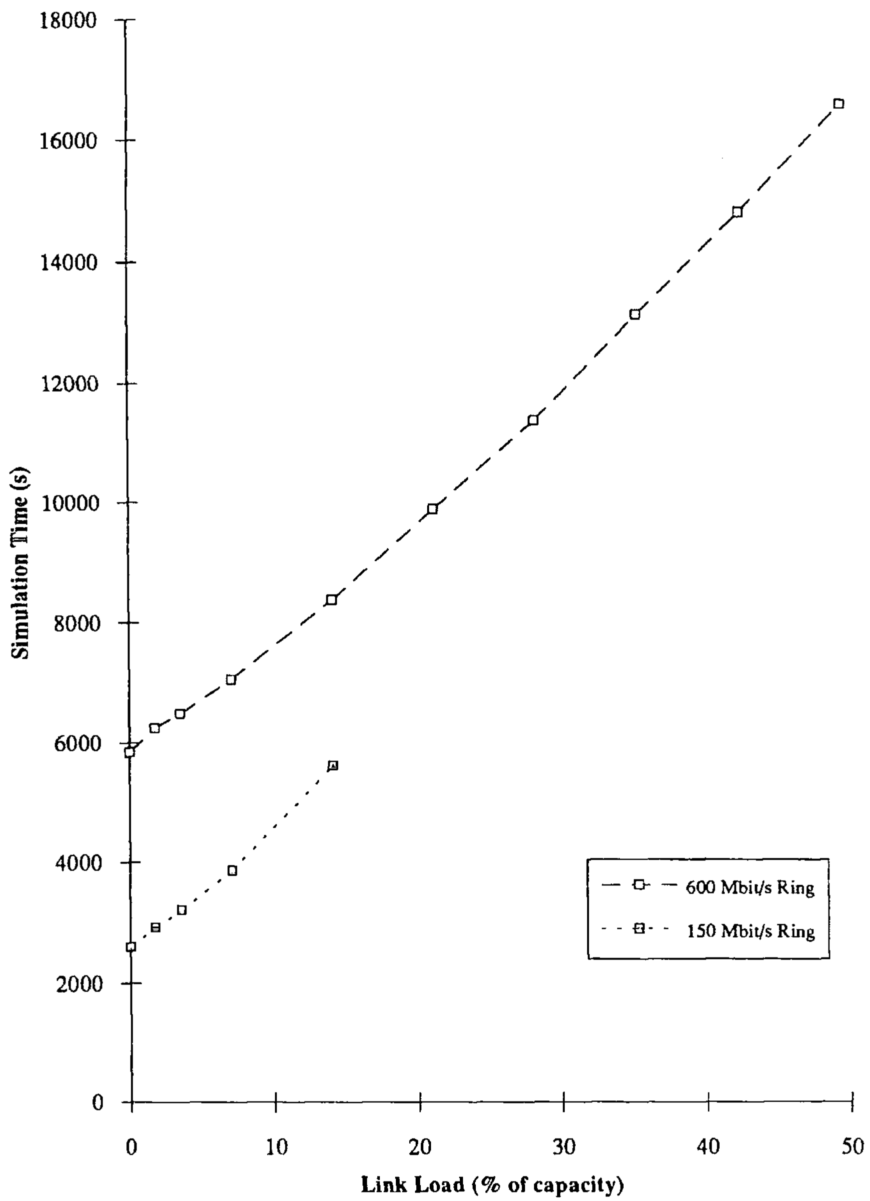


Figure 6.4: Parallel simulation run times as a function of traffic load for the twelve-node networks on twelve transputers. The 150 Mbits/s times are scaled to take account of the difference in simulation length.

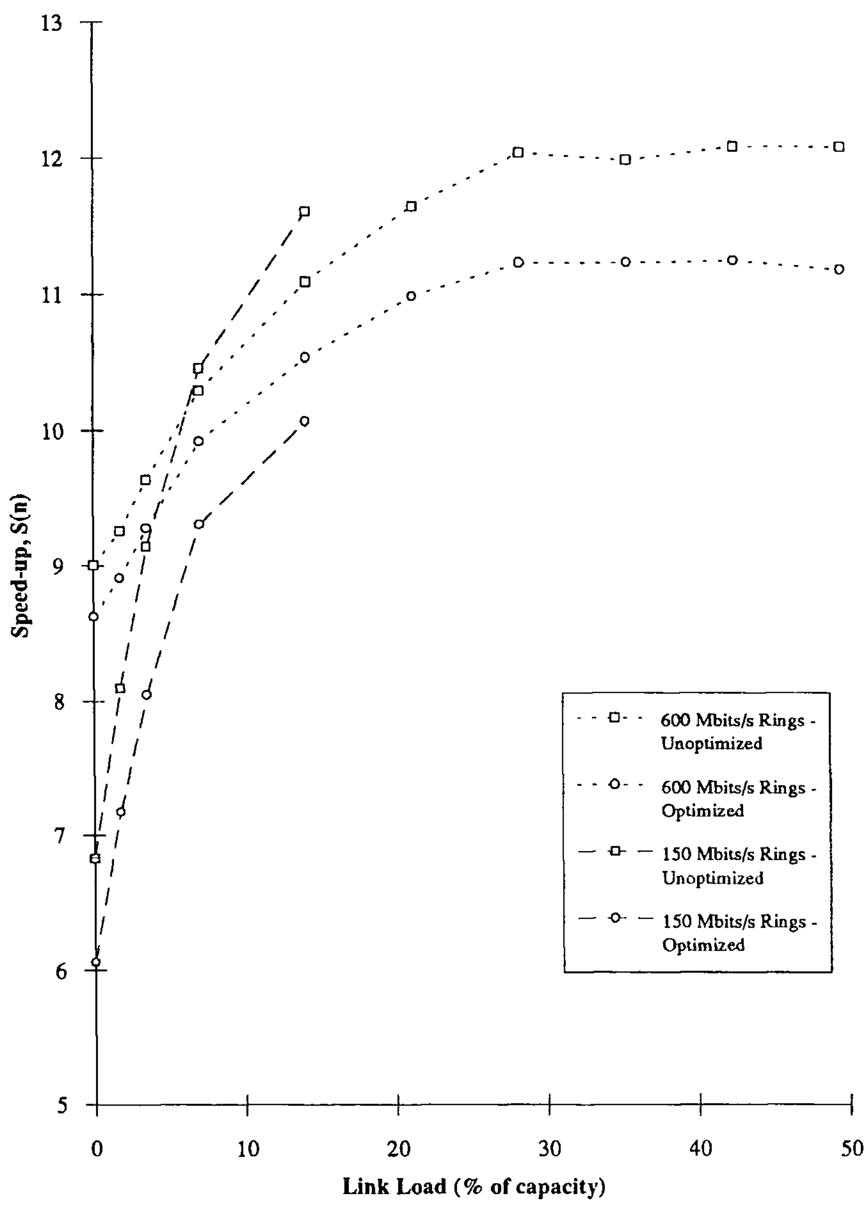


Figure 6.5: Speed-up curves as a function of traffic load. Speed-up is calculated relative to the optimized or unoptimized uniprocessor simulations.

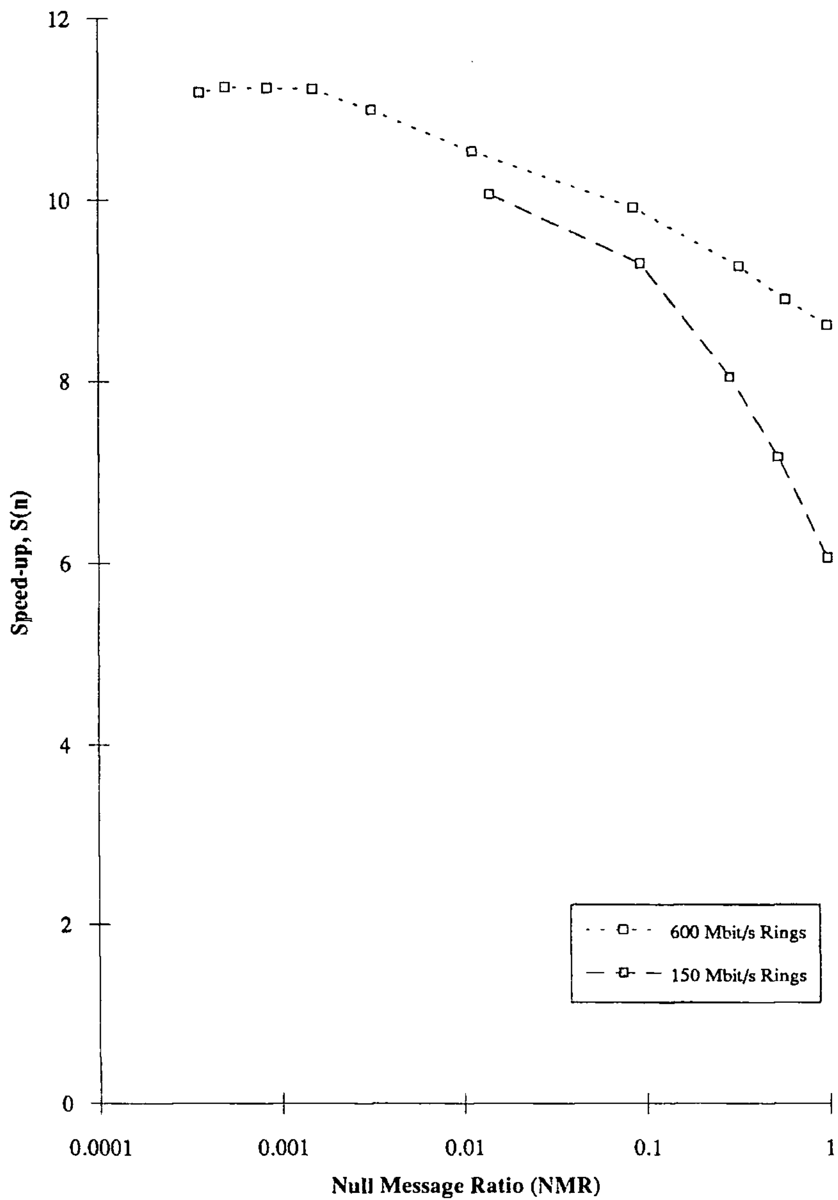


Figure 6.6: Speed-up (optimized) as a function of NULL-message ratio. The difference between the two curves represents the extra parallelism that can be extracted from the higher speed rings.

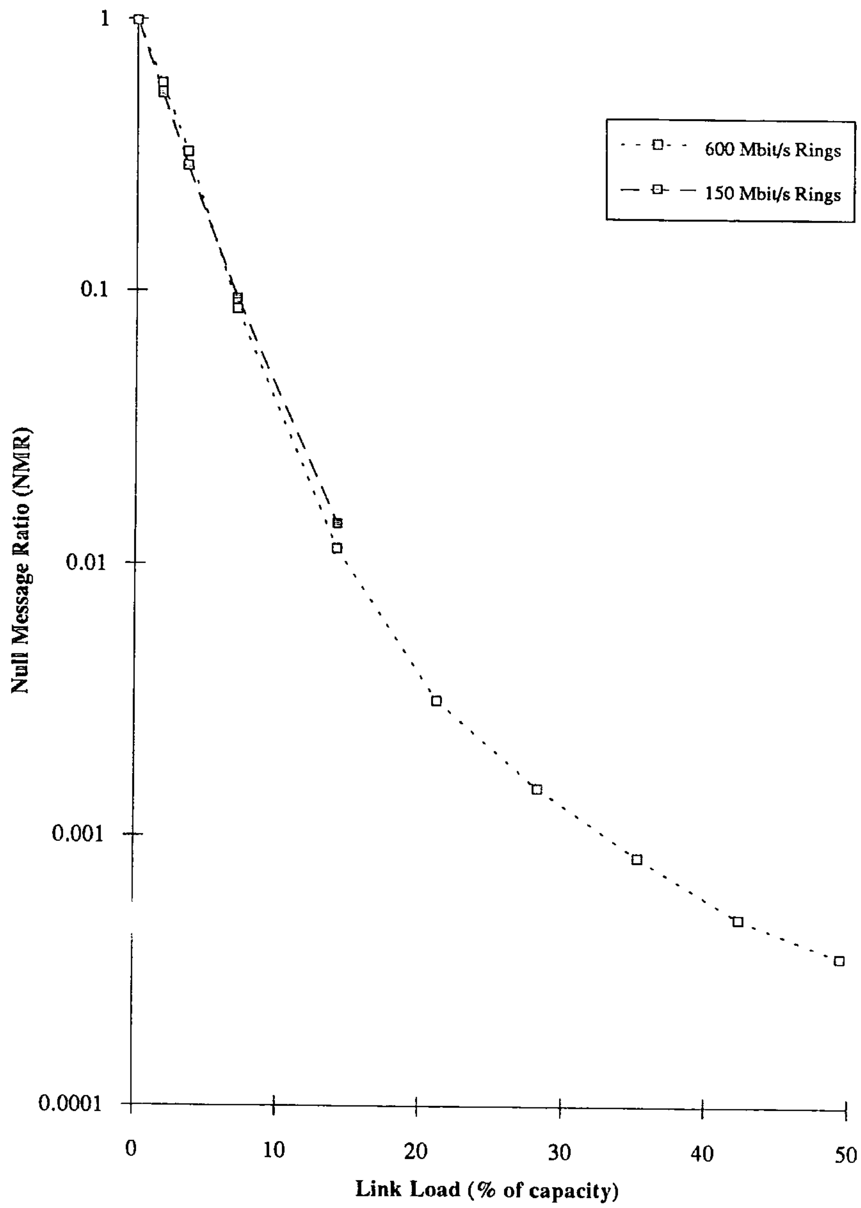


Figure 6.7: NULL-message ratio (NMR) as a function of load. As might be expected, the ratio is largely independent of the ring speed.

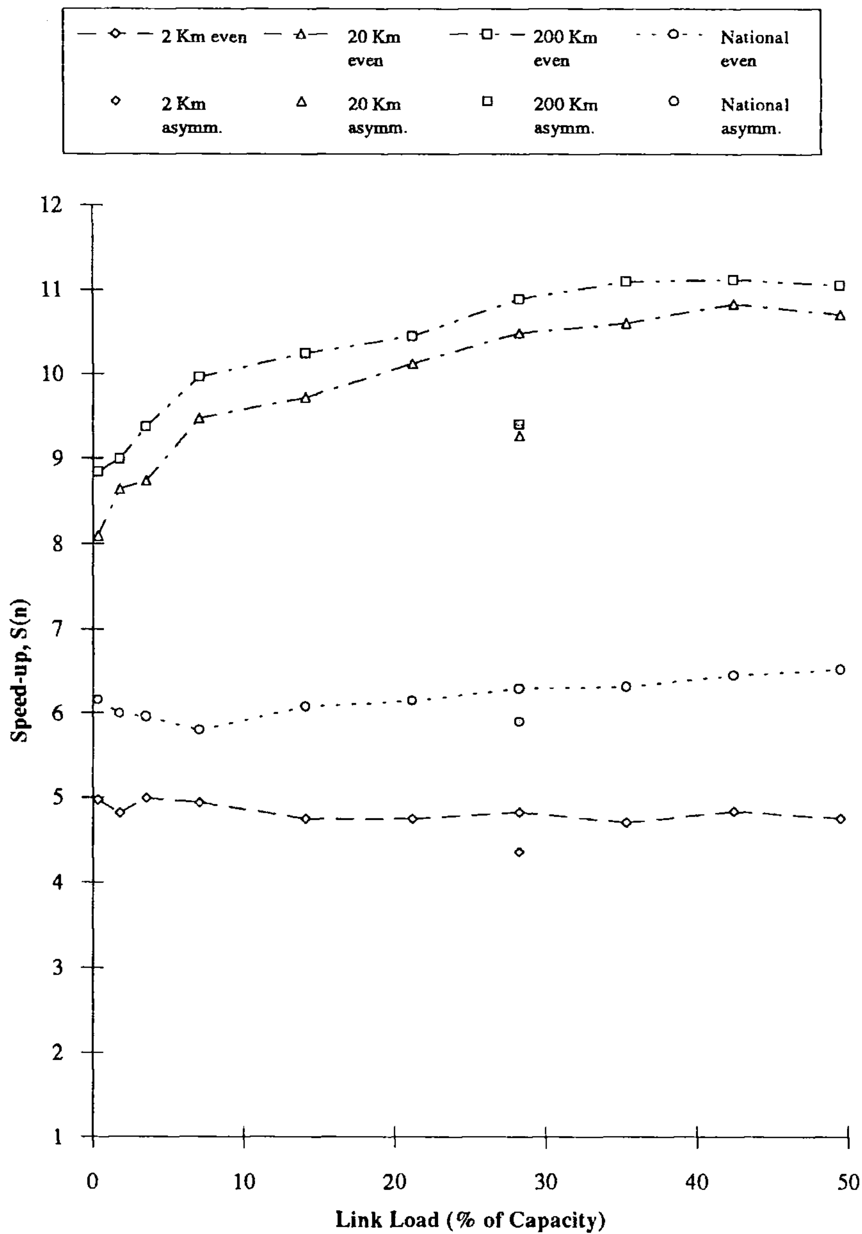


Figure 6.8: Speed-up (optimized) as a function of load for a range of lookahead values and symmetric and asymmetric traffic patterns.

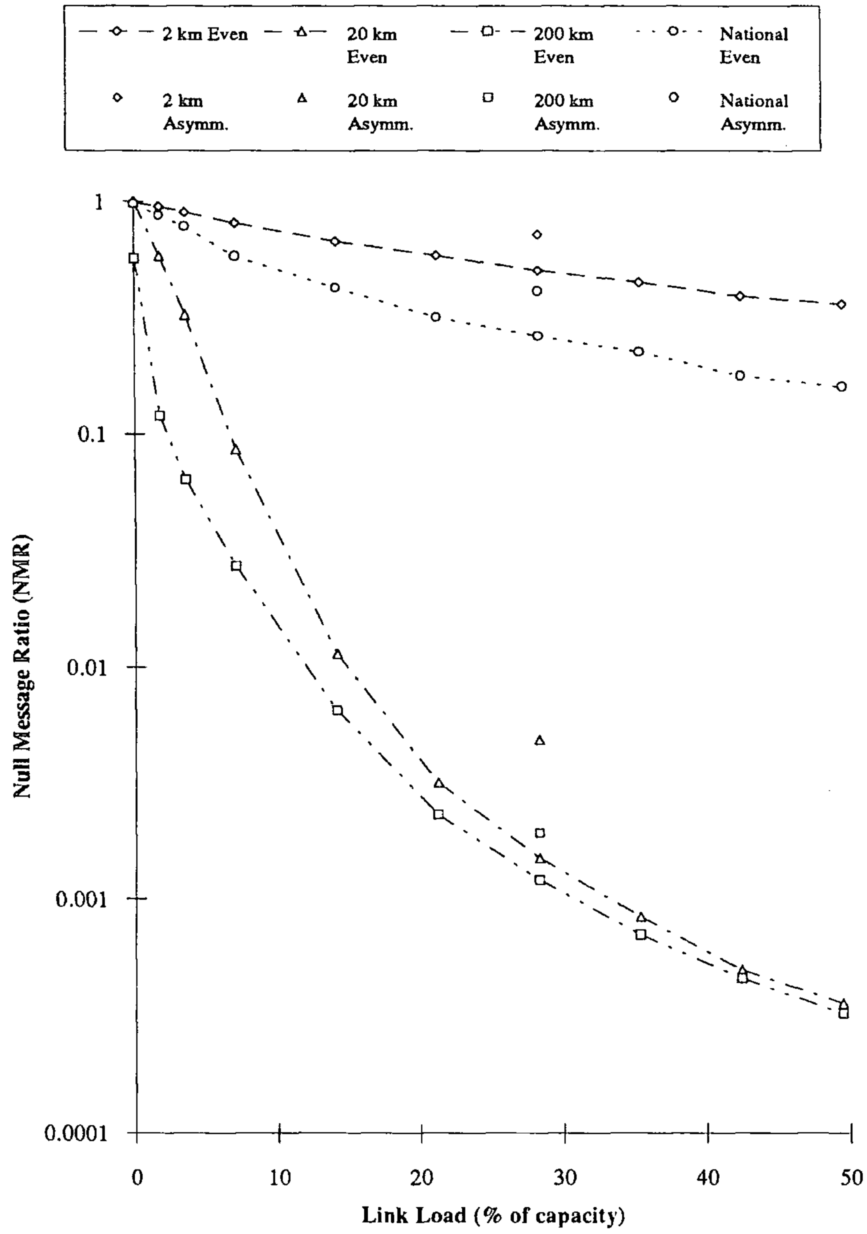


Figure 6.9: NULL-message ratio (NMR) as a function of load for a range of lookahead values and symmetric and asymmetric traffic patterns.

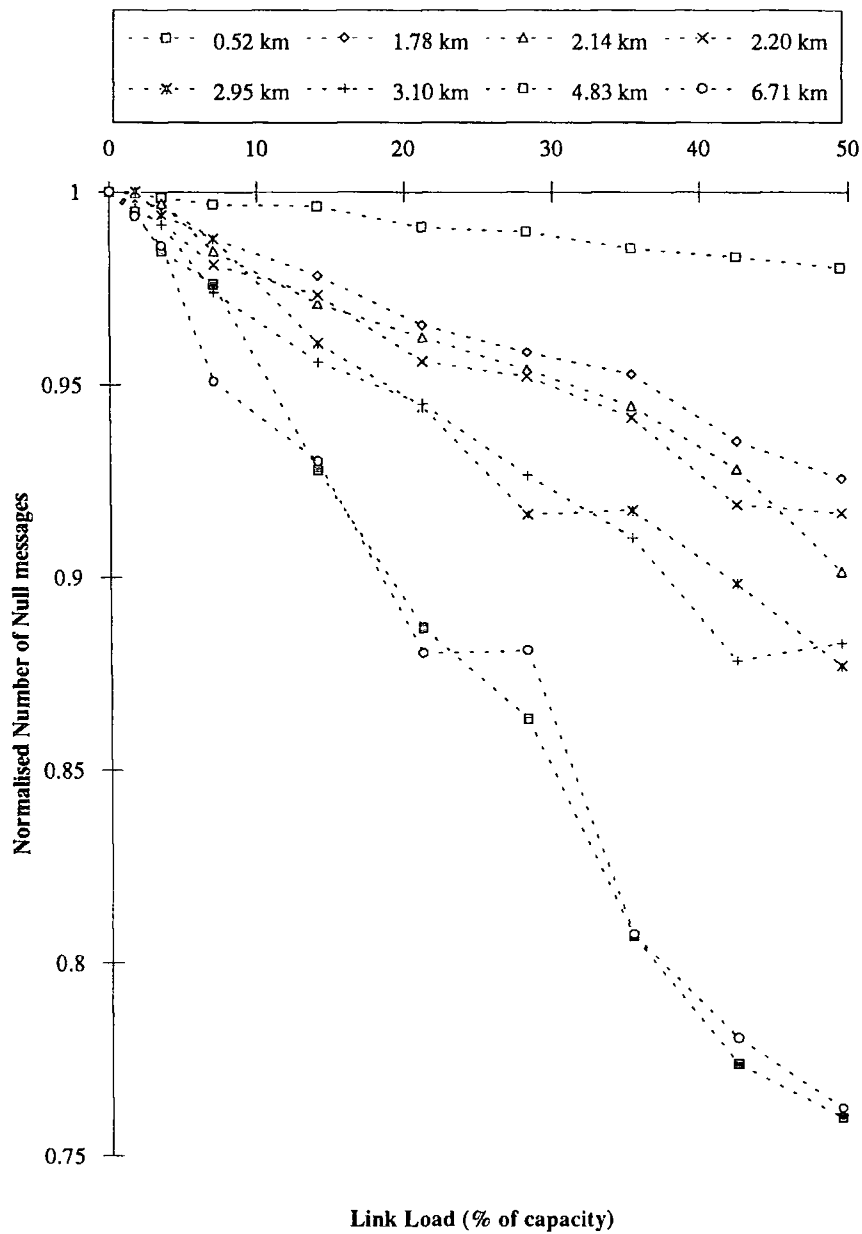


Figure 6.10: Number of NULL-messages normalised against the number at the lowest load (0.05 calls/source/s) plotted against link load for a selection of source-switch links.

Chapter 7

Conclusions and Further Work

7.1 Conclusions

In this thesis, we have considered the application of parallel simulation to the performance modelling of telecommunication networks. Parallel simulation is a rapidly growing area of research, with significant potential for increasing the size and complexity of models which can be simulated in a reasonable amount of time. It is also hoped that the reader has gained an insight into the practical difficulties of implementing a parallel simulator as well as the advantages in run-time performance.

The work described here has particularly focussed on the problem of synchronization; and this will continue to be an interesting area of study. A number of different approaches have been shown to work, albeit under varying circumstances and with varying degrees of success. The review highlighted the potential of the parallelizing compiler approach for simple models and the distributed model components approach for true parallel discrete event simulation. Both of these techniques have been explored.

The parallelizing compiler approach has a number of attractions. Firstly, it can be used like any other compiler; ignoring the machine architecture and any involvement with parallel programming. If an acceptable level of speed-up for an application can be obtained with such a cursory approach then this is most attractive; particularly to those with no real interest in learning about parallelism but with a real need for improved performance. The approach is also attractive for improving the performance of so-called "dusty decks"; old, tried and tested programs, often written in an obsolete language version, still in use; but, not enough to justify a complete re-write to improve the performance, even in a modern sequential

language. However, such applications very often use efficient sequential algorithms which usually do not (by definition) contain much scope for the compiler to extract parallelism.

For these reasons, the user should make use of execution profiling tools and any intermediate output of the compiler analysing the parallelism extracted. An appreciation of how the parallelizing compiler resolves dependencies by modifying or re-arranging the code is very useful. The parts to look for are where the compiler has failed to extract parallelism and the code is unchanged from it's original sequential execution. Often, quite small changes made in the source code can enable large portions of code to be parallelized. This is often described as "fighting the compiler's hooks"; an experience not unknown in conventional programming when attempting to optimize code for execution. Some knowledge of the machine architecture and, more particularly, the run-time model of the architecture, is also useful in giving insights into possible improvements in the source code which are often very easy to implement. More complex improvements may require the replacement of sequential with parallel algorithms or the re-writing of some parts of the application directly in the parallel language.

Thus, the user can choose to get involved at several levels of knowledge in order to improve the performance of the application. There is obviously a potential trade-off possible between time and effort expended and performance achieved. However, depending on the size and characteristics of the application, the law of diminishing returns will mean that an increasingly large amount of effort is expended to achieve a decreasingly small improvement. It is interesting that many of these conclusions were also arrived at by Thomborson [193] in a recent article on porting programs to supercomputers.

The work on parallel discrete event simulation reported here confirms results obtained by parallel simulation practitioners in similar fields using the distributed model components approach to model decomposition. The work on queueing networks using YADDES showed that conservative Chandy-Misra-Bryant synchronization gives good speed-up for models with good lookahead and high traffic loads. Conversely, poor lookahead and/or low traffic loads result in relatively poor performance. The results for optimistic virtual time synchronization were more modest, but were also more consistent, being relatively insensitive to the traffic load. Optimistic synchronization was easily the best performer for lower traffic loads only losing out to conservative Chandy-Misra-Bryant synchronization with NULL-message cancellation at the highest load.

The optimistic synchronization algorithm could use lazy or aggressive cancellation strat-

egies. Overall, with the closed stochastic queueing network models, there was little to choose between their performance except in the area of memory usage, where lazy cancellation used less, particularly at higher values of checkpoint interval. The checkpoint interval was also found to have an optimal value in terms of reducing memory usage and increasing speed-up as predicted by other parallel simulation practitioners.

The speed-up results for the asynchronous transfer mode network simulations using the conservative synchronization approach were reasonable throughout the series of experiments, the speed-up for the production runs being particularly good. The production runs again show that Chandy-Misra-Bryant message passing using NULL-messages and exploiting lookahead is very effective at higher traffic loads. Reduced lookahead, asymmetric traffic and asymmetric lookahead all caused reductions in the speed-up; asymmetric lookahead having the most impact as expected from the results on varying lookahead evenly. Given these results, it seems unlikely that using an optimistic synchronization approach would lead to improved speed-up.

Conservative methods thus offer good potential for certain classes of problems. Significant successes have also been obtained particularly when application-specific knowledge is applied to maximise the efficiency of the simulation mechanism. Conservative methods may well find success in packaged simulation systems (e.g. logic simulators and, possibly communication network simulators) in which the simulation code is optimized for the synchronization algorithm and users only configure the provided simulation modules into specific systems.

Which strategy should one use for a particular simulation problem? If state-saving overheads do not dominate, time warp has a good chance of success, assuming of course that the problem contains a reasonable degree of parallelism. If the application has good lookahead properties, conservative mechanisms may also perform well. If the application has both poor lookahead and large state-saving overheads, all existing parallel discrete event approaches will have trouble obtaining good performance even if the application contains copious amounts of parallelism. However, it is hoped that time warp aided with hardware support for state-saving will provide a viable solution for this situation in the future.

7.2 Further Work

The parallelizing compiler approach still has, potentially, a good future; albeit largely driven by application areas other than simulation. There are continuing improvements being made in parallelizing compilers and they are becoming almost mandatory on new massively parallel machines. Such machines hold the promise of simulating very large circuit-switched networks with hundreds or even thousands of switches; modelling the whole, or at least substantial portions, of national and international networks in reasonable amounts of time. The question which needs to be addressed is whether the speed-up will scale as the model size, the number of processors, and the level of model detail are increased.

A particularly interesting area, which has yet to be explored, is the application of a parallelizing compiler to the time parallel simulation models discussed in chapter 2. Time parallel models based on recurrence relations are currently under investigation for queueing networks and circuit-switched telecommunication networks. Another interesting comparison would be to compare the speed-up, obtained using a parallelizing compiler, with that obtained from a parallel discrete event simulation; executing the same simulation models, using a similar depth of model construction, but synchronized using a conservative or optimistic approach.

Much work could be done in the area of asynchronous transfer mode network in terms of investigating larger networks and different switch and traffic models. Also, the simulation models here were performed at the cell-level; that is, in relatively fine detail. Work could also be done looking at simpler models at the burst- or call-levels in order to speed-up execution times and allow more complex networks to be investigated. Indeed, work has already been done by Pitts et. al. [176–179] at Queen Mary and Westfield College, London, on using burst-level, or cell rate, simulation techniques.

An important application area that has not yet been adequately addressed by either optimistic or conservative simulation mechanisms is real-time applications. Theories of performances are not sufficiently developed to address this question, though significant progress has been made particularly by Ghosh et. al. [166]. This may well be an important issue in the future where high performance simulator/emulator packages are required to drive new switch architectures with simulated traffic and also to investigate network management strategies in future networks.

Most applications of parallel simulation reported thus far in research papers and confer-

ence proceedings tend to be simplified benchmarks. Therefore, they are not commercially motivated and the results of the simulation are not necessarily required by anyone. Parallel simulation will only prove itself on serious applications which need to be simulated and are only feasible in terms of of both time and cost on a parallel simulator; due to to time, size, complexity or all three. Telecommunication networks offers such grand challenges in abundance; eg. very large circuit-switched trunk networks, asynchronous transfer mode networks, high speed local and metropolitan area networks, heterogeneous internets and intelligent networks.

If the practice of parallel simulation is to become more widespread, most of the difficult details of synchronization must be embedded within a parallel simulation environment where they remain hidden from the simulation modeller. It seems that the critical problems for parallel simulation lie more in its automation. The important future work in synchronization protocol design lies in developing protocols whose application is transparent to a wide variety of simulation models, and whose overheads are minimal. This potential lies more with optimistic approaches such as time warp.

The vast majority of parallel simulators currently run on what are perceived by many to be specialist hardware platforms; either due to their cost, complexity or sheer novelty. Little work has been done to address the use of local area networks (LANs) of workstations and/or personal computers for parallel simulation. Though, there is no doubt, that the performances will not be as good, and the novelty will not be as great, the vast majority of potential users of parallel simulation will not consider buying specialist parallel hardware for a single, or small number, of applications. It is interesting that the Jade time warp simulation package, which is the only one commercially available at present, is now available for networks of workstations as well as a transputer based parallel computers. Unfortunately, there are no speed-up figures available for the workstation version as yet. Also, the recent work of Chai and Ghosh [183] at Brown Univerity, USA. on the distributed simulation of ATM networks on a LAN of standard workstations, has shown that good performance on such platforms for serious applications is possible.

Finally, perhaps the most challenging problem remaining to be explored is the application of these techniques, beyond the realm of discrete event simulation, and into the world of general purpose parallel computation. The degree to which the techniques described here can be applied to parallelizing arbitrary computations is only just beginning to be explored.

Bibliography

[1] S. Agrawal, E. Matalon, and R. Ramaswany, "BEST/1-SNA: An analytic tool for computer aided modelling of SNA MSNF networks," in *Proceedings of the IEEE International Conferences on Communications*, pp. 37.6.1–37.6.6, 1987.

[2] C. H. Sauer, E. A. McNair, and J. F. Kurose, "Queueing network simulation of computer communication," *IEEE Transactions on Selected Areas in Communications*, vol. 2, pp. 203–220, January 1984.

[3] J. F. Kurose and H. T. Mouftah, "Computer aided modeling, analysis, and design of communication networks," *IEEE Transactions on Selected Areas in Communications*, vol. 6, pp. 130–145, January 1988.

[4] V. S. Frost, W. W. Larue Jr., and K. S. Shanmugan, "Efficient techniques for the simulation of computer communications networks," *IEEE Transactions on Selected Areas in Communications*, vol. 6, pp. 146–157, January 1988.

[5] L. Schruben, "Detecting initialization bias in simulation output," *Operations Research*, vol. 30, pp. 569–590, 1982.

[6] P. Welch, "The statistical analysis of simulation results," in *The Performance Modeling Handbook* (S. Lavenberg, ed.), Academic Press, 1983.

[7] P. Heidelberger and P. D. Welch, "Simulation run control in the presence of an initial transient," *Operations Research*, vol. 31, pp. 1109–1144, 1983.

[8] A. M. Law, "Statistical analysis of simulation output data," *Operations Research*, vol. 31, pp. 983–1029, 1983.

[9] M. Crane and A. Lemoine, *An introduction to the regenerative method for simulation analysis.* Springer-Verlag, 1977.

[10] P. Heidelberger and P. D. Welch, "A spectral method for confidence interval generation and run length control in simulation," *Communications of the ACM*, vol. 24, pp. 223-245, 1981.

[11] R. Righter and J. C. Walrand, "Distributed simulation of discrete event systems," *Proceedings of the IEEE*, vol. 77, pp. 99–113, January 1989.

[12] D. P. Helmbold and C. E. McDowell, "Modeling speedup (n) greater than n," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 250–256, April 1990.

[13] J. G. Shanthikumar and R. G. Sargent, "A unifying view of hybrid simulation/analytic models and modeling," *Operations Research*, vol. 31, no. 6, pp. 1030–1052, 1983.

[14] C. McGeoch, "Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups," *ACM Computing Surveys*, vol. 24, no. 2, pp. 195–212, 1992.

[15] P. Heidelberger, "Statistical analysis of parallel simulations," in *Proceedings of the Winter Simulation Conference*, pp. 290–295, 1986.

[16] G. M. Amdahl, "Limits of expectation," *International Journal of Supercomputer Applications*, vol. 2, no. 1, pp. 88–97, 1988.

[17] R. Duncan, "A survey of parallel computer architectures," *IEEE Computer*, pp. 5–16, February 1990.

[18] M. J. Flynn, "Very high speed computing systems," *Proceedings of the IEEE*, vol. 54, pp. 1901–1909, December 1966.

[19] T. Feng, "A survey of interconnection networks," *IEEE Computer*, vol. 14, pp. 12–27, December 1981.

[20] H. J. Siegel, T. Schwederski, D. G. Meyer, and W. Tsun yuk Hsu, "Large-scale parallel processing systems," *Microprocessors and Microsystems*, vol. 11, no. 1, pp. 3–19, 1987.

[21] G. S. Almasi and A. J. Gottlieb, *Highly parallel computing.* Benjamin/Cummings, 1989.

[22] D. A. Patterson and J. L. Hennessey, *Computer architecture: A quantitative approach.* Morgan Kaufmann, 1990.

[23] A. Trew and G. Wilson, eds., *Past, present and Parallel: A survey of available parallel computing systems*. Springer-Verlag, 1991.

[24] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, pp. 30–53, October 1990.

[25] U. Bannerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceeding of the IEEE*, vol. 81, pp. 211–243, February 1993.

[26] A. Chandak and J. C. Browne, "Vectorization of discrete event simulation," in *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 359–361, 1983.

[27] D. A. Reed, "Parallel discrete event simulation: A case study," in *Proceedings of the 18th IEEE Annual Simulation Symposium*, pp. 95–107, 1985.

[28] Y. Escaig and W. Oed, "Analysis tools for micro- and autotasking programs on CRAY multiprocessor systems," *Parallel Computing*, vol. 17, pp. 1425–1433, 1991.

[29] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," in *Proceedings of the First International Conference on Supercomputing*, Springer-Verlag, June 1987.

[30] S. Tjang, M. E. Wolf, M. S. Lam, K. L. Pieper, and J. L. Hennessey, "Integrating scalar optimization and parallelization," in *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[31] R. J. Carpentar, "Performance measurement instrumentation for multiprocessor systems," *High Performance Computer Systems*, pp. 81–92, 1987.

[32] T. Kerola and H. Schwetman, "Monit: A performance monitoring tool for parallel and pseudo-parallel programs," in *Proceedings of the ACM SIGMETRICS Conference*, May 1987.

[33] B. P. Miller and C. Q. Yang, "IPS: An interactive and automatic performance measurement tool for parallel and distributed programs," in *Proceedings of the Seventh International Conference on Distributed Computing Systems*, September 1987.

[34] H. Burkhart and R. Millen, "Performance measurement tools in a multiprocessor environment," *IEEE Transactions on Computers*, vol. 38, pp. 725–737, May 1989.

[35] T. E. Anderson and E. D. Lazowska, "Quartz: A tool for tuning parallel program performance," in *Proceedings of the ACM SIGMETRICS Conference*, pp. 115–125, May 1990.

[36] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren, "The ParaScope parallel programming environment," *Proceeding of the IEEE*, vol. 81, pp. 244–263, February 1993.

[37] Parasoft, *Programming parallel computers using the Express system*, 1990.

[38] K. Ikudome, G. C. Fox, A. Kolawa, and J. W. Flower, *An automatic and symbolic parallelization system for distributed memory parallel computers*. Parasoft, 1990.

[39] S. Ahuja, N. Carriero, and D. Gelertner, "Linda and friends," *IEEE Computer*, August 1986.

[40] D. Gelertner, *Proceedings on Parallel Architectures and Languages in Europe*, vol. 2 of *Lecture Notes in Computer Science 366*, ch. Multiple tuple spaces in Linda. Springer-Verlag, June 1989.

[41] Perihelion Software Ltd., *The Helios operating system*. Prentice-Hall, 1989.

[42] D. L. Wyatt, "Simulation programming on a distributed system: A preprocessor approach," in *Proceedings of the SCS Conference on Distributed Simulation*, pp. 32–36, 1985.

[43] M. P. Papazoglou, P. I. Georgiadis, and D. G. Maritsas, "Designing a parallel Simula machine," *Computer design*, pp. 125–132, October 1983.

[44] P. Civera, G. Piccinini, and M. Zamboni, "Implementation studies for a VLSI Prolog Co-processor," *IEEE Micro*, pp. 10–23, February 1989.

[45] R. Rajagopal and J. C. Comfort, "Contrasting distributed simulation with parallel replication: a case study of a queueing simulation with a network of transputers," in *Proceedings of the Winter Simulation Conference*, pp. 746–755, December 1989.

[46] W. E. Biles, D. M. Daniels, and T. J. O'Donnell, "Statistical considerations in simulation on a network of microcomputers," in *Proceedings of the Winter Simulation Conference*, pp. 388–392, December 1985.

[47] D. L. Wyatt, S. V. Sheppard, and R. E. Young, "An experiment in microprocessor-based digital simulation," in *Proceedings of the Winter Simulation Conference*, pp. 271–277, December 1983.

[48] D. L. Wyatt and S. V. Sheppard, "A language directed distributed discrete simulation system," in *Proceedings of the Winter Simulation Conference*, pp. 463–464, December 1984.

[49] M. Krishnamurthi, U. Chandrasekaran, and S. V. Sheppard, "Two approaches to the implementation of a distributed simulation system," in *Proceedings of the Winter Simulation Conference*, pp. 435–444, December 1985.

[50] R. M. Reese, "A software development environment for distributed simulation," in *Proceedings of the SCS Conference on Distributed Simulation*, pp. 37–40, 1985.

[51] S. V. Sheppard and C. K. Davis, "Parallel simulation environments for multiprocessor architectures," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, pp. 109–114, July 1988.

[52] C. K. Davis, S. V. Sheppard, and W. M. Lively, "Automatic development of parallel simulation models in Ada," in *Proceedings of the Winter Simulation Conference*, pp. 339–343, December 1988.

[53] J. C. Comfort, "The design of a multi-microprocessor based simulation computer I," in *Proceedings of the IEEE Annual Simulation Symposium*, pp. 45–52, 1982.

[54] J. C. Comfort, "The design of a multi-microprocessor based simulation computer II," in *Proceedings of the IEEE Annual Simulation Symposium*, pp. 197–209, 1983.

[55] J. C. Comfort, "The simulation of a master-slave event set processor," *Simulation*, vol. 42, pp. 117–124, March 1984.

[56] J. C. Comfort and R. Rajagopal, "Environment partitioned distributed simulation with transputers," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, pp. 103–108, July 1988.

[57] A. I. Concepcion, "A hierarchical computer architecture for distributed simulation," *ACM Transactions on Computers*, vol. 38, pp. 311–319, February 1989.

[58] G. Zhang and B. P. Zeigler, "DEVS-Scheme supported mapping of hierarchical model onto multiple processors systems," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 64–69, 1989.

[59] D. W. Jones, "Concurrent simulation: An alternative to distributed simulation," in *Proceedings of the Winter Simulation Conference*, pp. 417–423, December 1986.

[60] B. A. Cota and R. G. Sargent, "An algorithm for parallel discrete event simulation using common memory," in *Proceedings of the 22nd IEEE Annual Simulation Symposium*, pp. 23–31, 1989.

[61] C. Hughes, U. Chandra, and S. V. Sheppard, "Two implementations of a concurrent simulation environment," in *Proceedings of the Winter Simulation Conference*, pp. 618–623, December 1987.

[62] D. M. Nicol, "Mapping a battlefield simulation onto message-passing parallel architectures," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, pp. 141–146, July 1988.

[63] R. T. Clarke, S. J. Nichols, and P. Mars, "Transputer-based simulation tool for performance evaluation of wide area telecommunications networks," *Microprocessors and Microsystems*, vol. 13, no. 3, pp. 173–178, 1989.

[64] S. J. Nichols, R. T. Clarke, and P. Mars, "Design of a high speed simulation tool for WAN using parallel processing," *Microprocessing and Microprogramming*, vol. 25, pp. 327–332, 1989.

[65] S. J. Nichols, *Simulation and analysis of adaptive routing and flow control in wide area communication networks*. PhD thesis, University of Durham, School of Engineering and Applied Science, 1990.

[66] R. W. Earnshaw and P. Mars, "Simulation of ATM networks on Transputer arrays," in *Seventh UK IEE Teletraffic Symposium*, 1990.

[67] K. M. Chandy and R. Sherman, "Space-time, and simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, pp. 53–57, March 1989.

[68] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani, "Algorithms for unboundedly parallel simulations," *ACM Transactions on Computer Systems*, vol. 9, no. 3, pp. 201–221, 1991.

[69] H. Ammar and S. Deng, "Time warp simulation using time scale decomposition," in *Proceedings of the SCS Multiconference on Advances on Parallel and Distributed Simulation*, vol. 23, pp. 11–24, January 1991.

[70] Y-B. Lin and E. D. Lazowska, "A time-division algorithm for parallel simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 73–83, January 1991.

[71] P. Heidelberger and H. Stone, "Parallel trace-driven cache simulation by time partitioning," Technical Report RG 15500, IBM Research, February 1990.

[72] D. M. Nicol, A. G. Greenberg, B. D. Lubachevsky, and S. Roy, "Massively parallel algorithms for trace-driven cache simulation," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, vol. 24, pp. 3–11, January 1992.

[73] Y-B. Lin, "Parallel trace-driven simulation of packet-switched multiplexer under priority scheduling policy," *Information Processing Letters*, vol. 77, pp. 197–201, September 1993.

[74] R. Baccelli and M. Canales, "Parallel simulation of stochastic petri nets using recurrence equations," in *Proceedings of the 1992 ACM SIGMETRICS Conference*, pp. 257–258, June 1992.

[75] B. Gaujal, A. G. Greenberg, and D. M. Nicol, "A sweep algorithm for massively parallel simulation of circuit-switched networks," Technical Report 92-30, ICASE, July 1992. To appear in the Journal of Parallel and Distributed Computing.

[76] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys*, vol. 18, pp. 39–65, March 1986.

[77] P. F. Reynolds Jr., "A spectrum of options for parallel simulation," in *Proceedings of the Winter Simulation Conference*, pp. 325–332, December 1988.

[78] R. M. Fujimoto and D. Nicol, "State of the art in parallel simulation," in *Proceedings of the Winter Simulation Conference*, pp. 246–254, December 1992.

[79] R. M. Fujimoto and D. Nicol, "Parallel simulation today." Presented as notes accompanying a tutorial session at the 7th Workshop on Parallel and Distributed Simulation, May 1993.

[80] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[81] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions in Software Engineering*, vol. SE–5, no. 5, pp. 440–452, 1979.

[82] K. M. Chandy, V. Holmes, and J. Misra, "Distributed simulation of networks," *Computer Networks*, vol. 3, pp. 105–113, 1979.

[83] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, pp. 198–206, April 1981.

[84] R. E. Bryant, "Simulation of packet communication architecture computer systems," Master's thesis, MIT, Computer Science Laboratories, 1977.

[85] J. K. Peacock, J. W. Wong, and E. Manning, "Distributed simulation using a network of processors," *Computer Networks*, vol. 3, pp. 44–56, 1979.

[86] J. K. Peacock, E. Manning, and J. W. Wong, "Synchronization of distributed simulation using broadcast algorithms," *Computer Networks*, vol. 4, pp. 3–10, 1980.

[87] R. W. Earnshaw, *Simulation of packet- and cell-based communication networks.* PhD thesis, University of Durham, School of Engineering and Computer Science, 1992.

[88] S. Manthorpe, C. I. Phillips, and L. G. Cuthbert, "High performance ATM network simulation using transputers," in *IEE Colloquium on Parallel Processing: Industrial and Scientific Applications*, June 1990.

[89] C. I. Phillips and L. G. Cuthbert, "Concurrent discrete event-driven simulation tools," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 477–485, April 1991.

[90] D. A. Reed and A. D. Maloney, "Parallel discrete event simulation: The Chandy-Misra approach," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, pp. 8–13, July 1988.

[91] W. L. Bain and D. S. Scott, "An algorithm for time synchronization in distributed discrete event simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, pp. 30–33, July 1988.

[92] W. K. Su and C. L. Seitz, "Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, pp. 38–43, March 1989.

[93] P. F. Reynolds Jr., "A shared resource algorithm for distributed simulation," in *Proceedings of the 9th IEEE Annual Symposium on Computer Architecture*, pp. 259–266, 1982.

[94] D. M. Nicol and P. F. Reynolds Jr., "Problem oriented protocol design," in *Proceedings of the Winter Simulation Conference*, pp. 471–474, December 1984.

[95] J. Misra, "Detecting termination of distributed computations using markers," in *Proceedings of the 2nd ACM Conference on Principles of Distributed Computing*, pp. 290–293, 1983.

[96] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, pp. 1–4, August 1980.

[97] B. Groselj and C. Tropper, "A deadlock resolution scheme for distributed simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, pp. 108–112, March 1989.

[98] L. Z. Liu and C. Tropper, "Local deadlock detection in distributed simulations," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 64–69, January 1990.

[99] D. Kumar, "An approximate method to predict performance of a distributed simulation scheme," in *Proceedings of International Conference on Parallel Processing*, vol. 3, pp. 259–262, August 1989.

[100] Y-B. Lin, E. D. Lazowska, and J-L. Baer, "Conservative parallel simulation for systems with no lookahead prediction," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 144–149, January 1990.

[101] R. M. Fujimoto, "Lookahead in parallel discrete-event simulation," in *Proceedings of the IEEE International Conference on ParallelProcessing*, vol. 3, pp. 34–41, 1988.

[102] R. M. Fujimoto, "Performance measurements of distributed simulation strategies," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, pp. 14–20, July 1988.

[103] D. A. Reed, A. D. Malony, and B. D. McCredie, "Parallel discrete event simulation using shared memory," *IEEE Transactions on Software Engineering*, vol. 14, pp. 541–553, April 1988.

[104] R. M. Fujimoto, "Performance measures of distributed simulation strategies," *Transactions of the Society for Computer Simulation*, vol. 6, pp. 89–132, April 1989.

[105] D. M. Nicol, "Parallel discrete-event simulation of FCFS stochastic queueing networks," *SIGPLAN Notes*, vol. 23, pp. 124–137, September 1988.

[106] K. M. Chandy and R. Sherman, "The conditional event approach to distributed simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 93–99, 1989.

[107] W. Cai and S. J. Turner, "An algorithm for distributed discrete-event simulation the "carrier null message" approach," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 3–8, January 1990.

[108] B. R. Preiss, W. M. Loucks, I. D. MacIntyre, and J. A. Field, "Null message cancellation in conservative distributed simulation," in *Proceedings of the SCS Multiconference on Advances on Parallel and Distributed Simulation*, vol. 23, pp. 33–38, January 1991.

[109] L. Gould, I. Bowler, and A. Purvis, "Real-time, multi-channel digital filtering on the transputer," in *Proceedings of the International Symposium on Computer Architecture and Digital Signal Processing*, 1989.

[110] D. B. Wagner, E. D. Lazowska, and B. N. Bershad, "Techniques for efficient shared-memory parallel simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 29–37, 1989.

[111] D. B. Wagner and E. D. Lazowska, "Parallel simulation of queueing networks: Limitations and potential," in *Proceedings of ACM SIGMETRICS and Performance '89*, vol. 17, pp. 146–155, May 1989.

[112] Y-B. Lin and E. D. Lazowska, "Exploiting lookahead in parallel simulation," Technical Report 89-10-06, University of Washington, Department of Computer Science, University of Washington, Seattle, Washington, 1989.

[113] W. M. Loucks and B. R. Preiss, "The role of knowledge in distributed simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 144–149, January 1990.

[114] R. Ayani, "A parallel simulation scheme based on the distance between objects," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, pp. 113–118, March 1989.

[115] B. C. Merrifield, S. B. Richardson, and J. B. G. Roberts, "Quantitative studies of discrete event simulation modelling road traffic," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 188–193, January 1990.

[116] R. L. Bagrodia and W-T. Liao, "Maisie: A language and optimizing environment for distributed simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 205–210, January 1990.

[117] R. L. Bagrodia and W-T. Liao, "A language for iterative design of efficient simulations," Technical Report UCLA-CSD-920044, University of California at Los Angeles, Department of Computer Science, UCLA, Los Angeles, California, 1992.

[118] B. D. Lubachevsky, "Efficient distributed event-driven simulations of multiple-loop networks," *Communications of the ACM*, vol. 32, pp. 111–123, January 1989.

[119] D. Nicol and S. Roy, "Parallel simulation of timed petri nets," in *Proceedings of the Winter Simulation Conference*, pp. 574–583, December 1991.

[120] J. Steinman, "Speedes: synchronous parallel environment for emulation and discrete event simulation," in *Proceedings of the SCS Multiconference on Advances on Parallel and Distributed Simulation*, vol. 23, pp. 95–103, January 1991.

[121] B. D. Lubachevsky, "Scalability of the bounded lag distributed discrete event simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, pp. 100–107, March 1989.

[122] B. Berkman and R. Ayani, "Parallel simulation of multistage interconnection networks on a SIMD computer," in *Proceedings of the SCS Multiconference on Advances on Parallel and Distributed Simulation*, vol. 23, pp. 133–140, January 1991.

[123] S. Bellenot, "Global virtual time algorithms," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 122–127, January 1990.

[124] Y-B. Lin and E. D. Lazowska, "Determining the global virtual time in a distributed simulation," Technical Report 90-01-02, University of Washington, Department of Computer Science, University of Washington, Seattle, Washington, 1989.

[125] B. R. Preiss, "The Yaddes distributed discrete event simulation specification language and execution environments," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, pp. 139–144, March 1989.

[126] A. Gafni, "Rollback mechanisms for optimistic distributed simulation systems," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, pp. 61–67, July 1988.

[127] O. Berry, *Performance evaluation of the time warp distributed simulation mechanism.* PhD thesis, University of Southern California, Department of Computer Science, May 1986.

[128] T. K. Som, B. A. Cota, and R. G. Sargent, "On analyzing events to estimate the possible speedup of parallel discrete event simulations," in *Proceedings of the Winter Simulation Conference*, pp. 729–737, December 1989.

[129] D. West, "Optimizing time warp lazy rollback and lazy re-evaluation," master's thesis, University of Calgary, Department of Computer Science, January 1988.

[130] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Di Loreto, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, and H. Younger, "The time warp operating system," in *The 11th Symposium on Operating System Principles*, vol. 21, November 1987.

[131] R. M. Fujimoto, "Time warp on a shared memory multiprocessor," in *Proceedings of the IEEE International Conference on Parallel Processing*, 1989.

[132] B. D. Lubachevsky, A. Weiss, and A. Shwartz, "An analysis of rollback-based simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 154–193, April 1991.

[133] V. Madisetti, J. Walrand, and D. Messerschmitt, "Wolf: a rollback algorithm for optimistic distributed simulation systems," in *Proceedings of the Winter Simulation Conference*, pp. 296–305, December 1988.

[134] V. Madisetti, D. Hardaker, and R. M. Fujimoto, "The MIMDIX operating system for parallel simulation," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, vol. 24, pp. 65–74, January 1992.

[135] D. M. Nicol, "Global synchronization for optimistic parallel discrete event simulation," in *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, pp. 27–34, May 1993.

[136] L. M. Sokol and B. K. Stucky, "MTW: experimental results for a constrained optimistic scheduling paradigm," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 169–173, January 1990.

[137] S. Turner and M. Xu, "Performance evaluation of the bounded time warp algorithm," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, vol. 24, pp. 117–128, January 1992.

[138] D. Ball and S. Hoyt, "The adaptive time warp concurrency control algorithm," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 174–177, January 1990.

[139] B. D. Lubachevsky, A. Shwartz, and A. Weiss, "Rollback sometimes works ...if filtered," in *Proceedings of the Winter Simulation Conference*, pp. 630–639, December 1989.

[140] P. F. Reynolds Jr, "An efficient framework for parallel simulations," in *Proceedings of the SCS Multiconference on Advances on Parallel and Distributed Simulation*, vol. 23, pp. 167–174, January 1991.

[141] J. V. Briner Jr., "Fast parallel simulation of digital systems," in *Proceedings of the SCS Multiconference on Advances on Parallel and Distributed Simulation*, vol. 23, pp. 71–77, January 1991.

[142] Y-B. Lin and E. D. Lazowska, "Reducing the state saving overhead for time warp parallel simulation," Technical Report 90-02-03, University of Washington, Department of Computer Science, University of Washington, Seattle, Washington, February 1990.

[143] B. R. Preiss, I. D. MacIntyre, and W. M. Loucks, "On the trade-off between time and space in optimistic parallel discrete-event simulation," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, vol. 24, pp. 33–42, January 1992.

[144] S. Bellenot, "State skipping performance with the time warp operating system," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, vol. 24, pp. 53–64, January 1992.

[145] Y-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska, "Selecting the checkpoint interval in time warp," in *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, vol. 23, pp. 3–10, May 1993.

[146] A. C. Palaniswamy and P. A. Wilsey, "An analytical comparison of periodic checkpointing and incremental state saving," in *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, vol. 23, pp. 127–134, May 1993.

[147] D. R. Jefferson, "Virtual time II: Storage management in distributed simulation," in *Proceedings of the Ninth Annual ACM Symposium on Princliples of Distributed Computing*, pp. 75–89, August 1990.

[148] Y-B. Lin, "Memory management algorithms for optimistic parallel simulation," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, vol. 24, pp. 43–52, January 1992.

[149] Y-B. Lin and B. R. Preiss, "Optimal memory management for time warp parallel simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 283–307, October 1991.

[150] I. F. Akyildiz, L. Chen, S. R. Das, R. M. Fujimoto, and R. Serfozo, "Performance analysis of time warp w[illegible] limited memory," in *Proceedings of the ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems*, vol. 20, May 1992.

[151] S. R. Das and R. M. Fujimoto, "A performance study of the cancelback protocol for time warp," Technical Report GIT-CC-92/50, Georgia Institute of Technology, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, October 1992.

[152] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. R. Jefferson, "Distributed combat simulation and time warp: The model and its performance," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 14–20, 1989.

[153] D. Baezner, C. Rohs, and H. Jones, "U.S. Army ModSim on Jade's TimeWarp," in *Proceedings of the Winter Simulation Conference*, pp. 665–671, December 1992.

[154] M. Ebling, M. Di Loreto, M. Presley, F. Wieland, and D. R. Jefferson, "An ant foraging model implemented on the time warp operating system," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 21–28, 1989.

[155] P. Hontalas, B. Beckman, M. Di Loreto, L. Blume, P. Reiher, K. Sturdevant, L. Van Warren, J. Wedel, F. Wieland, and D. R. Jefferson, "Performance of the colliding pucks simulation on the time warp operating system (Part 1: Asynchronous behaviour and sectoring)," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 3–7, 1989.

[156] M. Presley, M. Ebling, F. Wieland, and D. R. Jefferson, "Benchmarking the time warp operating system with a computer network simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 8–13, 1989.

[157] R. M. Fujimoto, "Performance of time warp under synthetic workloads," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, pp. 23–28, January 1990.

[158] D. Baezner, J. Cleary, G. Lomow, and B. Unger, "Algorithmic optimizations of simulations on time warp," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, pp. 73–78, March 1989.

[159] R. M. Fujimoto, "The virtual time machine," in *The IEEE International Symposium on Parallel Algorithms and Architectures*, 1989.

[160] K. Ghosh and R. M. Fujimoto, "Parallel discrete event simulation using space-time memory," in *Proceedings of the International Conference on Parallel Processing*, vol. 3, pp. 201–208, August 1991.

[161] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan, "Design and evaluation of the rollback chip: Special purpose hardware for time warp," *IEEE Transactions on Computers*, vol. 41, pp. 68–82, January 1992.

[162] C. M. Pancerella, "Improving the efficiency of a framework for parallel simulations," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, vol. 24, pp. 22–32, January 1992.

[163] P. F. Reynolds Jr. and C. M. Pancerella, "Making parallel simulations go fast," in *Proceedings of the Winter Simulation Conference*, pp. 646–656, December 1992.

[164] C. M. Pancerella and P. F. Reynolds Jr., "Disseminating critical target-specific synchronization information in parallel discrete event simulation," in *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, vol. 23, pp. 52–61, May 1993.

[165] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, *Time Warp Operating System 2.5.1: User's Manual*, jpl d-6493 rev. b ed., September 1991.

[166] J. Ghosh, R. M. Fujimoto, and K. Schwan, "Time warp simulation in time constrained systems," in *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, vol. 23, pp. 163–166, May 1993.

[167] P. L. Reiher, S. Bellenot, and D. Jefferson, "Debugging the time operating system and its application programs," in *Proceedings of the Second Unenix Symposium on Experience with Distributed and Multiprocessor Systems*, pp. 203–220, 1991.

[168] N. Eshragh, *Dynamic routeing in circuit-switched non-hierarchical networks*. PhD thesis, University of Durham, School of Engineering and Applied Science, May 1989.

[169] Encore Computer Corporation, *Encore parallel Fortran manual.* Encore Computer Corporation, 1983.

[170] A. Hind, "YADDES (Yet Another Distributed Discrete Event Simulator) user manual," research report, University of Durham, School of Engineering and Computer Science, September 1992.

[171] B. R. Preiss and I. D. MacIntyre, "YADDES – Yet Another Distributed Discrete Event Simulator: User manual," Technical report E-197, University of Waterloo, Computer Communications Networks Group, 1990.

[172] A. Hind, "On Porting YADDES (Yet Another Distributed Discrete Event Simulator)," research report, University of Durham, School of Engineering and Computer Science, September 1992.

[173] E. P. Rathgeb, "Modeling and performance comparison of policing mechanisms for ATM networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 325–334, April 1991.

[174] M. Butto, E. Cavallero, and A. Tonietti, "Effectiveness of the *Leaky Bucket* policing mechanism in ATM networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 335–342, April 1991.

[175] B. R. Preiss, "Performance of discrete event simulation on a multiprocessor using optimistic and conservative synchronization," in *Proceedings of the IEEE International Conference on Parallel Processing*, August 1990.

[176] J. M. Pitts and Z. Sun, "Burst-level teletraffic modelling and simulation of broadband multi-service networks," in *Proceedings of the Seventh IEE UK Teletraffic Symposium*, pp. 7/1–7/5, May 1990.

[177] J. M. Pitts, J. A. Schormans, and E. M. Scharf, "Burst level simulation: A comparison with cell level simulation and queueing analysis," in *Proceedings of the Ninth IEE UK Teletraffic Symposium*, pp. 8/1–8/6, May 1992.

[178] J. M. Pitts, L. G. Cuthbert, L. G. Bocci, and E. M. Scharf, "Modelling burst scale congestion in ATM networks," in *Proceedings of the First UK Workshop on Performance Modelling and Evaluation of ATM Networks*, pp. 4/1–4/10, June 1993.

[179] J. M. Pitts, L. G. Cuthbert, L. G. Bocci, and E. M. Scharf, "Cell-rate modelling: An accelerated technique for ATM networks," in *Proceedings of the Ninth UK Performance Engineering Workshop for Computer and Telecommunication Systems*, July 1993.

[180] R. W. Earnshaw and P. Mars, "Footprints for mobile communications," in *Proceedings of the Eighth IEE UK Teletraffic Symposium*, pp. 22/1–22/5, April 1991.

[181] J. E. Mellor, J. R. Chen, and M. Hansen, "Simulation support for the management networks," in *Proceedings of the Sixth RACE TMN Conference*, 1992.

[182] J. E. Mellor and A. Hind, "Performance of parallel simulators for ATM networks," in *Proceedings of the First UK Workshop on Performance Modelling and Evaluation of ATM Networks*, pp. 8/1–8/9, June 1993.

[183] A. Chai and S. Ghosh, "Modeling and distributed simulation of a broadband-ISDN network," *IEEE Computer*, vol. 26, pp. 37–51, September 1993.

[184] CCITT: COM XVIII, *228-E.* Geneva, March 1984.

[185] R. Händel and M. N. Huber, *Integrated broadband networks: An introduction to ATM-based networks.* Addison-Wesley, 1991.

[186] M. De Prycker, ed., *Asynchronous transfer mode: Solution for broadband ISDN.* Ellis Horwood, 1990.

[187] J. Chauhan, T. King, and A. C. Micallef, *Specification of the Orwell protocol.* British Telecom Research Laboratories, Martlesham Heath, Ipswich, Suffolk, UK. IP5 7RE.

[188] R. M. Falconer, J. L. Adams, and G. M. Walley, "A simulation study of the cambridge ring with voice traffic," *British Telecom Technology Journal*, vol. 3, April 1985.

[189] J. L. Adams and R. M. Falconer, "Orwell: A protocol for carrying integrated services on a digital communications ring," *Electronics Letters*, vol. 20, pp. 970–971, November 1984.

[190] R. M. Falconer and J. L. Adams, "Orwell: a Protocol for an Integrated Services Local Network," *British Telecom Technology Journal*, vol. 3, October 1985.

[191] R. W. Earnshaw and A. Hind, "A parallel simulator for performance modelling of broadband telecommunication networks," in *Proceedings of the Winter Simulation Conference*, pp. 1365–1373, 1992.

[192] B. M. McGeeney, "Performance of an integrated services ATM protocol over a broadband passive optical network," in *Proceedings of the Sixth IEE UK Teletraffic Symposium*, pp. 12/1–12/8, May 1989.

[193] C. D. Thomborson, "Does your workstation computation belong on a vector supercomputer?," *Communications of the ACM*, vol. 36, pp. 41–49,94, November 1993.

Appendix A

Simulation Model Files

A floppy disk formatted for an IBM. PC. compatible computer is available from the author which contains examples of the simulation model files described herein.

There are four directories on the disk, as follows.

- NAD — contains files which relate to the circuit-switched telecommunication network simulator originally written by Nadereh Eshragh. This was modified to run on modern workstations and on the Encore Multimax, a shared memory multiprocessor, using a parallelizing compiler.
- QUEUE — contains files used by the YADDES simulator to model a closed stochastic queueing network of a hypercube of queues. This runs on a hardwired cube of Inmos transputers.
- TANDEM — contains files used by the YADDES simulator to model a tandem queueing network. This also runs on a hardwired cube of Inmos transputers.
- ATM — contains files used by the Richard Earnshaw's ATM. simulator to model a network of twelve Orwell rings. This runs on a reconfigurable array of Inmos transputers.

For more information on these files contact:

Alan Hind, School of Computing and Mathematics, University of Teesside, Borough Road, Middlesbrough, Cleveland, TS1 3BA.

Telephone: (0642) 342673.

Email: alan.hind@teesside.ac.uk

Appendix B

Published Papers

The following papers have been published as a result of this work:

- A. Hind, "Parallel Discrete-Event Simulation of Engineering Systems", in Proceedings of *The Eighth International Conference on Mathematical and Computer Modelling, University of Maryland, USA.* , April 1991. (Published in *The Journal of Mathematical Modelling and Scientific Computing, Principia Scientia*, Vol. 2, Section A, pp. 228–233, February 1993.)
- A. Hind, "Parallel Simulation for Performance Modelling of Telecommunication Networks", in Proceedings of *The Eighth IEE. UK. Teletraffic Symposium, GEC-Plessey Telecommunications, Beeston, Nottingham*, pp. 9/1–9/6, April 1991.
- A. Hind, "Overview: Parallel Simulation Techniques for Telecommunication Network Modelling", in Proceedings of *The Ninth IEE. UK. Teletraffic Symposium, University of Surrey, Guildford*, pp. 5/1–5/6, April 1992.
- A. Hind, "Parallelization of a Circuit-Switched Telecommunication Network Simulator", in Proceedings of *The Ninth IEE. UK. Teletraffic Symposium, University of Surrey, Guildford*, pp. 7/1–7/7, April 1992.
- R. W. Earnshaw and A. Hind, "Parallel Simulation of Asynchronous Transfer Mode Networks", in Proceedings of *The Fourth IEE. Bangor Communications Symposium, University of Wales, Bangor, Wales*, pp. 58–62, May 1992.

- R. W. Earnshaw and A. Hind, "A Parallel Simulator for Performance Modelling of Broadband Telecommunications Networks," in Proceedings of *the Winter Simulation Conference, Arlington, Virginia, USA.* , pp. 1365–1373, December 1992.

- J. E. Mellor and A. Hind, "Performance of Parallel Simulators for ATM. Networks", in Proceedings of *the First UK. Workshop on Performance Modelling and Evaluation of ATM. Networks, University of Bradford, Bradford*, pp. 8/1–8/9, June 1993.

Math Modelling and Sci. Computing. Vol. 2, pp 228-233, 1993
Principia Scientia. Printed in U.S.A.

ISSN 1067-0688/93 $5.00 + 0.00

Parallel Discrete-Event Simulation of Engineering Systems

A. Hind

British Telecom Research Fellow in Parallel Simulation,
Telecommunication Networks Research Group, School of Engineering and Applied Science,
University of Durham, South Road, Durham, DH1 3LE, U.K.

Abstract

This review paper discusses some of the issues which need to be addressed by someone wishing to use a multiprocessor computer architecture to speed-up the discrete-event simulation of an engineering system. Some indications of future advances in this area are also discussed.

Keywords

Simulation; Methodology; Parallel Processing; Parallel Simulation; Distributed Simulation.

Introduction

The performance evaluation of engineering systems rapidly becomes analytically intractable as the complexity of the system increases. In addition, behaviour under transient conditions, such as traffic fluctuations or component failures, is difficult to express mathematically. Under such conditions the use of simulation techniques to determine relevant performance parameters becomes necessary. Conventional sequential simulations running on sequential (i.e. Von Neumann) computer architectures suffer from limitations imposed by the excessive processing time required to achieve the required depth of information and the intrinsic statistical nature of the results. These problems increase as functions of the activity in the system, its size and complexity. This leads to detailed simulations of large systems often being economically and even physically impossible to implement. Such engineering systems include communication systems, battlefield scenarios, manufacturing systems, road traffic systems, general queueing networks etc.

Dramatic advances are expected over the next decade in the extensive use of parallel multiprocessor architectures to speed-up simulation. However the use of parallel simulation has its own attendant issues. These include the hardware architecture, the decomposition approach used to produce the parallel software processes, mapping these processes onto the processors and the synchronization of the resulting parallel simulation.

Hardware Architecture

The last decade has seen the advent of a huge variety of new computer architectures for parallel processing. This variety can be bewildering to the non-specialist in computer architecture who needs to know which architecture is the most suitable for his application. In order to make an informed choice we need to be able to classify the different types of architecture which are possible along with their suitability for various applications. A useful taxonomy introduced recently is that of Duncan (1990). This is an informal high-level classification scheme, based on Flynn's Taxonomy (Flynn, 1966), which distinguishes between the principle parallel computer architectures which are currently being explored. Of the architectures described by Duncan the most useful, and the most used, architectures for parallel simulation are the synchronous vector (SIMD) architecture and multiprocessor (MIMD) architectures using shared or distributed memory. The synchronous vector architecture will be discussed later in connection with the parallelizing compiler.

A distributed memory architecture needs the processing nodes (processor plus local memory) to be connected using some interconnection network. This network may be static, dynamic or programmable (Almasi and Gottlieb, 1989). Various static interconnection network topologies have been explored to support various applications, eg. pipelines, meshes, trees, rings, cubes, hypercubes etc. Dynamic or programmable

topologies are also possible by using some programmable switching matrix. These can be single-stage. multi-stage or a crossbar. A disadvantage is that the communication overhead associated with this architecture can significantly reduce the performance, particularly where data has to be queued and forwarded by many intermediate nodes. This explains the current popularity of the hypercube topology as it minimises the number of hops between any two processors.

Shared memory architectures allow communication between processors via a common shared memory which each processor can access. Shared memory architectures thus replace message sending problems with data access synchronization and cache coherency problems. As in the case of distributed memory architectures, there are several alternatives for the interconnection of the multiple processors to the shared memory. Some major examples are time-shared bus interconnections, crossbar interconnections and various forms of multi-stage interconnection network.

Comparing the two architectures, the distributed memory architecture gives greater flexibility. Generally it is easier to develop, more easily extensible and, with the advent of more powerful microprocessors, gives a higher performance/cost ratio.

Model Decomposition

For a given simulation model, five ways of decomposing it for processing on a multiprocessor architecture have been identified (Righter and Walrand, 1989) together with combined approaches.

The Parallelizing Compiler

A parallelizing compiler can be used to compile a sequential simulation written in a conventional sequential language so that it will run on our chosen multiprocessor hardware. The compiler thus has the responsibility to recognize sequences in the source code which can be scheduled to run on separate processors in parallel. This definition thus distinguishes a parallelizing compiler from a compiler which takes a high-level parallel language and compiles it to run on multiple processors. The overwhelming advantage is that the approach is largely transparent to the user. A new parallel language does not have to be learned, the multiple processor architecture should not impact the program structure and existing sequential software may be ported. The disadvantage that has been found is that the problem has been coded in sequential form, thus ignoring any parallelism in the structure of the problem. This results in relatively small portions of the available parallelism in the problem being exploited and, hence, the speed-up in moving to a multiple processor architecture is generally disappointing.

There are at least two approaches to converting sequential code to run on multiple processor architectures. The provision of a parallelizing compiler which takes sequential code directly and produces parallel code to run on the target multiple processor system, or, intelligent run-time support and parallel routine libraries to provide the user with a programming environment which allows the conversion of sequential code into parallel code. The latter approach has been taken by several commercial products such as Express and Linda. The former approach is exemplified by the work concerning parallelization and optimization of code for synchronous vector architectures such as the Cray X-MP. Here, sequential Fortran 77 code is compiled and vectorized by the parallelizing compiler. Work has been done by Chandak and Browne (1983) which showed that any network of queues containing feedback loops cannot be vectorized. Because most simulation models of any interest are bound to contain feedback this is a disappointing result. This result was born out by Reed (1985) who also investigated the simulation of queueing networks using a Cray machine. The results were compared with the simulation's performance on a Vax 11/780 using the same sequential code. The results showed a speed-up of about 100 which is almost the same as the two computers rated performance on sequential code. It was suspected that the amount of vectorization was small and, using an execution monitor, it was found to be between 1 and 5%.

Distributed Experiments

Distributed experiments may be conducted by running separate simulations on separate processors in parallel. This is particularly efficient for stochastic simulations, as results can be averaged at the end of the run, and also for doing several "what-if" simulations simultaneously with slightly different parameters. This approach seems extremely efficient as no co-ordination is required between processors except for results averaging and presentation. Hence, for N processors we may approach an ideal speed-up of N. The only other overhead is loading the model into each processor which is often negligible compared with the simulation run time. Many of us have exploited this technique when several workstations or microcomputers are available, perhaps on an evening, overnight or at a weekend.

In terms of the hardware required, distributed experiments may not be possible due to the memory requirements. This has led to the use of networks of uni-processors to realise the approach. Nevertheless, if these memory deficiencies do not apply to the particular simulation application, the distributed experiments approach can be very efficient and can also use existing sequential simulation programs. Also variance reduction techniques such as antithetic sampling and common random number streams may be used to improve statistical efficiency (Frost *et al.*, 1988). The relative unpopularity of the approach is perhaps due to the most common need from simulations being fast and accurate results. As we are not exploiting any parallelism in the problem, speed-up is more in terms of statistical efficiency and simulation throughput.

Distributed Functions

Distributed functions involves different tasks of a simulation being placed on separate processors. For instance, processors may be dedicated to random number generation, event list processing, statistics collection etc. Also other functions may be desired such as animated graphics during simulation or intelligent supervision of the simulation process. Each of these functions may be supported by distributing them to individual processors. The processors may be identical, or may be tailored to each individual function. The advantages of this decomposition method is its freedom from the possibility of deadlock and its potential scaleability. The architecture may also be made transparent to the user as each function's code can be divided up and placed with each processor fairly easily. It could even be made an automatic process at compilation. This would obviously be much easier if identical processors were used. Its disadvantages are the communication overhead between functional processors, which becomes the limiting factor in performance above a handful of processors, and also the failure to exploit any parallelism in the system being modelled. The work on the distributed functions approach seems to indicate that this is a fruitful approach if the number of processes that the simulation is decomposed into is small (Rajagopal and Comfort, 1989).

Distributed Events

Distributed events uses a global event list, as in sequential simulation, to schedule available processors to process the next event on the list. The difficulty is maintaining consistency in the simulation as the next event available on the list may be pre-empted by other events currently being processed by other processors. The need for global simulation control points very much towards the use of a shared memory multiprocessor architecture so that all processors can have access to the global event list. The results for this approach seem to indicate that it is reasonable if there are only a small number of processes required and a large amount of global information used by the components of the system (Sheppard and Davis, 1988).

Distributed Model Components

The final, and most popular, method of decomposing a simulation is to decompose the simulation model into a number of components and assign the simulation of each component to a process. One, or many, processes can then be assigned to execute on each processor. Model decomposition usually follows the logical structure of the real system being simulated. Therefore his approach can take advantage of any parallelism inherent in the system to be modelled, so it seems to promise significant speed-up on a multiple processor system. However, this only holds if the simulation does not require a significant amount of global information and control. The major overhead will be communication between processes executing on different processors. This can be handled by message passing on a distributed memory architecture or global shared variables or message passing on a shared memory architecture.

The two major problems with distributed model components are the development of the model processes themselves and the synchronization of the processes during simulation. Model building is essentially a software problem, synchronization is a problem of both simulation and software. As we shall see the method employed to synchronize the distributed model impacts the way the model is developed and the performance of the simulation. The performance is affected as it is the synchronization overhead which prevents ideal speed-up. Distributed model components offers the greatest potential speed-up in terms of a single simulation. Also, the decomposition of the simulation model can follow the structure of the problem making it easier to understand and develop the models.

Combined Approaches

The ideal decomposition approach for a particular application may well be a combined approach integrating two or more of the above. Several scenarios are possible. For instance, the use of a parallelizing compiler could actually lead to a distributed event approach depending on how the compiler divided up and scheduled the processes. However, it is difficult to imagine how these two approaches could interact with the other

approaches. We could begin by decomposing our simulation model into our loosely coupled components and modelling each with a process. Then, instead of placing each component in a single processor, we could decompose each process into its simulation functions and place each function in a processor. Each component process will thus be executed in a cluster of processors. This seems a useful approach as the research on distributed simulation functions seems to be most efficient using a small number of processors. Also we are exploiting the parallelism of the system at a fine grain size. The disadvantages will be with code generation, loading and lack of flexibility and scaleability. The distributed simulation functions approach could also be exploited alongside distributed model components by using extra processors to handle global results collection, statistical calculations and animation. This combined approach is used in one of the telecommunication network simulators at Durham (Clarke *et al.*, 1989). The distributed experiments approach may be combined with the distributed functions or distributed model component approaches, or both. This simulator could then run several different simulations in parallel as well as exploiting the parallelism in each simulation.

Process Mapping

The mapping of software processes onto hardware processors can be an easy or difficult problem depending on the relative numbers of each. If we have the same number, or more, processors than processes then the mapping can be done fairly logically, particularly if the hardware topology is flexible. If however we have more processes to allocate than processors then the ideal mapping is rarely obvious. The two factors involved are the balance between processing loads and the amount of communication between processors. Any scheme for mapping processes must take these two factors into account. The number of processes to be placed and the structure of the simulation model are also significant. If the number of processes is small or the simulation model has a structure which points to an obvious placement, then the mapping is best done manually. If these criterion are not satisfied then an automatic mapping strategy may be employed.

There are three static mapping strategies which are commonly used at present; random partitioning, heuristic partitioning and simulated annealing. These are essentially a pre-processor approach to mapping processes onto processors. The two performance measures considered by the strategies are processor load and communications volume. They each work by starting at some random placement of processes and running the simulation for a short time to ascertain the two performance measures. A new placement is determined using the particular strategy and the run repeated. This is continued until an "optimum" placement is arrived at. Dynamic strategies are also possible, but the overheads incurred in transferring processes from one processor to another are considered to be too excessive in the vast majority of cases.

Random mapping concentrates entirely on load balancing and ignores communication considerations. It can be effective if allowed sufficient time to explore the whole design space (i.e. a very long time). Heuristic mapping attempts to minimise communication volume while maintaining a high degree of processor load balancing. The heuristic strategy, which searches for the optimum mapping, can find itself locked into a local minima depending on the initial placement. Simulated annealing weights load balancing and communication volume equally, attempting to find the global optimum by perturbation analysis. Thus avoiding the problem of the heuristic strategy. However, all of these strategies take significant amounts of processing time to achieve good results and may not give significantly better results than a manual placement done by a competent engineer (Chamberlain and Franklin, 1990). The author believes that the use of automated and manual process mapping needs to be integrated in much the same way as for printed circuit layout and routeing in semi-custom integrated circuits. The automated mapping can be used to reduce the tedious work but intervention by the engineer is required to apply a measure of common sense.

Simulation Synchronization

Before we discuss various synchronization schemes, it is important to review why it is such a difficult problem. In a sequential simulation, the synchronization of the simulation is maintained by manipulation of a data structure called the event list. This contains the pending events in the system in time stamped order. The simulation progresses by removing the event with the earliest time stamp from the list and processing it. If another event is generated, it is inserted into the event list at its time stamp position. Thus the simulator processes the events in synchronized chronological order. If we now distribute the simulation over several processes, it becomes possible for a processor to process an event which is not the earliest. Also, in processing this event we may affect conditions for earlier, as yet un-simulated events. Thus the future is affecting the past, which is clearly unacceptable, and is known as a causality error. Thus, synchronization schemes fall into two categories; conservative approaches and optimistic approaches.

CONSERVATIVE SYNCHRONIZATION

Conservative approaches avoid causality errors ever occurring by relying on some strategy of determining events which are "safe" to process. That is, they must determine when all events that could affect the event in question have been processed. An added problem which categorises various conservative approaches is that of deadlock. If processes do not have a "safe" event which they can process then they are blocked and cannot progress. If a cycle of blocked processes occurs then we have deadlock and the simulation will grind to a halt unless the deadlock can be broken. Generally conservative synchronization approaches can achieve good performance with sparsely connected systems which have less opportunity for deadlock and/or an application which contains good lookahead properties, eg. (Earnshaw and Mars, 1990). Lookahead refers to the ability to predict what will, or will not, happen in the simulated time future based on application specific knowledge. The worst case for a conservative synchronization approach is to be forced into almost sequential operation coupled with the synchronization mechanism overheads. This situation is not uncommon in some applications which have used these approaches. In their favour are simplicity and their chronological execution.

OPTIMISTIC SYNCHRONIZATION

Optimistic approaches allow causality errors rather than avoid them but when they are detected, a roll-back mechanism is employed to recover by re-simulating from the time of the error. Therefore optimistic approaches don't need to determine whether or not it is safe to proceed; they only need to detect the error and recover. The advantage of this is that the simulator can exploit the parallelism fully in applications which may produce causality errors but in reality rarely do. Obviously, the greater the amount of causality errors that a simulation produces, the greater the synchronization overhead.

The original work on optimistic synchronization was done by Jefferson (1985) on the mechanism called time warp, based on a concept of virtual time. In this case, virtual time is synonymous with simulated time. In the time warp mechanism, a causality error is detected whenever an event message is received by a process that contains a time stamp earlier than the processes' clock (i.e. the time of the last processed message). This is known as a straggler. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler. This is known as roll-back. Two things are affected by roll-back. The process state may be modified; this is accomplished by returning to the correct old state which is taken from a store of previous states. Also, previously sent messages must be unsent; this is achieved by sending anti-messages that cancel the effect of the original. If the original message has already been processed then that process in turn must also roll-back. This process continues until the effects of the causality error are cancelled. For even a moderate size of simulation this seems to imply a large amount of memory to store states for each process. However, as the earliest time stamped event is always safe to process, this is designated global virtual time and is used to discard all states before this time. This process of reclaiming memory, which is irrevocable, is known as fossil collection.

A variation on the above approach, which is said to use aggressive cancellation, is an approach which seeks to "repair the damage". This is known as lazy cancellation. In this case, instead of immediately sending out anti-messages, the process waits to see which messages that the re-execution of the process produces are different to those produced before. If the same message is produced, there is no need to send out an anti-message. It has been found that, depending on the application, lazy cancellation may improve or degrade the simulation performance. Improvement is usually due to processes with incorrect input still producing correct output. Degradation can be due to the additional message checking overheads and the fact that incorrect computations have longer to spread out.

MAKING THE CHOICE

The performance results for time warp approaches often look impressive. However it does have some problems. Time warp approaches do not lend themselves to fine grain parallelism due to the memory overheads required. Each process needs substantial memory capacity as well as the mechanism for fossil collection. Also, it is unproven that a continuous cycle of roll-backs may be possible for a particular simulation. Hence, a conservative approach is often preferred as being "safer". However, when time warp does well, it usually yields far greater speed-up than any conservative approach. There are, at present, no established rules for determining which of these two approaches will allow the greatest amount of speed-up for a given application. Much research effort is currently aimed at this area benchmarking various simulation applications using different synchronization approaches. Reviews of various synchronization approaches are to be found in (Righter and Walrand, 1989; Fujimoto, 1989).

THE FUTURE

The future of parallel simulation will most probably come in three areas; modular simulators, microprocessors for parallel processing and artificial intelligence (AI).

The development of simulation models and the writing and debugging of the simulator code are the two most costly simulation activities, in terms of both time and money. It is imperative in the future that factors like re-usability, modularity, model definition and validation be seriously addressed. Future simulators, whether they use multiprocessors or not, will need to be made of modular re-usable software components. Much attention is currently being given to the definition of object-oriented sequential languages. This work is now being adapted for parallel processing applications, including simulation, with languages such as C++ and Object-Oriented Communicating Sequential Prolog, both for the Inmos Transputer series of processors.

We have already seen that the most flexible approach to distributed simulation is that of the distributed memory multiprocessor. Using cheap modern microprocessors and a standardized communication protocol, such as that used by the Inmos Transputer, a flexible, easily extensible and powerful multiprocessor can be constructed. Future hardware developments lie with new parallel processing elements such as the Inmos T-9000 Transputer (formerly known as the H1 project) and the Intel iWarp and N11 projects.

Useful applications of AI techniques in simulation seem to appear in three categories; model development, simulation control and results analysis. Model development using a standard format could be aided by a software tool akin to an expert system. This would provide an interface to the model format for someone requiring to do a simulation who did not have the time to learn it. Also the tool could transparently make use of existing models from a library. Simulation control could also be aided by an embedded expert system. This could control a number of simulation functions, for instance; the management of simulation and analytical model libraries, analysis of simulation output and choice of suitable statistical methods to increase simulation efficiency, and the mapping of processes to processors. Results analysis and presentation may also be aided in this manner.

ACKNOWLEDGEMENTS

The author would like to thank British Telecom Research Laboratories for both their technical and financial support of this work.

REFERENCES

Almasi G. S. and Gottlieb A. J. (1989). *Highly Parallel Computing, Chap. 8.* Benjamin/Cummings.

Chamberlain R. D. and Franklin M. A. (1990). Hierarchical Discrete Event Simulation on Hypercube Architectures. *IEEE Micro, August*, 10–20.

Chandak A. and Browne J. C. (1983). Vectorization of Discrete Event Simulation. *Proc. of the IEEE Intl. Conf. on Parallel Processing*, 359–361.

Clarke R. T., Nichols S. J., and Mars P. (1989). Transputer-based Simulation Tool for Performance Evaluation of Wide Area Telecommunication Networks. *Microprocessors and Microsystems, Vol. 13, No. 3*, 173–178.

Duncan D. (1990). A Survey of Parallel Computer Architectures. *IEEE Computer, February*, 5–16.

Earnshaw R. W. and Mars P.(1990). Simulation of ATM Networks on Transputer Arrays. *Proc. of the 7th IEE UK Teletraffic Symposium*, 1/1–1/5.

Flynn M. J. (1966). Very High Speed Computing Systems. *Proc. of the IEEE, Vol. 54, No. 12*, 1901–1909.

Frost V. S., Larue Jr. W. W., and Shanmugan K. S. (1988). Efficient Techniques for the Simulation of Computer Communications Networks. *IEEE Trans. on Sel. areas in Comms., Vol. 6, No. 1*, 146–157.

Fujimoto R. M. (1989). Parallel Discrete Event Simulation. *Proc. of the IEEE Winter Simulation Conf.*. 19–28.

Jefferson D. R. (1985). Virtual Time. *ACM TOPLAS, Vol. 7, No. 3*, 404–425.

Rajagopal R. and Comfort J. C. (1989). Contrasting Distributed Simulation with Parallel Replication: A Case Study of a Queueing Simulation with a Network of Transputers. *Proc. of the IEEE Winter Simulation Conf.*, 746–755.

Reed D. A. (1985). Parallel Discrete Event Simulation: A Case Study. *Proc. of the 18th IEEE Annual Simulation Symposium*, 95–107.

Righter R. and Walrand J. C. (1989). Distributed Simulation of Discrete Event Systems. *Proc. of the IEEE, Vol. 77, No. 1*. 99–113.

Sheppard S. V. and Davis C. K. (1988). Parallel Emulation Environments for Multiprocessor Architectures. *Proc. of the SCS Multiconference on Distributed Simulation*, 109–114.

Parallel Simulation for Performance Modelling of Telecommunication Networks

A. Hind[°]

1 Introduction

The performance evaluation of telecommunication networks rapidly becomes analytically intractable as the complexity of the network increases. In addition, behaviour under transient conditions, such as traffic fluctuations or component failures, is difficult to express mathematically. Under such conditions the use of simulation techniques to determine relevant performance parameters becomes necessary. Conventional sequential simulations running on sequential (i.e. Von Neumann) computer architectures suffer from limitations imposed by the excessive processing time required to achieve the required depth of information and the intrinsic statistical nature of the results. These problems increase as functions of the traffic intensity and the size and complexity of the network. This leads to detailed simulations of large networks often being economically and even physically impossible to implement.

Dramatic advances are expected over the next decade in the extensive use of parallel multiprocessor architectures to speed-up simulation. However the use of parallel simulation has its own attendant issues. These include the hardware architecture, the decomposition approach used to produce the parallel software processes, mapping these processes onto the processors and the synchronization of the resulting parallel simulation. This is not an exhaustive list of the issues, but will suffice for the purposes of this review paper.

2 Hardware Architecture

The last decade has seen the advent of a huge variety of new computer architectures for parallel processing. This variety can be bewildering to the non-specialist in computer architecture who needs to know which architecture is the most suitable for his application. In order to make an informed choice we need to be able to classify the different types of architecture which are possible along with their suitability for various applications. A useful taxonomy introduced recently is that of Duncan [1]. This is an informal high-level classification scheme, based on Flynn's Taxonomy, which distinguishes between the principle parallel computer architectures which are currently being explored.

Parallel simulation of telecommunication systems tends to result in a relatively coarse grain model. The communication volume between software processes is high and the amount of memory required is also high. These factors push us towards the use of multiprocessor architectures using shared or distributed memory. The software processes executing on each processor are then synchronized by passing messages either via an interconnection network or via shared memory.

A distributed memory architecture needs the processing nodes (processor plus local memory) to be connected using some interconnection network. This network may be static, dynamic or programmable. Various static interconnection network topologies have been explored to support various applications, eg. pipelines, meshes, trees, rings, cubes, hypercubes etc. Dynamic or programmable topologies are also possible by using some programmable switching matrix. These can be single-stage, multi-stage or a crossbar. A disadvantage is that the communication overhead associated with this architecture can significantly reduce the performance, particularly where data has to be queued and forwarded by many intermediate nodes.

Shared memory architectures allow communication between processors via a common shared memory which each processor can access. Shared memory architectures thus replace message sending problems with data access synchronization and cache coherency problems. As in the case of distributed memory architectures, there are several alternatives for the interconnection of the multiple processors to the shared memory. Some major examples are time-shared bus interconnections, crossbar interconnections and various forms of multi-stage interconnection network.

[°]British Telecom Research Fellow, Telecommunication Networks Research Group, School of Engineering and Applied Science, University of Durham, South Road, Durham, U.K. DH1 3LE.

Comparing the two architectures, the distributed memory architecture gives greater flexibility. Generally it is easier to develop, more easily extensible and, with the advent of more powerful microprocessors, gives a higher performance/cost ratio.

3 Model Decomposition

For a given simulation model, five ways of decomposing it for processing on a multiprocessor architecture have been identified [2]. A parallelizing compiler approach, distributed experiments, distributed functions, distributed events and distributed model components. Combined approaches are also possible.

A parallelizing compiler can be used to compile a sequential simulation written in a conventional sequential language so that it will run on our chosen multiprocessor hardware. The compiler thus has the responsibility to recognize sequences in the source code which can then be executed in parallel and scheduled to run on separate processors in parallel. This definition thus distinguishes a parallelizing compiler from a compiler which takes a high-level parallel language and compiles it to run on multiple processors. The overwhelming advantage is that the approach is largely transparent to the user. A new parallel language does not have to be learned, the multiple processor architecture should not impact the program structure and existing sequential software may be ported. The disadvantage that has been found is that the problem has been coded in sequential form, thus ignoring any parallelism in the structure of the problem. This results in relatively small portions of the available parallelism in the problem being exploited and, hence, the speed-up in moving to a multiple processor architecture is generally disappointing.

There are at least two approaches to converting sequential code to run on multiple processor architectures. The provision of a parallelizing compiler which takes sequential code directly and produces parallel code to run on the target multiple processor system, or, intelligent run-time support and parallel routine libraries to provide the user with a programming environment which allows the conversion of sequential code into parallel code. The latter approach has been taken by several commercial products such as Express and Linda. The former approach is exemplified by the work concerning parallelization and optimization of code for synchronous vector and shared memory multiprocessor architectures. Taking for example the Cray X-MP, a synchronous vector architecture, sequential Fortran 77 code is compiled and vectorized by the parallelizing compiler. Work has been done by Chandak [3] which showed that any network of queues containing feedback loops cannot be vectorized. Because most simulation models of any interest are bound to contain feedback this is a disappointing result. This result was born out by Reed [4] who investigated the simulation of queueing networks. The results were compared with the simulations performance on a Vax 11/780 using the same sequential code. The results showed a speed-up of about 100 which is almost the same as the two computers rated performance on sequential code. It was suspected that the amount of vectorization was small and, using an execution monitor, it was found to be between 1 and 5%.

Distributed experiments may be conducted by running separate simulations on separate processors in parallel. This is particularly efficient for stochastic simulations, as results can be averaged at the end of the run, and also for doing several "what-if" simulations simultaneously with slightly different parameters. This approach seems extremely efficient as no co-ordination is required between processors except for results averaging and presentation. Hence, for N processors we may approach an ideal speed-up of N. The only other overhead is loading the model into each processor which is often negligible compared with the simulation run time.

In terms of the hardware required, distributed experiments may not be possible due to the memory requirements. This has led to the use of networks of uni-processors to realise the approach. Nevertheless, if these memory deficiencies do not apply to the particular simulation application, the distributed experiments approach can be very efficient and can also use existing sequential simulation programs. Also variance reduction techniques such as antithetic sampling and common random number streams may be used to improve statistical efficiency [5]. The relative unpopularity of the approach is perhaps due to the most common need from simulations being fast and accurate results. As we are not exploiting any parallelism in the problem, speed-up is more in terms of statistical efficiency and simulation throughput.

Distributed functions involves different tasks of a simulation being placed on separate processors. For instance, processors may be dedicated to random number generation, event list processing, statistics collection etc. Also other functions may be desired such as animated graphics during simulation or intelligent supervision of the simulation process. Each of these functions may be supported by distributing them to individual processors. The processors may be identical, or may be tailored to each individual function. This is very much like the approach taken in many personal computers; a general purpose processor,

an arithmetic co-processor for floating-point calculations, a graphics co-processor, other processors for controlling input/output functions with keyboards, printers or communications links.

The advantages of this decomposition method is its freedom from the possibility of deadlock and its potential scaleability. The architecture may also be made transparent to the user as each function's code can be divided up and placed with each processor fairly easily. It could even be made an automatic process at compilation. This would obviously be much easier if identical processors were used. Its disadvantages are the communication overhead between functional processors, which may become the limiting factor in performance, and also the failure to exploit any parallelism in the system being modelled. The work on the distributed functions approach seems to indicate that this is a fruitful approach if the number of processes that the simulation is decomposed into is small [6, 7]. The law of diminishing returns sets in at an early stage above a handful of processors due to communication overheads.

Distributed events uses a global event list, as in sequential simulation, to schedule available processors to process the next event on the list. The difficulty is maintaining consistency in the simulation as the next event available on the list may be pre-empted by other events currently being processed by other processors. The need for global simulation control points very much towards the use of a shared memory multiprocessor architecture so that all processors can have access to the global event list. The results for this approach seem to indicate that it is reasonable if there are only a small number of processes required and a large amount of global information used by the components of the system [8].

The final, and most popular, method of decomposing a simulation is to decompose the simulation model into a number of components and assign the simulation of each component to a process. One, or many, processes can then be assigned to execute on each processor. Model decomposition usually follows the logical structure of the real system being simulated. Therefore his approach can take advantage of any parallelism inherent in the system to be modelled, so it seems to promise significant speed-up on a multiple processor system. However, this only holds if the simulation does not require a significant amount of global information and control. The major overhead will be communication between processes executing on different processors. This can be handled by message passing on a distributed memory architecture or global shared variables or message passing on a shared memory architecture. The other major problem is the synchronization of events during the simulation. Generally speaking, the more loosely coupled the processes can be (i.e. asynchronous requiring little communication), the more likely the simulation is to be processor bound. On the other hand, the more tightly coupled the processes are, the more likely the simulation is to be communication bound.

The two major problems with distributed model components is the development of the model processes themselves and the synchronization of the processes during simulation. Model building is essentially a software problem, synchronization is a problem of both simulation and software. As we shall see the method employed to synchronize the distributed model impacts the way the model is developed and the performance of the simulation. The performance is affected as it is the synchronization overhead which prevents ideal speed-up. Distributed simulation model components offers the greatest potential speed-up in terms of a single simulation. Also, the decomposition of the simulation model can follow the structure of the problem making it easier to understand and develop the models. In the case of the simulation of communication networks we can also produce something akin to a software emulation. The synchronization problem mars the potential of the approach, but guide-lines are beginning to appear as to which will be best for various types of application.

The ideal decomposition approach for a particular application may well be a combined approach integrating two or more of the above. Several scenarios are possible. For instance, the use of a parallelizing compiler could actually lead to a distributed event approach depending on how the compiler divided up and scheduled the processes. However, it is difficult to imagine how these two approaches could interact with the other approaches. But let us now consider what approaches might work effectively together.

We could begin by decomposing our simulation model into our loosely coupled components and modelling each with a process. Then, instead of placing each component in a single processor, we could decompose each process into its simulation functions and place each function in a processor. Each component process will thus be executed in a cluster of processors. This seems a useful approach as the research on distributed simulation functions seems to be most efficient using a small number of processors. Also we are exploiting the parallelism of the system at a fine grain size. The disadvantages will be with code generation, loading and lack of flexibility and scaleability. The distributed simulation functions approach could also be exploited alongside distributed model components by using extra processors to handle global results collection, statistical calculations and animation. This combined approach is used in one of the telecommunication network simulators at Durham [9]. The distributed experiments approach may be combined with the distributed functions or distributed model component approaches, or both.

This simulator could then run several different simulations in parallel as well as exploiting the parallelism in each simulation.

4 Process Mapping

The mapping of software processes onto hardware processors can be an easy or difficult problem depending on the relative numbers of each. If we have the same number, or more, processors than processes then the mapping can be done fairly logically, particularly if the hardware topology is flexible. If however we have more processes to allocate than processors then the ideal mapping is rarely obvious. The two factors involved are the balance between processing loads and the amount of communication between processors. Any scheme for mapping processes must take these two factors into account.

The number of processes to be placed and the structure of the simulation model are also significant. If the number of processes is small or the simulation model has a structure which points to an obvious placement, then the mapping is best done manually. If these criterion are not satisfied then an automatic mapping strategy may be employed.

There are three static mapping strategies which are commonly used at present; random partitioning, heuristic partitioning and simulated annealing. These are essentially a pre-processor approach to mapping processes onto processors. The two performance measures considered by the strategies are processor load and communications volume. They each work by starting at some random placement of processes and running the simulation for a short time to ascertain the two performance measures. A new placement is determined using the particular strategy and the run repeated. This is continued until a "best" placement is arrived at. Dynamic strategies are also possible, but the overheads incurred in transferring processes from one processor to another are considered to be too excessive in the vast majority of cases.

Random mapping concentrates entirely on load balancing and ignores communication considerations. It can be effective if allowed sufficient time to explore the whole design space (i.e. a very long time). Heuristic mapping attempts to minimise communication volume while maintaining a high degree of processor load balancing. The heuristic strategy, which searches for the optimum mapping, can find itself locked into a local minima depending on the initial placement. Simulated annealing weights load balancing and communication volume equally, attempting to find the global optimum by perturbation analysis. Thus avoiding the problem of the heuristic strategy. However, all of these strategies take significant amounts of processing time to achieve good results and may not give significantly better results than a manual placement done by a competent engineer [10]. It is believed that the use of automated and manual process mapping needs to be integrated in much the same way as for printed circuit layout and routeing in semi-custom integrated circuits. The automated mapping can be used to reduce the tedious work but intervention by the engineer is required to sort out problems and apply common sense.

5 Simulation Synchronization

Before we discuss various synchronization schemes, it is important to review why it is such a difficult problem. In a sequential simulation, the synchronization of the simulation is maintained by manipulation of a data structure called the event list. This contains the pending events in the system in time stamped order. The simulation progresses by removing the event with the earliest time stamp from the list and processing it. If another event is generated, it is inserted into the event list at its time stamp position. Thus the simulator processes the events in synchronized chronological order. If we now distribute the simulation over several processes, it becomes possible for a processor to process an event which is not the earliest. Also, in processing this event we may affect conditions for earlier, as yet un-simulated events. Thus the future is affecting the past, which is clearly unacceptable, and is known as a causality error.

Thus, synchronization schemes fall into two categories; conservative approaches and optimistic approaches. Conservative approaches avoid causality errors ever occurring by relying on some strategy of determining events which are "safe" to process. That is, they must determine when all events that could affect the event in question have been processed. An added problem which categorises various conservative approaches is that of deadlock. If processes do not have a "safe" event which they can process then they are blocked and cannot progress. If a cycle of blocked processes occurs then we have deadlock and the simulation will grind to a halt unless the deadlock can be broken. Generally conservative synchronization approaches can achieve good performance with sparsely connected systems which have less opportunity for deadlock and/or an application which contains good lookahead properties (eg. [11]). Lookahead refers to the ability to predict what will, or will not, happen in the simulated time future based on application specific knowledge. The worst case for a conservative synchronization approach is to

be forced into almost sequential operation coupled with the synchronization mechanism overheads. This situation is not uncommon in some applications which have used these approaches. In their favour are simplicity, relative transparency to the user and their chronological execution.

Optimistic approaches allow causality errors rather than avoid them but when they are detected, a roll-back mechanism is employed to recover by re-simulating from the time of the error. Therefore optimistic approaches don't need to determine whether or not it is safe to proceed; they only need to detect the error and recover. The advantage of this is that the simulator can exploit the parallelism fully in applications which may produce causality errors but in reality rarely do. Obviously, the greater the amount of causality errors that a simulation produces, the greater the synchronization overhead.

The original work on optimistic synchronization was done by Jefferson [12] on the mechanism called time warp, based on a concept of virtual time. In this case, virtual time is synonymous with simulated time. In the time warp mechanism, a causality error is detected whenever an event message is received by a process that contains a time stamp earlier than the processes' clock (i.e. the time of the last processed message). This is known as a straggler. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler. This is known as roll-back. Two things are affected by roll-back. The process state may be modified; this is accomplished by returning to the correct old state which is taken from a store of previous states. Also, previously sent messages must be unsent; this is achieved by sending anti-messages that cancel the effect of the original. If the original message has already been processed then that process in turn must also roll-back. This process continues until the effects of the causality error are cancelled. For even a moderate size of simulation this seems to imply a large amount of memory to store states for each process. However, as the earliest time stamped event is always safe to process, this is designated global virtual time and is used to discard all states before this time. This process of reclaiming memory, which is irrevocable, is known as fossil collection.

A variation on the above approach, which is said to use aggressive cancellation, is an approach which seeks to "repair the damage". This is known as lazy cancellation. In this case, instead of immediately sending out anti-messages, the process waits to see which messages that the re-execution of the process produces are different to those produced before. If the same message is produced, there is no need to send out an anti-message. It has been found that, depending on the application, lazy cancellation may improve or degrade the simulation performance. Improvement is usually due to processes with incorrect input still producing correct output. Degradation can be due to the additional message checking overheads and the fact that incorrect computations have longer to spread out.

The performance results for time warp approaches often look impressive. However it does have some problems. Time warp approaches do not lend themselves to fine grain parallelism due to the memory overheads required. Each process needs substantial memory capacity as well as the mechanism for fossil collection. Also, it is unproven that a continuous cycle of roll-backs may be possible for a particular simulation.

There are, at present, no established rules for determining which of these two approaches will allow the greatest amount of speed-up for a given application. Much research effort is currently aimed at this area benchmarking various simulation applications using different synchronization approaches. Reviews of various synchronization approaches are to be found in [2, 13].

6 The Future

The future of parallel simulation will most probably come in three areas; modular simulators, cheap microprocessors for parallel processing and artificial intelligence (AI).

The development of simulation models and the writing and debugging of the simulator code are the two most costly simulation activities, in terms of both time and money. It is imperative in the future that factors like re-usability, modularity, model definition and validation be seriously addressed. Future simulators, whether they use multiprocessors or not, will need to be made of modular re-usable software components. Much attention is currently being given to the definition of object-oriented sequential languages. This work is also likely to be adapted for parallel processing applications including simulation.

We have already seen that the most flexible approach to distributed simulation is that of the distributed memory multiprocessor. Using cheap modern microprocessors and a standardized communication protocol, such as that used by the Inmos Transputer, a flexible, easily extensible and powerful multiprocessor can be constructed. Future hardware developments lie with new parallel processing elements such as the Inmos H1 Transputer and the Intel iWarp and N11 projects.

Useful applications of AI techniques in simulation seem to appear in three categories; model development, simulation control and results analysis. Model development using a standard format could be

aided by a software tool akin to an expert system. This would provide an interface to the model format for someone requiring to do a simulation who did not have the time to learn it. Also the tool could transparently make use of existing models from a library. Simulation control could also aided by an embedded expert system. This could control a number of simulation functions, for instance; the management of simulation model and analytical model libraries, analysis of simulation output and choice of suitable statistical methods to increase simulation efficiency, and the mapping of processes to processors. Results analysis and presentation may also be aided in this manner. The expert system approach is ideally suited where the knowledge domain is well defined and stable. This makes the approach more viable in the simulation of existing systems rather than the exploration of the performance of new technologies.

Acknowledgements

The author would like to thank British Telecom Research Laboratories for both their technical and financial support of this work.

References

[1] R. Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer*, pp. 5–16, February 1990.

[2] R. Righter and J. C. Walrand, "Distributed Simulation of Discrete Event Systems," *Proceedings of the IEEE*, vol. 77, pp. 99–113, January 1989.

[3] A. Chandak and J. C. Browne, "Vectorization of Discrete Event Simulation," in *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 359–361, 1983.

[4] D. A. Reed, "Parallel Discrete Event Simulation: A Case Study," in *Proceedings of the 18th IEEE Annual Simulation Symposium*, pp. 95–107, 1985.

[5] V. S. Frost, W. W. Larue Jr., and K. S. Shanmugan, "Efficient Techniques for the Simulation of Computer Communications Networks," *IEEE Transactions on Selected Areas in Communications*, vol. 6, pp. 146–157, January 1988.

[6] R. Rajagopal and J. C. Comfort, "Contrasting Distributed Simulation with Parallel Replication: A Case Study of a Queueing Simulation with a Network of Transputers," in *Proceedings of the IEEE Winter Simulation Conference*, pp. 746–755, 1989.

[7] J. C. Comfort and R. Rajagopal, "Environment Partitioned Distributed Simulation with Transputers," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 8–13, 1985.

[8] S. V. Sheppard and C. K. Davis, "Parallel Emulation Environments for Multiprocessor Architectures," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 109–114, 1988.

[9] R. T. Clarke, S. J. Nichols, and P. Mars, "Transputer-based Simulation Tool for Performance Evaluation of Wide Area Telecommunication Networks," *Microprocessors and Microsystems*, vol. 13, no. 3, pp. 173–178, 1989.

[10] R. D. Chamberlain and M. A. Franklin, "Hierarchical Discrete Event Simulation on Hypercube Architectures," *IEEE Micro*, pp. 10–20, August 1990.

[11] R. W. Earnshaw and P. Mars, "Simulation of ATM Networks on Transputer Arrays," in *The 7th IEE UK Teletraffic Symposium*, pp. 1/1–1/5, April 1990.

[12] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[13] R. M. Fujimoto, "Parallel Discrete Event Simulation," in *Proceedings of the IEEE Winter Simulation Conference*, pp. 19–28, 1989.

Overview: Parallel Simulation techniques for Telecommunication Network Modelling

A. Hind [*]

1 Introduction

The performance evaluation of telecommunication networks rapidly becomes analytically intractable as the complexity of the network increases. Modern networks are often large, geographically spread and subject to constant change and growth. They are increasingly likely to consist of interconnected network types running diverse protocols on diverse platforms over diverse connection types. In addition to this, the behaviour of interest to the performance engineer is often that which occurs under transient conditions, such as traffic fluctuations or component failures which are known to be difficult to express mathematically.

Under such conditions the use of simulation techniques to determine relevant performance parameters becomes necessary. Conventional sequential simulations running on sequential (i.e. Von Neumann) computer architectures suffer from limitations imposed by the excessive processing time required to achieve the required depth of information and the intrinsic statistical nature of the results. These problems increase as functions of the traffic intensity and the size and complexity of the network. This leads to detailed simulations, of traffic intensive and very large networks, often being economically and even physically impossible to implement.

Parallel simulation makes use of multiprocessor hardware architectures and various techniques to achieve execution *speed-up*. This paper reviews the current state of the art in parallel simulation indicating existing techniques and their known application to telecommunication network simulation. This is followed by a discussion of the problems still to be resolved.

2 Overview of Parallel Simulation

Parallel Simulation, sometimes called distributed simulation, refers to the execution of a single discrete event simulation program on a parallel multiprocessor computer. It has received much attention in recent years due to the number of disciplines which use discrete event simulation to study large scale systems; these include economics, computer science, engineering, transport and military studies as well as telecommunications. However, it has been found that, even though many such applications contain substantial amounts of parallelism, it is surprisingly difficult to realise significant speed-up when moving to a multiprocessor.

There are many additional issues to resolve in the development of a parallel simulator; these include the hardware architecture of the multiprocessor, the decomposition approach used to produce the parallel tasks (processes), mapping these tasks onto the processors and the synchronization of the resulting parallel simulation. This paper follows on from the review paper given on this subject at this symposium last year [1] Excellent general reviews of parallel simulation can be found in [2, 3].

3 Hardware Architecture

Parallel simulation of telecommunication systems tends to result in a relatively coarse grain model. That is, the software tasks written to model the system tend to be relatively large and complex. The communication volume between software tasks is high and the amount of memory required is also high. Discrete event simulation is also, by it's very nature, irregular and data-dependent. Thus, along with other disciplines, parallel simulation of telecommunication networks has proved to be an application area where parallelization or vectorization techniques using supercomputer hardware has had little success. These factors push us towards the use of multiprocessor architectures using shared or distributed memory.

[*] Alan Hind is the British Telecom Research Fellow, Telecommunication Networks Research Group, School of Engineering and Computer Science, University of Durham, South Road, Durham, U.K. DH1 3LE.

The software tasks executing on each processor are then synchronized by passing messages either via an interconnection network or via shared memory.

A distributed memory architecture needs the processing nodes (processor plus local cache memory) to be connected using some interconnection network. This network may be static, dynamic or programmable. Various static topologies have been explored to support various applications, e.g. pipelines, meshes, trees, rings, cubes, hypercubes etc. Dynamic or programmable topologies are also possible by using some form of programmable switching matrix. These can be single-, multi-stage or a crossbar. A disadvantage is that the communication overhead associated with this architecture can significantly reduce the performance, particularly where data has to be queued and forwarded by many intermediate nodes.

Shared memory architectures allow communication between processors via a common shared memory which each processor can access. Shared memory architectures thus replace message processing problems with data access synchronization and cache coherency problems. As in the case of distributed memory architectures, there are several alternatives for the interconnection of the processors to the shared memory. Some major examples are the time-shared bus, crossbar and various forms of multi-stage network.

Both types of architecture are available "off-the-shelf" [4] for those who don't wish to develop their own machine. Distributed memory multiprocessors are available either as complete systems with languages, tools and operating system, or as sets of parallel processor "building blocks". The latter are usually plug-in cards to workstations. Shared memory multiprocessors are normally only available as complete systems with their own operating system etc. This is now often a version of the Unix operating system, with it's associated set of software tools, allowing the machine to also be used as a general purpose computer. Comparing the two architectures, the distributed memory architecture gives greater overall flexibility. Generally it is easier to develop, more easily extensible and, with the advent of more powerful microprocessors, gives a higher performance/cost ratio.

The processor of choice for building distributed memory multiprocessor machines has for the last few years been the Inmos transputer. It still has much to recommend it in terms of hardware support for multitasking, concurrency and communications. However, it has now been surpassed in terms of raw processing power by a new crop of microprocessors such as Intel's i860 and i486. Inmos are seeking to redress this with the development of the T9000 transputer, which will also hopefully solve some of the outstanding communication difficulties with transputer networks with the provision of hardware support for message passing. In the mean time, it is interesting to note the number of processing cards now available which couple the communications and parallel processing capability of the transputer with the computing power of another, more modern, microprocessor.

Competitors for the transputer's parallel processing niche in the market have been slow in coming. But now there are the Texas Instruments TMS320C40 and the Intel iWarp. The Texas processor is primarily intended for parallel digital signal processing applications but could be used for novel communication simulation/emulation applications. The Intel iWarp is more of a direct competitor though it is currently being aimed at a finer grain applications than the transputer and a commercially available system based on it has yet to appear. Both are still unproven as far as parallel simulation is concerned.

4 Model Decomposition

For a given simulation model, five ways of decomposing it for processing on a multiprocessor architecture have been identified [2]. A parallelizing compiler approach, distributed experiments, distributed functions, distributed events and distributed model components. Some combinations of these approaches are also possible.

4.1 The Parallelizing Compiler Approach

A parallelizing compiler is a software tool that can be used to compile an application written in a conventional sequential language so that it will execute on a particular multiprocessor. The compiler thus has the responsibility to recognise sequences in the source code which can be executed in parallel and scheduled to run concurrently on separate processors. This definition thus distinguishes it from a compiler which takes a high-level parallel language and compiles it to run on multiple processors. The overwhelming advantage of the approach is it's transparency. A new parallel language does not have to be learned, the multiprocessor architecture should not impact the program structure and existing "dusty-decks" may be ported. The disadvantage is that is that the application has been coded in sequential form, often using inherently sequential algorithms, thus ignoring any potential parallelism. This results in relatively small portions of the available parallelism being exploited and (hence) the speed-up in moving to the multiprocessor is generally disappointing. Results for this approach are discussed in some detail

in [5] elsewhere in these proceedings. This paper also explores the parallelization of a sequential circuit-switched network simulator. The results show that given a simple enough model, in this case one which involves a great deal of matrix manipulation, significant speed-up can be achieved if there is sufficient parallelism in the model to mask synchronization overheads.

An alternative approach to converting sequential code to run on a multiprocessor is to provide the user with a programming environment, intelligent run-time analysis tools and parallel function libraries. This approach is thus a few steps further on than re-writing the application from scratch in a parallel language. This approach is typified by commercial products such as Express and Linda.

4.2 The Distributed Experiments Approach

Distributed experiments, often known as parallel replications, may be conducted by running the simulations on separate processors in parallel. This is particularly efficient for stochastic simulations, as results can be averaged at the end of the run, and also for doing several "what-if" simulations simultaneously with slightly different parameters. This approach seems extremely efficient as no co-ordination is required between processors except for results averaging and presentation. Hence, for n processors we may approach an ideal speed-up of n. The only other overhead is loading the model into each processor which is often negligible compared with the simulation run time.

In terms of the hardware required, distributed experiments may not be possible due to the memory requirements. This has led to the use of networks of uniprocessors to realise the approach. Nevertheless, if these memory deficiencies do not apply to the particular simulation application, the distributed experiments approach can be very efficient and can also use existing sequential simulation programs. Also variance reduction techniques such as antithetic sampling and common random number streams may be used to improve statistical efficiency [6]. The relative unpopularity of the approach is perhaps due to the most common need from simulations being fast and accurate results. As we are not exploiting any parallelism in the problem, speed-up is more in terms of statistical efficiency and simulation throughput.

4.3 The Distributed Functions Approach

Distributed functions involves different tasks of a simulation being placed on separate processors. For instance, processors may be dedicated to random number generation, event list processing, statistics collection etc. Each of these functions may be supported by distributing them to individual processors. The processors may be identical, or may be tailored to each individual function. The work on this approach seems to indicate that it can be fruitful if the number of tasks involved is small. The law of diminishing returns sets in at an early stage above a handful of processors due to communication overheads.

4.4 The Distributed Events Approach

Distributed events uses a global event list, as in sequential simulation, to schedule available processors to process the next event on the list. The difficulty is maintaining consistency in the simulation as the next event available on the list may be pre-empted by other events currently being processed by other processors. The need for global simulation control points very much towards the use of a shared memory multiprocessor architecture so that all processors can have access to the global event list. The results for this approach seem to indicate that it is reasonable if there are only a small number of tasks required and a large amount of global information used by the components of the system.

4.5 The Distributed Model Components Approach

The final, and most popular, method of decomposing a simulation is to decompose the simulation model into a number of components and assign the simulation of each component to a task. One, or many, tasks can then be assigned to execute on each processor. Model decomposition usually follows the logical structure of the real system being simulated. Therefore this approach can take advantage of any parallelism inherent in the system to be modelled, so it seems to promise significant speed-up. However, this only holds if the simulation does not require a significant amount of global information and control. The major overhead will be communication between tasks executing on different processors. The other major problem is the synchronization of events during the simulation. Generally speaking, the more loosely coupled the tasks can be (i.e. asynchronous requiring little communication), the more likely the simulation is to be processor bound. On the other hand, the more tightly coupled the tasks are, the more likely the simulation is to be communication bound.

The two major problems with distributed model components is the development of the model tasks themselves and the synchronization of the tasks during simulation. Model building is essentially a software problem, synchronization is a problem of both simulation and software. The method employed to synchronize the distributed model impacts the way the model is developed and the performance of the simulation. The performance is affected as it is the synchronization overhead which prevents ideal speed-up. Distributed simulation model components still offers the greatest potential speed-up in terms of a single simulation. Also, the decomposition of the simulation model can follow the structure of the problem making it easier to understand and develop the models. In the case of the simulation of communication networks we can also produce something akin to a software emulation. Synchronization is discussed in section 6.

4.6 Combined Approaches

The ideal decomposition approach for a particular application may well be a combined approach integrating two or more of the above. Firstly, let us discount the parallelizing compiler and the distributed event approaches as it is difficult to imagine how these two approaches could interact with any of the others. But let us consider what approaches might work effectively together. We could begin by decomposing our simulation model into our loosely coupled components and modelling each with a software task. Then, instead of placing each component in a single processor, we could decompose each task into its simulation functions and place each function in a processor. Each component task will thus be executed in a cluster of processors. This seems a useful approach as the research on distributed simulation functions seems to be most efficient using a small number of processors. Also we are exploiting the parallelism of the system at a finer grain size. The disadvantages will be with code generation, loading and lack of flexibility and scaleability. The distributed simulation functions approach could also be exploited alongside distributed model components by using extra processors to handle global results collection, statistical calculations or animation. The distributed experiments approach may be combined with the distributed functions or distributed model component approaches, or both. This simulator could then run several different simulations in parallel as well as exploiting the parallelism in each simulation.

5 Process Mapping

The mapping of software tasks onto hardware processors can be an easy or difficult problem depending on the relative numbers of each. If we have the same number (or more) processors than tasks then the mapping can be done fairly logically, particularly if the hardware topology is flexible. If however we have more tasks to allocate than processors then the ideal mapping is rarely obvious. The three factors involved are the balance between processing loads, the amount of communication between processors and the scheduling of the tasks when running. Any scheme for mapping tasks must take these factors into account. The structure of the simulation model is also significant. If the number of tasks is small or the simulation model has a structure which points to an obvious placement, then the mapping is best done manually. If these criterion are not satisfied then an automatic mapping strategy may be employed. However, automatic strategies take significant amounts of processing time to achieve good results and may not give significantly better results than a manual placement done by a competent engineer. It is believed that the use of automated and manual process mapping needs to be integrated in much the same way as for placement and routeing for printed circuit layout and semi-custom integrated circuits. Automated mapping can be used to reduce the tedious work but intervention by the engineer is required to sort out problems and apply common sense.

6 Simulation Synchronization

Before we discuss various synchronization schemes, it is important to review why it is such a difficult problem. In a sequential simulation, the synchronization of the simulation is maintained by manipulation of a data structure called the event list. This contains the pending events in the system in time-stamp order. The simulation progresses by removing the event with the earliest time-stamp from the list and processing it. If another event is generated, it is inserted into the event list at its time-stamp position. Thus the simulator processes the events in synchronized chronological order. If we now distribute the simulation over several tasks executing on separate processors, it becomes possible for a processor to process an event which is not the earliest. Also, in processing this event we may affect conditions for earlier, as yet unsimulated events. Thus the future is affecting the past, which is clearly unacceptable, and

is known as a *causality error*. Thus, synchronization schemes fall into two categories, conservative and optimistic approaches, which are distinguished by how they handle causality errors. Reviews of various synchronization approaches are to be found in [2, 3].

6.1 Conservative Synchronization

Conservative approaches avoid causality errors ever occurring by relying on some strategy of determining events which are *safe* to process. That is, they must determine when all events that could affect the event in question have been processed. An added problem which categorises various conservative approaches is that of *deadlock*. If tasks do not have a safe event which they can process then they are blocked and cannot progress. If a cycle of blocked tasks occurs then we have deadlock and the simulation will grind to a halt unless the deadlock can be broken. Generally conservative synchronization approaches can achieve good performance with sparsely connected systems which have less opportunity for deadlock and/or an application which contains good *lookahead* properties. Lookahead refers to the ability to predict what will, or will not, happen in the simulated time future based on application specific knowledge. The worst case for a conservative synchronization approach is to be forced into almost sequential operation coupled with the synchronization mechanism overheads. This situation is not uncommon in some applications which have used these approaches. In their favour are simplicity, relative transparency to the user and their chronological execution.

6.2 Optimistic Synchronization

Optimistic approaches allow task to proceed asynchronously but incorporate a detection and *roll-back* mechanism to catch causality errors and recover by re-simulating from the time of the error. Therefore optimistic approaches don't need to determine whether or not it is safe to proceed. The advantage of this is that the simulator can exploit the parallelism fully in applications which may produce causality errors but in reality rarely do. Obviously, the greater the amount of causality errors that a simulation produces, the greater the synchronization overhead due to roll-backs.

The original work on optimistic synchronization was done by Jefferson [7] on the mechanism called *time-warp*, based on a concept of virtual time. In this case, virtual time is synonymous with simulated time. In the time-warp mechanism, a causality error is detected whenever an event message is received by a task that contains a time-stamp earlier than the tasks' local clock. This is known as a *straggler*. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the task receiving the straggler. This is known as roll-back. Two things are affected by roll-back. The process state may be modified; this is accomplished by returning to the correct old state which is taken from a store of previous states. Also, previously sent messages must be unsent; this is achieved by sending *anti-messages* that cancel out the effect of the original. If the original message has already been processed then that task in turn must also roll back. This process continues until the effects of the causality error are cancelled. For even a moderate size of simulation this implies a large amount of memory to store states for each task. However, as the earliest time-stamped event is always safe to process, this is designated as the global virtual time and is used to discard all states before this time. This process of reclaiming memory, which is irrevocable, is known as *fossil collection*.

A variation on the above approach, which is said to use *aggressive cancellation*, is an approach which seeks to "repair the damage". This is known as *lazy cancellation*. In this case, instead of immediately sending out anti-messages, the task waits to see which messages that the re-execution of the task produces are different to those produced before. If the same message is produced, there is no need to send out an anti-message. It has been found that, depending on the application, lazy cancellation may improve or degrade the simulation performance. Improvement is usually due to tasks with incorrect input still producing correct output. Degradation can be due to the additional message checking overheads and the fact that incorrect computations have longer to spread out.

The performance results for optimistic approaches often look impressive. However it does have some problems. Optimistic approaches do not lend themselves to fine grain parallelism due to the memory overheads required. Each task needs substantial memory capacity as well as a mechanism for fossil collection. This problem has been addressed analytically by Lin and Lazowska [8] and also experimentally by Preiss et. al. [9]. The work of Lin and Lazowska showed that reducing the frequency of state saving (i.e. increasing the so called *checkpoint interval*) can save memory and also reduce simulation run time. This was borne out by the work of Preiss et. al. in their simulation of queueing networks. Further, they

showed that there can be a trade-off between memory and run-time, that there is an optimum checkpoint interval for a given simulation and that this point is predictable from the analytical model of Lin and Lazowska.

7 Existing Problems

The most successful parallel simulation speed-up results have been obtained using the distributed model components approach which leaves us with the issue of synchronization. There are, at present, no established rules for determining which of the basic synchronization approaches will lead to the greatest amount of speed-up for a given application. The consensus of opinion seems to be that optimistic synchronization is the more generally applicable of the two. Much research effort is currently aimed at this area analysing parallel simulation behaviour and benchmarking various simulation applications.

The behaviour of optimistic synchronization mechanisms is still not fully understood particularly with regard to the prediction of memory requirements and the occurrence of roll-back. The most worrying aspect of all is that it is difficult (if not impossible) to prove that a particular simulation may be prone to excessive cycles of roll-backs. However, it has seldom been reported by simulationists. Some analytical work has already been done in this area [10, 8].

Little work has been done in the area of process mapping and scheduling for parallel simulation. This is partly due to most simulators having a relatively small number of processors and tasks. However, it is known that these factors can have a significant impact on the simulation performance [9, 8].

Acknowledgements

The author would like to thank British Telecom Research Laboratories for both their technical and financial support of this work.

References

[1] A. Hind, "Parallel Simulation for Performance Modelling of Telecommunication Networks," in *Proceedings of the Eighth IEE UK Teletraffic Symposium*, pp. 9/1–9/6, April 1991.

[2] R. Righter and J. C. Walrand, "Distributed Simulation of Discrete Event Systems," *Proceedings of the IEEE*, vol. 77, pp. 99–113, January 1989.

[3] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, pp. 30–53, October 1990.

[4] A. Trew and G. Wilson, eds., *Past, present and Parallel: a survey of available parallel computing systems*. Springer-Verlag, 1991.

[5] A. Hind, "Parallelization of a Circuit-Switched Telecommunication Network Simulator," in *Proceedings of the Ninth IEE UK Teletraffic Symposium*, pp. 7/1–7/7, April 1992.

[6] V. S. Frost, W. W. Larue Jr., and K. S. Shanmugan, "Efficient Techniques for the Simulation of Computer Communications Networks," *IEEE Transactions on Selected Areas in Communications*, vol. 6, pp. 146–157, January 1988.

[7] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[8] Y. Lin and E. D. Lazowska, "A Study of Time Warp Rollback Mechanisms," *ACM Transactions on Modelling and Computer Simulation*, vol. 1, pp. 51–72, January 1991.

[9] B. R. Preiss, I. D. MacIntyre, and W. M. Loucks, "On the trade-off between Time and Space in Optimistic Parallel Discrete-Event Simulation," in *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, pp. 1–9, January 1992.

[10] B. Lubachevsky, A. Weiss, and A. Shwartz, "An Analysis of Rollback-based Simulation," *ACM Transactions on Modelling and Computer Simulation*, vol. 1, pp. 154–193, April 1991.

Parallelization of a Circuit-Switched Telecommunication Network Simulator

A. Hind*

1 Introduction

The need for parallel simulation is now well established [1]. However, there is still a marked reluctance for performance engineers to learn and use parallel languages and, indeed, to purchase or construct specialised multiprocessor hardware. This situation is gradually changing as many shared memory multiprocessor machines (see figure 1) are now appearing which can function as general purpose Unix machines or as parallel processing platforms, see table 1. Such machines are now becoming more competitively priced and, hence, more common [2].

Company	*Mark*	*Operating System*
Alliant Computer Systems	FX/2800	Concentrix (4.3 BSD) or AT&T System V.4
BBN Advanced Computers Inc.	ACI TC2000	nX or *pSOS*$^{+m}$ (4.3 BSD)
Convex Computer Corporation	C2	Convex Unix (4.3 BSD)
Encore Computer Corporation	Multimax and 91	UMAX 4.3 (4.3 BSD) or UMAX V (AT&T System V)
FPS Computing	System 500	FPX (4.3 or 4.4 BSD)
Sequent Computer Systems	Balance and Symmetry	DYNIX (4.2 BSD and AT&T System V)

Table 1: Currently available shared memory multiprocessor machines.

This paper explores the use of a parallelizing compiler on such a shared memory multiprocessor to speed-up the execution of a circuit-switched telecommunication network simulator. A parallelizing compiler is defined as one which takes an application written in a conventional sequential programming language, determines which parts can be executed in parallel, and produces the machine level code to run on the multiple processors. Thus, speed-up may be achieved without necessarily learning a new parallel language and re-writing the whole simulation, or knowing too much about the underlying hardware.

The approach taken was to discover the "path-of-least-resistance" to achieving significant speed-up. That is, to find out how to get the most speed-up from a simulation with the minimum of effort. With any multiprocessor architecture, the best way of fully exploiting the parallelism available is to re-write the application in the machine's own parallel language using inherently parallel algorithms wherever possible. In this case, we are trying to exploit the automated facilities to minimise development time as well as execution time.

The nearest relevant research to this exercise is that of Chandak and Browne [3] and that of Reed [4]. The work done by Chandak and Browne, showed that discrete event simulation cannot always be parallelized using this approach and, more specifically, that the discrete event simulation of any network of queues containing feedback loops cannot be parallelized. As most simulation models of any interest are almost bound to contain feedback this was a disappointing result. On a positive note though, they showed that careful optimization of event-list processing could produce a speed-up of two on a CDC Cyber 205 even for non-parallelized code. This result was born out by Reed. He used a Cray X-MP and Cray's Fortran 77 vectorizing compiler (see footnote over) to investigate the discrete event simulation of queueing networks. The results were compared with the simulation's performance on a Vax 11/780 using the same sequential code. The results showed a speed-up of about a hundred which is almost the same as the two computers rated performance on sequential code. It was suspected that the amount of vectorization was small and using an execution monitor it was found to be between one and five percent for different simulation models.

*British Telecom Research Fellow, Telecommunication Networks Research Group, School of Engineering and Computer Science, University of Durham, South Road, Durham, U.K. DH1 3LE.

Nevertheless, Reed did hold out some hope for this approach as he believed the current limitations were with the Cray Fortran vectorizing† compiler. There were many loops in the simulation which potentially could have been vectorized but weren't because the compiler couldn't vectorize any loops containing IF statements, function calls or data dependencies; even by rearrangement of the code. From a cursory study of the Cray and the Encore compilers it was felt that the Encore compiler offered better facilities for overcoming these problems (see section 2.2).

2 The Testbed Architecture

2.1 Hardware Architecture

The shared memory multiprocessor used in these experiments was an Encore Multimax. This was first designed in 1983 and is produced in several versions. The version available at the University of Newcastle's Computing Laboratory is a 520, known locally as Newton, with fourteen main processors and 160 Mbytes of shared memory. The basic architecture is illustrated in figure 1.

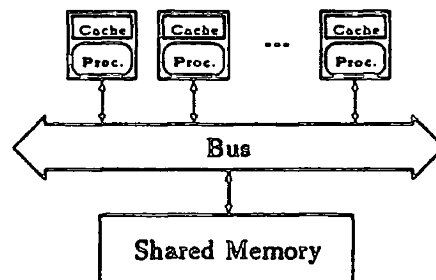


Figure 1: Shared memory multiprocessor architecture with a single bus and local caches.

The Multimax is controlled by a central processor which provides general monitoring and diagnostic facilities. This is based on a National Semiconductor NS32016 running at 10 MHz. The main processors are based on a processor/co-processor pair, the NS32532 and NS32381, running at 30 MHz. Encore claim a peak performance of 4 MFlops for a main processor. Each main processor has a 256 kbyte cache memory controlled from the memory management circuitry in the NSC32532.

The complete system is built around a bus designed by Encore called the nanobus. It is made up of three individual buses; a 32 bit address bus, a 64 bit data bus and a 14 bit vector bus. In addition, each bus has an extra byte wide channel for parity checking. The Multimax 520 is so called as it has space for 20 card slots on the nanobus. The clock speed of the bus is nominally 12.5 Mhz giving a total memory bandwidth of 100 Mbytes per second. Inevitably, the bus is the bottleneck for this type of machine partially alleviated by the local cache memories.

2.2 Software Architecture

For the Multimax, Encore developed two varieties of UNIX: UMAX 4.3 and UMAX V. The former is based on Berkeley UNIX 4.3 BSD and the latter is an implementation of the System V UNIX from AT&T. UMAX 4.3 is the system available on Newton. The languages available are C, UMAX Fortran and Encore Parallel Fortran (EPF).

The UMAX Fortran compiler can produce standard sequential code for running on a single processor, or parallel code for running on multiple processors using EPF. The sequential Fortran meets the ANSI Fortran 77 standard and also includes most of the VAX/VMS extensions. In order to support the parallelism of the machine, a number of Fortran 90 features have also been included. Parallelization of programs may either be done manually or automatically by the parallel optimizer. The optimizer is implemented as a preprocessor to the compiler. If required, the programmer may take the output from the preprocessor to perform further parallelization by hand. The compiler also has an option to generate execution profiling code. Thus, when the program is executed a trace file is produced which can be analysed using the utility gprof to produce a report. The report will contain exact call counts, call graph arcs and statistically approximate timing data for a process as well as other data to aid in optimizing the program.

† Vectorization and parallelization are not quite synonymous. Parallelization is a generic term for the automatic distribution of software tasks over a number of processors; vectorization also implies that the tasks are rendered suitable for execution on vector processors.

EPF analyses the source code to determine which program segments can be executed in parallel and converts them into parallel Fortran constructs. EPF uses explicit synchronization statements and local variables to improve concurrency automatically. Among the optimizations which EPF can perform are the following;

- Loop spreading – conversion of DO loops into DOALL (parallel) loops
- Loop splitting – loops split into simplified loops more amenable to loop spreading
- Statement re-ordering – to enable loop splitting and spreading
- Variable initiation – introduction of new variables to eliminate dependencies and enable loop spreading
- Synchronization – introduction of explicit synchronization and ordering controls

The extent to which any program can be parallelized in this manner is limited by four primary forms of data dependency.

- Flow dependency – assignment modifies a variable used in a later statement
- Anti-dependence – assignment is made after a variable is altered
- Output dependency – assignment must complete before another is made
- Control dependency – conditional statement dependent on prior statements

3 Modelling Speed-up

The speed-up, $S(n)$, of a multiprocessor with n processors is defined to be the ratio of the total execution time on a uniprocessor to the total execution time on the multiprocessor. In this case, we qualify this by stating that the uniprocessor version will not contain any optimizations to aid a parallelizing compiler. A hierarchy of terms has also appeared to describe various levels of speed-up [5]; Superunitary speed-up ($S(n) > n$), Unitary speed-up ($S(n) = n$) and Subunitary speed-up ($S(n) < n$).

Let us now define a simple model for the multiprocessor in order to gauge the success (or failure) of the exercise. Such models are defined in more detail in [5]. We define the total number of operations performed by the complete program as W. Various types of operation are involved, so we define W_i as the number of times that operations of type i are performed. Each operation has a cost in processor time, so we can define $C_i(n)$ as the time taken by all the processors to perform one operation of type i on an n processor system. Let W_s represent the number of sequential operations and $C_s(n)$ the cost of performing a single sequential operation on an n processor system. Likewise, let W_p and $C_p(n)$ be the amount and cost of work which exhibits n-fold parallelism. Therefore, speed-up is defined as;

$$S(n) = \frac{W_s C_s(1) + W_p C_p(1)}{\frac{W_s C_s(n)}{n} + \frac{W_p C_p(n)}{n}} \quad (1)$$

If we define the cost of one sequential operation to be one unit of processor time, then $C_s(1) = 1$. Also we can assume that when performing a single sequential operation on an n processor system, one processor is working and the other $n-1$ are idle; hence $C_s(n) = nC_s(1) = n$. Further, if we assume that the cost of one operation is the same for sequential and parallel operation then equation (1) reduces to Amdahl's law[1].

$$S(n) = \frac{W_s + W_p}{W_s + \frac{W_p}{n}} \quad (2)$$

Amdahl's law for various values of W_p, where W_p and W_s are expressed as a fraction of the total number of operations, is shown graphically in figure 2. Amdahl's law also leads to a theoretical upper-bound for speed-up for this model of;

$$\lim_{n \to \infty} S(n) = \frac{W_s + W_p}{W_s} = \frac{1}{1 - W_p} \quad (3)$$

However, this model does not take any account of any overheads. In the case of our shared memory multiprocessor there will be synchronization (sync) overheads and memory reference overheads due to cache misses. Therefore, we modify the work to be done as follows;

[1] In this particular case, given the software architecture of the Encore Multimax, both of these assumptions are valid.

$$W = W_s + W_p + W_m + W_{sync} \quad (4)$$

where W_m is the number of parallelizable memory reference operations, W_{sync} is the number of sync operations and W_p is now the number of non-memory and non-sync parallel operations. The extra cost functions are;

$$C_m(n) = 1 + \alpha_n miss(n) \quad (5)$$

$$C_{sync}(n) = 1 + \beta_n switch \quad (6)$$

where α_n is the fraction of memory references which cannot be satisfied by the cache. The cost of a miss is $miss(n)$. β_n is the fraction of times that a task must stop at a sync point. The cost of the subsequent context switch is *switch*.

Both the cache miss ratio α_n and the sync ratio β_n will depend on n and the size of the program and it's data. In this particular application it would seem sensible to expect α_n to decrease and β_n to increase as n increases. These trends were in fact confirmed by the performance monitor traces though there was no discernible pattern to the increase/decrease.

Both of the above models would seem to indicate that superunitary speed-up is not possible in this case and that Amdahl's law (2) and it's limit (3) can be treated as true upper-bounds. A more realistic model based on (4), (5) and (6) would indicate lower values of speed-up than for Amdahl's law.

4 The Simulator

The simulator used for these experiments was written by Nadereh Eshragh at Durham University Telecommunication Networks Research Group as part of her Ph.D. thesis looking into dynamic routeing in circuit-switched telecommunication networks [6]. The simulator was written in Fortran 77 for execution on an Amdahl 470 mainframe. The simulator was designed to handle a network of up to ten fully-connected exchanges, or nodes, though it can easily be extended to any number of nodes and also to networks which are not fully-connected. The routeing strategies available are fixed routeing, random routeing, automatic alternative routeing (AAR), least busy alternative (LBA), dynamic alternative routeing (DAR) and stochastic learning automata (SLA) using linear reward inaction (LRI) or linear reward penalty (LRP). These strategies, and their relative performance in are described in [7] and [6].

The information supplied to the simulator consists of the number of nodes, random number seeds, capacity and traffic matrices, simulation run title, routeing information, an overload factor (scaling factor for network traffic) and a trunk reservation parameter. Results are written into an output file along with the input information.

The simulation consists of a number of time units (usually taken to be seconds), each time unit consisting of a cycle of the following actions. Firstly, the inter-arrival time, and the origin-destination pair of the next call arrival are generated. Next, the calls which have been completed during this inter-arrival time are cleared down. Then, an attempt to route the new call is made according to the specified routeing algorithm. Finally, the acceptance or rejection (blocking) of the call, along with the origin-destination pair information, is recorded into the appropriate arrays. Results are written to the screen and the output file at the end of each time unit.

Thus, we are dealing with very simple model of a circuit-switched network. Many operational details of the network are ignored in order to focus on the performance of the routeing algorithms in relation to the overall blocking probability. As this model involves a great deal of matrix manipulation there seemed a good possibility that some speed-up would be possible using a parallelizing compiler.

The test network models used were of five-, ten- and twenty-node fully-connected networks. The figures for the five- and ten-node networks were chosen to reflect reasonably busy trunks in a realistic network. The link capacities were obtained using fixed route dimensioning using Erlang's formula constrained by a modularity factor of thirty circuits and random effects due to periodic upgrading of trunks. The twenty-node network was less realistic as the link capacities and traffic volumes needed to be much smaller to give lower overall network traffic and hence a reasonable execution time.

5 Results

5.1 Uniprocessor Simulation Results

For these simulations the fixed routeing strategy was used and the same random number seeds. Initial runs were done with various routeing options but there was little to choose between them in execution time and the fixed routeing strategy was generally the shortest. The simulation run length was 200 time units (seconds). The only change between implementations on different machines was the Fortran READ and WRITE statements which needed to be different depending on the Fortran implementation. In each case the results were checked against those obtained from the original implementation to ensure consistency. There were no problems observed in this respect for the uniprocessor or multiprocessor simulations. Execution times were obtained using the Unix Time command and the operating system (MTS) logging facilities on the Amdahl. The uniprocessor execution times are shown in Table 2.

A single processor on the Encore Multimax using the sequential option to compile the code yields roughly the same performance as the Amdahl mainframe. Thus the Encore Multimax times were used as the reference for the speed-up calculations. The more modern Sun SPARCstation is easily the fastest, as much as four times faster than the Amdahl or Encore Multimax.

Machine	*five-node model*	*ten-node model*	*twenty-node model*
Amdahl 470	607	782	3212
Encore Multimax 520	675	844	3308
Sun 4/460 SPARCstation	126	256	1173

Table 2: Execution times for the uniprocessor simulations in seconds.

5.2 Multiprocessor Simulation Results for the Five-node Model

The multiprocessor simulations were begun by using the parallelizing compiler EPF without any modifications to the source code. The results, in the form of a speed-up graph is shown in figure 2. Overall, the results are disappointing with no speed-up and with the trend being downwards as n increases.

These results were investigated by re-running the simulations with the profiler option. From careful examination of the profile reports the most expensive subroutines and functions could be noted, including the compiler generated functions which manage parallel execution. One of these, mtask_fork, which manages the re-synchronization of tasks before and after parallel execution phases, becomes more expensive as n increases. The overheads of this function effectively stifle any potential speed-up.

At this point, it was decided that work must be concentrated on reducing the cost of the most expensive tasks of the original program thereby reducing the impact of the mtask_fork. Only two tasks were involved, clearing down calls and calculating link loads, which were by far the most expensive. This was achieved in both cases by hand coding. Data dependencies which couldn't be removed by the compiler were removed using local independent variables and coding some of the function calls in-line. This allowed the compiler to achieve greater parallelization in the tasks by loop spreading.

This new version of the simulator yielded slightly better results as shown in figure 2. However, there is still no evidence of speed-up and the trend is still downwards as n increases.

The simulations were again profiled showing that the execution times of the hand-coded functions had decreased but the number of calls to mtask_fork were considerably increased due to the increased parallelism. Thus the synchronization overheads still stifled the speed-up as n increased. Therefore it was decided that a larger simulation model might lead to more significant speed-up as the synchronization overheads would remain about the same but the amount of parallelism would increase.

5.3 Multiprocessor Simulation Results for the Ten-node Model

The results shown in figure 2 using the new version of the simulator are much more encouraging with a net speed-up for all n. There is an intriguing dip in the speed-up graph for 6, 7 and 8 processors, picking up again for 9, peaking at 10 and dipping again above 10. This type of behaviour is not uncommon with machines employing parallelizing compilers. The greatest speed-up is most often observed when n coincides numerically with the granularity of the application's parallelism. Where it doesn't match more overheads are incurred as extra tasks are run on less than the full complement of processors. This decreases the effective utilization of the processors available. This phenomenon is described for Cray Parallel Fortran in [8].

The execution profiles bore out the assumption made after the five-node model experiments in that the compiler generated functions had much less impact on the execution time. That is, the processors spend much more time, than previously with the five-node model, executing tasks in parallel; thus making the synchronization overheads less significant. The fastest executions of the ten-node model were with 4, 5 and 10 processors which were equal to the nearest three seconds of execution time.

5.4 Multiprocessor Simulation Results for the Twenty-node Model

The twenty-node model was created to investigate whether the encouraging results found with the ten-node model would scale for larger models. The results shown in figure 2 would indicate that they do.

6 Conclusions

These experiments have highlighted the fact that there needs to be sufficient parallelism in a problem to make it possible to achieve reasonable speed-up when porting it to a multiprocessor architecture. The results for the five-node model were poor as the model did not contain enough parallelism to offset the cost of the parallel synchronization overheads. The threshold, or break-even point, was obviously passed with the ten-node model. The results for the twenty-node model indicate that this approach will scale to larger network models though ultimately more processors would be needed to take full advantage of the available parallelism.

As a measure of the "success" of the ten-node simulations, consider the following. The "worst" simulation was with 7 processors giving a speed-up of 1.65. This indicates, using Amdahl's law (2), that the percentage of the program parallelized was 45.6% giving a theoretical maximum speed-up, using equation (3), of 1.84. The "best" simulation was with 4 processors yielding a speed-up of 1.99. This indicates a percentage parallelized of 66.3% and a theoretical maximum speed-up of 2.97. These measures make the the results obtained look quite reasonable.

The complicating factor is of course due to the parallel synchronization overheads which are assumed to be negligible in the derivation of Amdahl's law, as has been shown. Interestingly, when the profile reports for the ten-node simulations were examined, between 22.6% ($n = 4$) and 44.3% ($n = 14$) of the execution times were expended on synchronization operations. Removing these overheads would yield speed-ups of 2.57 and 3.12 respectively. Furthermore, the percentage of the program parallelized indicated by these revised figures is closely grouped between 71.7% ($n = 6$) and 76.7% ($n = 10$); a theoretical maximum speed-up of 4.30. Thus, a revised graph of speed-up against n for the ten-node model would follow Amdahl's law for $Wp = 0.75$ quite closely. It must be noted, however, that overheads due to cache misses have not been removed though their effect must be much lower than the synchronization overheads.

The Encore Parallel Fortran package would seem to be quite reasonable for this kind of exercise. The most difficult aspect, as always with parallel processing, is debugging. In this case, the parallel simulation can be run on one processor with the debugging tool fdb. This was never severerly put to the test in this case. The other debugging alternative is to write the application as a sequential program initially and debug this with the standard tools before proceeding with the parallelization. The profiler gprof proved to be an essential tool in highlighting problem areas and assessing performance, echoing again the findings of Reed [4]. The hand coding performed as part of the exercise was not particularly arduous, only occupying a couple of working days for finding out how to do it, doing it and debugging successfully. The interesting point to note is that the key to success was knowledge, not of the machine's architecture, but of the compiler's "hooks".

Acknowledgements

The author would like to thank British Telecom Research Laboratories for both their technical and financial support of this work and also the staff of the Computing Laboratory at the University of Newcastle-upon-Tyne.

References

[1] A. Hind, "Parallel simulation for performance modelling of telecommunication networks," in *Proceedings of the 9th IEE UK Teletraffic Symposium*, pp. 9/1–9/6, April 1991.

[2] A. Trew and G. Wilson, eds., *Past, present and Parallel: a survey of available parallel computing systems.* Springer-Verlag, 1991.

[3] A. Chandak and J. C. Browne, "Vectorization of discrete event simulation," in *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 359–361, 1983.

[4] D. A. Reed, "Parallel discrete event simulation: a case study," in *Proceedings of the 18th IEEE Annual Simulation Symposium*, pp. 95–107, 1985.

[5] D. P. Helmbold and C. E. McDowell, "Modeling speedup (n) greater than n," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 250–256, April 1990.

[6] N. Eshragh, *Dynamic routeing in circuit-switched non-hierarchical networks.* PhD thesis, School of Engineering and Applied Science, University of Durham, May 1989.

[7] N. Eshragh and P. Mars, "Performance evaluation of de-centralized routeing strategies in circuit-switched networks," in *Proceedings of the 4th IEE UK Teletraffic Symposium*, May 1987.

[8] G. S. Almasi and A. J. Gottlieb, eds., *Highly Parallel computing.* Benjamin/Cummings, 1989.

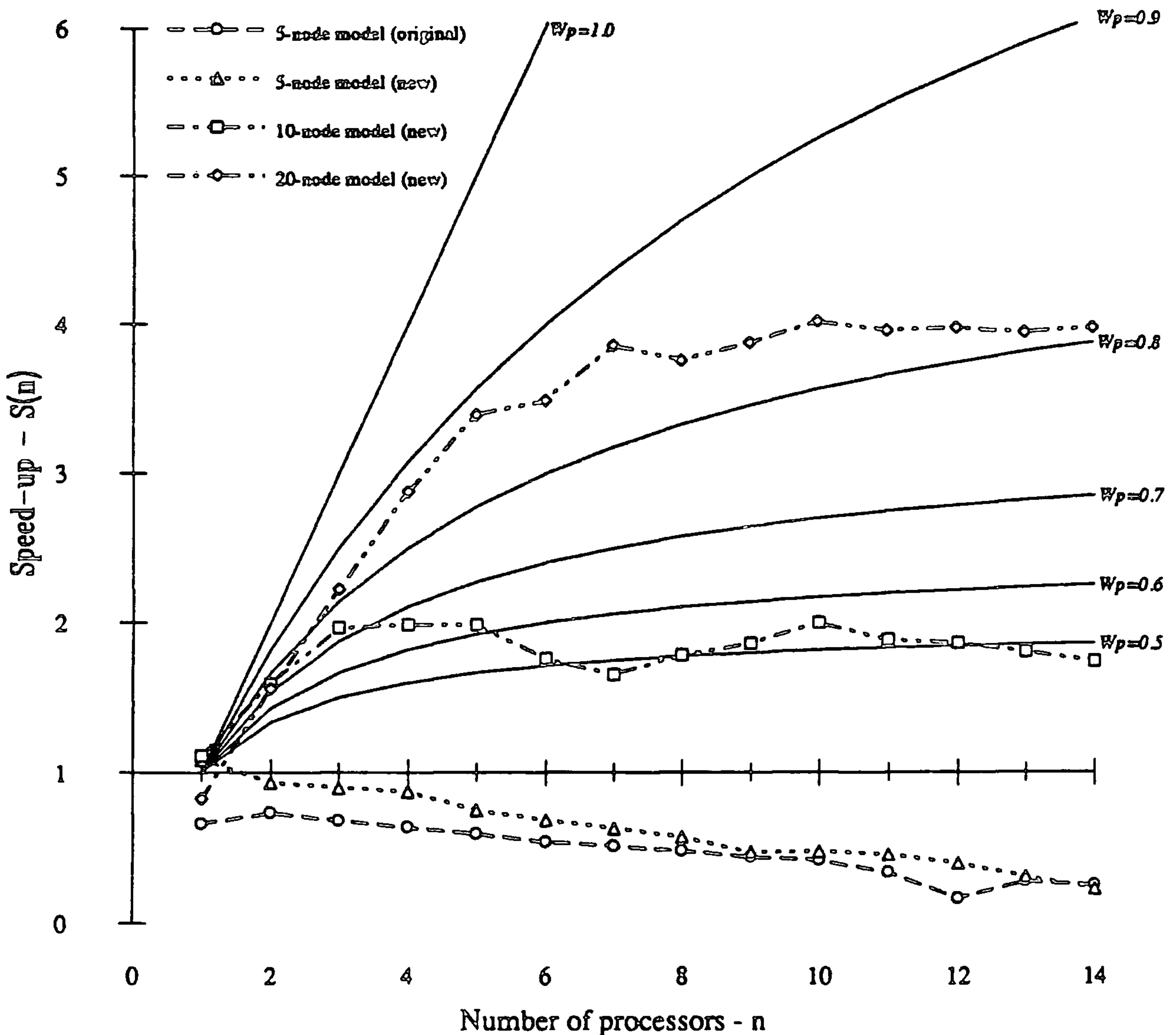


Figure 2: Speed-up graph for multiprocessor simulations

Parallel Simulation of Asynchronous Transfer Mode Networks

R. W. Earnshaw and A. Hind °

1 Introduction

The performance evaluation of telecommunication networks rapidly becomes analytically intractable as the complexity of the network increases. In addition to this, the behaviour of interest to the performance engineer is often that which occurs under transient conditions, such as traffic fluctuations or component failures which are known to be difficult to express mathematically.

Under such conditions the use of simulation techniques to determine relevant performance parameters becomes necessary. Conventional sequential simulations running on sequential (i.e. Von Neumann) computer architectures suffer from limitations imposed by the excessive processing time required to achieve the required depth of information and the intrinsic statistical nature of the results. These problems increase as functions of the traffic intensity and the size and complexity of the network. This leads to detailed simulations, of traffic intensive and very large networks, often being economically and even physically impossible to implement. Such problems have led to growing interest in parallel simulation using multiprocessor hardware.

The argument becomes most pointed when considering the performance evaluation of asynchronous transfer mode (ATM) networks. The complexity, traffic intensity and the potential size of the networks are all large. Simulation studies of ATM have thus far largely centred on the behaviour of single traffic sources, multiplexers or switching nodes. When complete ATM networks have been studied it has usually been at the call- or burst-level since cell-level simulation involves levels of complexity (and hence processing time) which are orders of magnitude greater.

Nevertheless, for many investigations of ATM network behaviour, cell-level simulation is unavoidable and has motivated the development of a parallel multiprocessor simulator by the University of Durham Telecommunication Networks Research Group. The simulator is being used for the study of network behaviour and, particularly, for studying of the integration of mobile communication protocols into the ATM environment [1].

The use of parallel simulation introduces many additional issues into the simulation design process. These include the hardware architecture, the decomposition approach used to produce the parallel software processes, mapping these processes onto the processors and the synchronization of the resulting parallel simulation. A review of the application of parallel simulation techniques to the performance evaluation of telecommunication networks can be found in [2]. An excellent general review of parallel simulation can be found in [3].

2 ATM Networks

The CCITT define an integrated services digital network (ISDN) as one " ... that provides end-to-end digital connectivity to support a wide range of services, including voice and non-voice services, to which users have access by a limited set of standard multi-purpose user-network interfaces" [4]. Initially, *basic access* was centred around two 64 kbits/s B channels and one 16 kbits/s signalling D channel. The requirement for supporting more advanced multi-media services within ISDN has led to the development of broadband ISDN (B-ISDN).

The asynchronous transfer mode is the CCITT's target solution for B-ISDN. ATM networks use a fixed-size data packet, known as a cell, which consists of 48 octets of data and 5 octets of header. They are typically transmitted, within the network, using multi-megabit-per-second media, such as fibre-optic links; such links will typically be running at data rates in excess of 150 Mbit/s. Further background on B-ISDN, and ATM networks in particular, can be found in [5].

The ATM switch used in the simulation study is based on the Orwell ring protocol which is a slotted ring protocol. The ring is divided into slots which circulate around the ring; a node wishing to transmit a message waits until an unfilled slot is found, changes the slot header and transmits the message in the body of the slot. A typical implementation of a slotted ring is the Cambridge ring protocol (British Standard BS6531). Examination of existing protocols has indicated that those based on a slotted ring are probably the best suited for carrying delay-sensitive speech, but simulation studies of high-bandwidth Cambridge Rings have indicated that there are

°Richard Earnshaw is a Research Assistant and Alan Hind is British Telecom Research Fellow. Telecommunication Networks Research Group, School of Engineering and Computer Science, University of Durham, South Road, Durham, U.K. DH1 3LE.

still significant limitations when operated under high load [6] and, further, load control is difficult since there is no relevant parameter that can easily be extracted from the ring. The Orwell protocol was developed after making a detailed study of the limitations of the Cambridge ring protocol: it was found that by introducing destination release of slots, and by adding a novel, distributed, load control mechanism to bound access delays, a viable level of performance could be obtained [7, 8]. For higher capacity networks multiple, synchronized, rings can be used and such a network is known as an Orwell Torus.

Whilst detailed simulations of a single ring have been made, under a variety of load and traffic services, there has, as yet, been very little investigation made into the behaviour of an Orwell torus, or ring behaviour in multi-ring systems. The reason for this, at least in part, is because of the large amount of simulation time required to investigate networks of Orwell rings: a single simulation run of one ring takes, typically, a couple of hours on a VAX 11/750, or three times as long on a Sun 3/50 workstation for just a couple of seconds of simulated time.

3 The Multiprocessor Testbed

The multiprocessor testbed used for the ATM simulator is based on a network of Inmos transputers. This was originally designed for use as a high speed circuit-switched network simulator, with code written in occam; subsequently, a traditional packet-switched network simulator was also developed using the same language [9, 10]. The transputer network consists of up to 31 simulation transputers, each with up to 16 Mbytes of memory (the current implementation consists of 13 T800 processors each with 1Mbyte of memory). The transputers are connected with a double layer of C004 cross-point link switches which enables any link on each of the simulation processors to be connected to a link on any of the other processors; this flexibility enables the network to be configured in arbitrary topologies so that the system being simulated can be mapped closely onto the processor network, and enables the path length required when passing messages between processors to be kept to a minimum. Finally, a layer of control processors are used to connect between the host transputer and the link switches; one is connected to the link-switch programming interface, while both can be connected, via the switches, to any of the simulation transputers. An optional transputer-based graphics card can also be connected at this layer.

4 The Software Architecture

To isolate the simulation model, as far as possible, from the implementation details of the hardware, the simulator was structured in a hierarchical manner; each layer builds on the abstraction of the layer below in a similar approach to that of the ISO seven layer model. At the lowest layer lie the transputer processors in a dynamically reconfigurable array. On top of this a multiplexor task on each processor provides the abstraction of virtual channels between each task in the simulation, regardless of where the tasks are mapped in the processor network. A simple packetizer layer hides the fact that the channels in the multiplexor (and, indeed, the physical channels of the transputer itself) work most efficiently when presented with large packets as opposed to a series of very small ones. A synchronization layer uses the packet layer processes; it ensures that each message is correctly marked with a time-stamp on dispatch and uses this at the receiver to maintain synchronization: the layer is optional, if there is no definable synchronization between two tasks (for example, diagnostic messages destined for the console) then the channel can be declared asynchronous and the packet layer accessed direct. Finally, in parallel with the simulation model and the synchronization layer, an event manager is responsible for scheduling components of the simulation model in the correct sequence. The overall hierarchy is summarised in figure 1. Further implementation details can be found in [11].

5 The Synchronization Mechanism

In a sequential discrete event simulation, the synchronization of the simulation is maintained by manipulation of a data structure called the event list. This contains the pending events in the system in time-stamped order. The simulation progresses by removing the event with the earliest time-stamp from the list and processing it. If another event is generated, it is inserted into the event list at its time-stamp position. Thus the simulator processes the events in synchronized chronological order. If we now distribute the simulation over several processors, each having a local event list, it becomes possible for a processor to process an event which is not the earliest. Also, in processing this event we may affect conditions for earlier, as yet unsimulated events. Thus the future is affecting the past, which is clearly unacceptable, and is known as a *causality error*.

Thus, synchronization schemes can fall into one of two categories; conservative approaches and optimistic approaches, see [3] for a fuller explanation of these terms. Conservative approaches avoid *causality errors* ever occurring by relying on some strategy of determining events which are "safe" to process. That is, they must determine when all events that could affect the event in question have been processed. An added problem which categorises various conservative approaches is that of deadlock. If processes do not have a "safe" event which they can process then they are blocked and cannot progress. If a cycle of blocked processes occurs then we have deadlock and the simulation will grind to a halt unless the deadlock can be broken. NULL messages can be used to avoid deadlock situations occurring. NULL messages are only used for synchronization purposes and do not correspond to any activity in the physical system being simulated and, hence, have no message content only a time-stamp t_{Null}. Thus, it is essentially a promise that the sending process will not send a real message to the destination process with a time-stamp less than t_{Null}. NULL messages are sent on each outgoing port whenever a process finishes processing an event; t_{Null} being a lower bound on the time-stamp of the next outgoing message on each outgoing port calculated from the time-stamp value associated with each incoming port and knowledge of the simulation performed by the process. Generally conservative synchronization approaches can achieve good performance with sparsely connected systems which have less opportunity for deadlock and/or an application which contains good lookahead properties. Lookahead refers to the ability to predict what will, or will not, happen in the simulated time future based on application specific knowledge.

For an ATM link it is possible to derive a simple formula that describes the number of cells that will be in transit across a given length at any one time (the link can be considered as a delay line):

$$N = \frac{LSn}{lc}$$

where L is the length of the link, S is its speed (adjusted to account for overheads such as framing), n is the refractive index of the transmission media (typically, about 1.5 for a glass fibre), l is the cell size and c is the speed of light. Considering, for example, a 15 km link running at 150 Mbit/s, then there may be up to twenty-six cells in transmission across the link at any time; longer, or faster, links would have correspondingly larger numbers of cells in transit. This "pipeline" is used to advantage as a method of lookahead within the simulator. Effectively, a destination task can see a small amount of future behaviour for the link: this can then be exploited for two ends; the avoidance of deadlock with fewer NULL messages and the improvement of concurrency between the processes.

6 The Simulator Results

The results produced by the simulator consists of sets of statistics for the simulator performance, the traffic patterns and the switching-node activity. The simulator performance can be assessed from the run time, processor usage and link usages. The performance of the synchronization mechanism is also monitored, along with several other aspects, by an event profiling process. This gives the number of instances and and percentage processing time spent on various simulation events. Such profiling is made easier as the transputer has hardware timers which allows the profiler to be run at fixed time intervals. Traffic patterns are reported as a set of histograms of the voice delay statistics for each source in the network. Switching-node activities are also reported as histograms of the input queue lengths to the Orwell rings, the ring reset and cell delay statistics.

7 Performance Analysis of the Simulator

With parallel simulation, the ultimate goal is to obtain a simulator that runs as fast as possible; if the speed of the parallel simulator is less than that of a sequential simulator then there is no reason for using it (and many good reasons for not doing so). However, it is normally impossible to directly compare parallel and conventional simulators since the two are written in an entirely different manner and the programmer rarely wants to write both. A good indication of the possible behaviour of the conventional simulator can sometimes be obtained, though, by running an optimised version of the distributed simulator on a single processor. The time taken for the single processor version to run can be compared with that for the multiprocessor version and the *speed-up* of the simulator is then the ratio of the time for the multiprocessor version to that for the single processor: normally this should lie in the range between one and n, when the multiprocessor version is run on n processors; a speed-up of n is said to be *linear*.

The performance results given here are for the ATM network simulator modelling the network shown in Figure 2: the network consists of four ATM exchanges in a fully connected trunk network and eight "local" exchanges each of which is dual-parented onto two trunk exchanges; each local exchange has two traffic generators. The exchanges were all running the Orwell ring protocol and the traffic generated was voice only. The

ring speed was 600 Mbits/s and the link speed was 150 Mbit/s; the propagation delay on all the links was set to 1×10^{-4} s (equivalent to about 20 km of glass fibre, or about 35 cells). The simulation time was 7.5 seconds with the statistics reset after 2.5 seconds. Two single processor simulations were run for each load: one with identical code to the multiprocessor version; the other an optimised version with the redundant multiplexors removed to speed message transfer. In the following graphs the load is shown expressed as the average percentage capacity of a single link.

Figure 3 shows the speed-up of the simulator as a function of load; it shows that, even for a load of just 15% of maximum capacity, the speed-up is approaching the ideal value of twelve for the unoptimised version, and is starting to level out at just over ten when compared with the optimised version. The difference between the two curves represents the proportion of the processing time that is taken up in switching the messages from one processor to another. The speed-up of the simulator relative to the unoptimised version can also be estimated from the CPU activity monitoring of each of the transputers in use: the results from doing this agree well with the upper curve shown. Unfortunately, the capacity of the trunk exchanges was not sufficient to permit higher loads than 50% to be simulated. It can be seen from figure 4, that the NULL message ratio remains very low for a large range of the load. This indicates, along with the impressive speed-up figures, that the synchronization mechanism works very efficiently.

8 Conclusions

The simulation of ATM networks at the cell-level would be prohibitive on anything but the highest performance sequential uniprocessor. The work described in this paper has shown that parallel simulation of such networks is not only possible but yields good performance for relatively little cost in hardware.

Acknowledgements

The authors would like to thank British Telecom Research Laboratories and the Science and Engineering Research Council for their technical and financial support of this work.

References

[1] R. W. Earnshaw and P. Mars, "Footprints for Mobile Communications," in *Proceedings of the eighth IEE UK Teletraffic Symposium*, pp. 22/1–22/5, April 1991.

[2] A. Hind, "Parallel Simulation for Performance Modelling of Telecommunication Networks," in *Proceedings of the Eighth IEE UK Teletraffic Symposium*, pp. 9/1–9/6, April 1991.

[3] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, pp. 30–53, October 1990.

[4] CCITT: COM XVIII, *228-E.* Geneva, March 1984.

[5] R. Händel and M. N. Huber, *Integrated Broadband Networks: An Introduction to ATM-based Networks.* Addison-Wesley, 1991.

[6] R. M. Falconer, J. L. Adams, and G. M. Walley, "A Simulation Study of the Cambridge Ring with Voice Traffic," *British Telecom Technology Journal*, vol. 3, April 1985.

[7] J. L. Adams and R. M. Falconer, "Orwell: A Protocol for Carrying Integrated Services on a Digital Communictions Ring," *Electronics Letters*, vol. 20, pp. 970–971, November 1984.

[8] R. M. Falconer and J. L. Adams, "Orwell: A Protocol for an Integrated Services Local Network," *British Telecom Technology Journal*, vol. 3, October 1985.

[9] S. J. Nichols, *Simulation and Analysis of Adaptive Routeing and Flow Control in Wide Area Communication Networks.* PhD thesis, University of Durham, March 1990.

[10] R. T. Clarke, S. J. Nichols, and P. Mars, "Transputer-based Simulation Tool for Performance Evaluation of Wide Area Telecommunications Networks," *Microprocessors and Microsystems*, vol. 13, pp. 173–178, April 1989.

[11] R. W. Earnshaw and P. Mars, "Simulation of ATM Networks on Transputer Arrays," in *Proceedings of the Seventh IEE UK Teletraffic Symposium*, pp. 1/1–1/5, April 1990.

Figure 1

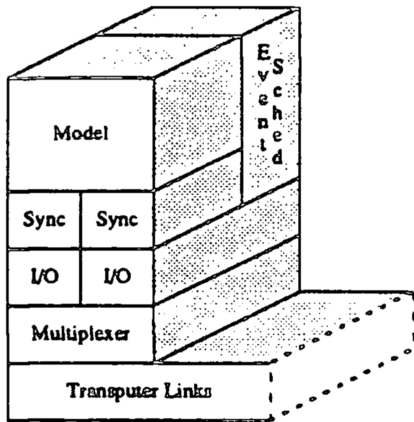


Figure 2

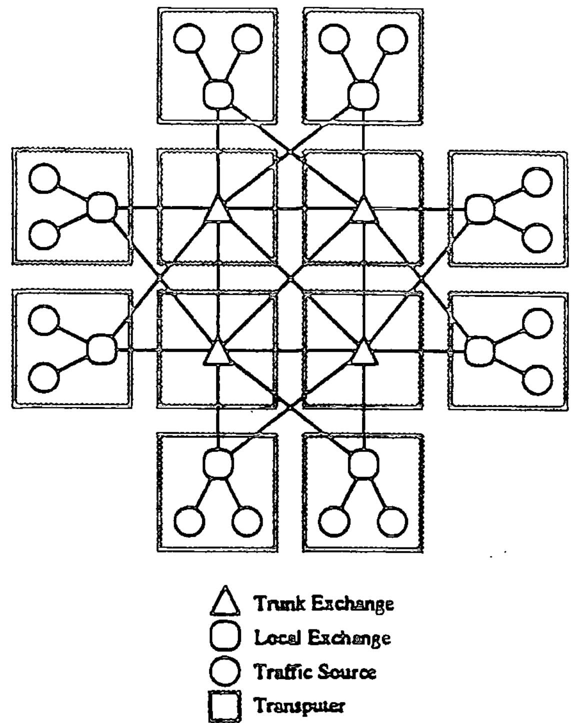


Figure 3

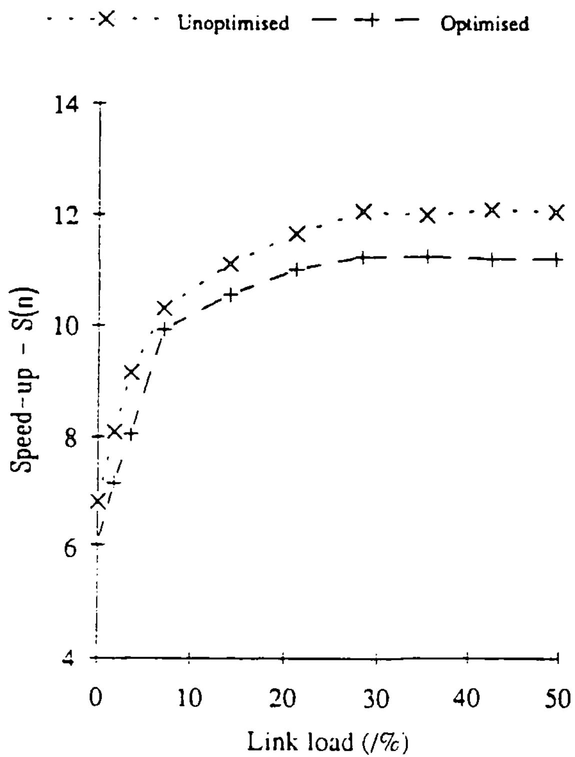
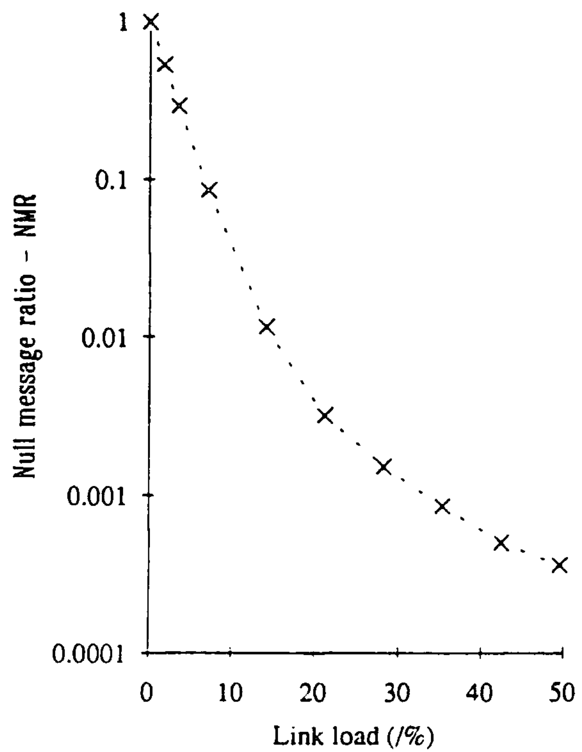


Figure 4



Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

A PARALLEL SIMULATOR FOR PERFORMANCE MODELLING OF BROADBAND TELECOMMUNICATION NETWORKS

Richard W. Earnshaw

Computing Laboratory
University of Cambridge
Cambridge, CB2 3QG, U.K.

Alan Hind

School of Engineering and Computer Science
University of Durham
Durham, DH1 3LE, U.K.

ABSTRACT

This paper describes the structure of a parallel simulator developed to investigate the performance of broadband telecommunication networks. The simulator hardware is based on a reconfigurable array of Inmos transputers. The software has a layered architecture and issues of efficient communication, deadlock avoidance and message routeing have been addressed. The synchronization mechanism used is conservative, based on the Chandy-Misra model, and exploits lookahead. The speed-up results are almost linear when compared with the same parallel simulation run on a single transputer and are still impressive when compared with an optimized single processor version.

1 INTRODUCTION

The performance evaluation of telecommunication networks rapidly becomes analytically intractable as the complexity of the network increases. In addition to this, the behaviour of interest to the performance engineer is often that which occurs under transient conditions, such as traffic fluctuations or component failures which are known to be difficult to express mathematically.

Under such conditions the use of simulation techniques to determine relevant performance parameters becomes necessary. Conventional sequential simulations running on sequential computer architectures suffer from limitations imposed by the excessive processing time required to achieve the required depth of information and the intrinsic statistical nature of the results. These problems increase as functions of the traffic intensity and the size and complexity of the network. This leads to detailed simulations, of traffic intensive and very large networks, often being economically and even physically impossible to implement. Such problems have led to growing interest in parallel simulation using multiprocessor hardware.

The argument becomes most pointed when considering the performance evaluation of broadband networks. The complexity, traffic intensity and the potential size of the networks are all large. Simulation studies of systems of this nature have thus far largely centred on the behaviour of single traffic sources, multiplexers or switching nodes. When complete networks have been studied it has usually been at the call- or burst-level since lower-level simulation involves levels of complexity (and hence processing time) which are orders of magnitude greater.

Nevertheless, for many investigations of network behaviour, lower-level simulation is unavoidable and has motivated the development of a parallel multiprocessor simulator by the University of Durham Telecommunication Networks Research Group. The simulator is being used for the study of network behaviour and, particularly, for studying the integration of mobile communication protocols into the broadband environment (Earnshaw and Mars 1991).

The use of parallel simulation introduces many additional issues into the simulation design process. These include the hardware architecture, the decomposition approach used to produce the parallel software processes, mapping these processes onto the processors and the synchronization of the resulting parallel simulation. Reviews of the application of parallel simulation techniques to the performance evaluation of communication networks have been written by Mouftah and Sturgeon (1991) and by Hind (1991). An excellent general review of parallel simulation has been written by Fujimoto (1990).

2 BROADBAND NETWORKS

The CCITT define an integrated services digital network (ISDN) as one "... that provides end-to-end digital connectivity to support a wide range of services, including voice and non-voice services, to which users have access by a limited set of standard multi-purpose user-network interfaces" (CCITT 1984). Initially, *basic access* was centred around two 64 kbits/s B channels and one 16 kbits/s signalling D channel. The requirement for supporting more advanced multi-media services within ISDN has led to the development of broadband ISDN (B-ISDN).

The asynchronous transfer mode (ATM) is the target solution for B-ISDN defined by the CCITT. ATM networks use a fixed-size data packet, known as a cell, which consists of 48 octets of data and 5 octets of header. They are typically transmitted, within the network, using multi-megabit-per-second media, such as fibre-optic links; such links will typically be running at data rates in excess of 150 Mbit/s. A good background text on B-ISDN, and

ATM networks in particular, has recently been published by Händel ıd Huber (1991).

The ATM switch used in the simulation study is based on the Orwell ring protocol which is a slotted ring protocol described by Chauhan, King and Micallef (1990). The ring is divided into slots which circulate around the ring; a node wishing to transmit a message waits until an unfilled slot is found, changes the slot header and transmits the message in the body of the slot. Slotted ring protocols have been unpopular in the past for several reasons. A monitor node is required to ensure that slots which become corrupted can be identified and regenerated, thus correct behaviour of the ring is critically dependent on correct behaviour of the monitor. To get a reasonable number of slots onto the ring delays have to be inserted at each node and one node, normally the monitor, has to be able to adjust its delay so that there are an integral number of slots. Finally, the efficiency of slotted rings is generally poor since the ratio of header to body is normally high. Its greatest advantage over token-based protocols, however, is that more that one node can be transmitting information at a time, using different slots on the ring. Acknowledgement of delivery is normally made by releasing the slot at the source (correct receipt there is taken to imply correct delivery at the destination); the node may not refill a slot that it has just released, ensuring that the slot is passed to the next node and thereby ensures fair access to all nodes on the ring. An earlier implementation of a slotted ring is the Cambridge ring protocol (British Standard BS6531).

Examination of existing protocols has indicated that those based on a slotted ring are probably the best suited for carrying delay-sensitive traffic such as speech, but simulation studies of high-bandwidth Cambridge Rings have indicated that there are still significant limitations when operated under high load (Falconer, Adams and Walley 1985). Further, load control is difficult since there is no relevant parameter that can easily be extracted from the ring. The Orwell protocol was developed after making a detailed study of the limitations of the Cambridge Ring protocol: it was found that by introducing destination release of slots, and by adding a novel, distributed, load control mechanism to bound access delays, a viable level of performance could be obtained. These developments are discussed by Adams and Falconer (1984) and Falconer and Adams (1985). For higher capacity networks multiple, synchronized, rings can be used and such a network is known as an Orwell Torus.

Whilst detailed simulations of a single Orwell ring have been made, under a variety of load and traffic services, there has, as yet, been very little investigation made into the behaviour of an Orwell torus, or ring behaviour in multi-ring systems. The reason for this, at least in part, is because of the large amount of simulation time required to investigate networks of Orwell rings: a single simulation run of one ring takes, typically, a couple of hours on a VAX 11/750, or three times as long on a Sun 3/50 workstation for just a couple of seconds of simulated time.

3 SIMULATOR ARCHITECTURE

3.1 The Multiprocessor Testbed

The multiprocessor testbed used for the ATM simulator is based on a network of Inmos transputers. This was originally designed for use as a high-speed circuit-switched network simulator, with code written in occam; subsequently, a traditional packet-switched network simulator was also developed using the same language (Nichols 1990 and Clarke, Nichols and Mars 1989). The transputer network consists of up to 31 simulation transputers, each with up to 16 Mbytes of memory (the current implementation consists of 13 T800 processors each with 1Mbyte of memory). The transputers are connected with a double layer of cross-point link switches which enables any link on each of the simulation processors to be connected to a link on any of the other processors; this flexibility enables the network to be configured in arbitrary topologies so that the system being simulated can be mapped closely onto the processor network, and enables the path length required when passing messages between processors to be kept to a minimum. Finally, a layer of control processors are used to connect between the host transputer and the link switches; one is connected to the link-switch programming interface, while both can be connected, via the switches, to any of the simulation transputers. An optional transputer-based graphics card can also be connected at this layer.

3.2 The Software Architecture

To isolate the simulation model, as far as possible, from the implementation details of the hardware, the simulator was structured in a hierarchical manner; each layer building on the abstraction of the layer below in a similar approach to that of the ISO seven layer model. At the lowest layer lie the transputer processors in a dynamically reconfigurable array. On top of this a multiplexer task on each processor provides the abstraction of virtual channels between each task in the simulation, regardless of where the tasks are mapped in the processor network. A simple packetizer layer hides the fact that the channels in the multiplexer (and, indeed, the physical channels of the transputer itself) work most efficiently when presented with large packets as opposed to a series of very small ones. A synchronization layer uses the packet layer processes; it ensures that each message is correctly marked with a time-stamp on dispatch and uses this at the receiver to maintain synchronization: the layer is optional, if there is no definable synchronization between two tasks (for example, diagnostic messages destined for the console) then the channel can be declared asynchronous and the packet layer accessed direct. Finally, in parallel with the simulation model and the synchronization layer, an event manager is responsible for scheduling components of the simulation model in the correct sequence. The overall hierarchy is summarised in figure 1. The implementation is described by the authors in more detail else-

where (Earnshaw and Mars 1990, Hind 1990 and Earnshaw 1992).

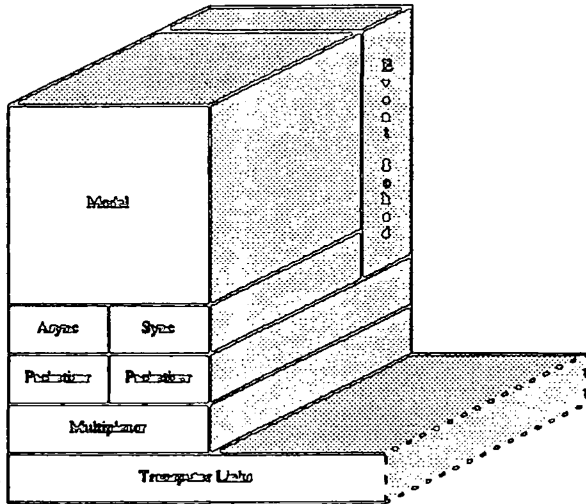


Figure 1: The overall hierarchy of the simulation model. The Event scheduler is a control-plane for the upper layers.

3.2.1 The Multiplexer

The multiplexer is the lowest layer of the simulator kernel; it is responsible for the delivery of messages from one task in the simulator to another, regardless of the topological mapping of either the tasks or the processors upon which they are running. Each transputer in the network is allocated exactly one multiplexer task; all other tasks desiring to communicate with tasks on another processor do so by communicating indirectly via the multiplexer (figure 2). If two tasks that communicate are on the same processor then it is, of course, possible for them to be directly connected. The result is that for large simulations a simulation task may have many of its channels connected to the multiplexer; the routeing decisions that the multiplexer makes are based solely upon the channel from which each message is received.

3.2.2 The Flow Control Mechanism

The flow control mechanism has to ensure two things: firstly that the multiplexer routeing, as a whole, can operate within a fixed amount of memory, i.e. a finite number of buffers (deadlock free); and secondly that all messages will be eventually delivered, regardless of the other traffic in the multiplexer (livelock free). The algorithm adopted for the implementation of the deadlock- and livelock-free routeing is based on that of Toueg and Ullman (1979), using a forward state controller.

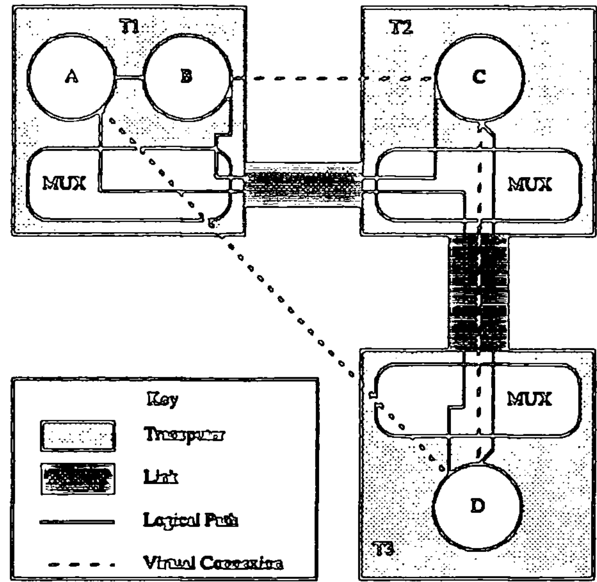


Figure 2: Multiplexer processes run on each node to provide virtual links between each task in the simulator.

3.2.3 The Packetizer

Messages between the simulation tasks commonly consist of several small pieces of information: for example, a cell has associated with it not only the time of transmission and the data and header fields but also the time of creation, the size of the data field in use (for efficiency) and an optional series of trace information packets that can be used when debugging the simulator. If each item were to be transmitted individually across the processor network then the efficiency of the multiplexer would be extremely poor; each packet in the multiplexer would contain perhaps as little as four bytes of information and an overhead of eight bytes (four bytes for each of the packet-header and the packet-size fields). To overcome this inefficiency, each simulation message (e.g. a cell) is concatenated into a single packet (or a series of packets if this would exceed the maximum size of a single multiplexer packet) which is then transmitted to the receiving process.

In addition to the inefficiency associated with using the multiplexer with small units of data there would also be an overhead due to the establishment of the occam channel for passing data between one task and the next. Each communication requires that both ends (the sender and the receiver) are ready to proceed before any data can be sent: if one end is not ready the other task blocks whilst waiting. Because of the way in which the transputer's process scheduler works this can mean a large number of process switches, each switch having an overhead in terms of processor time; in addition, each time a process is descheduled it is placed at the back of the relevant queue (either high or low priority) and has to wait its

turn for further access to the processor. It is clearly more efficient if the number of times a channel communication has to be initiated is kept to a minimum; work by Gould, Bowler and Purvis (1989) has shown that the throughput of the channels increases dramatically as the size of the data block is increased.

3.2.4 The Event Manager

The event list is normally maintained using the twin-list method described by Blackstone, Hogg and Phillips (1981), but it is possible to convert the procedures to be functionally the same as a single list manager by setting the initial length of the first list to infinity. It was found that for the T4 series of transputers (which do not support floating-point arithmetic in hardware), using the twin list method approximately halved the amount of time spent maintaining the event list, but for the T800 transputer (which does support floating-point arithmetic) the change was negligible; indeed, for some configurations, the twin list procedure was slower by about 0.5%.

3.2.5 Configuring the Simulator

For any simulation tool to be useful it must be capable of being run with a series of different configurations, the extent of which has to be borne in mind when the simulator is designed. For a truly flexible system it is not normally sufficient for these to be parameters that are 'hard coded' into the simulator itself; instead, they should be made available from a separate file (or by interactive prompting) at the time the simulator is invoked. In the ultimate case, not only parameters such as load and various delays should be configurable, but also the entire topology of the network itself: this can require substantial effort being expended on making the simulator easier to use, but, consequently, significantly more powerful.

The method employed here is a parse-able grammar that describes the simulation parameters (and some of their dependencies) in a human comprehensible format: in such environments it is rarely necessary for the information to be in a totally fixed order since each parameter will have a tag associated with it that describes it uniquely. Comments are normally easily supported. An example entry might contain:

```
link 5:
% Link between nodes 1 and 6
      prop_delay = 100u S
      speed = 100M bitps
      ;
```

Parsers for grammars of this type are easily produced sing tools such as yacc and lex and would, probably, be nplemented using a preprocessor for the simulator that roduces the configuration tables that the simulator itself ads. Another advantage of this approach is that default lues can now be used: a special entry (for example 'link fault:') might contain a series of fields that should be ed when a real definition omits a parameter.

The ATM Network Simulator currently parses two files when it starts to run: the first describes the topology of the network being simulated and how the individual processes should be mapped onto the processors of the transputer network; the second contains the various parameters required by each individual process. Both files are of the 'table of values format'. A parser is available for generating the first file that understands a superset of the *3L configurer* language (Parallel C User Guide 1989); the extensions are mainly aimed at supporting the reconfigurability of the transputer array used. The second file has to be generated by hand, but a built-in preprocessor parses the special symbols '%date' and '%seed', replacing them with the current date and an unique seed respectively. The seeds are generated using a different random number generator from the one used during simulation in order to avoid, as far as possible, correlations between the random number streams.

The compiler package supplied by 3L Ltd is described in the Parallel C User Guide (1989) and consists of three main components for use with multi-transputer networks: the compiler, which produces object modules from the source files; a linker, which links object modules and libraries to create tasks; and a configurer, which binds several tasks together to form an executable application. A task is a program in its own right: it is allocated a stack and an area of memory, and has its own global variables; it must always run on one processor, but can spawn *threads* which execute part of the code of the task in parallel and share the memory (they each, however, have their own stack); a task can only communicate with other tasks by using the occam channels implemented in the processor hardware: the collection of program threads in a task are collectively referred to as a process. The configurer is responsible for allocating tasks to processors, creating initial stacks and heap areas, and for mapping the connexions between tasks onto occam channels (both internal and external).

Unfortunately the configurer supplied with the compiler does not support the link-switch mechanism in the transputer network used and, therefore, cannot be used in the traditional sense to boot the entire network. The approach used in the simulator, is to have a small main application, which runs on the fixed topology part of the network, and a series of un-configured tasks. The main application does on-the-fly configuration of the remainder of the application using a single file that describes the simulation run. To do this it uses the low-level configurer execution primitives to load the tasks directly into each processor.

Once each task has been loaded and has started to run, the simulation parameter files have to be loaded. Unlike traditional simulators this poses a large problem: part of the information contained in the parameter file is used by the multiplexers to control the switching of messages; until this is loaded they cannot operate properly. Similarly, none of the other tasks knows any information about where it lies in the overall topology, since to provide this information would require 'hard coding'. Indeed, the only

information that each process has is its own array of channels for use in communicating, but even this has little meaning unless some conventions are used. Fortunately, 'false' channels can be created during the configuration process and their values set to represent something other than a genuine channel. With this information, known as a 'tag', each task in the simulator can be uniquely identified, enabling it to extract the relevant information from the parameters file.

At this stage a task still does not know on which input channel it will receive the configuration information; further, it does not know on which output channels, if any, it must forward the information so that it can reach its neighbours. To obtain this information a boot-tree is built which starts at the task connected to the fixed topology part of the network (there is exactly one such task) and extends outwards until all the tasks know a parent and any children they might have. The protocol for doing this in the presence of loops is quite complex if the use of timeouts are to be avoided; the petri-net in figure 3 represents the code running on just one channel pair of one task (all of the channels in the simulator are paired, one input and one output, to the same remote task), the same code runs on each channel pair throughout the simulator.

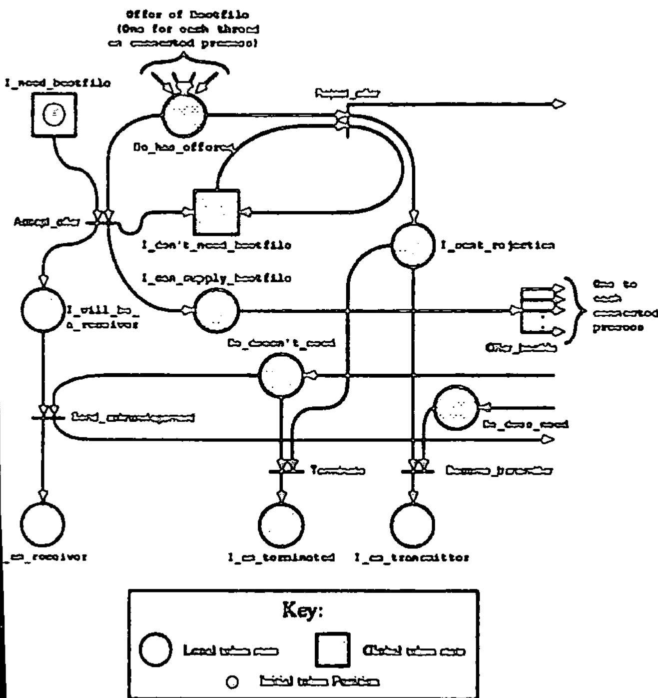


Figure 3: Petri-net showing the state transitions for a single channel while determining the download path for the simulator. The 'square' states are shared by all of the channels, which have been omitted for clarity, making it impossible for more than one channel to be activated as a receiver.

The parameters file contains a few lines of global information, such as the title of the simulation run, the size of the network, and for how long the run must last, followed by a series of entries, one for each task in the simulator. To avoid the need for each task to have to be able to interpret information for other tasks (which may well be of a different class), each task scans the parameters file looking for a string of the form 'class xxx:', where the class is the type of task ('SRCE' for a traffic generator, 'MUX' for a multiplexer, etc.) and xxx is the tag-value that was bound to the false link. On finding this string, the task then interprets the following parameters as its personal configuration file. Special routines are used to parse the file while ensuring that at the same time the entire file is passed on to its children in the boot-tree without modification or loss. Once the entire file has been read and interpreted, the configuration process is complete and the simulation can begin.

4 THE SYNCHRONIZATION MECHANISM

In a sequential discrete event simulation, the synchronization of the simulation is maintained by manipulation of a data structure called the event list. This contains the pending events in the system in time-stamped order. The simulation progresses by removing the event with the earliest time-stamp from the list and processing it. If another event is generated, it is inserted into the event list at its time-stamp position. Thus the simulator processes the events in synchronized chronological order. If we now distribute the simulation over several processors, each having a local event list, it becomes possible for a processor to process an event which is not the earliest. Also, in processing this event we may affect conditions for earlier, as yet unsimulated events. Thus the future is affecting the past, which is clearly unacceptable, and is known as a *causality error*.

Thus, synchronization schemes can fall into one of two categories; conservative approaches and optimistic approaches, see Fujimoto (1990) for a fuller explanation of these terms. Conservative approaches avoid causality errors ever occurring by relying on some strategy of determining events which are "safe" to process. That is, they must determine when all events that could affect the event in question have been processed. An added problem which categorises various conservative approaches is that of deadlock. If processes do not have a "safe" event which they can process then they are blocked and cannot progress. If a cycle of blocked processes occurs then we have deadlock and the simulation will grind to a halt unless the deadlock can be broken. In the Chandy-Misra conservative approach used here (Chandy, Holmes and Misra 1979, Chandy and Misra 1979 and Chandy and Misra 1981), NULL-messages are used to avoid deadlock situations occurring. NULL-messages are only used for synchronization purposes and do not correspond to any activity in the physical system being simulated and, hence, have no message content only a time-stamp t_{Null}.

Thus, it is essentially a promise that the sending process will not send a real message to the destination process with a time-stamp less than t_{Null}. NULL-messages are sent on each outgoing port whenever a process finishes processing an event; t_{Null} being a lower bound on the time-stamp of the next outgoing message on each outgoing port calculated from the time-stamp value associated with each incoming port and knowledge of the simulation performed by the process. Generally conservative synchronization approaches can achieve good performance with sparsely-connected systems which have less opportunity for deadlock and/or an application which contains good lookahead properties. Lookahead refers to the ability to predict what will, or will not, happen in the simulated time future based on application specific knowledge.

For an ATM link it is possible to derive a simple formula that describes the number of cells that will be in transit across a link of given length at any one time (the link can be considered as a delay line):

$$N = \frac{LSn}{lc}$$

where L is the length of the link, S is its speed (adjusted to account for overheads such as framing), n is the refractive index of the transmission media (typically, about 1.5 for a glass fibre), l is the cell size and c is the speed of light. Considering, for example, a 15 km link running at 150 Mbit/s, then there may be up to twenty-six cells in transmission across the link at any time; longer, or faster, links would have correspondingly larger numbers of cells in transit. This "pipeline" is used to advantage as a method of lookahead within the simulator. Effectively, a destination task can see a small amount of future behaviour for the link: this can then be exploited for two ends; the avoidance of deadlock with fewer NULL-messages and the improvement of concurrency between the processes.

In the Chandy-Misra simulation, there is not normally an event processor in the classical sense. Instead, events are replaced exclusively by messages and the order of processing is determined by selecting the message with the oldest time-stamp: there must be a message available from each incoming link in order to be able to do this; the absence of a message causes the node to block. In the ATM Network Simulator an event manager is used; consequently, in addition to adding dependence on the link mechanisms to the code of the event manager, monitoring for messages would be inefficient. To overcome this, the synchronization routines are implemented as normal events that run in the same manner as all other events in the simulator: two events are required for each link to a remote process; these are a NULL-message generator and a process blocker.

The NULL-message generator runs on the output of a link: it compares the current simulation time with the time when a message was last sent to the remote process; if this is less than a propagation delay it simply reschedules itself to a time one propagation delay later than the time at which the last message was sent; otherwise, it must be exactly one propagation delay since a message was last sent, so a NULL-message is generated to the remote process and the generator reschedules itself one propagation delay later. The process blocker compares the simulation time against the time when a message was last received across a link from the remote process; if this is less than a propagation delay then it simply reschedules itself for one propagation delay after the time the last message was received; otherwise it blocks the current process until a message is received and then reschedules itself accordingly. The process blocker appears to the rest of the simulation as a routine that takes just sufficiently long to execute that the process remains in synchronization with its neighbours; however, while blocking, it consumes no processing time.

5 THE SIMULATOR RESULTS

The results produced by the simulator consists of sets of statistics for the simulator performance, the traffic patterns and the switching-node activity. The simulator performance can be assessed from the run time, processor usage and link usages. The performance of the synchronization mechanism is also monitored, along with several other aspects, by an event profiling process. This gives the number of instances and and percentage processing time spent on various simulation events. Such profiling is made easier as the transputer has hardware timers which allows the profiler to be run at fixed time intervals. Traffic patterns are reported as a set of histograms of the voice delay statistics for each source in the network. Switching-node activities are also reported as histograms of the input queue lengths to the Orwell rings, the ring reset and cell delay statistics.

6 PERFORMANCE ANALYSIS OF THE SIMULATOR

The ultimate goal with parallel simulation is to obtain a simulator that runs as quickly as possible; if the speed of the parallel simulator is less than that of a conventional simulator then there is no reason for using it (and many good reasons for not doing so). However, it is normally impossible to directly compare parallel and sequential simulators since the two are written in an entirely different manner and the programmer rarely wants to write both. A good indication of the possible behaviour of the conventional simulator can sometimes be obtained, though, by running an optimized version of the parallel simulator on a single processor. The time taken for the single processor version to run can be compared with that for the multiprocessor version and the *speed-up* of the simulator is then the ratio of the time for the multiprocessor version to that for the single processor: normally this should lie in the range between one and n, when the multiprocessor version is run on n processors; a speed-up of n is said to be *linear*, as defined by Helmbold and McDowell (1990). If the speed-up is greater than n we have

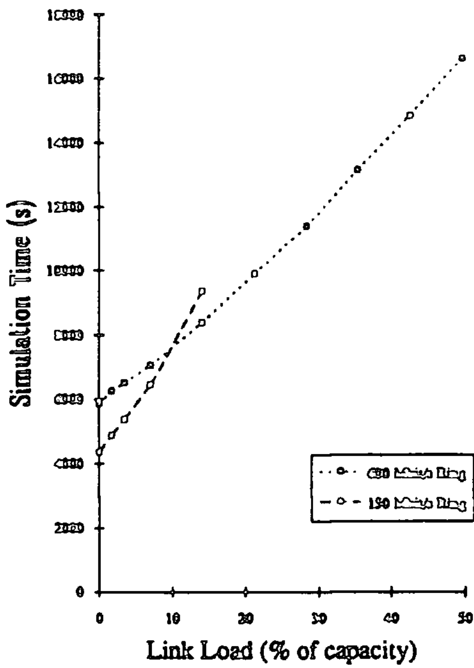


Figure 4: Simulation times for the twelve-node networks of 150 and 600 Mbit/s rings on twelve transputers.

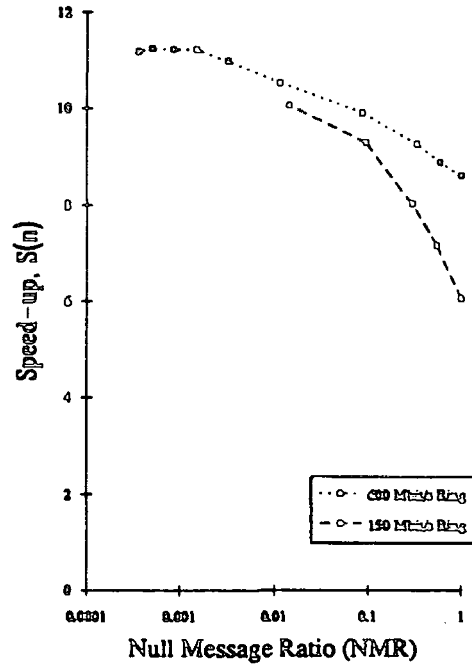


Figure 6: Speed-up as a function of NULL-message ratio. The difference between the two curves represents the extra parallelism that can be extracted from the higher speed rings.

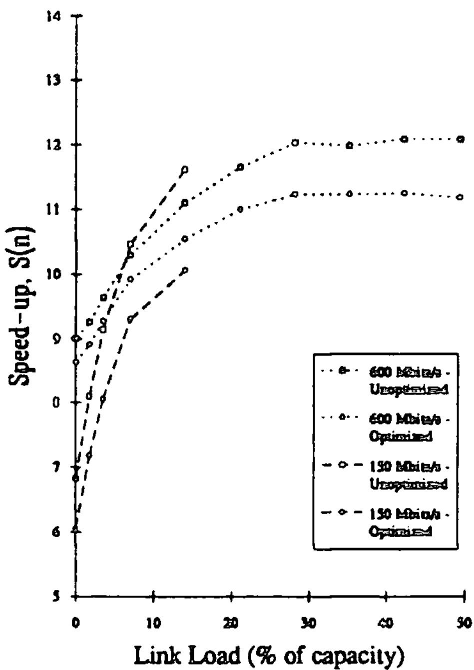


Figure 5: Speed-up for the 150 Mbit/s rings carrying mixed mobile and voice traffic and the 600 Mbit/s rings carrying voice traffic only.

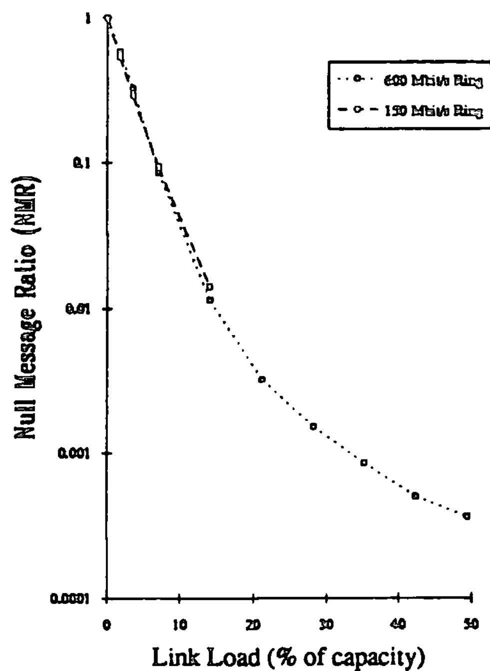


Figure 7: NULL-message ratio (NMR) as a function of load. As might be expected, the ratio is independent of the ring speed.

superlinear speed-up and, if the speed-up is less than n, we have *sub-linear speed-up*.

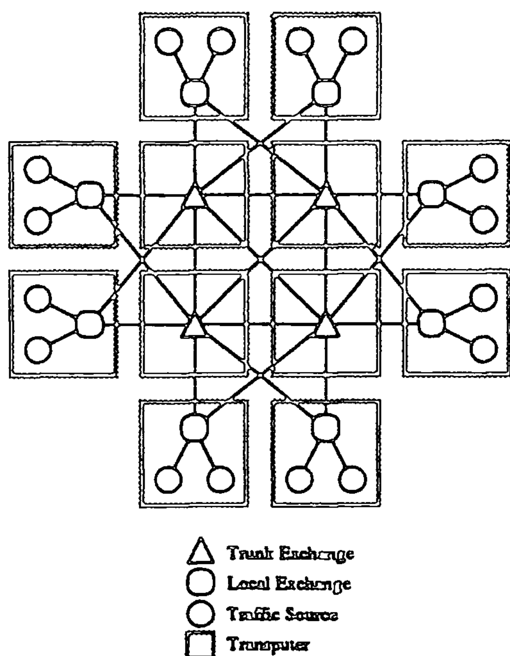


Figure 8: Network topology used for the simulator performance analysis runs. The basic processor assignments are also shown; a traffic source presents a very small load to a processor so it may be safely combined with a local exchange without unduly affecting the load balance.

The performance results given here are for the ATM Network Simulator configured as shown in figure 8: the network consists of four ATM exchanges in a fully-connected trunk network and eight 'local' exchanges each of which is dual-parented onto two trunk exchanges; each local exchange has two traffic generators. The exchanges were all running the Orwell ring protocol (see section 2). Two sets of results were taken with differing switch capacities and traffic mixes. In both cases the links were running at 150 Mbit/s and the propagation delay was set to 1×10^{-4} s (equivalent to about 20 km of glass fibre, or about 35 cells). The results for the lower traffic loads were taken using 150 Mbit/s Orwell rings for the switches and with a mixture of voice and mobile traffic; the results for the higher loads used purely voice traffic and a ring speed of 600 Mbit/s. With the smaller capacity switches the maximum link loading was about 15%, but was increased to about 50% for the high-capacity rings. Two single processor simulations were run for each load: one with identical code to the multiprocessor version, the unoptimized version; the other with the redundant multiplexers removed to speed message transfer, the optimized version. In the following graphs, when the load is shown it is expressed as the average percentage of the capacity of a link.

Figure 4 shows the time taken to simulate the two models on twelve processors. The fact that the two curves do not pass through the origin has two causes: the NULL-message traffic for low loads and the overhead of simulating the ring slot-rotation action for the Orwell protocol. That it is the latter that represents the largest factor can be inferred from the fact that the NULL-message traffic generated for each of the two curves is almost identical for a given link loading (see figure 7); so if this was the cause the two curves would cut the axis at the same point.

Figure 5 shows the speed-up of the simulator as a function of load. It shows, for the 150 Mbit/s rings, that even for a load of just 15% of maximum capacity, the speed-up is approaching the ideal linear value of 12 for the unoptimized version, and is starting to level out at just over 10 when compared with the optimized version. The difference between the two curves represents the proportion of the processing time that is taken up in switching the messages from one processor to another. The speed-up of the simulator relative to the unoptimized version can also be estimated from the processor activity monitoring of each of the transputers in use: the results from doing this agree well with the upper curve shown. Figure 5 shows, for the 600 Mbit/s rings that in comparison with the unoptimized single processor version the speed-up is greater than 9 for all loads simulated, and for link loads greater than 30% it is almost linear.

It can be seen from figure 6 that the speed-up degrades gracefully with increasing NULL-message ratio; but, fortunately, as can be seen from figure 7, the NULL-message ratio remains very low for a large range of the load.

ACKNOWLEDGEMENTS

The authors would like to thank British Telecom Research Laboratories and the Science and Engineering Research Council for both their technical and financial support of this work.

REFERENCES

3L Ltd, *Parallel C User Guide, Version 2.1*. Peel House, Ladywell, Livingston EH54 6AG, Scotland, 1989.

J. L. Adams and R. M. Falconer, "Orwell: A Protocol for Carrying Integrated Services on a Digital Communications Ring," *Electronics Letters*, vol. 20, pp. 970–971, November 1984.

J. H. Blackstone Jr, G. L. Hogg, and D. T. Phillips, "A Two-List Synchronization Procedure for Discrete Event Simulation," *Communications of the ACM*, vol. 24, pp. 825–829, December 1981.

CCITT: COM XVIII, *228-E*. Geneva, March 1984.

K. M. Chandy, V. Holmes, and J. Misra, "Distributed Simulation of Networks," *Computer Networks*, vol. 3, pp. 105–113, 1979.

K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 440–452, September 1979.

K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, pp. 198–206, April 1981.

J. Chauhan, T. King, and A. C. Micallef, *Specification of the Orwell Protocol.* British Telecom Laboratories, Martlesham Heath, Ipswich, Suffolk, UK. IP5 7RE, May 1990. Revision C.1(05/90).

R. T. Clarke, S. J. Nichols, and P. Mars, "Transputer-based Simulation Tool for Performance Evaluation of Wide Area Telecommunications Networks," *Microprocessors and Microsystems*, vol. 13, pp. 173–178, April 1989.

R. W. Earnshaw and P. Mars, "Simulation of ATM Networks on Transputer Arrays," in *Proceedings of the Seventh IEE UK Teletraffic Symposium*, pp. 1/1–1/5, April 1990.

R. W. Earnshaw and P. Mars, "Footprints for Mobile Communications," in *Proceedings of the Eighth IEE UK Teletraffic Symposium*, pp. 22/1–22/5, April 1991.

R. W. Earnshaw, *Simulation of Packet- and Cell-based Communication Networks.* PhD thesis, University of Durham, UK, May 1992.

R. M. Falconer, J. L. Adams, and G. M. Walley, "A Simulation Study of the Cambridge Ring with Voice Traffic," *British Telecom Technology Journal*, vol. 3, April 1985.

R. M. Falconer and J. L. Adams, "Orwell: A Protocol for an Integrated Services Local Network," *British Telecom Technology Journal*, vol. 3, October 1985.

R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, pp. 30–53, October 1990.

L. Gould, I. Bowler, and A. Purvis, "Real-Time, Multi-Channel Digital Filtering on the Transputer," in *Proceedings of the 1989 International Symposium on Computer Architecture and Digital Signal Processing*, 1989.

R. Händel and M. N. Huber, *Integrated Broadband Networks: An Introduction to ATM-based Networks.* Addison-Wesley, 1991.

D. P. Helmbold and C. E. McDowell, "Modeling Speedup (n) Greater than n," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 250–256, April 1990.

A. Hind, "A Multiprocessor Testbed for Parallel Simulation of Telecommunication Networks," technical report, University of Durham (SEAS), December 1990.

A. Hind, "Parallel Simulation for Performance Modelling of Telecommunication Networks," in *Proceedings of the Eighth IEE UK Teletraffic Symposium*, pp. 9/1–9/6, April 1991.

H. T. Mouftah and R. T. Sturgeon, "Distributed Discrete Event Simulation for Communications Networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 1723–1734, December 1991.

. J. Nichols, *Simulation and Analysis of Adaptive Routeing and Flow Control in Wide Area Communication Networks.* PhD thesis, University of Durham, UK, March 1990.

S. Toueg and J. D. Ullman, "Deadlock-Free Packet Switching Networks," in *Proceedings of the ACM Symposium on the Theory of Computing, Atlanta, Georgia*, pp. 89–98, May 1979.

AUTHOR BIOGRAPHIES

Richard W. Earnshaw received the degree of B.Sc. in Applied Physics from the University of Durham, UK, in 1987 and the degree of Ph.D. from the University of Durham, UK, in June 1992. His research interests include performance engineering of telecommunication networks, particularly broadband ISDN and mobile systems, and parallel discrete event simulation. He is currently a Research Assistant in the Computing Laboratory at the University of Cambridge, UK, where he is engaged on the *Fairisle* fast packet switching project.

Alan Hind received the B.Sc. degree in Electronic Communications from the University of Salford, UK, in 1982. He is currently the British Telecom Research Fellow in Parallel Simulation in the School of Engineering and Computer Science at the University of Durham, UK. His research interests include parallel discrete event simulation and the performance engineering of telecommunication networks. He is a member of the Institute of Electrical and Electronic Engineering, the Association of Computing Machinery and the Society for Computer Simulation and an associate member of the Institute of Electrical Engineers in the UK.

Performance of parallel simulators for ATM networks.

John Mellor and Alan Hind
Telecommunication Networks Research Group
University of Durham, UK.

Abstract

Recent work has concentrated on the use of parallel processing to achieve high speed simulation of Asynchronous Transfer Mode (ATM) networks. Initially a model for a network of ATM switches based upon the Orwell ring was developed. Near linear speed up of processing with the addition of further processors was achieved through the exploitation of physical network parameters during the design of the synchronisation mechanism. This technology was then applied to a burst level ATM simulator. The implementation of a conservative synchronisation scheme and the mapping of B-ISDN functions to the parallel discrete event simulator have been achieved through the design of a layered modular architecture.

Introduction

The performance evaluation of telecommunication networks becomes analytically intractable as the complexity of the network increases and transient conditions are investigated. Conventional sequential simulations suffer from excessive processing time which is required to gather sufficient data to achieve the required degree of confidence. These problems increase as the size, complexity and traffic intensity of the network increases and as behaviour of interest to the performance engineer such as component failure is modelled in greater detail. The network is modelled by multiple queues and servers operating concurrently. One approach to speeding up the simulation in this parallel environment is to use a multiprocessor array as the simulation engine.

The multiprocessor architecture described in this paper was initially developed to study, at the cell level, the detailed operation of switch behaviour, network behaviour and particularly the integration of mobile communication protocols into the Broadband environment [7].

A research programme funded by the European Commission for Research in Advanced Communication technologies in Europe (RACE) was charged with the development of an ATM simulator. The RACE simulator was designed to operate at the burst level to further speed up the simulation process [13]. Near real time generation of results and interfacing to other projects would provide an ATM network emulator. It incorporates features reflecting the growing understanding in the Telecommunication Management Network (TMN) community of the likely functioning of the Broadband Integrated Services Digital Network (B-ISDN). It may be used to prove implementation issues such as the performance improvement in switch processing that accrues from the use of virtual paths (VP).

The issues of parallel simulation have been widely studied. The hardware architecture, the decomposition of the model to produce the parallel software processes, mapping the processes onto the processors and the synchronisation have been reviewed by Mouftah and Sturgeon [15] and by Hind [11]. A general review of parallel simulation is provided by Fujimoto and Nicol [9].

Simulator Architecture

The general architecture chosen for the two simulators is basically similar. The techniques were initially proved using networks of Orwell rings [1], [8] and toruses to model a B-ISDN network at the cell level. Background information on ATM networks and cell structure can be found in the books by Handel and Huber [10] and by Prycker [17]. At the burst level of the RACE simulator there is no need to model the detailed operation of the switch. This provides opportunities to investigate higher level issues such as call acceptance functions and network management.

The simulators consist of multiprocessor simulation engines, a layered software architecture which includes the messaging system and synchronisation, and a host computer for user interaction.

Simulation engine.

The ATM simulation engine consists of a network of Inmos transputers. This was initially developed as a high speed circuit-switched network simulator with code written in OCCAM, and later as a traditional packet-switched simulator [3]. The testbed consists of up to 31 transputers each with 16 Mbytes of memory. The results given for the cell level simulator used 13 T800 processors, one with 16 Mbytes and the others with 1 Mbyte. The testbed has a double layer of cross-point link switches which enables the network to be configured in any arbitrary topology so that the system being simulated can be mapped closely onto the processor network. This also means that the path length for message passing is kept to a minimum; there is no shared memory in the system.

A requirement for the RACE simulator was that it use commercially available and supported components. The transputer arrays chosen for use in Sun and PC workstations had less flexible connectivity but the arrays were smaller and the messaging system was designed to achieve virtual full connectivity without the need to physically reconfigure switches. The transputer is an ideal building block for distributed memory multiprocessors employing message passing. It has hardware support for multitasking, concurrency and communications.

Software architecture

Commercially available software ranges from general purpose simulation languages to specific network simulation packages. The general purpose languages provide great flexibility but consume large computing resources. The network packages are often built from general purpose languages (and hence consume large amounts of computing resource) and were found to have insufficient flexibility for our purposes. The simulators were written using standard compilers. A graphical user interface was provided for the RACE simulator to enhance 'ease of use'.

The simulators have been structured in a hierarchical manner. Functionality has been separated into layers with each layer building upon the abstraction of the layer below. The lowest layer consists of the physical links between the transputers. These links are reconfigurable but are set to a given topology at the start of each simulation. Above this is a multiplexor task. This runs on each processor and provides the abstraction of virtual channels between each task regardless of where they are in the processor network. The multiplexor provides message buffering, routeing, and a virtual topology configuration that matches the simulated network. The cell level simulator contains a simple packetizer layer that reduces the communication overhead by collecting cells into larger and more efficient packets.

In the RACE simulator a processing node will typically host one or more simulated network nodes. The simulated network nodes consist of several processes and are

programmed during the initial configuration to represent switches, sources, network management, sinks and loop-back nodes.

The synchronisation layer time stamps each message on despatch and uses this at the receiver to maintain synchronisation. For control and diagnostic messages this layer can be bypassed. An event manager for each node schedules the components of the simulation model. A full description of the ATM cell level simulator is given in [5], [6], and for the RACE simulator in [14].

Synchronisation

Time in a discrete-event simulator is advanced upon the execution of each task from the event list. The aim is for the simulation to move forward monotonically in time by selecting the event that is the least distance into the future. The simulation of the event will generate zero or more further events for execution at some time in the future.

If a single event list is maintained no synchronisation is required although management of the list can be a problem. For distributed simulation the event list is maintained local to each processor and synchronisation is required to avoid causality errors. Conservative synchronisation avoids causality error by only processing the current event when there is certainty that no other event could affect it. There is a possibility that the system could deadlock by waiting indefinitely until it is safe to proceed.

Optimistic synchronisation proceeds with the local event list until a causality error occurs at which point a rollback scheme is implemented to recover from the error. A fuller explanation of these terms can be found in Fujimoto and Nicol [9].

A conservative approach was chosen because it gives good performance with sparsely connected systems which have less opportunity for deadlock. Null messages are used to avoid deadlock in the Chandy-Misra [2], [16] conservative scheme. These indicate to connected processors that no event will arrive with a time stamp less than that of the null message. Significant lookahead was possible because of the propagation delays of an ATM network. This allowed greater concurrency to be achieved with fewer null messages. It has also been shown analytically and in practice that the scheme is deadlock free [4].

Performance of the cell level simulator

The performance of the cell level simulator has been determined from results gathered during production runs. A useful network topology was constructed, a warm up period was allowed before gathering results and the simulation run was long enough for the statistics gathered to be significant. The results were used to validate the simulator, examine the performance of the Orwell switches and verify the operation of the protocol developed to manage mobile voice traffic on the network [7].

The speed-up of the cell level simulator relative to a single processor is shown as a function of load in figure 1. In all the figures, traffic load is expressed as the average percentage of the capacity of a single link. For the 600 Mbit/s rings the speed-up is almost linear with link loads of greater than 30%. This shows that the communication overheads in the parallel simulator are effectively hidden and that the synchronisation method is very efficient.

Figure 2 shows that the speed-up degrades gracefully as the null message ratio (NMR) is increased; fortunately the null message ratio remains low for a large range of the link load as can be seen in figure 3.

Further performance measurements were taken using symmetrical distances between switches of 2 km, 20 km and 200 km; and using a UK national network. The national network has the same topology but the inter-switch distances range from 51 km to 343 km and the source to switch distances from 0.5 km to 7.5 km. The national network thus has asymmetric propagation delays and lookahead. These measurements were repeated with an asymmetric traffic pattern; half of the sources transmitted and received at twice the rate of the other half. The network average load was 28.27%.

These experiments showed that lookahead was the key factor in speed-up. Figure 4 shows a large increase in speed-up between 2 km and 20 km, with a further smaller increase between 20 km and 200 km. The speed-up for the national network was comparatively low. Further results show that the number of null messages required becomes large when the lookahead is low and this reduces the speed-up (figure 5).

RACE simulator features

Following the encouraging preliminary results from the Durham cell level simulator, it was decided to utilise a similar architecture for the RACE simulator. A coarser granularity of simulation was decided upon and a burst level was used for the traffic. The simulator was targeted at several different users and this required a robust modular construction so that different models and synchronisation schemes, developed by different partners, could be used in particular configurations.

The programmable burst generator can be configured to represent one of the following traffic profiles:

> PCM coded telephony at 64 kbps leading to 171 cells per second.
>
> Tasi coded telephony talk spurts of 32 kbps exponentially distributed with a mean of 0.35 sec., and silence of 0 kbps exponentially distributed with a mean of 0.65 sec.
>
> File transfer modelled as a call that consists of one or more bursts. In the case of more than one burst the burst length and time between bursts are exponentially distributed. The bit rate for these bursts is user selectable.
>
> Video traffic is modelled as either one way as in video broadcast or two way as in video conferencing. A separate PCM telephone connection may be used for sound. The picture is characterised by bursts of 0.04 sec. duration with a mean bit rate of 22 Mbps and a maximum rate of 78 Mbps. In a three party conference there will be six video and six sound connections to the conferencing node.

All messages between concurrent processes, whether they are on the same transputer or not, are required to pass through a kernel, to simplify messaging and synchronisation. The kernel orders task execution according to simulation needs. The flow control mechanism has to ensure two things: first that the routeing can operate with a finite number of buffers, i. e. a fixed amount of memory (deadlock free); and second that all messages will eventually be delivered, regardless of other traffic (livelock free).

The RACE simulator thus has a highly structured hierarchy and parallelism is mainly achieved through the kernel module. The other modules such as traffic profiles and generation, burst processing in the switches and network management are C programmes developed and tested in a conventional serial manner. Conversion to the parallel environment often requires only minor changes such that a high degree of confidence about the correct functioning of the module is gained before it is run on the transputers.

Burst level processing allows the RACE simulator to generate results faster than real time. However events of interest to network management occur infrequently in a well designed B-ISDN due to the low bit error rate and general capabilities of the network. Further speedup in the generation of events is achieved by abnormally loading the network and setting call acceptance parameters so that they do not reject any calls. This leads to buffer overflow and cell loss. Extensive work on the effects of scaling has been conducted by Kesidis and Walrand [12] to show that when carefully applied it can give a good indication of the likely performance of a real network.

Conclusions

The results show that, when carefully optimised for a particular task, parallel simulation can provide significant advantages. The results for the production runs with the cell based simulator being particularly good. The RACE simulator was an ambitious project built using software modules from different parties. The overhead required to ensure encapsulation and interworking of the modules means that the simulator hasn't reached its potential. Sequential simulators at Durham University providing equivalent functionality currently outperform the parallel RACE simulator.

The transputer has been surpassed in terms of raw processing power by newer microprocessors and certainly for less detailed simulations it is preferable to use single sequential processors rather than transputer arrays. The attraction of a scaleable parallel processing environment remains for high speed processing, however. Newer processors are being equipped with features to support parallel operation. Alternatively hybrid combinations of powerful processors with transputers for messaging are possible. Inmos are seeking to redress the balance with the introduction of the T9000, however we await delivery of the processor at the moment.

The approaches outlined in this paper have lead to the production of simulators that can produce results faster than real time (depending on level of detail). The flow of execution in the parallel processors is controlled by completion of communications without the need for polling and interrupts which would add further processing overheads and delay. The simulators have been used to assess the performance of ATM switches, networks, call handling procedures and protocols for mobile operation.

References

[1] Adams, J. L., and Falconer, R. M. 'Orwell; a protocol for carrying integrated services on a digital communications ring'. Electronics Letters 20, 3 (November 1984), 970-971.

[2] Chandy K. M. and Misra J. 'Asynchronous distributed simulation via a sequence of parallel computations.' Communications of the ACM 24, 4 (April 1981), 198-206.

[3] Clarke R. T., Nichols S. J. and Mars, P. 'A transputer-based simulation tool for performance evaluation of wide area telecommunications networks.' Microprocessors and Microsystems 13, 3 (April 1989), 173-178.

[4] Earnshaw R. W. 'Simulation of Packet- and Cell-based Communication Networks.' PhD Thesis, University of Durham, UK, May 1992.

[5] Earnshaw R. W. and Hind, A. 'A parallel simulator for performance modelling of broadband telecommunication networks.' In proceedings of the Winter Simulation Conference (1992), 1365-1373.

[6] Earnshaw R. W. and Mars P. 'Simulation of ATM networks on transputer arrays.' In Proceedings of the Eighth IEE UK Teletraffic Symposium (April 1991), 1/1-1/5.

[7] Earnshaw R. W. and Mars P. 'Footprints for mobile communications.' In Proceedings of the Eighth IEE UK Teletraffic Symposium (April 1991), 22/1-22/5.

[8] Falconer R. M. and Adams J. L. 'Orwell; a protocol for an integrated services local network.' British Telecom Technology Journal 3, 4 (October 1985).

[9] Fujimoto R. M. and Nicol D. 'State of the art in parallel simulation.' In proceedings of the Winter Simulation Conference (1992), 246-254.

[10] Handel R. and Huber M. N. 'Integrated Broadband Networks; an introduction to ATM-based Networks.' Addison-Wesley, 1991.

[11] Hind A. 'Overview: Parallel simulation techniques for telecommunication network modelling.' in Proceedings of the Ninth IEE UK Teletraffic Symposium (April 1992), 9/1-9/6.

[12] Kesidis G. and Walrand J. 'Large deviations of traffic in high speed digital networks with a view towards network control', EECS, University of California, Berkeley, 1991.

[13] Mellor J. 'A high speed simulation engine for BISDN' In proceedings of 3rd Bangor Communications Symposium, May 1991, 219-224.

[14] Mellor J., Chen J. R. and Hansen M. 'Simulation support for the management network' In proceedings of the 6th RACE TMN Conference, 1992.

[15] Mouftah H. T. and Sturgeon R. T. 'Distributed discrete event simulation for communications networks' IEEE Journal on Selected Areas in Communications 8, 9 (December 1991), 1723-1734.

[16] Reynolds P. F. 'A spectrum of options for parallel simulation' In proceedings of the 1988 Winter Simulation Conference.

[17] Prycker M. D., Ed. 'Asynchronous Transfer Mode: solution for Broadband ISDN' Ellis Horwood, 1990.

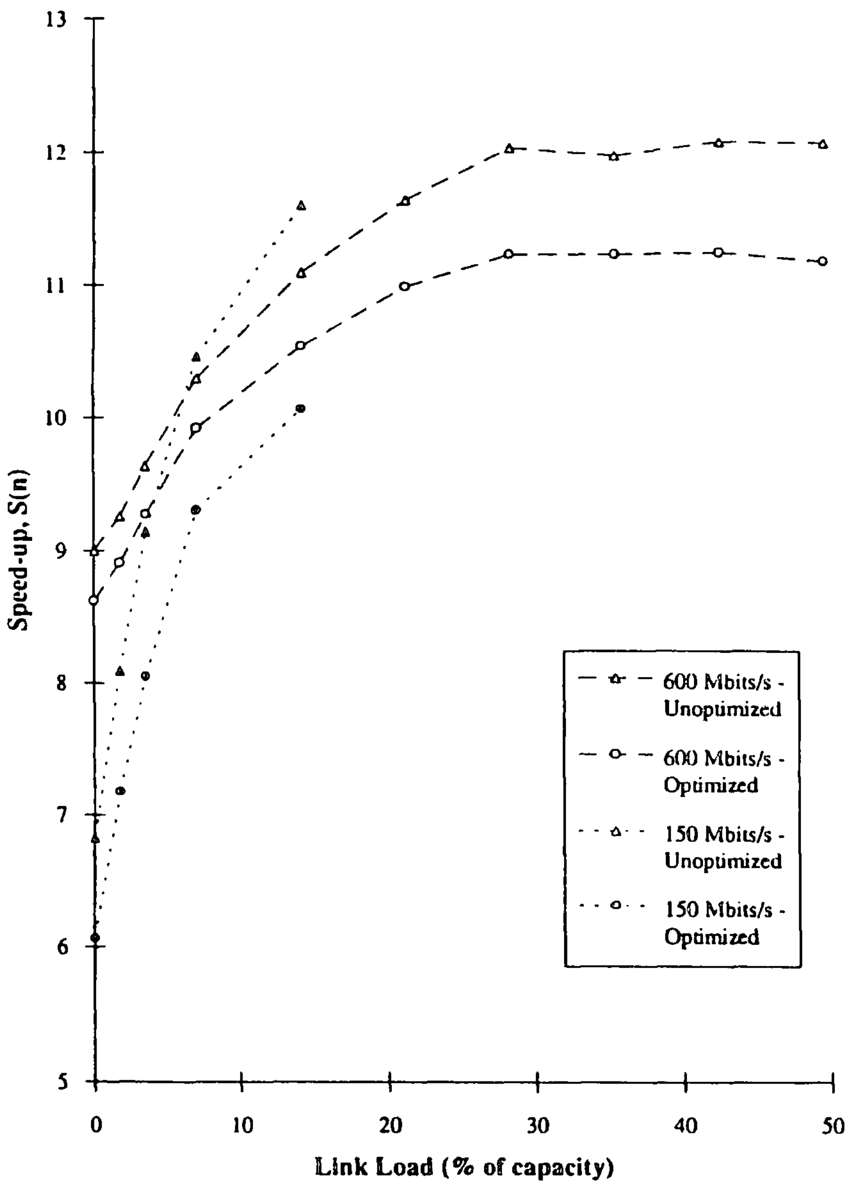


Figure 1: Speed-up curves as a function of traffic load. Speed-up is calculated relative to the optimized or unoptimized uniprocessor simulations.

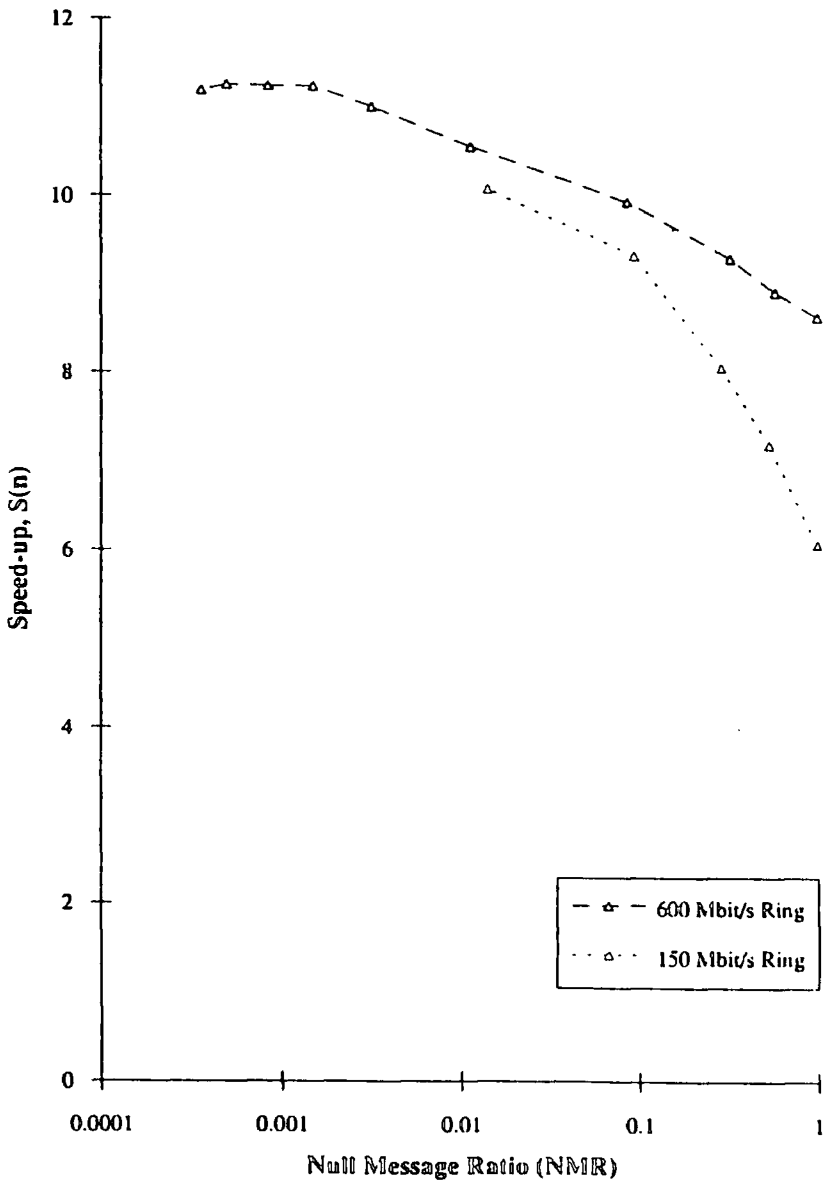


Figure 2: Speed-up (optimized) as a function of NULL message ratio. The difference between the two curves represents the extra parallelism that can be extracted from the higher speed rings.

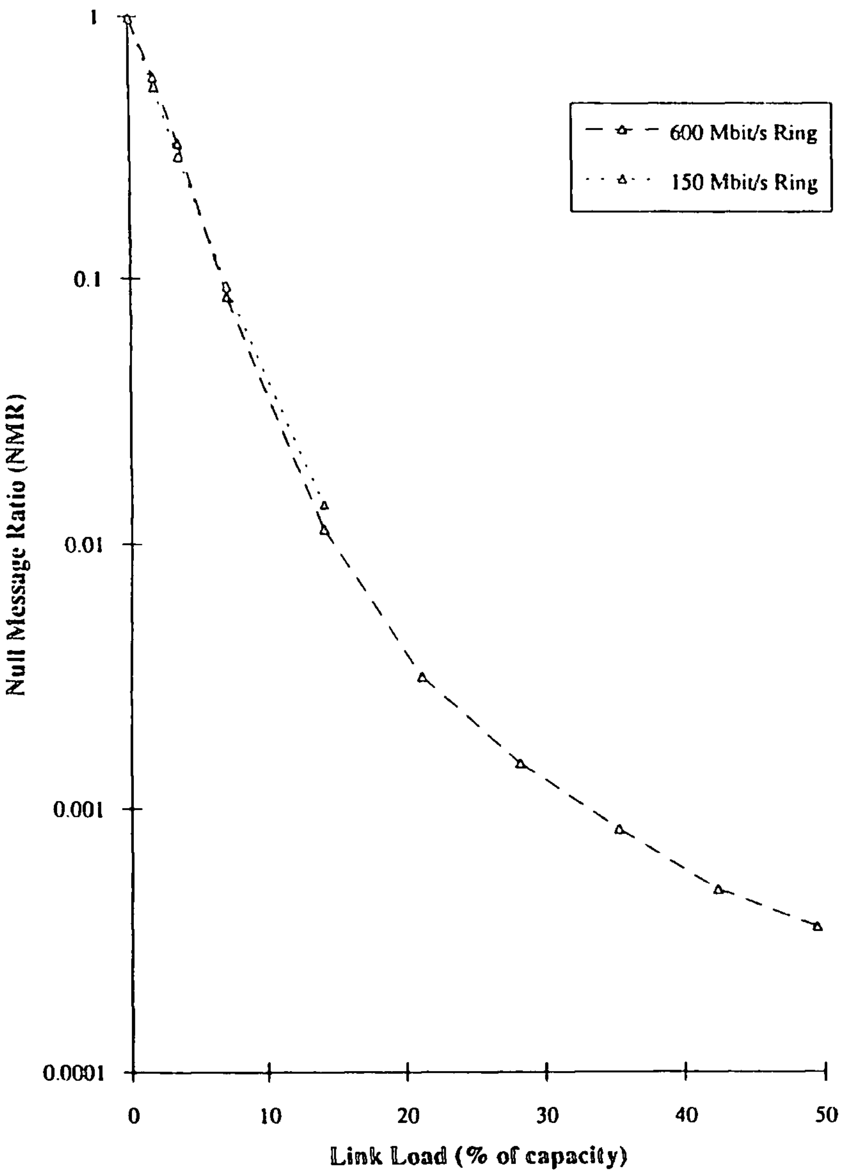


Figure 3: NULL message ratio (NMR) as a function of load. As might be expected, the ratio is largely independent of the ring speed.

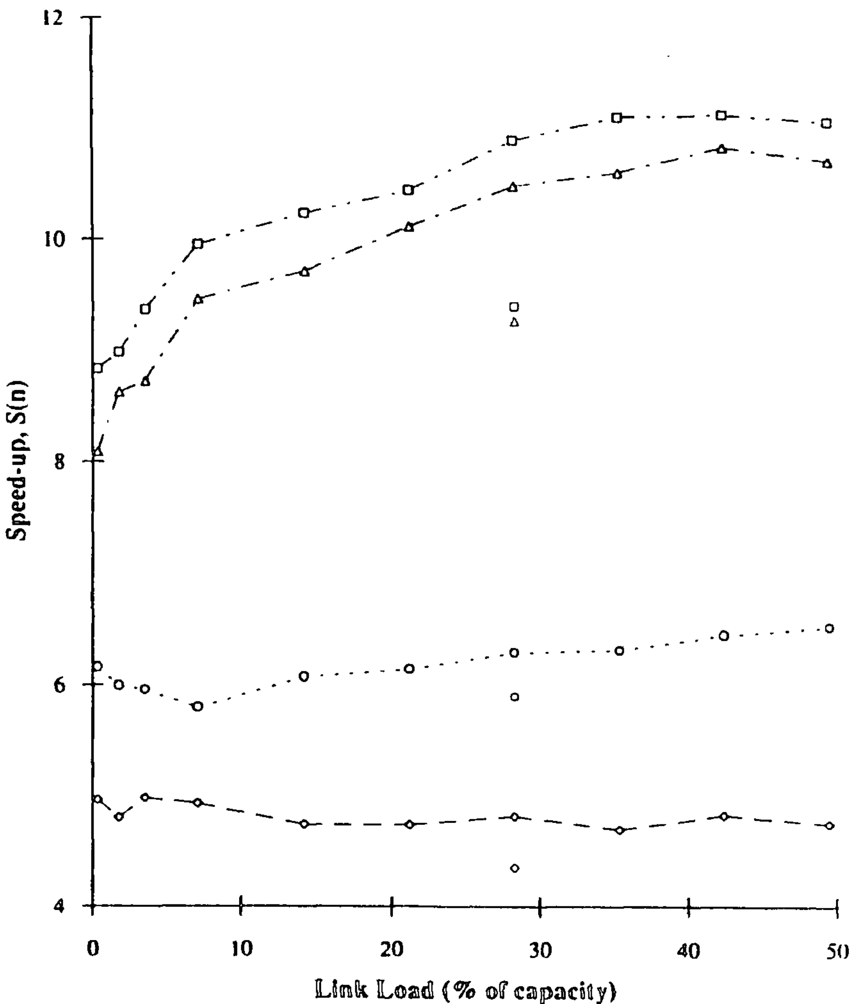
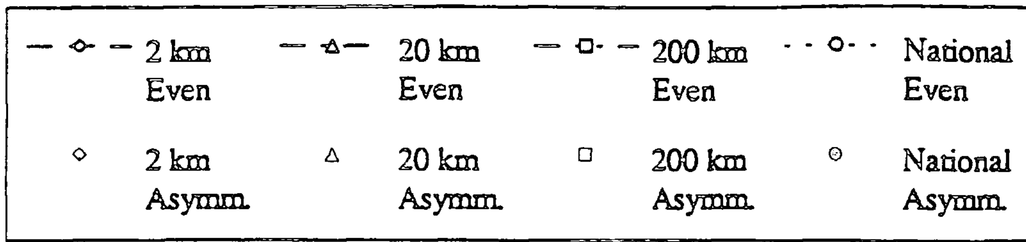


Figure 4: Speed-up (optimized) as a function of load for a range of lookahead values and symmetric and asymmetric traffic patterns.



Null Message Ratio (NMR)

1
0.1
0.01
0.001
0.0001

0 10 20 30 40 50

Link Load (% of capacity)

Figure 5: NULL message ratio (NMR) as a function of load for a range of lookahead values and symmetric and asymmetric traffic patterns.