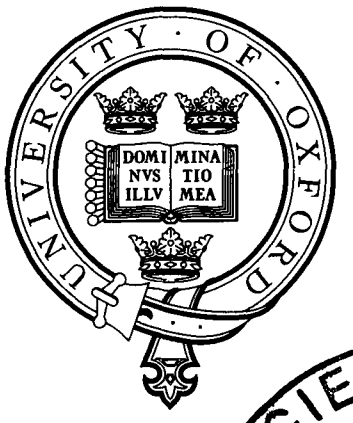
Global Optimisation of Communication Protocols for Bulk Synchronous Parallel Computation

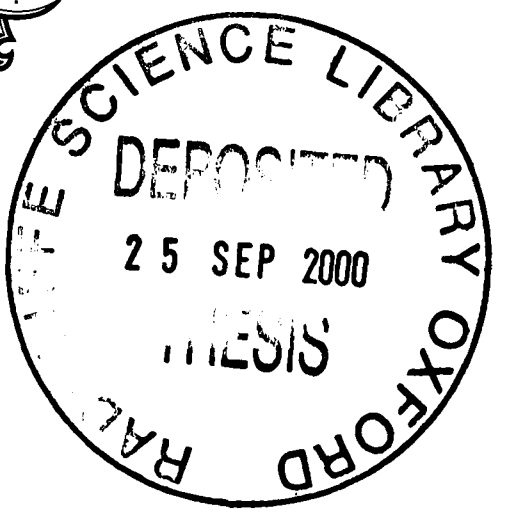
A THESIS SUBMITTED TO
THE FACULTY OF MATHEMATICAL SCIENCES
AT THE UNIVERSITY OF OXFORD
IN FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE
DOCTOR OF PHILOSOPHY

By
Stephen Richard Donaldson
Wolfson College, Oxford
Michaelmas, 1999

Supervised by
Professor W. F. McColl

Programming Research Group
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD





Abstract

GLOBAL OPTIMISATION OF COMMUNICATION PROTOCOLS FOR BULK SYNCHRONOUS PARALLEL COMPUTATION

D.PHIL DISSERTATION

Stephen Richard Donaldson, Wolfson College, Oxford

Michaelmas, 1999

In the Bulk Synchronous Parallel (or BSP) model of parallel communication represented by *BSPlib*, the relaxed coupling of the global computation, communication and synchronisation, whilst providing a definite semantics, does not prescribe exactly when and where communication is to be carried out during the computation. It merely states that it cannot happen before requested by the application and that at certain points local computation cannot proceed unless updates have been applied from the other participating processors.

The nature of the computation and this framework is open to exploitation by the implementation of the runtime system and can be made to suit particular physical environments without requiring application program changes. This bulk and global view of parallel computation can be used to implement protocols that both maintain and take into account global state for optimising performance. Such global protocols can provide performance improvements which are not easily achieved with local and greedy strategies and may in turn be locally sub-optimal.

This global perspective and the exploitable nature of BSP computation is applied to congestion avoidance, transport layer protocols suitable for BSP computation, global stable check-pointing, and work process placement and migration, to achieve a better overall performance.

An important consideration for the compositionality of parallel computer systems into larger systems is that in order for the composite to exhibit good performance, the individual components must also do so. However, it is not obvious how the individual components contribute to the global performance. Already mentioned is that non-locally optimal strategies might lead to globally optimal performance, but also of importance is that variance observed at the local level also influences performance. A number of decisions in the transport protocol design and implementations have been made in order that the observed variance in the protocol's behaviour is minimised. It is demonstrated why this is required using the BSP model. The analysis also suggests a regression technique which can be applied to sampled global performance data.

Acknowledgements

I am grateful for the encouragement, insight and supervision Prof. Bill McColl has so readily given over the last few years. The BSP Research Group, within the Programming Research Group under the leadership of Prof. Bill McColl, created an environment where the expression of ideas was encouraged. Participation within this research group was made particularly enjoyable for me by the efforts of Radu Calinescu, Vasil Vasilev, Yuguang Huang, Mauricio Marin, Alex Gerbessiotis, Jon Hill and Fabrizio Petrini. Both the quality and profile of my work has benefited enormously by the critique and participation of David Skillicorn and Jon Hill. Geraint Jones, as College Advisor, allayed many fears and anxieties with his thoughtful and desperately needed advice.

Financially, I was partially supported by the Harry Crossly Foundation. This was augmented by work opportunities afforded me by Clive Dall, Trevor Howcroft, Achi Racov and Rajan Pillay. This was made all the more enjoyable by friends and colleagues Dave Keating, Jan Grove and Patrick Power, amongst others.

I am grateful for the friendship and encouragement of Vanessa and Peter Winchester. I would also like to acknowledge my parents Lionel and Margaret; and friends Patrick and Karen Hayward, Patrick Marais and all the Ockendens. Last but not least, special thanks to Melanie for her editorial assistance and unfailing support and especially for helping to keep things in perspective.

Contents

Chapter 1

Introduction

The rise of Networks of Workstations and the renaissance of loosely-coupled multi-processing systems (or Clusters) have provided a new vehicle for parallel computation. The resultant intensive use of popular transport protocols over supporting network infrastructures has provided an opportunity to think of protocol requirements for parallel computation in another way. This new way treats communications globally and co-ordinates at the local level according to global implications.

Often in communication one finds that local optimisations or greedy strategies do not translate into global optimisations. Examples of this include the better bandwidth utilisation achieved by non-persistent CSMA over 1-persistent CSMA, where at the expense of longer delays, the non-persistent protocol does not attempt to gain access to the medium the instant it is found to be quiescent (for example, Tanenbaum [117]); or congestion avoidance with Valiant's two-phased randomised routing in low diameter networks over direct or greedy routing, even though the paths the messages take are not the shortest between communicating partners (for example, Valiant [121], Leighton [84]). In both these examples a better global strategy is obtained even though the protocols are sub-optimal at the local level.

In the most general setting, point-to-point protocol instances are oblivious of their connected application's usage of any other instance of the protocol or other components, either locally or remotely. Communication protocols are generally designed so that they cannot take this 'bigger picture' into account. Thus, as is evident in the design of TCP [99], point-to-point protocols tend to be greedy and optimised for what is known locally about the circuit to which they are attached.

Global optimisation, on the other hand, would require some global state distributed among the end-points where global choices can be made jointly

and as an online process. The communication of this additional state data, as an extra communication cost, and the inference of any global state, as an extra computation cost, must be kept to a minimum. Fortunately, both the possible extra communication and computation costs can be amortised over larger amounts of communication and computation in the Bulk Synchronous Parallel (or BSP) model assumed in this thesis. This is because the communication primitive in this BSP model, the h-relation, and the computation 'operation', the super-step, are bulk global operations. The optimisations for communications described in this thesis exploit the BSP computation structure in globally optimising communication for BSP computation. Aspects of the structure exploited include the type of communication primitives and the bulk nature of both the communication and computation parts of the BSP computation, as well as the nature of the synchronisation of the participating processors.

Whilst the BSP model of Valiant [120] was introduced as a bridging model between hardware and software or between real machines and the PRAM model, no real machines have been built according to the model. However, as was originally observed, almost all parallel machines could be parameterised in terms of the BSP machine model. Consequently, libraries that completed this conceptual mapping have appeared. Whilst these libraries usually define a perturbation of the original model, they can all loosely be defined as providing a mechanism for bulk communication and for a global synchronisation.

As has been pointed out on numerous occasions, thinking globally when designing programs for these libraries often results in better algorithms. It did not take long before the same was realized of the libraries themselves. For example, Skillicorn *et al.* [108] applied a number of global optimisation techniques to enhance the performance of *BSPlib* [66].

While the original BSP or XPRAM [121] model may be universally applicable to parallel applications given its emulation of the PRAM, the same is not true of its realisation in BSP programming libraries such as *BSPlib*. However, the somewhat restricted style of programming that it supports has some important aspects which can be exploited for performance gains. For example, when compared to implementations such as MPI and PVM, *BSPlib* has very few primitives. This has been beneficial not only from the point of view of the porting effort, but also the nature of the operations themselves has meant that certain optimisations can be implemented which cannot be applied in a more general setting.

One example is that two sided communication primitives define a two process (half-) synchronisation. The lazy approach to communication in *BSPlib* delays communication from the send primitive until the global synchronisation is requested. Naturally, the two types of primitives are not interchangeable

and globally synchronising on each pair-wise communication is not efficient. Amortising global synchronisation over many communications is certainly a benefit on architectures such as the Cray T3E which provide hardware memory barrier primitives. But more importantly, the delayed communication approach used by *BSPlib* provides an opportunity for optimising communication which far outweighs the loss of the potential factor of two gain by overlapping communication with computation. Hill and Skillicorn [67] describe the packing of messages to get efficient usage of the available bandwidth by minimising startup costs and generating global schedules using Latin squares to avoid conflicts at the processors. The performance improvements achieved almost always does better than the factor of two lost by not overlapping communication with computation.

This thesis is about the exploitation of the BSP model as refined by the *BSPlib* programming library and where it pertains to networks of workstations and clusters. This exploitation appears at three different levels and in general is about making choices that optimise the performance of the computation. Firstly, at the library level there is an opportunity to manipulate the transmission of messages, not only the ordering of the messages, but also the timing of messages. Secondly, by taking into consideration the nature of the computation and the use of workstations, a protocol more suited to BSP computation than more general network protocols is derived and its implementation discussed. Finally, by considering the nature of a group of loosely coupled workstations in reasonable proximity in local area networking terms, a scheme for keeping a computation on the most suitable set of processes is presented. While much of the work is experimental and the implementations should be considered as prototypes, a significant portion has extended *BSPlib* [61].

Transport protocols such as TCP/IP [99, 98] are designed to operate well in the point-to-point case, taking into account only the requirements of the protocol instance's circuit. By ignoring the usage of the medium by other users and adopting a greedy approach to recovery, they render themselves unsuitable for parallel computation. Even in situations where the underlying network on which TCP/IP is implemented has a high mean bandwidth and a low latency, if the protocol stack does not take this into account, it is likely that the protocol mean bandwidth and latency have a large variance. The importance of this second order effect and how it impacts scalability is demonstrated in this thesis by considering communication from a global BSP perspective; it forms the basis of an empirical study by Hill *et al.* [62].

1.1 Thesis overview

Chapter 2 provides a general review of parallel computation as it pertains to the BSP model. It includes a brief historical background to parallel computing, concentrating on what is relevant to the BSP model. Naturally, synchrony is fundamental to the BSP model and its role in the development of the BSP model is also discussed. Chapter 2 also reviews the BSP cost model, but stops short of presenting any details of concrete implementations such as *BSPlib*.

Chapter 3 reviews the BSP programming library *BSPlib*. It also describes the porting of the library for message passing systems or transport protocols such as those described in Chapter 5. *BSPlib* had two immediate benefits for the present work. Firstly, by porting and extending *BSPlib* rather than developing a library from scratch, ideas could be implemented and tested with a minimum of effort. Secondly, the contribution to *BSPlib* meant that the developments were immediately usable by a body of researchers and programmers, thus extending the base of implementations for a large and growing body of existing applications.

A means of congestion control is presented in Chapter 4. This control is generally applicable to many networks which show a reduction in the successful load as applied load is increased. Such networks include frame relay and ATM or cell relay networks [109]. In particular CSMA/CD has this behaviour [55] and Chapter 4 uses this to develop a mechanism suitable for controlling congestion in a BSP computation over a dedicated local area network. This chapter is based on the paper "Predictable communication on unpredictable networks: Implementing BSP over TCP/IP and UDP/IP" by Donaldson *et al.* [32].

Chapter 5 discusses the development of a BSP communication protocol by considering the implementation of a number of *BSPlib* messaging layers. The first messaging layer discussed is based on TCP/IP and is referred to in this thesis as *BSPlib*/TCP. This messaging layer also demonstrates the unsuitability of TCP/IP to high-performance computing, which prompted the development of a protocol to address the problems encountered using TCP/IP. The result is a reliable sequenced packet protocol built upon an unreliable datagram protocol, but instead of concentrating on optimising the protocol on a point-to-point basis, a number of global considerations are taken into account. The protocol provides its own recovery mechanism and includes buffer management, flow control and the congestion control mechanism described in Chapter 4.

Two implementations of the protocol are described. The first, *BSPlib*/UDP, is based on UDP/IP and makes as few assumptions about the implementa-

tion of UDP/IP as possible. This implementation captures the largest class of machines and is preferred over TCP/IP which has to make use of certain esoteric socket options in order to guarantee proper behaviour. The second implementation, *BSPlib*/NIC, is only a prototype and was implemented as an experiment to test a number of ideas to achieve further reductions in latency and increases in bandwidth. Of the three implementations, this is the least portable as there is a strong dependency on the software and hardware on which it was developed. The material in this chapter is based, in part, upon the paper "BSP Clusters: high performance, reliable and very low cost" by Donaldson *et al.* [33]. Some of the material also appeared in "Exploiting global structure for performance on clusters" by Donaldson *et al.* [30].

The performance of the various messaging layers and protocol implementations are compared in Chapter 6. These implementations of *BSPlib* are also benchmarked against public and proprietary message passing interfaces used in high performance computing. Two classes of benchmark are considered. Firstly, low level micro-benchmarks are used to show the raw efficiency of the communication equipment that can be achieved. These benchmarks measure the bandwidths and latencies that can be achieved independent of any application. The experiments measure link bandwidths between pairs of processors, round-trip delays for an echoed packet, and half-round-trip latencies per packet. Each experiment is also carried out for various message sizes. Secondly, at the application level, the NAS parallel benchmarks [4] are used to compare the various 'aggregated' BSP computers, `mpich`, proprietary message passing libraries and proprietary implementations of MPI. These benchmarks compare various configurations under the load of certain applications. The applications are standardised and considered by the authors of the NAS parallel benchmarks as being typical of classes of applications used in practice. The benchmark in this case measures the runtime of the application and quotes the mega-flop rate per process achieved. This is a more reasonable basis for the comparison of parallel computers as it is a single metric directly related to the expectation of the users of the machine.

Also in Chapter 6, a survey of other low latency protocols that have appeared in the literature is presented, and these results are compared with those presented in this thesis. The material in Chapter 6 is based upon the paper "BSP Clusters: high performance, reliable and very low cost" by Donaldson *et al.* [33]. Some of the material also appeared in "Performance Results for a Reliable Low-Latency Cluster Communication Protocol" by Donaldson *et al.* [31].

Chapter 7 deals with restrictions on scalability pertaining to parallel composition induced by the second order metric of variance. This effect is ad-

dressed from two separate perspectives both of which are independent of any implementation. Firstly, from the point of view of an abstract interconnect, it is clear that there must be some notion of $p(p-1)/2$ circuits among the p processors of a parallel computer. By assuming the best case interconnect, i.e. fully connected, the effects of variance at the link level alone on the global performance of the interconnect can be shown. The second approach takes only the BSP cost model into account (independent of any particular interconnect) using the implicit assertion that the cost of communication is incurred at the edges of the interconnect. A strictly BSP analysis, independent of any interconnect assumption, demonstrates the importance of minimising variance. These approaches also reveal a regression technique which can be used to assess the parallel composition of protocol stacks. An example is given comparing the two implementations of the BSP transport protocol presented in Chapter 5. As a consequence of this work, many decisions in the protocol design were made in order that variance be minimised. Variance as an architectural parameter is the basis of an empirical study of the interconnects of parallel machines by Hill *et al.* [62] from which examples are drawn. Part of Chapter 7 is based upon the paper "Communication performance optimisation requires minimising variance" [29].

Chapter 8 describes a scheme implemented in *BSPlib* that enables BSP programs using the library to migrate processes among a network of workstations. This affords *BSPlib* jobs not only fault tolerance, but also makes it possible to continually schedule the migration of BSP processes onto the machines with the greatest available capacities. It manages this by using a load manager that maintains an approximation of the global load of the system.

The simplicity of the structure of BSP programs provides a convenient point at which local checkpoints capture the global state of the entire BSP computation. Process migration and check-pointing can be achieved without changes to the users' programs. The chapter describes how the processes make check-pointing and migration decisions, and how computations are restarted on different machines. The determination of the global load approximation is described and the quality of the approximation is analysed. The chapter is based on the paper "Process migration and fault tolerance of *BSPlib* programs running on Networks of Workstations" by Hill *et al.* [60].

The final chapter, Chapter 9, concludes the thesis by reviewing the major results, and discusses possible future work and outstanding problems.

Declaration As a number of chapters are based on jointly authored papers, this declaration clarifies this author's contribution. All co-authored work is joint work unless stated otherwise here. In particular the benchmarking has been joint work. Of the co-authored papers, only those papers in which this author's contribution is significant have been included. Works in which this

author's contribution is less significant have not been included, although a number of examples and discussions have been drawn from them [107, 63, 62].

All implementations are by this author with the exception of the integration into *BSPlib* and the global load algorithm described in Chapter 8 which are by Hill. In addition all analyses using probabilistic and Markov arguments, simulations, literature surveys, background work and protocol designs are due to this author. The observation that TCP timeouts are responsible for the observed TCP/IP performance, and a number of fine tuning adjustments of the protocols, are due to Hill. Specifically, and notwithstanding editorial assistance, the entire contributions of Chapter 4 and Chapter 7 are this author's.

Chapter 2

Bulk Synchronous Parallel Computation

> *In the long run this inherent advantage of the parallel mode was to triumph, and the serial mode essentially disappeared.*
> Herman H. Goldstine on the von Neumann machine [48]

The purpose of this chapter is to review the BSP model of parallel computation. In doing so, a fine temporal line is traced from the model of sequential computation to the PRAM model and the accepted view of the BSP model. The purpose is not so much to provide a history, but to draw parallels between the objectives of the von Neumann and BSP models; and to highlight the role that synchrony (see, for example, Hwang and Xu [73]) has played in the development of the model.

2.1 Sequential Computation

The *Random Access Machine* or RAM is a formal model of sequential computing based on the von Neumann machine and its cost model (the RAM model introduced the notions of logarithmic and unit cost on elementary arithmetic operations; Cook and Reckhow [25]). The relationship between the machine model and the programming model, RAM-Algol, demonstrated that algorithmic complexity in the machine model *could* be determined simply by inspection of the source code in this high-level programming model.

The relationship with the von Neumann model is demonstrated in the same paper using RAM and *Random Access Stored Program* or RASP machine simulations. Prior to the introduction of index registers by Kilburn [80],

the power of the von Neumann model was derived from the ability of the program to modify itself [37]; see also Goldstine [48].

The important point was that this model was offered as a *bridging* model which had a number of desirable properties:

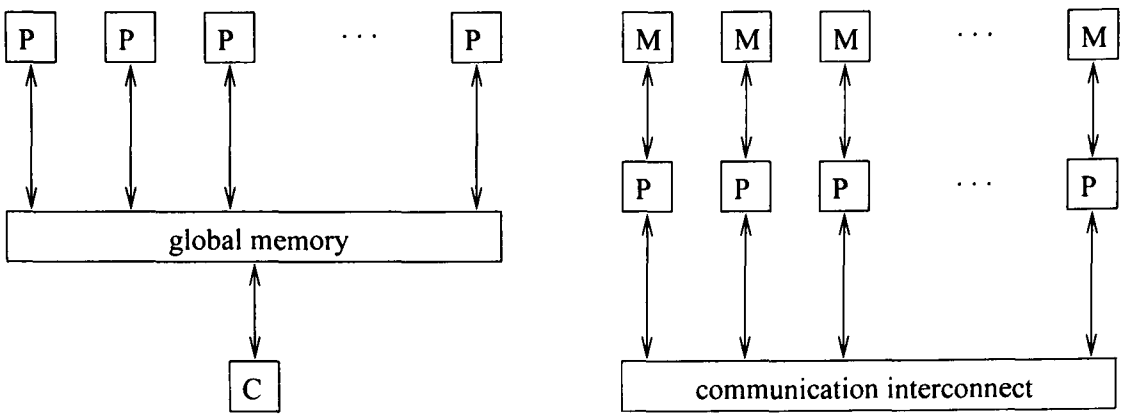
- Relationship with abstract models of complexity and computability.
- Relationship with real machines based on the von Neumann model.
- Programming notation (language) convenience which admitted complexity analysis by inspection.

2.2 The PRAM Model

One model of general purpose parallel computation is the idealised or abstract *Parallel Random Access Machine* or PRAM [40, 47] (see also McColl [86] for an exposition of *special* purpose parallel computation and McColl [85] for a treatment of general purpose parallel computation).

The PRAM consists of a 'sufficient' or unbounded collection of RAM processors connected together with some form of global memory, capable of executing, in parallel, a single stream of instructions. These instructions are executed by the distributed processors in a synchronised lock-step fashion. The PRAM is idealistic in that access to a memory cell can be accomplished within an instruction cycle independent of the number of processors in the computation or the location of the memory cell. Hence the placing of the memory is unimportant and the popular configurations shown in Figure 2.1 are equivalent. With respect to the global memory, the architecture is *load-store* and in each cycle the processor can initiate a read from or a write to the global or remote memory, or it can perform an operation on data held locally in the processor (two- or three-address architectures are equivalent).

Simultaneous access to a single cell or location of remote memory is unrealistic and a number of approaches have been devised to deal with this problem, each of which defines a refinement of the model. Either such access for read or write is allowed (concurrent), forbidden (exclusive), and if concurrent access is allowed on writes, then these have to be arbitrated to decide which update is applied. Thus there are classes, CRCW (or C), EREW (or E), CREW and ERCW, of PRAMs. The simplest and most reasonable, from an architectural point of view, is simply to forbid concurrent access as in the case of the EPRAM (EREW-PRAM). The original models of Goldschlager [47] and Fortune and Wyllie [40] were CREW-PRAMs, but by assuming the EPRAM, the widest portability of algorithms is assured as an EPRAM program can be executed without loss of efficiency on the



(a) Goldschlager [47]

(b) Valiant [121]

(c) McColl [85]

(d) Karp and Ramachandran [79]

P = processor, L = private memory, M = distributed memory, C = central processor

Figure 2.1: Possible configurations of the PRAM

other PRAM models and any dependence on the arbitration methods is avoided (for some problems the CPRAM or CRCW-PRAM behaves as an unrealistically powerful oracle in terms of current technology).

One can view dependence on memory locality in parallel computers whose architectures are based on regular graphs in a similar manner. There are a number of problems that can be solved efficiently on specific networks such as the butterfly or hypercube (see, for example, McColl [86]), but this efficiency can be heavily dependent on locality, making the algorithm non-portable. The required model must support portable parallel computation on a broad range of implementations and hence dependence on locality should be avoided.

The PRAM has proved popular in algorithm design and complexity theory (see, for example, Gibbons and Rytter [46], Jájá [76], Greenlaw *et al.* [51], Karp and Ramachandran [79]). Many results on specific networks as well as their simulation of PRAMs are presented in Leighton [84]. The style of

simulation of the PRAM that will be looked at is dealt with in Section 2.3 and comes from Valiant [121].

The assumption that a global or remote memory access can be completed in a single instruction cycle is also simplistic and unrealistic. It is interesting to observe that the same caution exhibited by Cook and Reckhow [25] in their motivation of the RAM was not exercised in the PRAM. The RAM did not cast in concrete the cost model for elementary operations but in terms of some function of the length of the operands of the instructions, and proposed both a *logarithmic* and a *unit cost* model. Furthermore, instructions in the RAM are limited to those that can be reasonably executed without violating the relationship with complexity theory.

For complexity purposes, the PRAM obeys this limitation on elementary operations with the unit cost model universally applied and in, for example, Greenlaw *et al.* [51] multiplication only by constants is allowed. However, as far as communication costs are concerned, the best case and certainly most unrealistic costs are used, rather than leaving it unspecified.

Algorithm design pursued two main streams. Firstly, PRAM efficient algorithms which continued to ignore the communication problem, perhaps in the hope that architectural technology would improve to the point where, for practical purposes, the PRAM can be implemented efficiently (see, for example, Gibbons and Rytter [46], Jájá [76], Greenlaw *et al.* [51], Karp and Ramachandran [79]). Secondly, architectural efficient algorithms were designed with specific network topologies in mind and typically depended on being able to exploit locality exposed by the specific architecture (see, for example, Leighton [84], McColl [86]).

The tightly coupled synchronisation of the PRAM was a style of parallel computation called *Single Instruction Multiple Data* or SIMD in Flynn's taxonomy [39] for parallel computer architectures (see, for example, Hwang and Xu [73], Culler *et al.* [28]). In this mode the parallel computer executes as though a stream of instructions is broadcast from a single control unit to the multiple parallel processing units. Other styles included *Single Instruction Single Data* or SISD (sequential computation) and *Multiple Instruction Multiple Data* or MIMD, a more loosely coupled mode usually associated with the message passing style of parallel computation.

2.3 Costing Communication

There are a number of PRAM derivatives which take into account the shortfalls in realizing the PRAM as a viable architecture. Chin [19] surveys the contribution of these models to efficient parallel computation identifying

problems such as contention and congestion at the nodes and in the communication interconnect, communication latency, pipelining, asynchrony and hardware failure. He identified certain models considered as (more) realistic for parallel computation.

One of these models was the XPRAM of Valiant [121]. Valiant introduced the XPRAM as a generalisation of the PRAM and viewed it at the intersection of realistic models of parallel computation and the idealised PRAM.

The tight synchrony of the PRAM is observable in two respects. Firstly, the instructions are scheduled and executed synchronously according to some global clock or a broadcast instruction stream. Secondly, when a processor accesses global memory or the local memory of another processor, the processor and the memory (or the two processors) synchronise.

In the XPRAM, the instruction synchronisation required by SIMD architectures is replaced by that required by the more generic MIMD architecture. The two models can simulate each other within a constant factor (see, for example, Savage [106]).

The XPRAM consists of a number of processors, p (it is necessary to think of a fixed number of processors in any realistic model as some of the architectural parameters are a function of the number of processors) each consisting of a local processor and a local memory. (All the symbols used in the discussion of the models in this chapter are summarised in Table 2.1 at the end of the chapter.) This model charges a single time unit for a local operation (including a memory access) and charges $g \geq 1$ time units for non-local accesses. The PRAM assumption of an unbounded number of processors is satisfied by allowing $v \geq p$ virtual processors as required by the computation.

The p-processor XPRAM executes in *supersteps* during which each processor executes a number of local operations and issues a number of global read and write instructions. Within each set of L operations the processors do not synchronise, but at the end of the period each processor knows whether or not the superstep has ended. L reflects the latency of the non-local memory accesses.

In the XPRAM model a superstep is a programming notion whilst the synchronisations are performed automatically by the machine. Thus whilst synchronisation is removed from the instruction level, the notion of supersteps allows the code to synchronise at a coarser level of granularity. The term *bulk-synchrony* was introduced to describe the way in which the machine periodically (with period L) synchronised memory references.

Figure 2.2 illustrates operation of the XPRAM. In the figure two processors are shown (i and j) separated by an interconnect in which, in this case, the

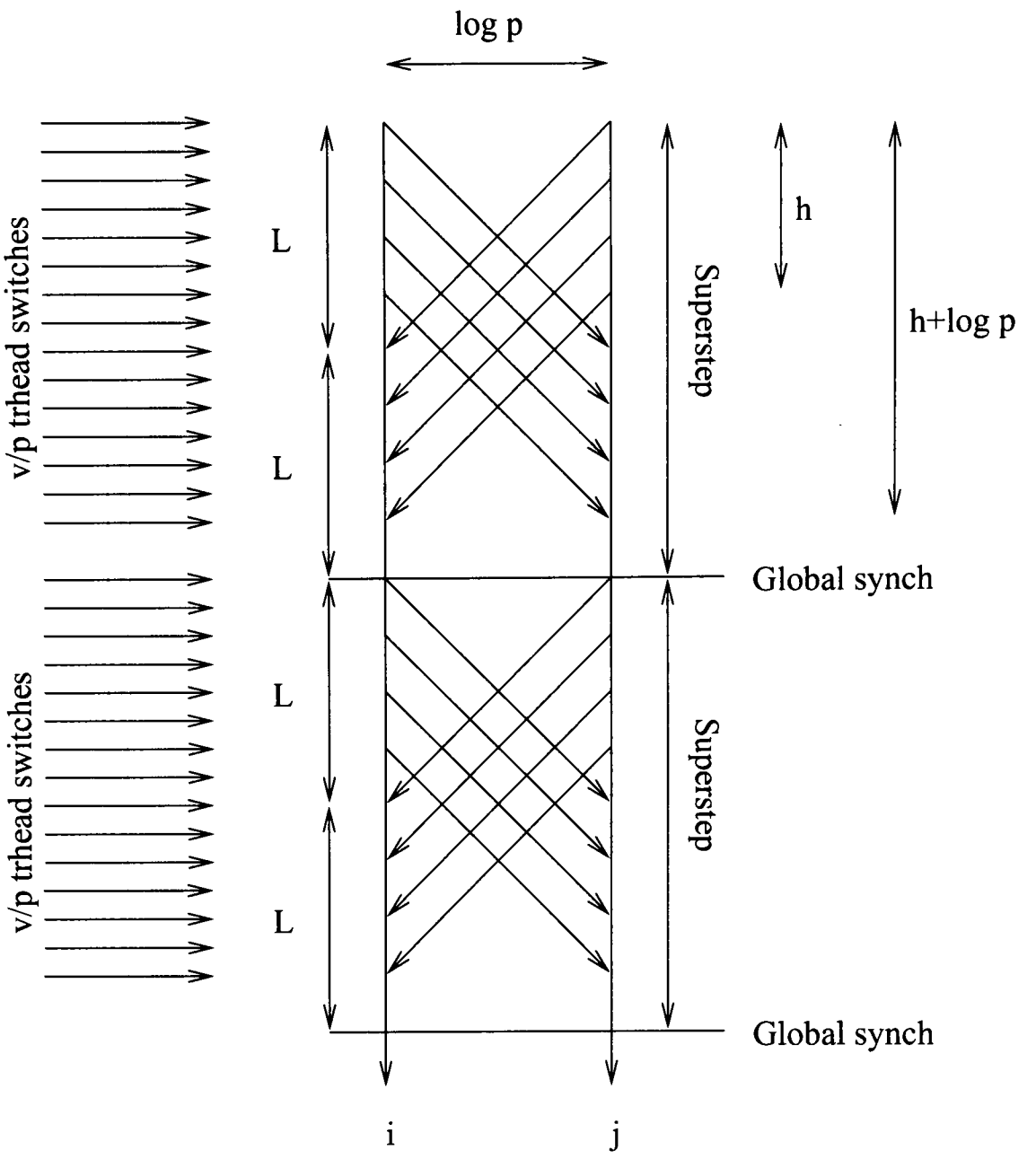


Figure 2.2: Operation of the XPRAM

time for a message sent from i to j will take time $\mathcal{O}(\log p)$. The time cost of a superstep in the XPRAM model can also be determined. If processor i performed w_i local operations, sent out s_i messages and received r_i messages during a single superstep, then the total time required by processor i in terms of global operations during this superstep is

$$t_i \quad = \quad w_i/g + s_i + r_i.$$

If $t = \max\{t_i \mid i \in \langle p \rangle\}$ then the time to complete the superstep is $(\lceil t/L \rceil)Lg$ (where $\langle n \rangle = \{0, \ldots, n-1\}$). The amount of communication illustrated in Figure 2.2 implies that in this case each of the supersteps takes time $2L$.

Valiant [121] showed that a p-processor XPRAM could simulate a v-processor EPRAM with optimal efficiency in the presence of $\log p$ *parallel slackness*

(that is, the ratio v/p). Also, in general in the cases where the interconnect between the processors is based on either cube-connected-cycles, a butterfly, 2D array or hypercube, the time required for each processor to send a message to a unique destination processor, a *permutation*, is proportional to the diameter of the network. And, in particular, on the hypercube with p-nodes, each processor can send $\log p$ messages and receive $\log p$ messages, a $(\log p)$-*relation*, in $O(\log p)$ time using two phase random routing (see McColl [85]). Similar results exist for the CPRAM [120] (also, Gerbessiotis and Valiant [45])

For the simulation of the EPRAM by the XPRAM the global memory of the EPRAM is distributed among the p processors according to a *good fast hash function.* The v virtual processors are assigned to the p physical processors in an arbitrary manner such that no physical processor is assigned more than $\lceil v/p \rceil$ processors. A cycle of the EPRAM is simulated as a superstep of the XPRAM. During each superstep a processor on the XPRAM switches between the v/p EPRAM processors or threads assigned to it. The hash function and the required slackness $(\log p)$ ensure that during a cycle of the EPRAM, or a superstep of the XPRAM, each processor will not get more than $3 \log p$ memory requests with high probability and an expected number of $\log p$ [120, 121].

The XPRAM model shows that the vast body of algorithms devised for the idealised EPRAM could be realized on the XPRAM computer. Hence, provided sufficient parallel slackness is achieved and provided such a computer is feasible, such algorithms would demonstrate efficient speed-up. However, while choice of graph, routing algorithm and required level of slackness had solved the EPRAM problem, there still remained the problem of fast context switching among the virtual processors or threads in the superstep in order to achieve the efficiency. This is necessary as the interleaving of the threads was at least at the instruction level as required by the SIMD nature of the EPRAM.

Furthermore, with the interleaving at the instruction level, the size of the messages are the size of the operands of the EPRAM instruction's operands, typically 32-bit or 64-bit words. With such small packets the overhead within the processor tends to dominate the cost of the communication. For example, Juurlink and Wijshoff [77] found that intra processor delays meant that the cost of communication appeared to be $O(h \times \log p)$ rather than $O(h + \log p)$ (see Figure 2.2).

2.4 The BSP Model

In Valiant [120] this model was generalised into the the *Bulk Synchronous Parallel* or BSP model. The essential differences appear to be that the synchronisation granularity is optionally controllable. Although, as Valiant points out, this scheme affects the simulation analysis only by small constant factors, it is an important development for programming. The BSP model allows a style of programming that is different from the EPRAM style. Valiant called the EPRAM style the *automatic mode*, a reference to the fact that management of the global memory, communication and context switching of the threads of the virtual processes is handled automatically by some intermediate layer between the XPRAM and the EPRAM program. The *direct mode*, in which the program itself manages the placement of data, the communication and the synchronisation, is suitable for achieving efficiency with less slack. This would be the case when the g parameter of the machine is high (a value of $g \approx 1$ is required for the automatic mode), multiplicative runtime factors are important (for example, when the overheads mentioned above are significant), or the L value of the machine is too high for the automatic mode [120, 45].

Returning to Figure 2.2, in this example all the virtual processors or threads on real processor i are sending messages only to virtual processors on real processor j. In the direct mode, a single process on processor i is in a position to send all of the h messages destined for processor j in one large packet of h words rather than h packets of one word each. The effect of this is that the overhead in message transmission is paid only once rather than h times. For example, Hill [58] shows that for the Cray T3D and Cray T3E machines, a value of $g \approx 1$ is possible, but only by packing messages in this way. By reducing the slackness in this manner, the demand for fast context switching is also reduced. In fact, slackness can be shown to be merely a matter of programming convenience in the model of BSP computation which will be assumed in this thesis. He *et al.* [56] show that running any of the parallel partners of a superstep sequentially end-to-end is sound. Reducing slackness in this manner amounts to, in their terminology, a partial conversion to *normal form* (i.e. a sequential program corresponding to the BSP program) in which all *phases* corresponding to *parallel partners* on the same physical processor are combined into a single phase (and hence one barrier synchronisation). The structure of BSP programs allows this normalisation to be carried out automatically with this context switching costing a little more than a subroutine call (and certainly not a system call).

A similar model to the BSP model is the LogP model [27]. In this model the intra processor overhead of sending and receiving messages is explicitly taken into account.

Surprisingly few problems are excluded from efficient BSP implementations. It is clear that for PRAM type programs in which the communication is regular, a good BSP implementation is feasible particularly if the pattern can be determined by some off-line process. Problems for which the communication pattern cannot be determined off-line and for which the communication cannot easily be assembled into this *bulk* fashion are harder, for example the parallel simulation of general random processes.

Valiant proposes the BSP model as a bridging model between hardware and software. The aim is to provide a practical model on which agreement could be established, based upon the strong theoretical results of abstract parallel machines. This could be thought of in much the same way as the von Neumann model (see, for example, Goldstine [48]) is a pragmatic realization of the Universal Turing machine [119] along the lines of the abstract models of the register machine of Minsky [93] and the RAM [25]

Thus there has been a systematic reduction in the tightness and the nature of synchronisation. In the PRAM the global or remote memory read and write requests are implemented as instructions within the PRAM architecture and are executed atomically. In the XPRAM such operations are *scheduled* or requested by such instructions, with the hardware synchronising independently of these communication requests with a defined and fixed synchronisation granularity L. And finally, the BSP model in which the control of the synchronisation could be left to the program as can the memory management and placement of data. This implies that the references to remote memory could be scheduled by the program. Just as communication and synchronisation are separated in the XPRAM model, so is communication and computation in the BSP model. Without further weakening of the model, strong gains can be made by separating and collecting the requests for communication from its realization.

A *BSP computer* consists of a collection of p identical processor-memory pairs with an interconnect for communication. The interconnect allows point-to-point communication among the processors and also implements a mechanism by which the processors barrier synchronise and wait for all pending communication requests to be completed. The operation of the interconnect is parameterised by its global bandwidth g and its latency l. In the model, the units of these parameters are in terms of the processor speeds given as their instruction execution rates or, more usually, as their mega-flop ratings. Thus l measures the synchronisation time in terms of how many instructions (or floating point operations) on a single processor to complete the synchronisation. Similarly, the g parameter measures the number of operations required to realize an h-relation (a communication pattern in which h is the maximum over all the processors of the number of words sent or received by the individual processors).

The g and l parameters of the cost model depend on the performance of the underlying architecture. For example, g depends on: (1) the bisection bandwidth of the communication network topology; (2) the protocols used to interface with and within the communication network; (3) buffer management by both the processors and the communication network; and (4) the routing strategy used in the communication network. The l parameter also depends on these properties of the architecture, as well as specialised barrier synchronisation hardware, if this exists. A contribution of this work is to further the exploitation of the BSP model by the runtime system to improve these parameters.

For purposes of algorithm analysis, the three parameters p, l and g are sufficient. However, in order to compute real execution times and to compare various BSP computers the normalisation to number of operations is not sufficient. Hence a fourth parameter s is often quoted and is a measure of the speed of the computer, usually in terms of mega-flops, and can be used to normalise to real time.

The programming style of the BSP computer is MIMD message passing, usually with a single program text being instantiated on each of the processors. This *Single Program Multiple Data* or SPMD mode of MIMD is simply for programming convenience and does not weaken the model. It could be used, for example, to ensure consistency by making sure that all processors have the same version of the text or by having the runtime support distribute the same copy to all the processors.

The execution of a BSP program proceeds in supersteps according to the BSP model. During a superstep, each processor executes operations on data held locally and may read or write (or at least make requests to read or write) data from or into the memories of the remote processors. The completion of the superstep is marked by a barrier synchronisation requested by each instance of the program amongst the processors. The program emerges from the barrier on all the processors once global read and write operations have been applied to their respective remote memories. Figure 2.3 shows semantically how the superstep operation can be viewed. The many overlapping and dashed arrows in the figure represent the potential for unstructured communication which may overlap with computation, but each word communicated would incur a startup cost. In contrast, Figure 3.1 on page 27 illustrates an arrangement involving the same amount of communication and computation in which the startup costs are paid only once per processor per superstep because of the structuring of the communication. In this case none of the communication is overlapped with the computation.

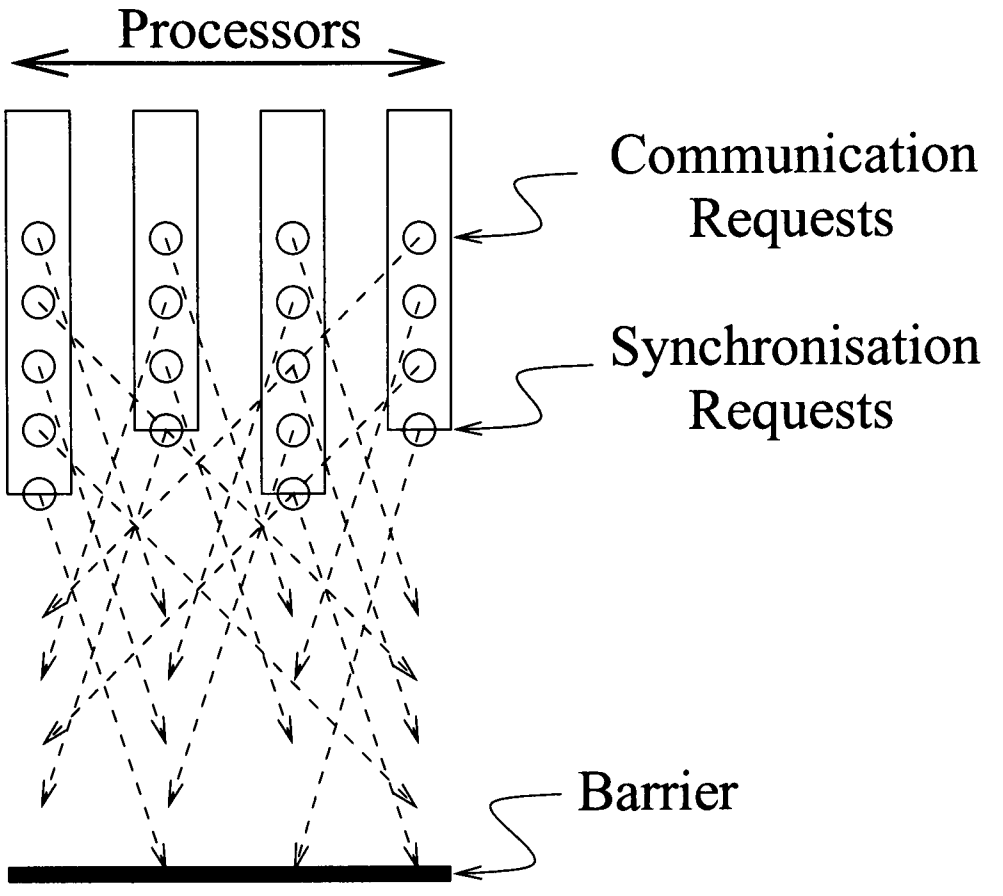


Figure 2.3: Operation of the BSP computer illustrating possible unstructured yet overlapped communication amongst the participating processors

2.5 The BSP Cost Model

The (time) cost in executing a BSP program is the sum of the costs of each of the supersteps executed during a run of the BSP program. This cost can be determined by inspection of the source code of the program and is a useful way of providing BSP algorithmic complexity.

If during the execution of a superstep, each processor, i, performs w_i operations on locally held data, receives r_i words from the other processors to update the local memory, and sends s_i words to other processors to update their local memories, then the size of the h-relation is

$$h = \max\{\max\{r_i, s_i\} \mid i \in \langle p \rangle\}$$

and the amount of computation charged to the superstep is

$$w = \max\{w_i \mid i \in \langle p \rangle\}.$$

The cost of the superstep, including the cost of the barrier, is then

$$C = w + h \cdot g + l$$

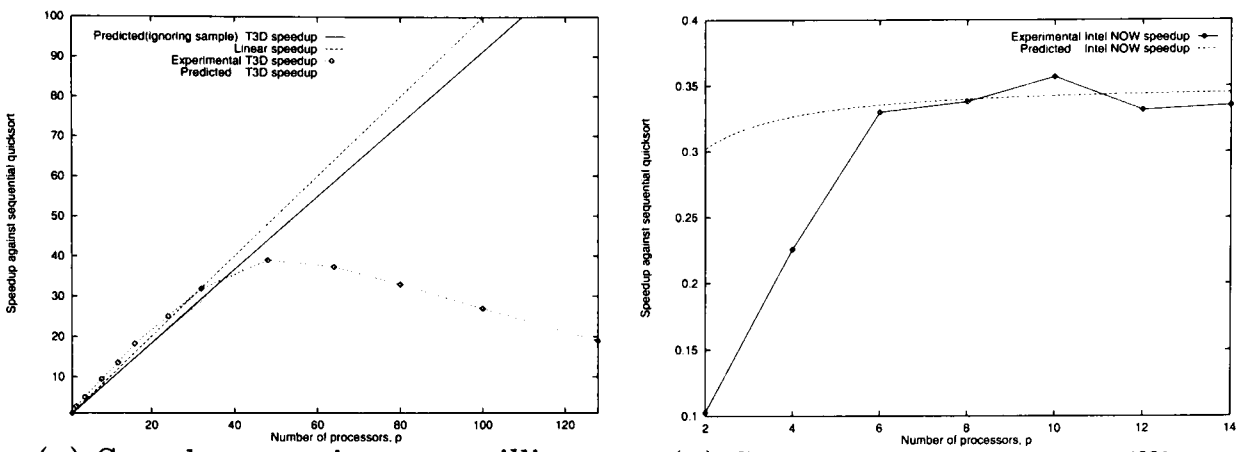
in number of operations. This expression for C does not take into account the fact that some of the communication and computation may be overlapped. When this is taken into account it is more usual to use the max function rather than addition. The use of the addition operation simplifies the analysis and, in general, the difference between the two costs is not significant [85]. Asymptotic analysis is also preserved by using addition instead of the max function as the same terms dominate. Since a BSP program is the sequential composition of supersteps, the cost of the program is the sum of the individual supersteps. Real time estimates for the execution of a superstep, and hence a BSP program, can be obtained by factoring in the processor speed parameter s. In this case the run time of the superstep would be

$$T = w/s + h \cdot g/s + l/s.$$

Note that an h-relation may involve transferring a total volume of data ranging from h (all but one h_i are 0) to ph (all of the h_i are equal). Hence the BSP cost model is implicitly asserting that communication performance is determined by the time to get data into and out of the network, and not by congestion inside it. The costs given by this model are not only theoretical costs, but closely match the observed execution times over a wide variety of applications and target architectures (see, for example, Crumpton and Giles [26], Reed *et al.* [101]).

Both MPI [90] and PVM [42] are message passing libraries with a large number of primitives, and both are implemented on a large number of platforms. Proponents of MPI and PVM thus claim that because of this prevalence of implementations of the library, it is a means of attaining portable parallel computation. However, BSP programs can also be viewed as portable and not only in a syntactic and semantic sense. While it is possible to run a single MPI or PVM code on a large number of platforms unchanged, the suitability of a particular set of primitives may not be optimal for all platforms. The architecture of *BSPlib* makes it possible for applications to be portable in a performance sense. For example, Figures 2.4(a) and (b) from Hill *et al.* [63] show predicted speed-ups of parallel sample sort [41, 72, 102, 45] over sequential quicksort [69]. These predictions are based entirely on the benchmarked BSP parameters of the 'machine' and an analysis of the parallel sample sort algorithm.

Figure 2.4(a) shows the predicted and actual speed-up achieved when sorting one million elements on various numbers of processors. Two sets of graphs are shown in the first figure. The two straight lines show an ideal linear speed-up and the speed-up achieved by ignoring the lower-order terms accounting for the cost of computing and communicating the samples used as splitters. The other two curves include these costs and demonstrate that



(a) Speedups sorting one million elements on the Cray T3D.

(b) Speedups sorting ten million elements on a NOW.

Figure 2.4: Parallel sample sort predicted versus actual speed-up

from approximately fifty processors, lower-order terms begin to dominate the total cost and reduce both the BSP cost model predicted and achieved speed-ups.

Figure 2.4(b) shows results using the same analysis and an experiment sorting ten million elements on a network of workstations (NOW) using 10Mbps shared Ethernet. In this case, the BSP cost model of the problem accurately predicts the achievable speed-up and shows that a speed-up of greater than one is neither predicted nor achieved.

2.6 Conclusion

The BSP model and its cost model reviewed in this chapter are assumed throughout the thesis. In arriving at the model, its genealogy has been traced out from the foundations of the constituent RAMs. This definition of BSP is quite well accepted and forms the basis of a number of implementations of the model on real machines. The relaxation of synchrony has played a major role in the derivation of the model. Computation and synchronisation in the XPRAM model are decoupled to allow the efficient simulation of certain PRAM algorithms. In the BSP model this is known as the automatic mode and supports a PRAM style programming in which memory and communication are controlled by the computer to support a large global memory. The BSP model also allows a direct mode in which the program controls the communication and the allocation of memory. This is the BSP model assumed in this thesis and represents a further loosening of synchronisation than is present in the XPRAM.

The direct method has allowed the BSP model to be used with algorithms

which showed considerably less slack than is required for efficient simulation of PRAM algorithms in the automatic mode. This direct mode has also reduced slackness to a means of convenience for the programmer. For machines with larger relative values of the BSP parameters l and g, one can still find algorithms for which reasonable speed-up can be achieved.

One can also see in the BSP model a similarity with the objectives of the RAM. Certainly the BSP cost model expression can be projected onto a computation only component, thus allowing parallel time complexity claims in terms of speed-up and computation efficiency to be made. But, it is also possible to study an algorithm's communication and synchronisation efficiencies.

Although the parameters l and g are fixed for a particular computer, the types of machines dealt with are typically only approximations to the BSP computer, in which a substantial runtime system is at work making this approximation as true as possible. The determination of the BSP characteristics (that is, the parameters l, g and s) of such a 'BSP computer' is by sampling using suitable benchmarks. The results of these benchmarks return sampled values of l and s with reasonably small variances, however the model asserts that the cost of communication is independent of volume and hence largely independent of the pattern of communication. In practice this is not automatically so and it is up to the runtime system to make sure this parameter behaves more like the constant its definition implies than a random variable.

Symbol	Description
g	Cost of sending a single word to the remote memory of another processor.
h	$\max_{i \in \langle p \rangle}\{h_i\}$. The resultant communication pattern is called an h-relation.
h_i	$\max\{s_i, r_i\}$. The contribution to the h-relation by processor i.
i, j	Processor indices.
l	Latency of the BSP communications interconnect. Related to the XPRAM parameter L.
L	Time to synchronise a remote memory reference in the XPRAM. Also referred to as the periodicity of the XPRAM.
p	Number of processors involved in a computation or present in a machine.
r_i	Number of messages received by processor i in a particular superstep.
s	Instruction rate of the processor, usually measured as a megaflop rate.
s_i	Number of messages sent by processor i in a particular superstep.
t	$\max\{t_i \mid i \in \langle p \rangle\}$. The total work performed in a particular superstep.
t_i	$w_i/g + s_i + r_i$. The work performed by processor i in a particular superstep.
v	Number of virtual processors or threads involved in computation. The ration v/p is called the Parallel slackness
w_i	Number of local operations performed on locally held data by processor i in a particular superstep.

Table 2.1: Summary of symbols used in the descriptions of the BSP and XPRAM models

Chapter 3

BSP Programming Libraries

In Chapter 2 a variant of the BSP model was arrived at which was SPMD, message passing, operated in supersteps, communicated in h-relations and synchronised globally. There have been a number of implementations of this BSP model, either in the form of imperative programming languages with special constructs for the BSP primitives or as libraries for calls from programming languages such as FORTRAN and C.

This chapter documents the background to BSP programming libraries and describes the instrument for the experiments in this thesis: *BSPlib* [66, 67, 108]. In doing so the background of earlier BSP programming libraries and programming languages is sketched.

As part of the work for this thesis, three additional ports of *BSPlib* were produced:

- A TCP/IP version, *BSPlib*/TCP, which uses a messaging or transport layer based on stream sockets using the BSD socket interface. Although one would expect that such an implementation could be universally applicable on UNIX systems, this is not the case. Problems with the implementation or interpretation of some of the more esoteric features of the TCP/IP are not universally or uniformly implemented. An example of one such feature is the setting of send and receive high-water marks.

- The UDP/IP implementation, *BSPlib*/UDP, solves certain problems with the scalability in the presence of many TCP/IP circuits which arise out of the nature of the protocol. Because the connectionless UDP/IP protocol operates at a lower level it allowed a greater degree of control by the *BSPlib* runtime system. The need for this greater control and the requirement for reliability lost in dispensing with the TCP/IP protocol led to a reliable transport protocol being developed

which made very few assumptions on the implementation of the BSD datagram socket interface. The result was an implementation of the library which is somewhat more portable than the TCP/IP version. Chapter 4 describes both the TCP/IP and UDP/IP implementations and how performance is optimised using the BSP superstep structure and the global state of the communicating parallel partners in a BSP computation to control congestion.

- The reliable packet protocol developed in the UDP/IP implementation was re-implemented directly as a device driver on a popular 100BASETX network interface card, the 3COM 3C905B-TX NIC [1]. This prototype cluster version of *BSPlib*, referred to as *BSPlib*/NIC, is discussed in Chapter 5 with performance benchmarks discussed in Chapter 6.

It has also been necessary to extend the runtime support of *BSPlib* to support the network of workstations and cluster versions of the library. These extensions are done in a manner which preserves the user interface of the *BSPlib* runtime system (Section 3.3).

Benchmarks measuring the BSP parameters g and l show that these parameters behave as random variables sampled over numerous trials. Typically, their mean values under a particular benchmark are cited as the machines' g and l parameters [58]. This random behaviour is particularly evident in these NOW and cluster versions of *BSPlib* [62] and makes it difficult to separate l and g in a benchmark when $hg \approx l$. Consequently, the benchmarks are typically devised so that $hg \gg l$. The effect of this is to place an emphasis on g, arguing that l is a term that can be overcome, not by parallel slackness, but by the equivalent notion of bulk communication operations, as pointed out in Chapter 2. A treatment of g as a random variable is considered in Chapter 7 where it is demonstrated how the effects of variance in the parameter detract from the compositionality of systems in parallel to form large ensembles.

The arguments in Chapter 7 motivate the minimising of variance in protocol throughput in order to improve the value of g. They also suggest a regression technique against which such protocols can be measured for the detection of unexplained conflicts or congestion. The two implementations of the reliable packet protocol, *BSPlib*/UDP and *BSPlib*/NIC, are tested in this way.

3.1 *BSPlib* and BSP programming libraries

An earlier library, *The Oxford BSP Library* by Miller [92], implemented the model with communication semantics and primitives similar to the Cray

SHMEM Direct Remote Memory Access or DRMA [6] one sided communication. The DRMA primitives in this library allow the storing and fetching of static (in the C sense) data in the instances of the SPMD program on other processing nodes participating as parallel partners of the same computation. The library also includes collective communication primitives such as broadcast and reduce. The Oxford BSP Library has been implemented on TCP/IP, PVM, PARMACS, MPL and SYSV shared memory.

Other implementations of BSP libraries include the *Green BSP Library* [50], BSP-L [17] and the Paderborn University BSP Library, PUB [10]. Implementation proposals for BSP programming languages also appeared in the form of GPL [87] and OPAL [83].

In 1995 a discussion on the BSP World-Wide mailing list distilled these efforts into a proposal for a standard set of primitives for a BSP library [66]. This proposal was influenced by PVM, MPI and Split-C projects as well as the Cray SHMEM primitives.

The result of the standardisation was a small set of primitives supporting one sided communication as DRMA primitives (which could reference heap and stack data as well as static data) in addition to message passing. Collective communication primitives, although specified, are left to a higher level library implemented using these primitives, or where available using low level primitives provided by the architecture. The primitives, summarised in Table 3.1 (from Hill *et al.* [66]), provide an efficient and simple programming interface in which programming errors are both minimised and easy to detect. This is because circular dependencies, causing deadlocks, cannot be created with these one sided communication primitives. Also, because of the superstep structure, message passing errors such as unmatched send or receives are quickly detected at the next barrier, localising the point of failure.

BSPlib is an implementation of the proposal that has been ported to many machines and provides interfaces to both FORTRAN and C. Some of these machines with their accompanying BSP machine parameters are listed by Skillicorn *et al.* [108] and Hill [58].

In general *processor* synchronisation is not required to facilitate the style of BSP computation outlined in this thesis, but some form of synchronisation of *data* is required. Consequently, in the presence of data, the cost of synchronisation is the latency of the network (including protocols) plus some small overhead (as a processor can be considered synchronised immediately after receiving the last of the messages from the other processes at the end of the superstep). In *BSPlib* the amount of communication in the superstep is determined by a reduction in which each processor learns of the global communication load of the superstep. Since this reduction cost depends only

Class	Primitive	Meaning
SPMD	`bsp_begin`	Start of SPMD part
	`bsp_end`	End of SPMD part
	`bsp_init`	Simulate dynamic processes
	`bsp_abort`	One process halts all
	`bsp_nprocs`	Number of processes
	`bsp_pid`	My process identifier
	`bsp_time`	Elapsed local time
	`bsp_sync`	Barrier synchronisation
DRMA	`bsp_push_reg`	Make area globally visible
	`bsp_pop_reg`	Remove global visibility
	`bsp_put`	Copy to remote memory
	`bsp_hpput`	Unbuffered put
	`bsp_get`	Copy from remote memory
	`bsp_hpget`	Unbuffered get
BSMP	`bsp_set_tagsize`	Set tag size
	`bsp_send`	Send to remote queue
	`bsp_qsize`	Number of messages in queue
	`bsp_get_tag`	Get message tag
	`bsp_move`	Move message from queue
	`bsp_hpmove`	High performance move

Table 3.1: *BSPlib* primitives

on p, it could be included in the cost of the synchronisation, l, for purposes of the BSP cost model.

It will be shown that performing this reduction first in order to determine the residual outstanding communication (rather than, for example, tagging the last message communication) has significant benefits for the implementations presented in this thesis (Chapters 4, 5 and 6). This allows each processor involved in the communication to be aware, at all times, of the global applied load. Before the reduction each processor expects to be involved in whatever communication is required by the reduction; and after the reduction each processor is aware of the outstanding communication until the start of the next computation phase.

This operation of *BSPlib* does not impact the semantics of the program and the model shown in Figure 2.3 on page 18 is equivalent to operation of *BSPlib* shown in Figure 3.1.

Hill and Skillicorn [67] show that this structure and the delaying of messages is what facilitates the packing of small messages into large messages, amortising start-up costs over larger messages and allowing the interconnect to operate near the limiting bandwidth. This scheme also allows the scheduling of communication, either by the random selection of partner nodes or

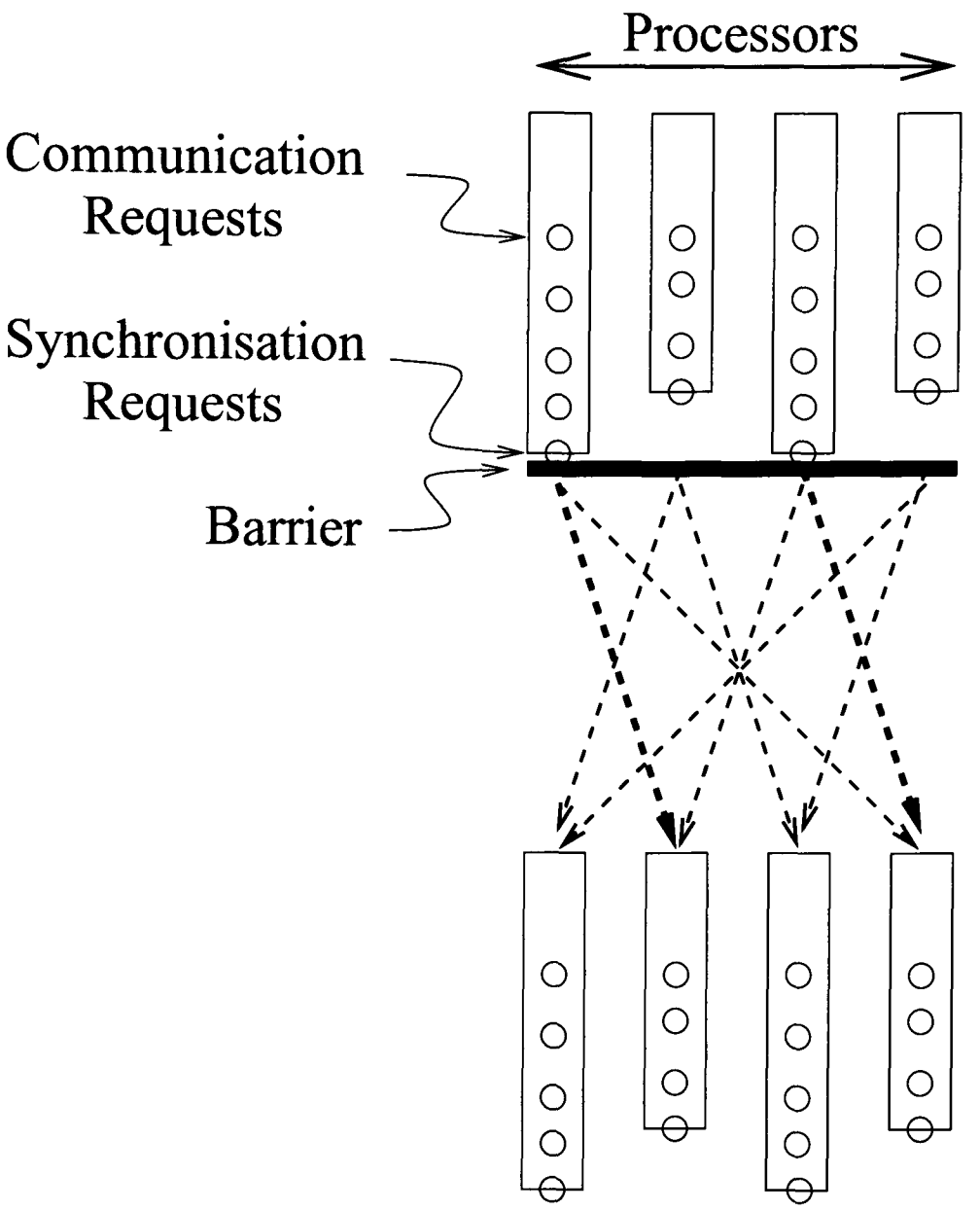


Figure 3.1: *BSPlib* superstep structure

according to the successive rows of a $p \times p$ Latin square, to avoid congestion at the destination nodes. Such tactics are required in realizing the BSP cost model. Consider, as an example, the situation where p processors are involved in a computation where each sends out $h/(p-1)$ messages to each of the other processes:

```
for (p = 0; p < bsp_nprocs(); p++)
   if (p != bsp_pid())
      bsp_put(p,&src,&dst,offset,nbytes);
```

The $p-1$ messages of size $h/(p-1)$, without rescheduling, would all be sent to the first processor, followed by $p-1$ messages of the same size being sent to the second, and so on. The result would be that the first processor would have to deal with h-messages not all properly overlapped with the

messages sent to the other processors. The resulting cost of the h-relation would be αphg (for $\alpha > 1$) rather than the cost model expectation of hg. Other reasons for performing this operation according to such schedules include minimisation of buffer overruns, recovery from which is particularly expensive in workstation and cluster environments.

As well as packing messages and choosing good schedules, it is also possible, with global knowledge of the outstanding communication, to exert an applied load on the network or interconnect which is optimal in terms of throughput (Chapter 4). Such global knowledge (a theme in this thesis) is used not only in this higher-level control of applied load, but also to improve global communication using implicit acknowledgements and buffer sizing, also taking advantage of the characteristics of the local network to 'bracket' the unreliable portions of the network (Chapter 5); as well as fault tolerance and process migration (Chapter 8). These can all be considered as exploitations of the BSP superstep structure. This exploitation, together with the combining and scheduling mentioned above is also covered by Donaldson *et al.* [30].

3.2 Porting *BSPlib*

There are two extremes of machine classes on which *BSPlib* has been implemented: shared memory systems and distributed systems. All the implementations discussed in this thesis are based on the distributed memory model as the 'machine' consists of a loosely coupled collection of workstations either in the open sense of *Networks of Workstations* of the Berkeley NOW project [2]; or in the more tightly coupled sense of a closed collection forming a cluster with a dedicated interconnect [97]. In both cases the operating system images on each processor are independent and synchronisation between them is very loose.

Similar to the abstract device interface upon which certain versions of MPI are based [53], the distributed and shared memory implementations of *BSPlib* depend on an abstraction layer, mapped to the primitives of the underlying 'architecture'. These primitives for the distributed memory implementation are listed in Table 3.2. A port of *BSPlib* can be accomplished by writing an intermediate message passing layer which provides these primitives. These would then be used by an upper layer of *BSPlib*. The string `xxx` in names of the primitives in the figure, represent the lower layer upon which the mapping layer has been implemented (for example, `xxx` is replaced by `tcp` in the TCP/IP implementation and `udp` in the UDP/IP implementation).

The informal semantics of this message passing layer might be inferred from

Primitive	Meaning
`bspxxx_set_buffer_size`	Negotiate buffer size
`bspxxx_env`	Set and distribute environment variables
`bspxxx_init`	Spawn processes on remote nodes
`bspxxx_nonblock_send`	Send data to remote peer
`bspxxx_block_recv`	Receive data from remote peer
`bspxxx_wait_sends`	Wait for re-usability of send buffers
`bspxxx_okto_send`	Check whether lower layer can accept a send
`bspxxx_probe`	Check message presence by type or source
`bspxxx_messg_stopall`	Abort: one process stops all
`bspxxx_messg_exit`	Tidy exit from the message passing system

Table 3.2: Low-level message passing primitives of the message passing layer

the table and message delivery is guaranteed if the message is accepted for transportation. This means each message has to be accepted or rejected in its entirety and the rejection of a message can be used to indicate a temporary condition such as the local send queue is full and awaiting draining by the network, or the remote receive queue is full awaiting draining by the process. Clearly, any symmetric code where the senders simply temporarily suspend packet injection will cause deadlock unless such suspension is interleaved with attempts to drain the local receive queues, thus alleviating a potential similar problem on a peer process, allowing progress to be maintained. The usage of this intermediate message passing layer within *BSPlib* is restricted in order to avoid deadlock in the presence of large amounts of data and resembles the following code fragment:

```
i = 0;
while (1)
   {
   if (sndbytes < limit) /* arbitrary large */
      /* sending phase */
      do {
         sd = 0;
         for (; !sd; i = (i+1)%nprocs)
            if (i != mypid && okto_send(i))
               sd = bspxxx_nonblock_send(outbuf,
                           BLOCK_SIZE,i,0,1);

         if (sd) sndbytes += BLOCK_SIZE;
         }
      while(!sd);
```

```
    /* receiving phase */
    frompid = -1; buftype = 0;
    while(bspxxx_probe(&frompid,&buftype,0,0)))
       {
       frompid = -1; buftype = 0;
       bspxxx_block_recv(inbuf,BLOCK_SIZE,
                 &frompid,&buftype,&nbytes);
       recbytes += nbytes;
       frompid = -1; buftype = 0;
       }
    if (sndbytes >= limit && recbytes >= limit)
       if (mypid == 0)
          exit(0);
       else
          bspxxx_messg_exit(0);

} /* while(1) */
```

This code fragment is taken from a test of the messaging layer. This particular test is designed to stress the system by passing arbitrarily large amounts of data between two processors. The variable `limit` is set to the number of bytes which each processor must send and receive before the test terminates successfully.

Provided the **`bspxxx_nonblock_send`** operation fails only when the 'data pipe' is full, making the condition transient if a partner process is sure to consume the data, and that in each case either all the data or none of the data are accepted for transmission in each call, deadlock cannot occur. By having each processor back-off sending and attempt to consume any data queued for reception, the failure of **`bspxxx_nonblock_send`** is guaranteed to be transient. Comments on a formal demonstration that this behaviour is required in order that the protocol be deadlock free appear in Chapter 5. This is more fully dealt with by Simpson *et al.* [107].

3.3 Support for *BSPlib* over NOWs and Clusters

The low level messaging libraries used in the NOW implementations are implemented using the BSD transport layer interface, BSD Sockets [23, 24], which has also been implemented on top of the SVR4 STREAMS transport layer interface. The *BSPlib*/NIC prototype cluster messaging implementation uses an interface that resembles the UDP/IP sockets interface and hence the IP terminology is also applicable.

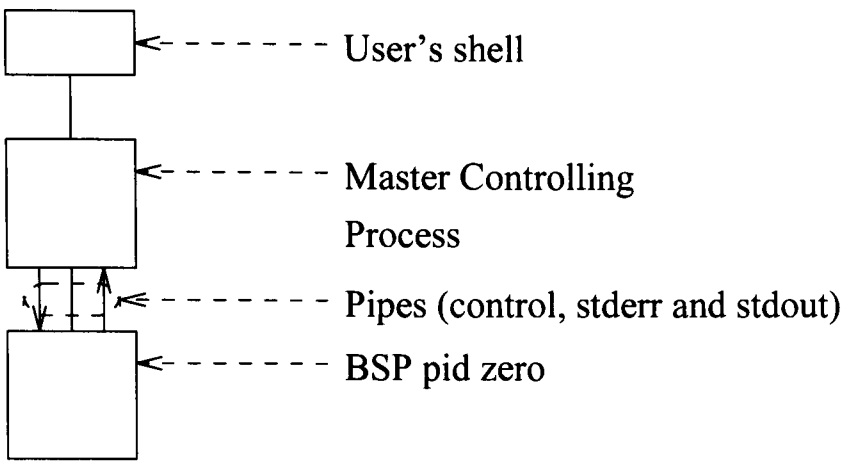


Figure 3.2: Process structure on workstation initiating BSP computation.

In the NOW environment, no additional hardware or software is required apart from the homogeneous set of UNIX workstations, their attached IP network and *BSPlib*. The applicability of the cluster implementation, however, is less universal and the implementation assumes the Linux 2.0 kernel and specific network interface cards. Once the runtime system is configured, the user interface is identical to all the other implementations of *BSPlib*. In order to keep the interface the same and the configuration as simple as possible, a scheme is used where only one port on a workstation need be known. When the initiating process makes the `bspxxx_init` call an attempt is made to connect to this well-known port on a number of machines whose names are listed in a file. Each of the machines in the list is tried in turn until the total number of positive responses is equal to the number of BSP processes requested for the execution, or until the list is exhausted, in which case the number of available workstations becomes the value for `bsp_nprocs`. The initiating process (as part of the `bspxxx_init` call) then spawns a child creating the process structure shown in Figure 3.2.

On each processor participating in the BSP 'machine' there is a daemon listening in on the well-known port for requests. For each incoming connection accepted, a slave process is spawned. This process in turn spawns a copy of the SPMD application as the BSP process, creating the process structure shown in Figure 3.3.

The library routines communicate with the controlling parent processes using pipes and the master and slave processes keep the originating sockets open for controlling the BSP job and communicating output back to the originating process. Part of this control information in the *BSPlib*/TCP *BSPlib*/UDP implementations is to communicate the locally bound dynamically assigned ports to all the partners in the BSP computation. On return from the `bspxxx_init` function in each of the BSP processes the process structure is as illustrated in Figure 3.4.

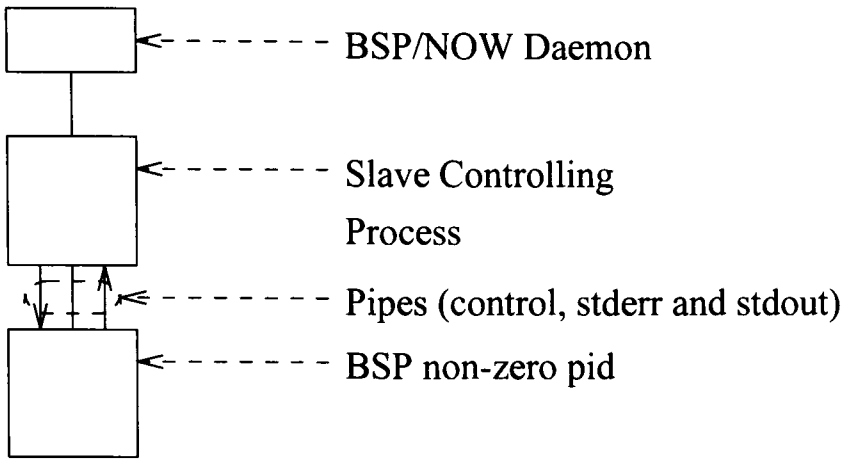


Figure 3.3: Process structure on workstation running daemon.

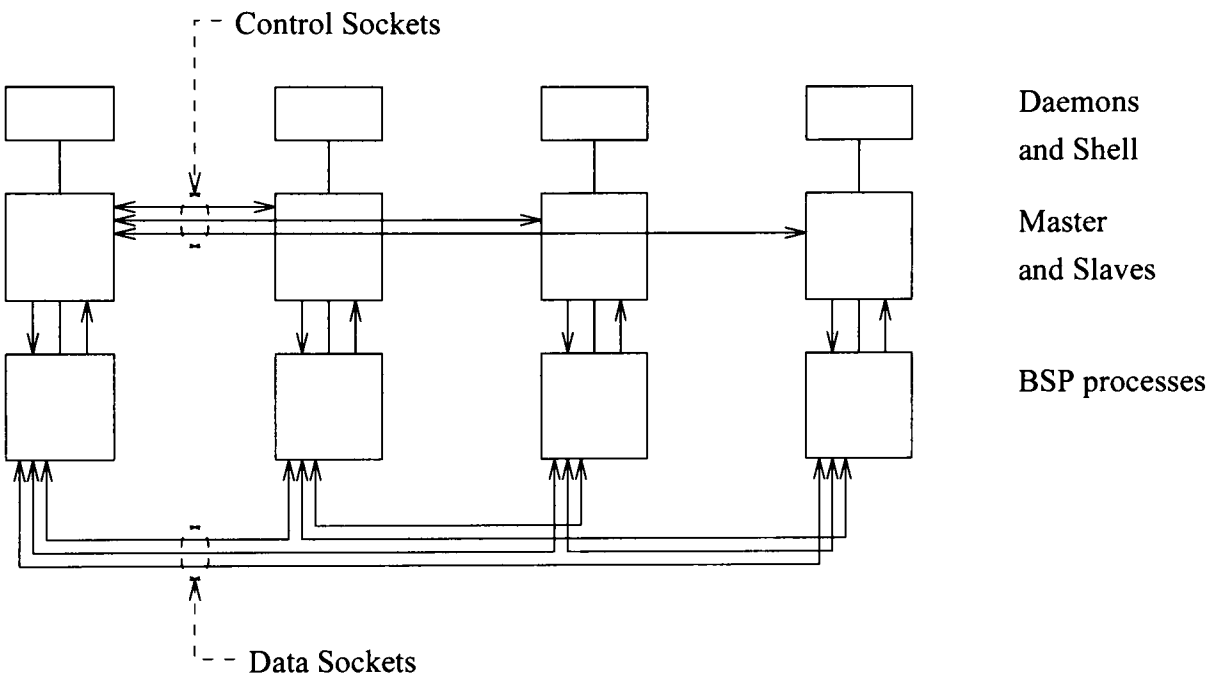


Figure 3.4: Process and circuit structure on the BSP "machine".

In the process described above, the daemon processes will accept any incoming connections, and will spawn a slave process for each connection, regardless of local workload, up to an arbitrary ceiling in the number of active slave processes. The daemon could easily be extended to reject connections based on the local workload of the processor, the time of day, or the presence of any logged in and active users.

3.4 Other message passing libraries

Two popular message passing libraries used in support of parallel computation are PVM and MPI. Of these two libraries, it is MPI that is generally

supported by commercial hardware and software manufacturers. PVM is a popular public-domain, self contained message passing system often used to run parallel applications on networks of heterogenous Unix systems, symmetric multi-processors, massively parallel processors and clusters of workstations.

Public-domain versions of MPI are also available, but it is probably its commercial success that has lead to it becoming ubiquitous and also to certain benchmarks based on it becoming popular. One such class of benchmarks are the NAS Parallel Benchmarks [4] used in this thesis.

In MPI, communication amongst processors is either point-to-point or collective. Unlike *BSPlib*, the collective communication in MPI can take place amongst a subset of the processors or tasks. Each communication primitive includes a *communicator* which specifies the *group* and *context* of the communication. This arrangement, plus the MPI primitives for creating and querying communicators allows not only for the elegant partitioning of problems but also for the sharing of code. It facilitates the development and use of libraries based on MPI which can be incorporated in further libraries, also based on MPI, or used in MPI programs. The collective communications of MPI are similar to those described in the *BSPlib* definition [66], except that in *BSPlib* collective communication necessarily includes all the processors.

There are also a number of different styles of point-to-point communication in MPI. These are all two sided in which a send primitive must match a corresponding receive primitive. This matching can be conditional on the value of a *tag* or source processor (or both). Communicating in this manner is similar to the *BSPlib Bulk Synchronous Message Passing* (or BSMP) style. This style of programming in the absence of the superstep structure makes deadlock resolution difficult. Support for various types of paired communication include synchronous, asynchronous, blocking, non-blocking, buffered and unbuffered (see, for example, the description by Hwang and Xu [73]).

MPI and *BSPlib* are different in another important respect. One can view the various instances of *BSPlib* send (`bsp_put`) and receive (`bsp_get`) primitives in a superstep as the specification of a gather and scatter to be executed during the following synchronisation (`bsp_synch`). However, any data communicated are simply treated by *BSPlib* as sequences of bytes regardless of the underlying data type. This was a conscious decision amongst the authors of *BSPlib* and excludes the possibility of performing a *BSPlib* computation over a heterogenous collection of processors. MPI on the other hand, includes an abstract type on all communication primitives. This allows the datum to be translated into a suitable concrete type for transfer to a processor which has a possible different concrete representation for the

abstract type. MPI also has the notion of *derived type* in which a structured type descriptor can be dynamically built from existing type descriptors and includes information on how to construct an item of the derived type. For example, a datum of the derived type might be constructed by selecting entries in an array which are spaced apart by a given stride.

A later version of the MPI standard, called MPI-II, added the ability to dynamically create tasks, a feature not found in *BSPlib*. MPI-II also added primitives for one sided communication similar to the DRMA primitives of *BSPlib* shown in Table 3.1. This version, however, is not as widely supported by hardware and software manufacturers as the first version is.

3.5 Conclusion

This chapter provided a review of BSP programming libraries and programming languages with special attention being paid to the *BSPlib* programming library of Hill *et al.* [66], Skillicorn *et al.* [108].

Also described was the porting effort for message passing systems as well as the structure created to support execution of *BSPlib* programs in a cluster or network of workstations environment.

A significant amount of software re-use was achieved by extending *BSPlib* and porting rather than re-implementing a BSP programming library. The benefits to this approach are that new implementations deploying new ideas can be implemented quickly and can be benchmarked against a significant existing base of implementations. Also, because of the popularity of *BSPlib*, new implementations are immediately usable by the many existing *BSPlib* applications, providing new platforms for that body of code to run on unchanged.

Chapter 4

Global Congestion Control

The BSP computation model and its implementation in *BSPlib* reviewed respectively in Chapters 2 and 3 provide a framework which can be exploited for the purposes of communication optimisation, not just in a local or point-to-point manner, but at a global level. This is so because the bulk communication and computation nature of BSP and the *BSPlib* primitives encourage program and algorithm development which seeks to balance both communication and computation among the participating processors. Furthermore, only a global synchronisation is provided in which all the processors of the computation must participate.

Consequently, without any application changes, the existing and required communication can be augmented with very little extra state information pertaining to application communication requirements. The resultant distributed global state can be used to make independent and consistent decisions at each of the nodes in the parallel computation to improve the performance of the communication.

This chapter describes the use of such a derived global state to solve the problem of congestion in communication interconnects. This is achieved by making each of the participating nodes aware of the global communication requirements and to have them act in a manner which controls the global applied load. The types of interconnect which can benefit from this technique include all communication interconnects whose protocol performance demonstrates a drop in successful load as applied load increases. This includes most data network protocols [109]. But the technique is not restricted to interconnects approaching saturation and is also applicable wherever this behaviour is observed at the application level as it reduces the probability

---

This chapter is based on the paper "Predictable communication on unpredictable networks: Implementing BSP over TCP/IP and UDP/IP" by Donaldson *et al.* [32].

of buffer starvation, overruns and underruns in the end-points as well as within the interconnect.

To put the work presented in this chapter into perspective, the technique represents the controlling of the global load on an interconnect at a level which is either above or inside the communication protocol layers, but in either case below the application level. That is, it is implemented within *BSPlib*. Implementation above the protocol layer provides only an indirect control of the applied load on the network medium whereas an implementation within the transport protocol facilitates direct control. Further, protocols tend to be point-to-point in nature and in the absence of any global information are optimised for this case even if this leads to a solution which is less than globally optimised.

The theme of exploiting the global nature of BSP computation for communication performance is taken up again in Chapter 5 where it is applied to the protocol layer itself. Even above the application, taking the BSP structure into account can be useful in making process migration decisions as it provides a natural and convenient setting for consistent checkpoints and migration decisions. Augmented with a scheme for maintaining global computation loads in a network of workstations, this provides a convenient and transparent mechanism for the efficient use of workstations and is the subject of Chapter 8.

The congestion problem addressed here is most acute in shared-media networks and protocols such as TCP/IP and UDP/IP, where there is far greater potential to waste bandwidth. For example, if two processors try to send more or less simultaneously, collision in the Ether means that neither succeeds, and transmission capacity is permanently lost. The problem is compounded because it is hard for each processor to learn anything of the global state of the network. Nevertheless, significant performance improvements are possible and will be demonstrated.

The BSP runtime system makes an important contribution to the value of the BSP parameters by acting to improve the effective value of g and l by the way it uses the architectural facilities. For example, Hill and Skillicorn [67] show how orders of magnitude improvements in g can be obtained for architectures using point-to-point connections by packing messages before transmission, and by altering the order of transmission to avoid contention at receivers [67]. Also, Hill and Skillicorn [68] show how careful construction of barriers can reduce the value of l.

Techniques specific to the implementation of *BSPlib* are described that ensure that the variation in g is minimised for programs running over bus based Ethernet networks. Compared to alternative communications libraries such as Argonne's implementation of MPI [53], these techniques have an absolute

performance improvement over MPI in terms of the mean communication throughput, but also have a considerably smaller standard deviation. Good performance over such networks is of practical importance because networks of workstations are increasingly used as practical parallel computers.

Section 4.1 describes a technique for minimising delays due to contention, and also minimising the standard deviation of such delays. Section 4.2 describes how to tune the technique to maximise throughput.

4.1 Minimising g in bus based Ethernet networks

Ethernet (IEEE 802.3) is a bus based protocol in which the media access protocol, 1-persistent CSMA/CD (Carrier Sense Multiple Access with Collision Detection) proceeds as follows. A station wishing to send a frame[1] listens to the medium for transmission activity by another station. If no activity is sensed, then the station begins transmission and continues to listen on the channel for a collision. After twice the propagation delay, 2τ, of the medium, no collision can occur, as all stations sensing the medium would detect that it is in use and will not send data. However, a collision could occur during the 2τ window. On detection of a collision, the transmitting station broadcasts a jamming signal onto the network to ensure that all stations are notified of the collision. The station recovers from a collision by using a *binary exponential back-off* algorithm that re-attempts the transmission after $t \times 2\tau$, where t is a random variable chosen uniformly from the interval $[0, 2^k]$ (where k is the number of collisions this attempted transmission has experienced). For Ethernet, the protocol allows k to reach ten, and then allows another six attempts at $k = 10$ (at which point transmission of the packet is aborted) (see, for example, King [81]).

Analysis of this protocol [118] shows that $S \to 0$ as $G \to \infty$ (where S is the rate of successful transmissions and G the rate at which messages are presented for delivery—symbols introduced in this discussion are summarised in Table 4.1 at the end of this chapter), whereas for a p-processor BSP computer one would expect that $S \to B$ as $G \to \infty$. One should expect $g = p/B$, where B is some measure of the bandwidth and is constant.

In the case of BSP computation over Ethernet, the effect of the exponential backoff is exaggerated (larger delays for the same amount of traffic) because the access to the medium is often synchronised by the previous barrier synchronisation and the subsequent computation phase. For perfectly-balanced

[1] In the UDP/IP implementation of *BSPlib* using sockets, a packet is the IEEE 802.3 frame that includes the payload, which can be up to 1430 bytes, a *BSPlib* header of 20 bytes, a UDP header of 8 bytes, an IP header of 20 bytes, the MAC header of 22 bytes and the MAC checksum of four bytes at the end of the frame.

computations and barrier synchronisations, all processors attempt to get their first message onto the Ethernet at the same time. All fail and back off. In the first phase of the exponential back-off algorithm, each of p processors choose a uniformly-distributed wait period in the interval $[0, 4\tau]$. Thus the expected number of processors attempting the retransmits in the interval $[0, 2\tau]$ is $p/2$, making secondary collisions very likely. If the processors are not perfectly balanced, and a processor gains access to the medium after a short contention period, then that process will hold the medium for the transmission of the packet, which will take just over $1000\mu s$ for 10Mbps Ethernet. With high probability, many of the other processors will be synchronised by this successful transmission due to the 1-persistence of this protocol. The remaining processors will then contend as in the perfectly-balanced scenario.

In terms of the performance model, this corresponds to a high applied load, G, albeit for a short interval of time. When $S \approx G$ then this burstiness of the applied load would not be detrimental to the throughput and would average out. It is always the case that $S \leq G$ as the successful load is a subset of the applied load.

Fortunately, the BSP model allows assumptions to be made at the global level based on local data presented for transmission. At the end of a superstep and before any user data are communicated, *BSPlib* performs a reduction, in which all processors determine the amount of communication that each processor intends sending. From this, the number of processors involved in the communication is determined. For the rest of the communication the number of processors involved and the amount of data are used to regulate (at the transport level) the rate at which data are presented for transmission on the Ethernet. By using BSD Socket options (`TCP_NDELAY` or `IP_TOS_LOWDELAY`), the data presented at the transport layer are delivered immediately to the MAC layer (ignoring the depth of the protocol stack and the availability of a suitable window size). Thus, by pacing the transport layer, pacing can be achieved at the MAC or link layer. This has the effect of removing burstiness from the applied load.

Most performance analyses give a model of random-access broadcast networks which provide an analytic, often approximate, result for the successful traffic, S, in terms of the offered load, G. Hammond and O'Reilly [55] present a model for slotted 1-persistent CSMA/CD in which the successful traffic, S, can be determined in terms of the offered load:

$$\begin{aligned} W &= \left(\frac{E}{\tau}+1\right) G\tau\, e^{-3\frac{G\tau}{E}} \left(3-2\,e^{-\frac{G\tau}{E}}\right) E^{-1}+3 \\ &\quad -3\left(\frac{E}{\tau}+1\right) G\tau\, e^{(-\frac{E}{\tau}-1)G\tau E^{-1}} E^{-1} \end{aligned}$$

$$
\begin{aligned}
& -3\,e^{-\frac{G\tau}{E}} + 3\,G e^{-\left(\frac{E}{\tau}+2\right)G\tau E^{-1}} + e^{-3\,\frac{G\tau}{E}} \\
& -\left(\frac{E}{\tau}-2\right) G\tau\, e^{-\left(\frac{E}{\tau}+4\right)G\tau E^{-1}} E^{-1} \\
S(G) \;=\; & G e^{-3\,\frac{G\tau}{E}} \left(3-2\,e^{-\frac{G\tau}{E}}\right) W^{-1} \qquad (4.1)
\end{aligned}
$$

where τ is the end-to-end propagation delay along the channel (bounded by $25.65\mu s$, i.e. the time taken for a signal to propagate $2500m$ of cable and 4 repeaters), and E is the frame transmit time (which for 10Mbps Ethernet with a maximum frame size of 1500 bytes is approximately $1200\mu s$). This formula also assumes that the jamming time is equal to τ.

In this closed form solution, the process being modelled can be thought of as an approximation to the use of 1-persistent CSMA/CD proposed here. The artificial assumption of slotting is very roughly related to the use of slotting proposed below. In the slotted version of the protocol the stations synchronise their attempts to gain access to the medium and each frame is exactly the length required to fill a slot. While such synchronisations are not feasible in the configurations under consideration, the slotted versions of the media access protocols perform better than their non-slotted counterparts [55]. The full-frame assumption of the model is preserved by *BSPlib*'s packing of messages and the arrival process is justified in Section 4.2. One would expect such a model to provide an upper bound on the throughput that can be achieved. Any system that achieves a throughput close to these bounds in spite of not being able to synchronise must be useful. The model is used to compare the experimental results obtained in Section 4.2.2.

Since both S, the rate of successful transmissions, and G are normalised with respect to E, S is also the channel efficiency achieved on the cable. T, shown in Figure 4.1, also normalised with respect to E, is the load applied by *BSPlib* on the transport layer. The objective is to pace the injection of messages into the transport layer such that T, on average, is equal to a steady-state value of S without much variance. The value of T determines the position on the S–G curve of Figure 4.2 in a *steady state*; in particular, T can be chosen to maximize S. If the applied load is to the right of the maximum throughput in Figure 4.2, then small increases in the mean load lead to a decrease in channel efficiency which in turn increases the backlog in terms of retries, and further increases the load. Working to the right of the maximum therefore exposes the system to these instabilities which manifest themselves in variances in the communication bandwidth. This is a metric requiring minimisation, particularly in a parallel setting where it can have an impact on the compositionality of systems (Chapter 7). On the other hand, when working to the left of the maximum, small increases in the applied load are accompanied by increases in the channel efficiency which helps cope with the increased load and therefore instabilities are unlikely. As

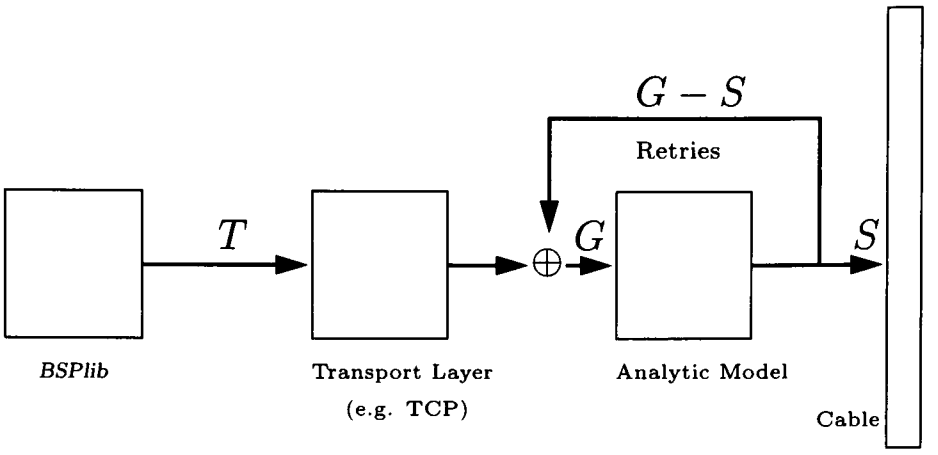


Figure 4.1: Schematic of aggregated protocol layers and associated loads

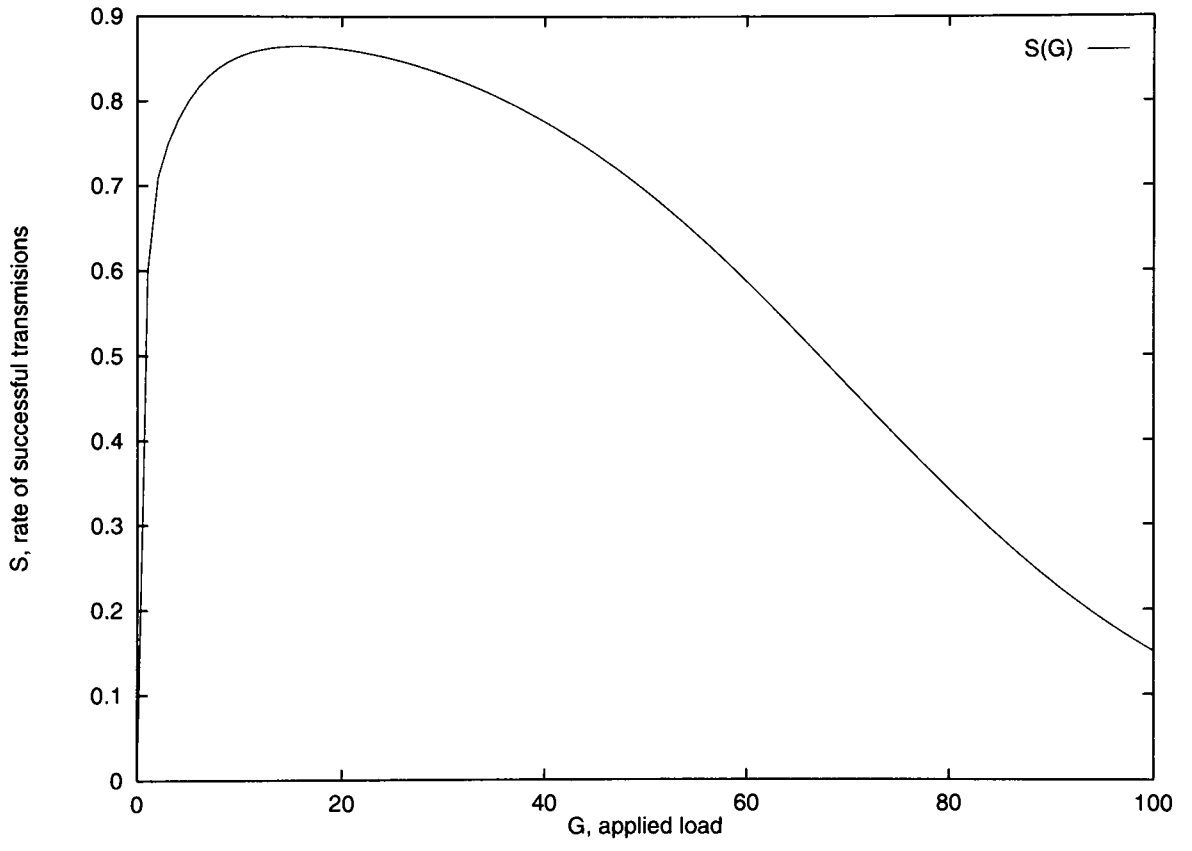


Figure 4.2: Plot of applied load (G) against successful transmissions (S)

an aside, the Ethernet exponential backoff handles the instabilities towards the right by rescheduling failed transmissions further and further into the future, which decreases the immediate applied load.

In *BSPlib*, the mechanism of pacing the transport layer is achieved by using a form of statistical time-division multiplexing that works as follows: The frame size and the number of processors involved in the communication are known. As the processors' clocks are not necessarily synchronized, it is not possible to allow the processors access in accordance with some permutation,

a technique applied successfully in more tightly-coupled architectures [67]. Thus the processors choose a *slot number*, q, uniformly at random in the interval $[0 \ldots Q-1]$ (where Q is the number of processors communicating at the end of a particular superstep), and schedule their transmission for this slot. The choice of a random slot is important if the clocks are not synchronized as it ensures that the processors do not repeatedly choose a bad communication schedule. Each processor waits for time $q\varepsilon$ after the start of the *cycle*, where ε is the duration of a slot, before passing another packet to the transport layer. The duration of the slot is chosen based on the maximum time that the slot can occupy the physical medium, and takes into account collisions that might occur when good throughput is being achieved. This mechanism is designed to allow the medium to operate at the steady state that achieves a high throughput. Since the burstiness of communication has been smoothed by this slotting protocol, the erratic behaviour of the low-level protocol is avoided, and a high utilization of the medium is ensured.

An alternative protocol, not considered here, would be to implement a deterministic token bus protocol in which each station can send data only whilst holding a 'token'. This scheme is not viable, as it is inefficient for small amounts of traffic because of the explicit token pass when the token-holding processor has no message for the 'next to go' processor. In the worst case this would double the communication time. Also, token mechanisms protect shared resources such as a single Ethernet bus. However, a network may be partioned into several independent segments, or processors may be connected via a switch. In this case, the token bus protocol would give only a single processor access to the medium at any time, therefore wasting bandwidth. In contrast, the parameters used in the slotting mechanism can be trivially adjusted to take advantage of a switch based medium. For example, for a full-duplex cross-bar switch, Q can be assumed to be 1 ($Q = 2$ for half-duplex), and ε encapsulates the rate at which the switch and protocol stacks of sender and receiver can process messages in a steady state. If the back-plane capacity of the switch is less than the capacity of the sum of the links, then Q and ε can be adjusted accordingly. Therefore, the randomised slotting mechanism is superior to a deterministic token bus scheme, as Q and ε can be used to model a large variety of LAN interconnects.

4.2 Determining the value of ε

In any steady state, $T = S$ because, if this were not the case, then either unbounded capacity for the protocol stacks would be required, or the stacks would dispense packets faster than they arrive, and hence contradict the steady-state assumption. Since ε is the slot time, packets are delivered with

a mean rate of $1/\varepsilon$ packets per microsecond. Normalising this with respect to the frame size E gives a value for $T = E/\varepsilon$ packets per unit frame-time. Hence, an S value from the curve can be chosen and a value of the slot size $\varepsilon = E/S$ as $S = T$ in a steady state can be inferred. Choosing a value of $S = 80\%$ and $E = 1200\mu s$ gives a slot size of $1500\mu s$.

In practice, while the maximum possible value for τ is known, the end-to-end propagation delay of the particular network segment is not, and this influences the slot size via the contention interval modelled in Equation (4.1). The analytic model assumes a Poisson arrival process, whereas for a finite number of stations the arrival process is defined by independent Bernoulli trials (the limiting case of this arrival process with parameters n', p' is Poisson with parameter $\lambda = n'p'$, and approximates the finite case after the number of processors reaches about 20—see Hammond and O'Reilly [55]). More complicated topologies could also be considered where more than one segment is used.

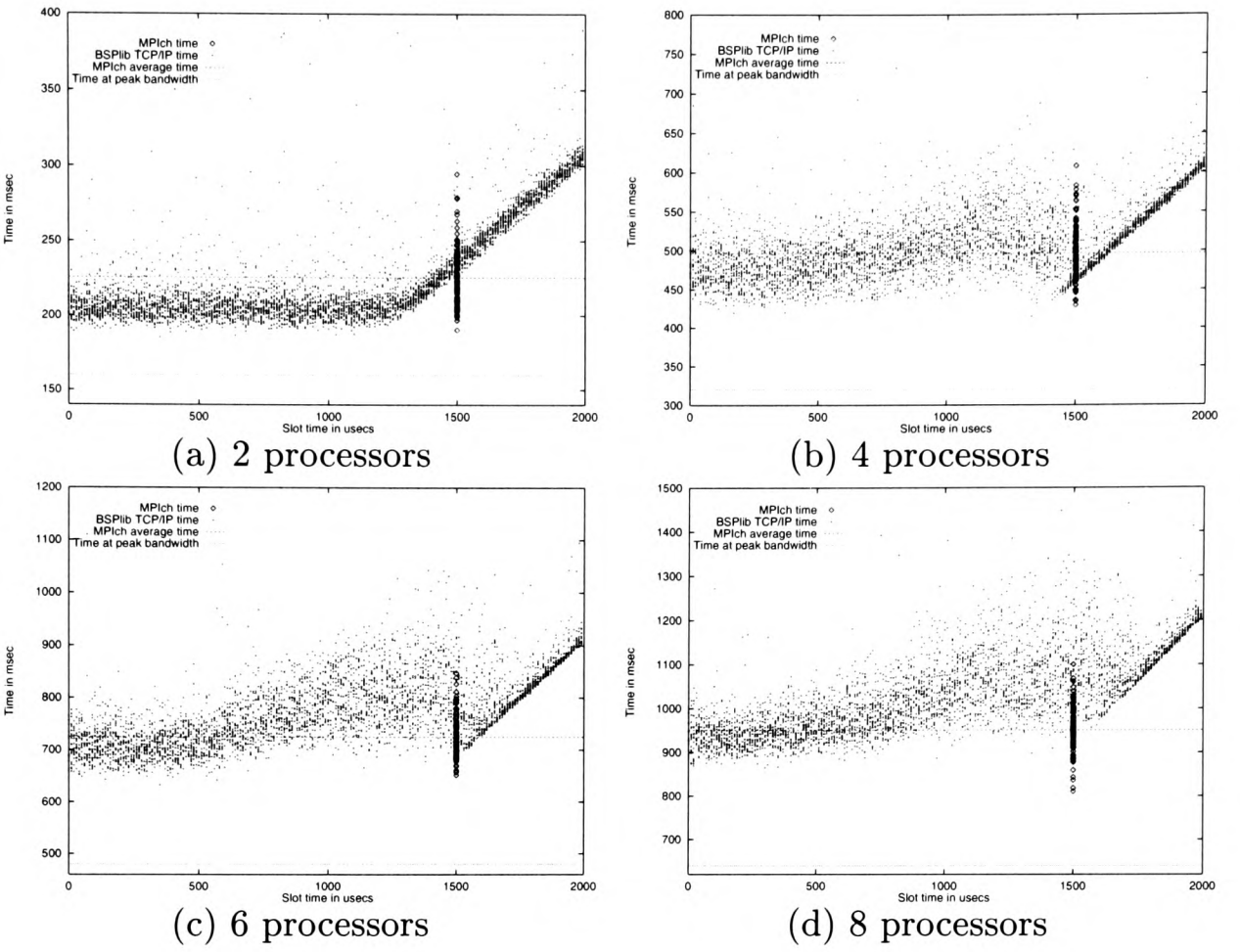
The slot size ε has been determined empirically by running trials in which the slot size is varied and its effect on throughput measured. The experiments involved a 10Mbps Ethernet networked collection of workstations. Each workstation was a 266MHz Pentium Pro processor with 64MB of memory running Solaris 2. The experiments were carried out using the TCP/IP (see Section 4.2.1) and the UDP/IP implementation (see Section 4.2.2) of *BSP-lib*. The machines and network were dedicated to the experiment, although the Ethernet segment was occasionally used for other traffic as it was a subset of a teaching facility.

4.2.1 The value of ε for *BSPlib*/TCP

Figures 4.3(a) to 4.3(d) plot the time it takes to realize a cyclic-shift communication pattern (each processor `bsp_hpputs` a $25,000$ word message into the memory of the processor to its right) for various slot sizes ($\varepsilon \in [0, 2000]$) and for 2, 4, 6 and 8 processors. The figures show the delivery time as a function of slot size, oversampled 20 times (that is, for each slot size the experiment was repeated 20 times—each of the results is plotted to give an idea of the spread for that slot size). The horizontal line towards the bottom of each graph gives the minimum possible delivery time based on bits transmitted[2] divided by physical bandwidth.

Results for an MPI implementation of the same algorithm running on top of the Argonne implementation of MPI (`mpich`) [53] are also shown on these

[2]That is, $25,000 \times 32 \times p$ of *user* payload, which ignores protocol overhead of: (i) message headers; (ii) acknowledgement packets; (iii) retransmissions; and (iv) packets required to implement the barrier in *BSPlib*. Therefore the actual minimum delivery time will be slightly higher than those shown in the figures.



(a) 2 processors (b) 4 processors

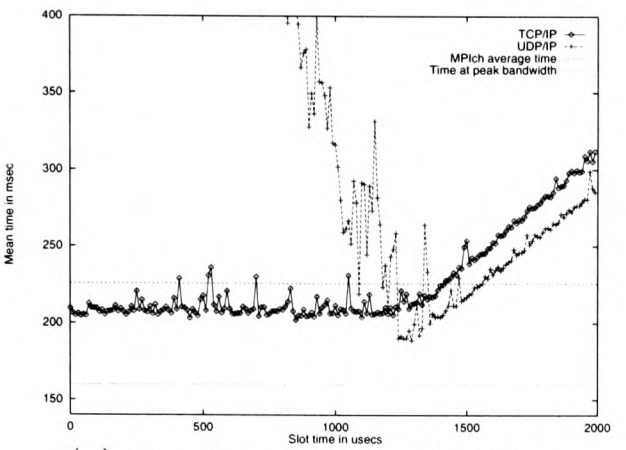
(c) 6 processors (d) 8 processors

Figure 4.3: TCP/IP implementation delivery time as a function of slot time for a cyclic shift of $25,000$ words per processor, $p = 2, 4, 6, 8$, data for `mpich` shown at 1500 although slots are not used

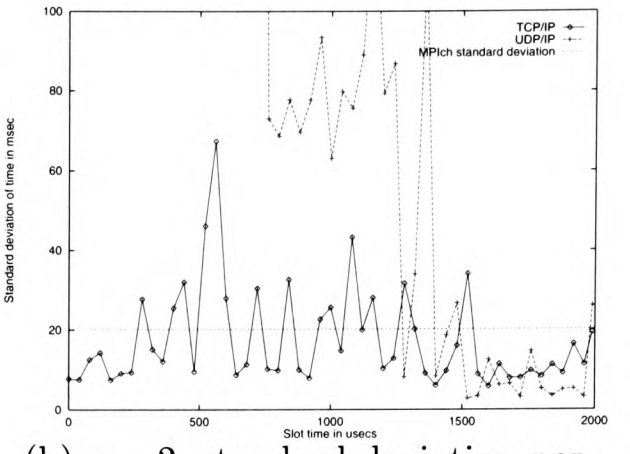
graphs. In this case, the data are presented at a slot size of $1500\mu s$ (even though `mpich` does not slot), so only one (oversampled) measurement is shown. The dotted horizontal line in the centre of these figures is the mean delivery time of the MPI implementation.

The BSP slot time should be chosen to minimise the mean delivery time. Choosing a small slot time gives some good delivery times, but the scatter is large. In practice, a good choice is the smallest slot time for which the scatter is small. For the TCP/IP implementation and $p = 2$ this is $1200\mu s$, for $p = 4$ it is $1450\mu s$, for $p = 6$ it is $1650\mu s$, and for $p = 8$ it is $1700\mu s$. Notice that these points do not necessarily provide the minimum delivery times, but they provide the best combination of small delivery times and small variance in these times.

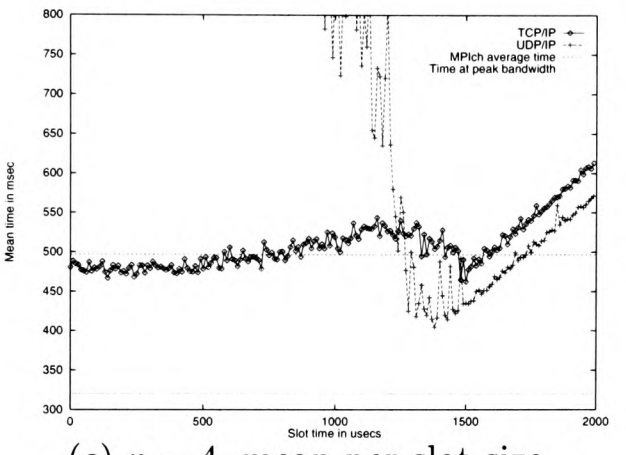
Figure 4.3(b) shows a particularly interesting case at $\varepsilon = 1500$, as both the mean transfer rate and standard deviation of the *BSPlib* benchmark is much smaller than those of the corresponding `mpich` program. This slot-size can be clearly seen in Figure 4.4(c) and Figure 4.4(d) where the scatter caused by the oversampling at each slot size in Figure 4.3(b) has been removed by



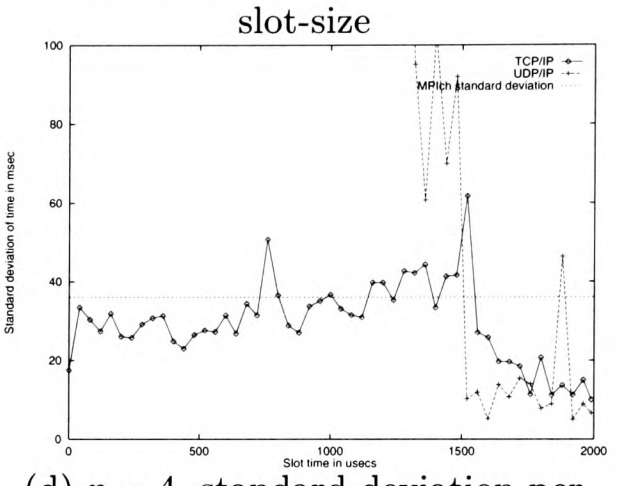
(a) $p = 2$, mean per slot-size



(b) $p = 2$, standard deviation per slot-size



(c) $p = 4$, mean per slot-size



(d) $p = 4$, standard deviation per slot-size

Figure 4.4: Mean and standard deviation of delivery times of data from Figures 4.3(a,b) and 4.6(a,b)

displaying only the mean and standard deviation of the oversampling. In contrast, the mean and largest outlier of the `mpich` program in Figure 4.3(d) is clearly lower than the corresponding *BSPlib* program when a slot size of 1500 is used. For larger configurations, the slot size that gives the best behaviour increases and the mean value of g for *BSPlib* quickly becomes worse than that for `mpich`.

An increase in the best choice of slot size from Figure 4.3(a) to 4.3(d) should be expected as the probability $P(n)$ of n processors choosing a particular slot is binomially distributed. Thus as p increases, so does the expectation $E\{X \geq 2\}$ of the amount of contention for the slot, where

$$\begin{aligned} P(n) &= \binom{p}{n}(1/p)^n(1-1/p)^{p-n} \\ E\{X \geq 2\} &= \sum_{i=2}^{p} iP(i). \end{aligned} \tag{4.2}$$

Figure 4.5 shows that for $p \approx 20$ and greater, the dependence on p is min-

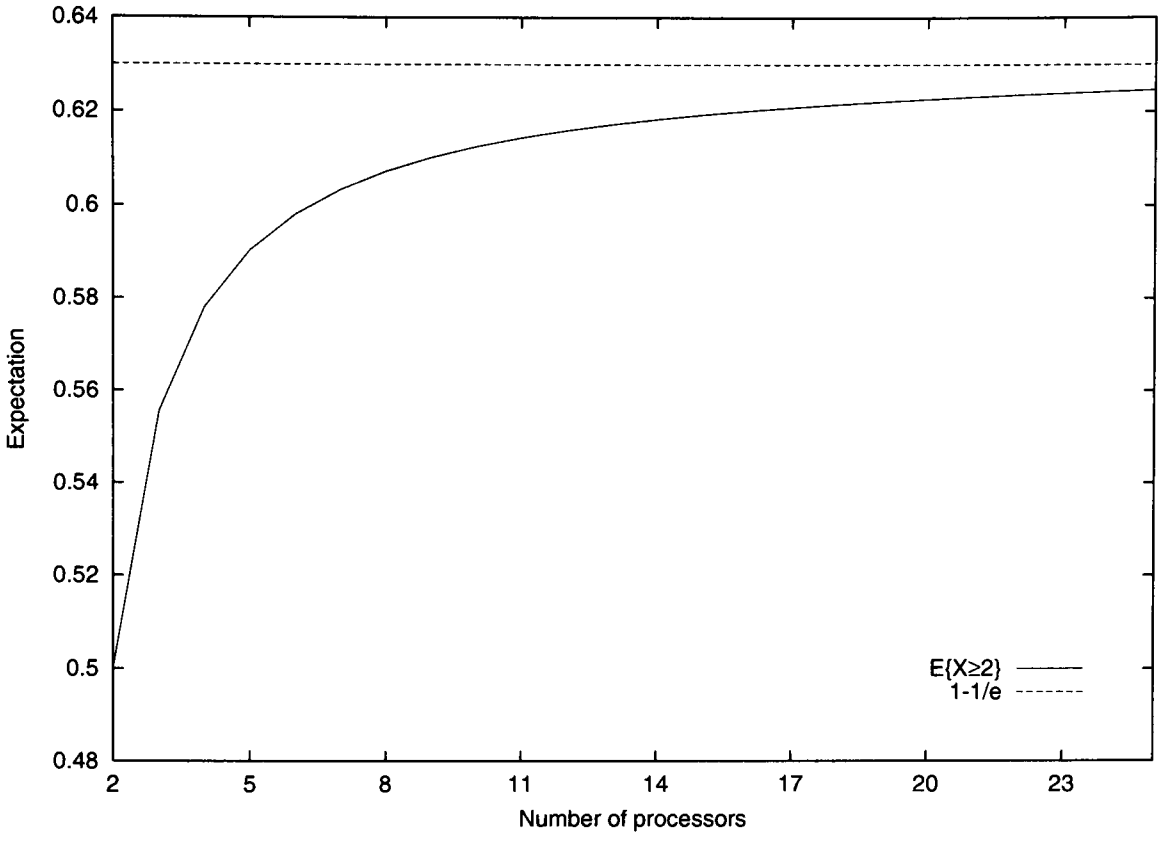


Figure 4.5: Contention expectation for a particular slot as a function of p

imal, and therefore the increase in slot size reaches a fixed point. Below twenty processors the dependence varies by at most 26%. The limit of Equation (4.2) as $p \to \infty$ gives $E\{X \geq 2\} \to 1 - 1/e \approx 0.63$, as shown in the figure. The same is true of the probability of contention, but the range is very small, from 0.25 at $p = 2$, and as $p \to \infty$, $P\{X \geq 2\} \to 1 - 2/e \approx 0.26$.

In the `mpich` implementation [54] of MPI, large communications are presented to the socket layer for communication in a single unit. However, in the *BSPlib* implementation all communications are split into packets containing at most 1418 bytes, so that the submission of packets can be paced using slotting. For this benchmark, each *BSPlib* process sends 71 small packets in contrast to `mpich`'s single large message. Therefore, when p is small one would expect *BSPlib* to perform worse than `mpich` due to the extra passes through the protocol stack, and for larger values of p one would expect that the benefits of slotting out-weigh the extra passes through the protocol stack. Figures 4.3(a) to 4.3(d) show an opposite trend.

As can be seen from Figures 4.3(b)–(d), as p increases, there is a noticeable 'hump' in the TCP/IP experimental data as the slot size increases. This phenomenon is not explained by the discussion above. The problem arises because communication is being modelled as though it were directly accessing the Ethernet, without taking into account the TCP/IP stack. What

is being observed are the TCP acknowledgement packets, which interfere with data traffic as they are not controlled by the slotting mechanism. The effect of this is to increase the optimum slot size to a value that ensures that there is enough extra bandwidth on the medium such that the extra acknowledgement packets do not negatively impact the transmission of data packets.

Implementations of TCP use a delayed acknowledgement scheme where multiple packets can be acknowledged by a single acknowledgement transmission. To minimise delays, a $200ms$ timeout timer is set when TCP receives data [127]. If during this $200ms$ period data are sent in the reverse direction then the pending acknowledgement is piggy-backed onto this data packet, acknowledging all data received since the timer was set. If the timer expires, the bytes received up to that point are acknowledged in a packet without a payload (a 512 bit packet).

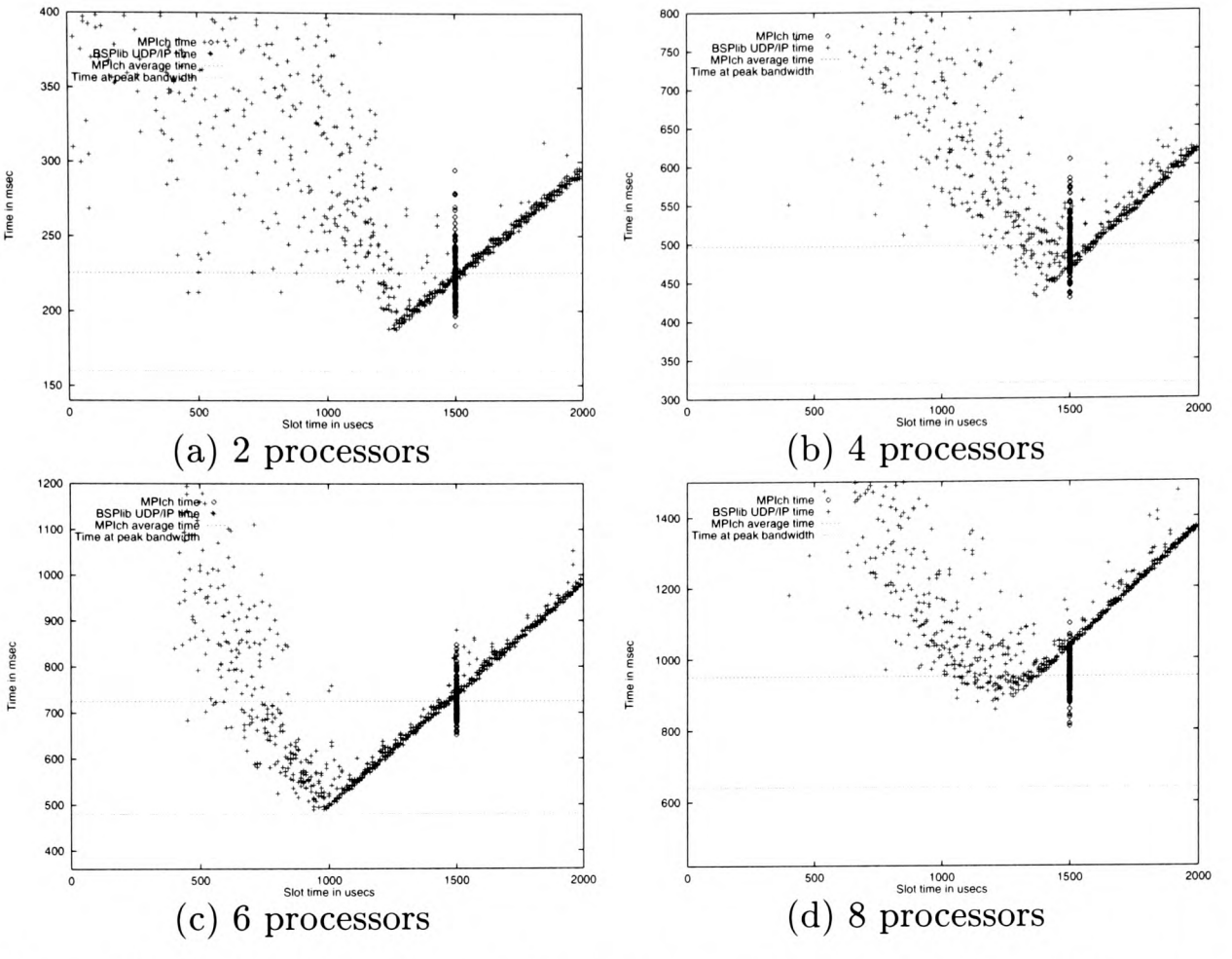
In the benchmark program that determines the optimal slot size, a cyclic shift communication pattern is used. When $p > 2$ there is no reverse traffic during the data exchange upon which to piggy-back acknowledgements. If the entire communication takes less than $200ms$ then only p acknowledgement packets will be generated for each superstep; as the total time exceeds $200ms$, considerably more acknowledgement packets are generated. In Figure 4.3(a) the communication takes approximately $200ms$ and a minimal number of acknowledgements are generated as can be seen by the lack of a hump. In Figures 4.3(b)–(d), the size of the humps increases in line with the increased number of acknowledgements. The `mpich` program does not suffer as severely from this artifact as *BSPlib*. When slotting is not used (for example in `mpich`) there is potential for a rapid injection of packets onto the network by a single processor for a single destination, which means that it is likely that more packets arrive at their destination before the delayed acknowledgement timer expires. This reduces the number of acknowledgement packets. When slotting is used, packets are paced onto the network with a mean inter-packet time between the same source-destination pair of $p\varepsilon$. This drastically decreases the possibility of accumulated delayed acknowledgements. For example, in Figure 4.3(c), as the total time for communication is approximately 800ms, and as the slot size steadily increases, the number of acknowledgements increases. This in turn steadily increases the standard deviation and mean of the communication time. From the figure it can be seen that this suddenly drops off when the slot size becomes large as the probability of collision decreases due to the under-utilisation of the network.

4.2.2 The value of ε for *BSPlib*/UDP

The global nature of BSP communication means that data acknowledgement and error recovery can be provided at the superstep level as opposed to the packet-by-packet basis of TCP/IP. In the UDP/IP implementation, a higher-level light-weight protocol has been implemented which takes responsibility for TCP's functions of ensuring packet delivery, in sequence and without duplication. The major advantage of this light-weight transport protocol is that it allows acknowledgement packets or retransmissions to be slotted, therefore reducing the possibility of conflict between data and control packets. Another advantage of the UDP/IP implementation is that it can be tailored to the physical network. For example, if the LAN guarantees delivery, then the recovery components can be disabled. Alternatively, if delivery is not guaranteed, but message order is preserved, then any gaps can safely be interpreted as dropped packets.

The TCP/IP protocol stack on the other hand cannot take advantage of the nature of the local LAN as the TCP/IP stack is general-purpose and must include support for long-haul traffic where nothing is known about the intermediate networks. Buffer resources are also more efficiently used in the UDP/IP implementation. Each processor uses only one socket to communicate with the $p-1$ other processors, rather than $p-1$ sockets, and buffer space is kept in a single pool that may be used by all sessions. This means that skewed h-relations do not cause unnecessary buffer starvation. By viewing the communication from the superstep point of view, it is possible to forego many of the acknowledgement packets of TCP/IP. For example, the UDP/IP implementation does not send non-piggy-backed acknowledgements if traffic in the reverse direction is not forthcoming within a timeout period. Instead the UDP/IP implementation sends non-piggy-backed acknowledgements based on space usage when the sender needs them to re-use buffer space. Also, the global view of the computation allows implicit acknowledgements of data. For example, if a processor has not received acknowledgements from a particular processor for a while, but received a message from another processor which indicates that the computation is proceeding, then because of the global synchronous nature of the BSP computation, the processor holding the unacknowledged buffers is assured that the messages have safely arrived at their destination and the buffer space can be re-used.

Figures 4.6(a)–(d) shows experimental results for the slotting benchmark described in Section 4.2.1 for the UDP/IP implementation of *BSPlib*. The profile of this data is very different from the TCP/IP results in Figures 4.3(a)–(d). When $\varepsilon > 1200\mu s$ the shape of the curve is approximately the same for both implementations. However, below this point the UDP/IP implementation is a lot less tolerant of small slot sizes and fares much worse than the



(a) 2 processors (b) 4 processors

(c) 6 processors (d) 8 processors

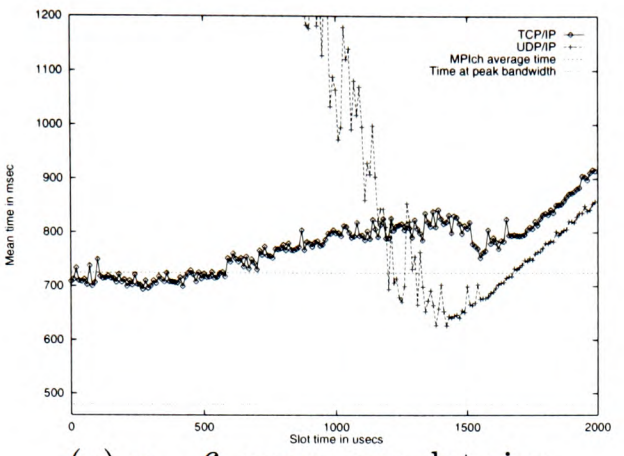
Figure 4.6: UDP/IP implementation delivery time as a function of slot time for a cyclic shift of 25,000 words per processor, $p = 2, 4, 6, 8$, data for `mpich` shown at 1500 although slots are not used

---

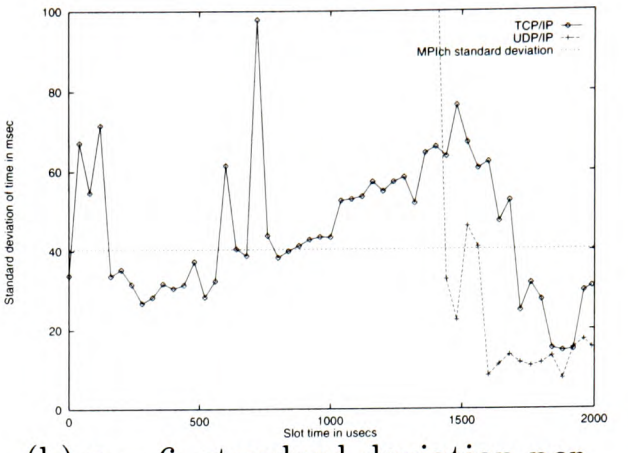
TCP/IP implementation.

Unlike the TCP/IP results, the UDP/IP results reflect the Ethernet analytical model shown in Figure 4.2 in so much as the projections of the inverses of both axes are related to the UDP/IP results. In contrast, the better TCP/IP values at small slot sizes are not explained by the analytic model. The slotting protocol can be viewed as a global flow control mechanism as the rate of message injection is coupled to that of reception and the protocol is run in the steady state where $T = S$ (T, S and G as depicted in Figure 4.1 are aggregated loads over all the participating workstations). However, when slotting is not deployed, the UDP/IP results suffer as there is no flow control protocol and the shallow depth of the protocol stack allows a single process to inject packets at a fast rate which increases the applied load G. In the TCP/IP implementation, the windowing point-to-point flow control of TCP/IP provides an approximation to global flow control when p is small.

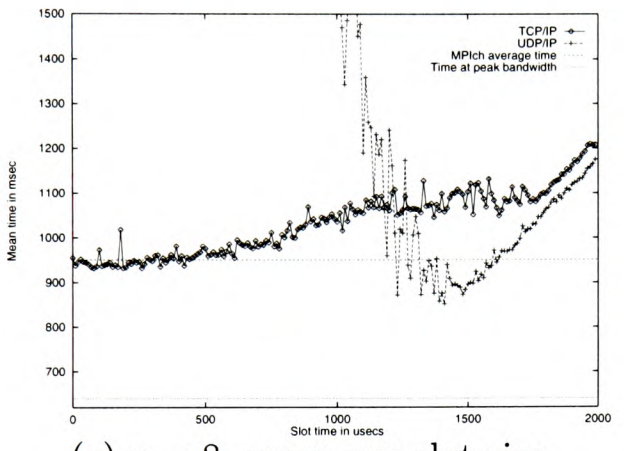
From Figures 4.4 and 4.7, the values of ϵ that maximise throughput are $1250\mu s$ for $p = 2$, $1300\mu s$ for $p = 4$, $1400\mu s$ for $p = 6$ and $1400\mu s$ for $p = 8$.



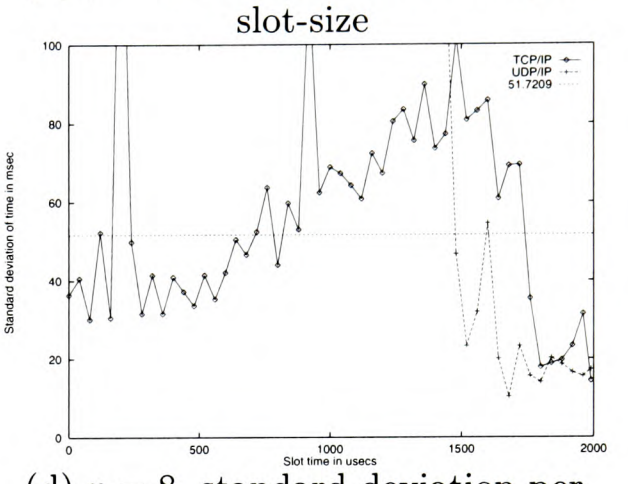
(a) $p = 6$, mean per slot-size



(b) $p = 6$, standard deviation per slot-size



(c) $p = 8$, mean per slot-size



(d) $p = 8$, standard deviation per slot-size

Figure 4.7: Mean and standard deviation of delivery times of data from Figures 4.3(c,d) and 4.6(c,d)

However, a value of $1500\mu s$ minimises variance in all cases. Surprisingly, at slot sizes greater than $1500\mu s$ where the variance is at a minimum in both implementations, there is a phase shift of approximately $80ms$, which is attributed to the smaller amount of protocol data required in the UDP/IP implementation (i.e. smaller headers, fewer acknowledgements, and a simpler barrier mechanism). This value of $\varepsilon = 1500\mu s$ correlates to the value that was chosen in the analytic model (see Section 4.2) at 80% utilisation of the medium. The effective utilisation achieved when modelling only user transmitted data[3] is approximately 72%, in comparison to the 66% utilisation of the `mpich` experiment.

There are some disadvantages of the UDP/IP implementation: If the slotting value is wrong, the performance can degrade quite severely as is evident from Figures 4.6(a)–(d). Also, recovery in the UDP/IP implementation

[3]There are two levels of protocol overhead: (1) At the message level approximately 5% of each message contains protocol information for the various levels; and (2) At the *BSPlib* level extra messages in support of the computation are exchanged.

is across the User-Kernel space interface and performance depends on the signal delivery mechanism and scheduling of the BSP computation processes. This means that other processes can seriously affect the performance of the implementation. TCP/IP on the other hand has the advantage of being able to implement recovery in the kernel.

4.3 Conclusion

The ability of the BSP runtime system to improve the performance of shared-media systems using TCP/IP and UDP/IP has been addressed in this chapter. Using BSP's global perspective on communication allows each processor to pace its transmission to maximise throughput of the system as a whole. A global view of communication also allows protocols and recovery to be based on the super-step structure and not the $p(p-1)/2$ independent point-to-point link structure of LAN communication protocols. A significant improvement over MPI on the same problem has been shown.

In this chapter the 10Mbps shared bus CSMA/CD physical and link layers were used as an example of an unreliable network. The technique, however, is not restricted to solving problems in such configurations and is applicable to any configuration where the throughput can fall after reaching some peak as the applied load increases. Such a reduction in throughput need not be as a result of the use of the media, as in CSMA/CD, but could be as a result of any down-stream congestion or errors. For example, TCP/IP or UDP/IP over switched Fast Ethernet (where the switch does not drop packets) also benefits from this technique by preventing congestion and overruns at the receivers. For this reason the technique has also been implemented in a special-purpose reliable packet protocol designed to support BSP computation (Chapter 5; also Donaldson *et al.* [33]). It should be noted that while CSMA/CD is particularly prone to congestion problems, congestion is not unique to CSMA/CD. The congestion problem also arises with similar profiles to CSMA/CD (Figure 4.2) in packet switching, frame relay and ATM or cell relay networks (see, for example, Stallings [109]).

Only homogeneous collections of processors have been considered (notwithstanding processor speed). The main reason for this is that BSP, or at least its manifestation in *BSPlib*, is untyped. This means that it is not possible to convert from the concrete representations of one host to that of another, and all communicated data structures are viewed as byte streams. Another problem with considering heterogeneous collections of machines is that BSP programs are typically written to balance communication and computation. Thus a well balanced program tends to run at the speed of the slowest component. Even so a scheme for selecting processors in a network based on the

predicted available cycles has been implemented. This prediction is based on some estimate of the load averages on the various machines. In this scheme if the estimate is too far out, the processors can reject the request to host the computation (and the next processor will be attempted), or if the load averages change over time (making another choice of processors significantly better) the processes can be safely moved to other processors (Chapter 8; also Hill *et al.* [60]).

It is clear that the achievable communication bandwidth, and therefore the value of g using these techniques, is not constant, but is normally distributed with a small standard deviation (this issue is explored at length in Hill *et al.* [62]). It is important that the effective value of g be as small as possible, because this directly affects the performance of programs. However, it is also important that the standard deviation of g should be small for two reasons:

- It makes the cost of programs *predictable.* This in turn means that sensible design decisions can be made, for example preferring one algorithm over another, or one target architecture over another [63].
- It increases the scalability of systems. For example, when the effective value of g for a small system has a large standard deviation, then the effective value of g for a larger ensemble built from these small systems has a larger *average* value. This is because the barrier synchronisation requires that all subsystems have finished communicating. When the g value for small systems has large standard deviation, it becomes increasingly likely that some of the subsystems will have 'chosen' a relatively large value of g, which is reflected in the overall g of the larger ensemble (Chapter 7; also Donaldson *et al.* [29]).

Symbol	Description
B	Bandwidth of the bus.
E	Frame transmit time.
ε	Slot size taking into account the frame size and any congestion the frame may encounter.
G	Applied load.
k	Number of times a transmission has backed-off because of a collision detection.
Q	Number of stations participating in the communication at the end of a particular superstep.
q	A random slot number drawn uniformly in $[0 \ldots Q-1]$.
S	Successful load or utilisation of the medium.
T	Applied load at the transport level.
τ	The end-to-end signal propagation delay.

Table 4.1: Summary of symbols used in the discussion of congestion avoidance

Chapter 5

BSP Communication Protocols

BSPlib and MPI are examples of programming models implemented as programming libraries rather than by compilation as is the case with GPL [87] and Opal [83]. While the library implementation style precludes certain higher level optimisations because the library code cannot easily take into account its calling context, it does make implementation and experimentation easier. Such libraries represent a mapping between a set of primitives (whose semantics provide the realization of the particular model of parallel computation) and a transport layer.

There are, however, fundamental differences between libraries such as MPI and *BSPlib*. The thread based nature of MPI with a large number of primitives including two-sided communication, makes it difficult to implement the type of global optimisation described in Chapter 4. Nor is it obvious whether the global protocol techniques described in this chapter can be applied in situ to an MPI setting.

Two aspects of BSP make a partial evaluation of the global state of communication possible, and these can be exploited for performance. Vertically, within a processor, there is a choice of only a few primitives and the realization of communication is decoupled from the point at which the communication is requested. This provides both an opportunity for a local assessment of the local communication requirements requested up to the next barrier as well as some local optimisation of the realization of the communication such as buffer packing, scheduling [67] or destination combining [44]. As

The material in this chapter is based, in part, upon the paper "BSP Clusters: high performance, reliable and very low cost" by Donaldson *et al.* [33]. Some of the material also appeared in "Exploiting global structure for performance on clusters" by Donaldson *et al.* [30].

was demonstrated in Chapter 4, this local summary of the communication requirements can be put to good effect in achieving a global optimisation of the communication interconnect.

The fact that there are only a small number of primitives also means that the interaction of the layers vertically (that is, the application's usage of *BSPlib* and *BSPlib*'s interaction with the transport layer) is restrictive. This means, for example, that a transport layer designed for *BSPlib* need not be general purpose and can take into account its usage by the higher layer.

The second aspect of the BSP model which is fundamentally different from MPI is that the superstep structure means that one can think of a BSP computation as a sequence of horizontal operations filling the entire machine. The alternative is the freedom of writing programs as individual interacting threads using pair-wise communication primitives.

Chapter 4 demonstrated both vertical and horizontal aspects of the BSP computation model. Vertically, communication requests are made and summarised as local state data and, horizontally, these are composed into some global state for the optimal utilisation of the communication interconnect.

The exploitation of the global nature of BSP computation is extended in this chapter and is applied to the design of a transport layer and of a communication protocol suitable for BSP computation. Global knowledge, both implicit and explicit, can play a role in designing a transport layer protocol and an interface to it. Chapter 6 demonstrates with a series of benchmarks precisely where performance gains arise, and how large they are.

Three implementations of messaging layers for *BSPlib* are described. The first is based on TCP/IP and the second two are based on the new transport protocol designed specifically for *BSPlib*. The first of these implementations is based on UDP/IP and the second directly on top of a Network Interface Card or NIC. (The importance of minimising variance in communication protocols for parallel computation and how it impedes compositionality by placing limits on global communication performance is covered in Chapter 7. The two implementations of the protocol are assessed in terms of the analysis presented there.)

Clusters (that is networks of processors using lightweight protocols that interface directly with NICs instead of conventional protocol stacks and buffers) are becoming more commonly used as extremely cost-effective alternatives to manufactured parallel machines. Performance results are presented in Chapter 6 and show just how cost-effective clusters can be.

The new transport layer is defined and provides a reliable packet interface, built upon a lower-level unreliable datagram protocol with similar semantics to UDP/IP. This layer has to provide services to ensure the reliability

required by higher layers, while the lower level interfaces directly with the network interface card. The novel features of the protocol are:

- a demonstration that user-space protocol processing is not necessary to achieve low-latency (i.e. tens of microseconds) data transfer;
- a new error-recovery algorithm that is receiver- rather than sender-driven, and works by hole-filling, rather than go-back-n;
- a fine-grained acknowledgement reduction technique based on explicit information about communication patterns;
- a further reduction in acknowledgements based on implicit knowledge of superstep structure;
- management of buffers on a per-processor basis, rather than a per-processor-pair basis.

For a 400MHz processor, the CPU overhead in the protocol of scheduling a packet for transmission is $1.5\mu s$. Packet upload takes $3.6\mu s$ of CPU time per packet, excluding the time required to copy the packet into user data structures. These times mean that it is possible to continually send small packets through a 100Mbps fast Ethernet switch, to a destination processor every $21\mu s$ in a reliable fashion (less than $10\mu s$ unreliably). A sustained point-to-point bandwidth of 93Mbps is achieved for full frame packets (1400 bytes), which scales to (at least) eight processors communicating at 91Mbps per link, to give a sustained global bandwidth of 728Mbps for large packets over switched full-duplex fast Ethernet.

Section 5.1 describes the messaging layer of the *BSPlib* implementation and three designs for it, specialised for different transport layers. Section 5.2 discusses how the messaging layer handles reliability when the transport layer is datagram based. Certain desirable properties of the abstract protocol have been demonstrated by Simpson *et al.* [107]. Section 5.3 discusses these results.

5.1 Network messaging layers for *BSPlib*

Figure 5.1, shows a high-level schematic of the structure of the implementation of *BSPlib* and `mpich`. It shows how a certain degree of isolation from the transport layer can be achieved by including a mapping layer termed the *BSP messaging layer*. This layer is responsible for providing an interface on which the upper layer *BSPlib* primitives can be implemented. In many ways, the BSP messaging layer is the same as the Abstract Device Interface [52] used in the `mpich` implementation of MPI.

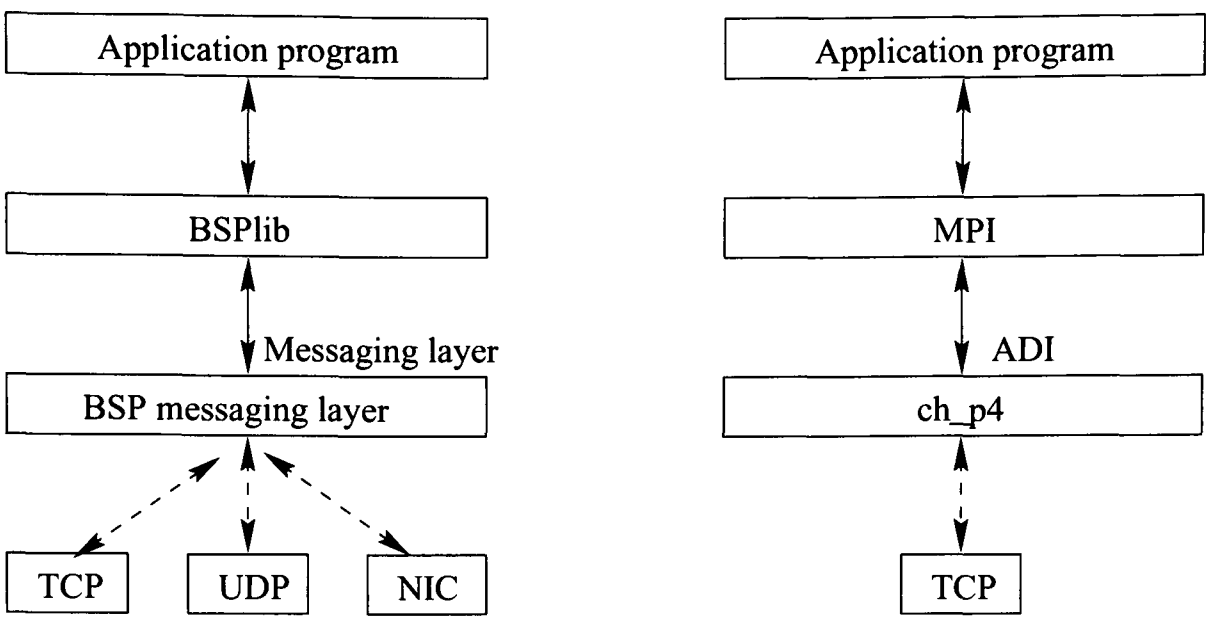


Figure 5.1: Comparing the *BSPlib* and `mpich` protocol hierarchy

The bulk of the work in porting *BSPlib* amounts to rewriting the messaging layer, and providing a binding to the new transport layer (there is also a similar structure for shared memory implementations of *BSPlib*; Hill and Skillicorn [68]). The important primitives supplied by this layer are: hand-shaking and initialisation of the computation on the various processors, mechanisms for orderly and unorderly shutdown, a non-blocking send primitive, a blocking receive primitive and a probe to test for message presence. There are also calls to manage re-use of communication buffers. It is important to realize that this is not a general interface, and in particular, the layer takes into account how it is used by the upper layers of the *BSPlib* implementation. This restriction in the interaction of the two layers allows *BSPlib* to guarantee that the computation is both deadlock and livelock free as has been demonstrated by Simpson *et al.* [107] and discussed in Section 5.3. Indeed, this can be seen as a direct result of uncoupling the user one-sided communication requests and their implementation by *BSPlib* on the available transport layer.

Currently, the message-passing layer has been implemented on top of a number of proprietary message-passing libraries, as well as generic tools such as MPI, and on TCP/IP and UDP/IP using BSD Sockets. In most of these implementations, the messaging layer is nothing more than a collection of macros that provide the necessary interface required by *BSPlib*. However, when the transport layer is lightweight, the BSP messaging layer becomes significantly more complex. For example, if UDP/IP is used as the transport layer, then the BSP messaging layer is a significant piece of software that handles error recovery and packet acknowledgements.

The BSP messaging layer uses two techniques to improve performance, both enabled by the semantics of the BSP model. Because all of the communication that is initiated during the computation phase of a superstep is not actually begun until the computation has ended, it is possible to pack all small communications between each processor pair into larger messages. This means that the startup penalty that must typically be paid for each message transmitted, is paid once per combined message (which depends upon the frame size of the transport medium) for each pair of processors rather than once per user message. On many architectures, the performance gain that results dramatically outweighs the (at most) factor of two that can be achieved if computation and communication are fully overlapped [67].

The second technique also depends on postponing communication until after computation. Congestion effects are much more likely to occur at the boundaries of the interconnection network than in its interior, since each processor typically has only a single connection to the network. Consider a total exchange communication pattern, where each processor has data for every other processor. If each processor tries to send a message to Processor 0 first, then its message to Processor 1, and so on, then Processor 0 will be bombarded with many messages in a short time, will be unable to extract more than the first one, and will therefore create congestion back through the network in its vicinity. A similar thing will happen slightly later to Processor 1, and so on. This behaviour is more or less independent of network topology and routing techniques. The net effect is that the time taken for the total exchange is likely to be $\mathcal{O}(p^2)$, rather than the $\mathcal{O}(p)$ expected from the BSP cost model. However, if processors all choose to send their messages in a different order, there will be approximately one message arriving at each destination at all times. This can be achieved in a variety of ways, e.g., randomising destinations, or by using a Latin square as a transmission schedule, but depends critically on the global knowledge of superstep communication [67]. The BSP messaging layer implements the semantics of supersteps in the following way:

1. Calls to **bsp_sync** result in all outgoing messages being packed as described above.
2. Each processor creates a bit-matrix that specifies the processors with which it will be communicating. A reduction of these matrices with bitwise-or produces a matrix that contains all of the information about who will communicate with whom; and each processor can compute this result with the same time complexity using a variant of parallel prefix or scan. (Note that this reduction *is* the barrier of the superstep).
3. A total exchange takes place between those processors involved in the superstep, so that each processor can inform the others of the number

and size of messages it plans to send to them.

4. Each processor sends its messages using an appropriate schedule as described above. The exact details of the rate and order used depends on the transport layer.
5. When each processor has received all the messages it expects, it continues to the computation phase of the next superstep.

In the following sections, the three implementations of the messaging layer used by *BSPlib* are presented: (1) the *BSPlib*/TCP implementation that uses BSD stream sockets as an interface to TCP/IP; (2) the *BSPlib*/UDP implementation that uses BSD datagram sockets as an interface to UDP/IP; and (3) the *BSPlib*/NIC implementation that uses the same error recovery and acknowledgement protocol used in the *BSPlib*/UDP implementation, but replaces the BSD datagram interface with a lightweight packet transmission mechanism that interfaces directly with the network interface card. In all the implementations, any *BSPlib* communications posted during a superstep are delayed until the end of the superstep. The actual physical communication in *BSPlib* therefore reduces to the problem of routing a collection of packets between all the processors within the `bsp_sync` procedure that marks the barrier at the end of the superstep.

5.1.1 *BSPlib*/TCP: a messaging layer built upon TCP/IP

The BSD stream socket interface provides a reliable, full-duplex, connection-oriented byte stream between two end-points using the TCP/IP protocol. As well as providing error recovery to deliver this reliability, TCP/IP uses a sliding window protocol for flow control which inhibits a fast sender from swamping a slow receiver with large amounts of data.

General and reliable transport protocols such as TCP/IP provide a portable implementation path for BSP. However, the functionality of TCP/IP is too rich for BSP-style computation using a dedicated set of processors. For example, the TCP/IP protocol stack cannot take advantage of the nature of the local LAN as the stack also includes support for long-haul traffic, where nothing may be known about the intermediate networks. This makes TCP/IP unsuitable for high-performance computation (Chapter 4; also Donaldson *et al.* [32]).

The *BSPlib*/TCP implementation of *BSPlib* uses the `send` function to push data into the TCP/IP protocol stack. At the level of the messaging layer, reception of messages is *not* asynchronous. When the higher level requires a packet, the messaging layer waits for a packet to arrive at the process by issuing a `select`. When the `select` completes, the received packet is copied into user space by issuing a `recv`. Of course, message reception is still

asynchronous with respect to the lower levels of the TCP/IP protocol stack. However, messages will be buffered within the stack until they have been selected by the messaging layer. Having the actual reception at a lower level makes it particularly difficult for the messaging layer to make any sensible decisions about buffering. For example, if packets are dropped due to insufficient buffer resources, the upper layer is unaware of it, and cannot plan to circumvent the problem by, for example, using global knowledge of the communication pattern. The only way of avoiding an excessive number of packets being dropped by the TCP/IP layer is to associate large buffers with each of the $p-1$ sockets that form the end-points on each process ($p(p-1)$ in total). This may be an extremely wasteful use of memory if an application uses only skewed communication patterns.

Another problem with TCP/IP is that the way it handles recovery and flow control based on timeouts, means that it *is not suitable for scalable* communications libraries. The problem was introduced in Chapter 4 and can best be understood in terms of a density of circuits in use argument. Given p processors, there will need to be $p(p-1)/2$ circuits to implement point-to-point messaging between all pairs of processors. Of these $\mathcal{O}(p^2)$ circuits only $\mathcal{O}(p)$ can be in use at any one time. Therefore, in the limit, as the majority of circuits ($p(p-1)/2 - p = \mathcal{O}(p^2)$) are idle, and only p circuits have traffic flowing over them, the majority of circuits have no reverse traffic on which acknowledgements can be piggy-backed. If the circuits become idle for too long, which will be the case if the communication is long-running over the active circuits, then a timeout associated with the receivers will expire, and an explicit acknowledgement will be sent. This phenomenon accounts for the poor performance seen when benchmarking the TCP/IP implementation of *BSPlib* (Chapter 4; Donaldson *et al.* [32]). On shared 10Mbps Ethernet, it would be expected that, as the number of processors increases, the available bandwidth from any single processor would decrease as $10/p$ Mbps. Because of an excessive number of non-piggy backed acknowledgements generated by TCP/IP, the observed bandwidth is significantly worse than this, even for a modest number of processors ($p = 8$).

5.1.2 *BSPlib*/UDP: a messaging layer built upon UDP/IP

The BSD datagram socket interface provides an unreliable, full-duplex, connectionless packet delivery mechanism between machines using the UDP/IP protocol. Unlike TCP streams, UDP datagrams have a fixed maximum transmission unit size of 1500 bytes for Ethernet frames. The link between the machines is unreliable as UDP/IP provides no form of message acknowledgement, error recovery, or flow control.

Protocols such as UDP/IP have a shallower protocol stack and therefore

have more potential for high-performance. However, they do not provide reliable delivery of messages. If user APIs such as *BSPlib* are implemented over UDP/IP, they must handle the explicit acknowledgement of data and error recovery themselves[1]. In the protected kernel/user-space model of UNIX, this must be done in user space.

To establish a reliable send primitive, the *BSPlib*/UDP implementation of *BSPlib* copies a user data structure referenced in a send call into a buffer that is taken from a free queue. If there are no free buffers available, then the send fails, and it is up to the higher level *BSPlib* layer to recover from this situation[2]. The buffer is placed on a send queue and a communication is attempted using the `sendto` datagram primitive. Queueing the buffer on the send queue ensures that it does not matter if the packet communicated by `sendto` fails to reach its destination, as the error-recovery protocol can resend it if necessary. Only when a send has been positively acknowledged (either explicitly or implicitly, see Section 5.2.1) by the partner process, are buffers reclaimed from the send queue to the free queue.

Message delivery is asynchronous and accomplished by using a signal handler that is dispatched whenever messages arrive at a processor (SIGIO signals). Within the signal handler, messages are copied into user space by using `recvfrom`. To perform the error recovery as soon as possible, it executes from within the signal handler (see Section 5.2).

The *BSPlib*/UDP implementation of *BSPlib* achieves better bandwidth than *BSPlib*/TCP because: (1) packets destined for a processor can be acknowledged without explicit communication from that processor by utilising the superstep structure of a program; (2) error recovery can be (receiver) resource based rather than (sender) timeout based. In contrast, in a TCP/IP client-server application, the server cannot make a distinction between missing client requests (an error situation) and no client requests (not an error situation).

The protocol runs on a dedicated set of machines with a very specific workload. The receiver is primarily responsible for recovery and is always aware of the amount of outstanding communication. Before the first user-data communication of a superstep, information concerning the number of packets destined for each processor is distributed among all processors using a total exchange. This communication acts as the barrier synchronisation of the superstep. Thereafter, the upper layers are aware of the circuits on which outstanding data can be expected. A processor will remain involved

[1] Implementations that do not do this are interesting only as research vehicles for showing minimal attainable latencies.

[2] *BSPlib* recovers by either: (1) receiving any packets that have been queued for *BSPlib*; or (2) asking those links that have a large number of unacknowledged send buffers to return an acknowledgement.

in the communication phase of a superstep while there are outgoing packets to be communicated, and until all incoming packets have arrived. Thereafter, the processor can drop through into the computation phase of the next superstep before all acknowledgements of data sent during the present superstep.

The *BSPlib*/UDP implementation achieves better bandwidth because explicit packets required for flow control are less frequent than for *BSPlib*/-TCP, and error recovery is based at the receiver where complete knowledge of expected packets is available. There are some drawbacks to using UDP/IP however. One is that in order for a message to be received asynchronously, a signal has to be dispatched to the user process and the user process has to be scheduled. To receive the message, the user process then makes a `recvfrom` system call. All this is on top of the height of the UDP/IP protocol stack. Since a larger portion of the protocol executes in user space, more scheduling events (i.e. ready-queue processing and context switches) may be required to process the same workload than when *BSPlib*/TCP is used. This is particularly relevant when applications are compute-bound and can be clearly seen in the NAS benchmark results presented in Chapter 6. Compared with kernel-level protocols, the height of the protocol stack and the execution of the protocol in user space means that triggering recovery is sluggish. The results show that although user-space communication is a good way of minimising communication latency, it should not be considered in isolation; and in the *BSPlib*/UDP case communication latency is being reduced at the expense of the effective computation rate.

5.1.3 *BSPlib*/NIC: a NIC based transport layer for *BSPlib*

A Network Interface Card (NIC) provides the hardware interface between a system bus and the physical network medium. NICs range from general purpose, to those tied to a specific link-layer protocol such as Ethernet, ATM and FDDI. The general-purpose variety (e.g., Myrinet), are controlled by dedicated microprocessors that can run user-supplied programs concurrently with the host microprocessor. Typically these NICs are produced in low volume, and used predominantly in research. In contrast, high-volume fixed-protocol NICs have a fixed firmware program, whose parameters can be altered via a memory-mapped register file. Due to the low cost and installed user base (millions of units) of such NICs, these devices have been targeted for the prototype.

The *BSPlib*/NIC implementation of the *BSPlib* messaging layer uses exactly the same error-recovery and acknowledgement protocol as the *BSPlib*/UDP implementation. The only difference between the implementations is that *BSPlib*/NIC is built upon a lightweight packet transmission mechanism that

interfaces directly to the NIC. This is achieved by having a portion of memory used for communication buffers and shared data structures mapped into both the kernel and the user's address space. The region is created with the `memmap` system call. The backing pages for the virtual region are supplied from contiguous reserved system memory. This allows both the kernel virtual storage (same as the physical address space in the Linux 2.0 kernel) and the user virtual storage to be contiguous, with the mapping between the two being accomplished simply by adding or subtracting an offset.

With the sophistication of modern network interface cards, UDP-like protocols can be implemented in such a way as to achieve very high bandwidth utilisation of the network. In the *BSPlib*/NIC implementation, the problems of error recovery mentioned above are alleviated by pushing most of the protocol processing dealing with error recovery back into the kernel. All outgoing packets contain a protocol header which contains piggy-backed acknowledgement and error recovery data. As the *BSPlib* layer can queue a potentially large number of packets for transmission by the NIC, the protocol information may become stale as incoming packets update the protocol state. This problem can be alleviated by allowing the NIC to perform a gathered send, whereby the protocol information is only read when the packet is about to be transmitted. Processor usage is also improved by using the features of the NIC to make the communication as asynchronous as possible. For example, the standard technique in device drivers is to use bounded send and receive descriptor rings which the NIC traverses. In the *BSPlib*/NIC implementation, the NIC traverses an arbitrary sized (linked-list) queue of buffers (the NIC's transmit queue).

Although moving the communication handling back into the kernel appears to be against the trend of user-space communication protocols, it is possible to have the benefits of kernel based reception and error recovery outlined above, as well as low-latency message sending that is normally associated with user-space communication. This is achieved without a system call in the implementation by having the *BSPlib* layer enqueue an element onto the NIC's transmit queue (the packet will also be enqueued onto a send queue for recovery purposes). This is serviced asynchronously by the NIC, which will poll for work if none is available.

The resulting implementation has additional desirable properties for the support of BSP computations. For example, many algorithms need a certain amount of parallel slackness. If the synchronous cost of sending a message (the LogP parameter o; Culler *et al.* [27]) is too large, then no amount of parallel slackness can be expected to improve the situation. In the implementation described here, the CPU cost of sending messages is very small ($\approx 1.5\mu s$).

5.2 A reliable packet protocol for *BSPlib*/UDP and *BSPlib*/NIC

Two types of packets are used in the protocol: (1) Sequenced data packets that contain a fixed-sized header with MAC, sequence number, payload length and type, and piggy-backed recovery information. This is followed by a variable-length payload, whose length is bounded so that the entire packet is no larger than the physical layer's frame size; and (2) Unsequenced control packets that contain the same fixed-sized header, but no data. Control packets are used for explicit acknowledgements and *prods*, as described below. Each process contains a free queue of packets (in the *BSPlib*/NIC implementation these are mapped into kernel and user space), and send and receive queues associated with each communication channel ($p-1$ in total). The send queue contains unacknowledged sent data-packets, and the receive queue contains data-packets not yet consumed by the upper *BSPlib* layer.

A messaging or transport protocol is described which aims at minimising the number of redundant retransmits of data packets, and the number of control packets required to achieve this. Consider two processors, A and B, communicating over a channel that loosely guarantees packet ordering[3], but that may drop packets. Suppose process A sends the sequence of packets $[0, 1, \ldots, 9]$ (which will be queued on A's send queue associated with B until acknowledged) to B, and only the sequence $[0, 1, 4, 5, 6, 9]$ arrives at B (and are queued onto a receive queue associated with A). If a go-back-n protocol (as used by TCP/IP) is used to recover from the lost packets, then processor A, after learning that the highest in-sequence packet received has sequence number 1, retransmits the packet sequence $[2, 3, 4, 5, 6, 7, 8, 9]$, which contains four redundant packets. A more sophisticated selective retransmit scheme would resend only the sequence $[2, 3, 7, 8]$. However, in both cases, it is the responsibility of the recovery protocol to ensure that the sending processor A learns of these missing packets. This can either be done by piggy-backing recovery information on packets travelling in the reverse direction, or by sending an explicit control packet that informs A of the problem if there is no reverse traffic.

In the selective retransmit protocol, the fixed header contains two fields for piggy-backed sequence number information: (1) An *acknowledgement* field such that if B sent a message back to A after having received $[0, 1, 4, 5, 6, 9]$, this field would contain 1 (this allows A to remove all packets up to this value from the send queue); and (2) An *end-of-hole* field which specifies the sequence number of the packet after the last in-sequence packet, i.e. 4. When no unaccounted-for packet loss has occurred, the acknowledged

[3]The test platform guarantees ordering. However, the software can handle out-of-order arrivals within a configurable tolerance.

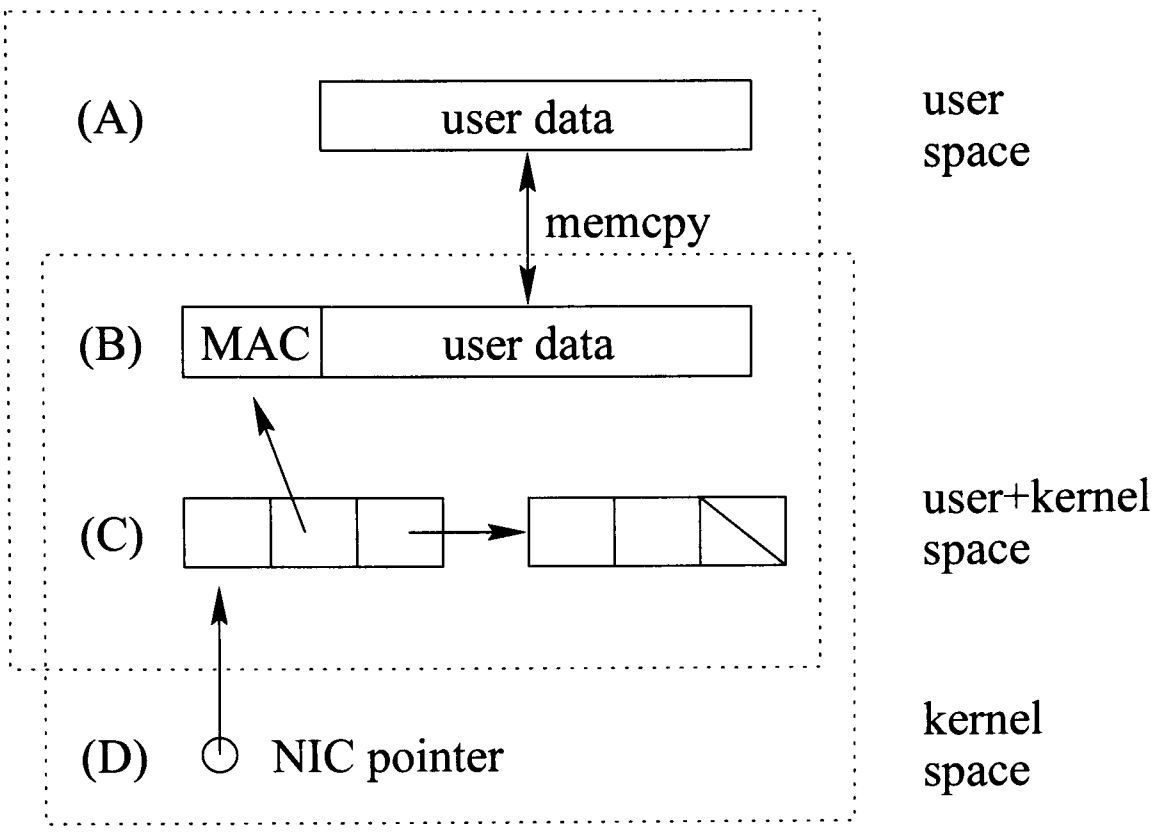


Figure 5.2: Kernel-User space views and NIC download data structures

sequence number and this field are the same. If A and B are involved in symmetric communication, then A will learn of the first hole from this piggy-backed data and resend the packets 2 and 3. After these have been received by B, further piggy-backed information will trigger the resending of 7 and 8. To prevent potentially stale in-flight information from being re-used to cause repeated resends, a double retransmission of data is delayed for at least a round-trip delay of the network.

If there is no reverse traffic from B to A, then in the situation where the upper *BSPlib* layer has consumed packets $[0, 1]$ and now requires packet 2, B will *prod* process A with a control packet. This allows A to learn of the hole and resend packets 2 and 3. In case the control packet goes missing, this prodding is repeated no more frequently than the round trip interval of the network. This prodding can also arise without packet loss when the computation on a processor runs ahead of another and hence enters its communication phase earlier. In this case the prodded processor responds with an acknowledgement rather than missing data. The prodding processor then realizes that it has run ahead and is being impatient, and enters an exponential back-off of prods.

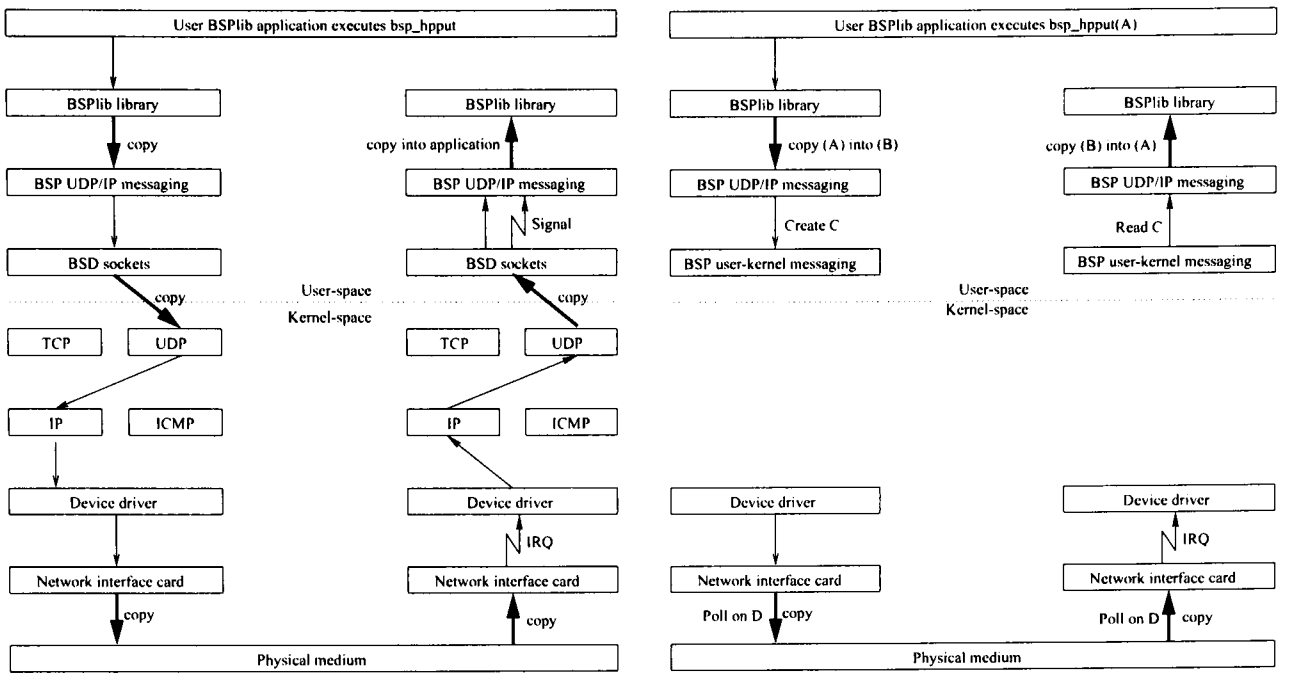
There can be a problem with the protocol described above if the tail of a sequence of packets has been dropped. For example, consider if B received only the packets $[0, 1]$ and packets $[2, 3, \ldots, 9]$ are dropped. Then, after the

upper layer consumes the first two packets, it will send a prodding control packet to A with an acknowledgement and end-of-hole number of 1. When A receives this control packet, packets $[0, 1]$ will be acknowledged and removed from the send queue, but as there are more packets on the send queue, A will assume they must have been dropped. Instead of just resending them all (which amounts to a go-back-n protocol), A resends the first and last packets from the send queue (i.e. 3 and 9). If there was no real hole then only two redundant messages are sent. If there really is a hole, the receiver (eventually) gets an out-of-sequence message and can report the extent of the hole the next time the sender is prodded.

The benefits of this scheme over a more-general selective retransmission scheme that could, for example, enumerate all the sequence numbers of packets dropped, is that only one integer field is required in the packet header. In the worst case, as many prodding control packets are generated as holes in the data; in the best none. The scheme relies upon the observed fact that dropped packets will be clustered into holes. The network used, and the scheme employed, ensures that packet loss is quite rare (0.05% packet drops are experienced in the NAS benchmarks in Section 6.1.4) and any packets lost are actually dropped by the receivers when, for example, available buffers run low. When buffers run low, only the next expected packets are accepted, but the acknowledgement data on all arriving packets is still inspected so that buffers held in the send queues can be freed, and progress is guaranteed [107].

Figure 5.2 shows the overlap of memory between user- and kernel-space, and the download data structures processed by the NIC. Figure 5.3 shows schematics of, respectively, *BSPlib*/UDP and *BSPlib*/NIC. The labels associated with the arrows in Figure 5.3(b) relate to data structures shown in Figure 5.2, and determine the point at which data are copied between user memory and memory which is shared between the user and kernel. Because the receive data structures are prepared by the kernel in the *BSPlib*/NIC implementation, some measure of system integrity can still be maintained. The gap between the lower layers of Figure 5.3(b) and the upper user-space layers not only represents a shallow protocol stack, but the missing flow arrows represents the asynchronous behaviour of the protocol with respect to the application process and the communication process.

There are standard light protocols which employ a selective retransmission scheme. Once such protocol is the *Service Specific Connection Oriented Protocol* (or SSCOP) (see, for example, Tanenbaum [117] or Brebner [12]). This protocol operates over ATM AAL5 in one of two modes, an unreliable mode and a reliable delivery mode. In this reliable delivery mode, some attributes of SSCOP are similar to the *BSPlib* messaging protocol when considered at a point-to-point level. Messages in SSCOP can be up to 64k bytes, contain a



(a) *BSPlib*/UDP implementation (b) *BSPlib*/NIC implementation

Figure 5.3: Comparison between the *BSPlib*/UDP and *BSPlib*/NIC implementations of the protocol

24-bit sequence number and are controlled using a dynamic sliding window flow control protocol. In this sliding window protocol a receiver maintains a list of message sequence numbers that it is prepared to receive and a bit mask of the messages it has already received. Because out of order arrivals are not possible in ATM, any missing data are permanently lost. In general, re-ordering is possible in other networks and this assumption could not be made in the *BSPlib* messaging protocol. Hence, the scheme required to discover the end of a sequence dropped of messages when this is not indicated by the acknowledgement data. Unlike the *BSPlib* messaging protocol, acknowledgements are not piggy-backed in SSCOP and recovery is initiated by the sender. After a certain amount of time has elapsed or a certain number of messages have been sent, the sender polls the receiver to discover which messages have been received and to update flow control information. The sender responds to these status or unsolicited status messages (which indicates that a previous poll message went missing) by sending the missing messages.

5.2.1 Non piggy-backed acknowledgements

Two techniques are used, one fine-grained, the other coarse-grained, for the acknowledgement of messages in the situation where there is no reverse traffic on which to piggy-back them. The overall strategy is to try as hard as possible to avoid sending acknowledgements. This involves waiting much

longer than a conventional implementation would before sending explicit acknowledgements.

The coarse-grained acknowledgements utilise the superstep structure of *BSPlib* programs, and acknowledge data implicitly. When a receiver notices that its local processor has ended a computation phase, and has begun the next communication phase (recall that this involves a reduction of those processors involved in the superstep), it can immediately treat all packets on all send queues, which must have been left over from the previous communication phase, as acknowledged.

The fine-grained scheme is based upon volume of communication, and not timeouts as in TCP/IP, and has both a sender and receiver component to the algorithm. From the sender's perspective, if the number of buffers on a free queue of packets becomes low, then outgoing packets are marked as requiring acknowledgements. This mechanism is triggered by calculating how many packets can be consumed (both incoming and outgoing) in the time taken for a message round-trip. The sender is anticipating when buffer resources will run out, and attempting to force an acknowledgement to be returned just before buffer starvation. This strategy minimises both the number of acknowledgements, and the time a processor stalls trying to send messages. From the receiver's viewpoint, if n is the total number of send and receive buffers, then a receiver will send a non-piggy-backed acknowledgement only if it is requested, and at least $\frac{n}{2p}$ elements have come in over a link since the last acknowledgement (either piggy-backed or explicit).

The fine-grained scheme can be refined further by using information from the upper *BSPlib* level. At the end of each communication phase (superstep), the protocol also marks the last data item sent to a process as requiring an explicit acknowledgement. This improves the usefulness of acknowledgements. Such last transmissions are likely to result in acknowledgements anyway, and explicitly requesting them gets them sent earlier and at a time when traffic on the interconnect is quiescing. Also, this increases the number of free buffers at the start of the next communication phase and makes non-piggy-backed acknowledgements less likely during that communication phase.

Because the coarse-grained scheme relieves the situations that trigger the fine-grained scheme's acknowledgements, and because the coarse-grained scheme's acknowledgements are implicit, the coarse-grained scheme limits the number of explicit acknowledgement packets when the number of free-queue packets is sufficiently large.

5.2.2 Asynchronous sending in the *BSPlib*/NIC implementation

As discussed earlier, delaying communication to the end of the computation phase of each superstep makes it possible to globally schedule the communication to avoid conflicts at NICs (e.g., using Latin squares to choose partners in communication; Hill and Skillicorn [67]). Also, the bandwidth utilisation and corresponding communication performance achieved by combining messages far outweighs the cost of not sending messages immediately. Delaying communication has another benefit: since the NIC used (a 3COM 3C905B-TX) can be configured to automatically poll for work when the send queue dries up, the poll rate can be set to slow or turned off during the computation phase of the superstep and can be changed to a much more aggressive polling rate during the communication phase. Each of these changes would cost a system call, but these two calls would be amortised over the whole communication phase of the superstep. In practice, however, these calls are not necessary, as the delaying of the communication allows the polling rate to be set at an intermediate level (between slow and aggressive, with the mean delay between work arriving on an empty queue and the NIC realizing that work has arrived, being half the polling interval) and data arrives on the send queue at such a rate that the NIC does not go into polling too often. There are other benefits too: since the communication code and data structures are localised, better cache locality can be achieved. Using this lightweight protocol, it is possible to use 93.62% of the raw bandwidth, which can be achieved only if the computational part of communication associated with download and reception of packets is pipelined with the actual communication on the wire; and this can be achieved only by foregoing the opportunity of overlapping user computation with communication.

Between each pair of processes a full-duplex link is maintained. Sending a packet to another processor is achieved by allocating a descriptor from a pool in the shared user-kernel memory, pointing the shared descriptor at the buffer to be sent, and setting certain control fields in a per-processor session block. The new descriptor is then placed on the queue already being serviced by the NIC. At this level, there are no data copies until the buffer is copied by the NIC. With many details omitted, the following code summarises this operation:

```
function NonBlockingSend(request : requests): boolean;

var buffer  : buffers;
    resend  : buffers;

begin { NonBlockingSend }

   buffer := DeQueue(FreePool);
   if buffer <> nil then begin
```

```
      PackBuffer(buffer,request);
      if IsAckRequiredFrom(DestOf(buffer)) then
         MarkAckRequiredIn(buffer);

      if IsRecoveryRequiredTo(DestOf(buffer)) then begin
         UnconditionallyAquireLocks;
         resend := HeadOf(SendQueue[DestOf(buffer)]);
         while (resend <> nil)
               and not IsQueued(DeviceTxQueue,resend) do begin
            if not (ProtocolMsgControl in AttributesOf(buffer)) then
               ChooseAndWaitForSlot;
            EnQueue(DeviceTxQueue,resend);
            resend := BufferAfter(resend)
            end;
         ReleaseLocks
         end;

      if not (ProtocolMsgControl in AttributesOf(buffer)) then
         ChooseAndWaitForSlot;
      EnQueue(SendQueue[DestOf(buffer)],buffer);
      EnQueue(DeviceTxQueue,buffer);

      NonBlockingSend := true
      end
   else
      NonBlockingSend := false

end; { NonBlockingSend }
```

The NIC is set to interrupt the processor whenever a packet arrives. As part of the interrupt handler, the messages are split into the correct receive queues. If any recovery should be initiated based on returning information from the sender, then data held on any of the send queues may be resent. Also, any buffers holding acknowledged sent data on the send queues are freed. Buffers removed from the NIC's receive descriptors are replaced by free buffers. By ensuring that there are sufficient free buffers and sufficient buffers on the NIC's receive queue, receive overruns are infrequent. The operation of the interrupt handler is sketched in the following code:

```
procedure InterruptHandler;

var buffer    : buffers;
    RepliedTo : boolean;
    resend    : buffers;

begin { InterruptHandler }

   if AquireLocks then begin
      while IsFilledInByDevice(HeadOf(DeviceRxQueue)) do begin
         buffer := DeQueue(DeviceRxQueue);
         RepliedTo := false;
         FreeBuffers(AckedBy(buffer),SendQueue[SourceOf(buffer)]);

         case ProtocolMsgTypeOf(buffer) of
            ProtocolMsgProd:
               if HighestSeenBy(buffer) = AckedBy(buffer) then begin
                  resend := HeadOf(SendQueue[SourceOf(buffer)]);
                  if (resend <> nil)
```

```
                and not IsQueued(DeviceTxQueue,resend) then begin
            ChooseAndWaitForSlot;
            EnQueue(DeviceTxQueue,resend);
            RepliedTo := true
            end;
         resend := TailOf(SendQueue[SourceOf(buffer)]);
         if (resend <> nil)
                and not IsQueued(DeviceTxQueue,resend) then begin
            EnQueue(DeviceTxQueue,resend);
            RepliedTo := true
            end
         end
      else begin
         resend := HeadOf(SendQueue[SourceOf(buffer)]);
         while (resend <> nil)
                and (SequenceNumberOf(resend) < HighestSeenBy(buffer)) do
            if not IsQueued(DeviceTxQueue,resend) then begin
               EnQueue(DeviceTxQueue,resend);
               RepliedTo := true;
               resend := BufferAfter(resend)
               end
         end;

   ProtocolMsgBackOff: SlowProdPaceTo(SourceOf(buffer));

   ProtocolMsgData: begin
      MaximiseProdPaceTo(SourceOf(buffer));

      if (not IsDuplicate(buffer,ReceiveQueue[SourceOf(buffer)]))
         and ((QueueLength(FreePool) > ThreshHold)
            or (SequenceNumberOf(buffer)
                  = NextExpectedFrom(SourceOf(buffer)))) then begin
            EnQueue(ReceiveQueue[SourceOf(buffer)],buffer);
            EnQueue(DeviceRxQueue,DeQueue(FreePool))
            end
         else
            EnQueue(DeviceRxQueue,buffer)
         end

      end; { case }

      if OkToSendNonPiggyBackAckTo(SourceOf(buffer))
          and IsAckRequestedFrom(DestOf(buffer)) then
         ScheduleAckTo(SourceOf(buffer));

      if IsDuplicate(buffer,ReceiveQueue[SourceOf(buffer)])
         or ((ProtocolMsgTypeOf(buffer) = ProtocolMsgAck) and not RepliedTo)
         or IsResendDataFromSourceOf(buffer) then begin
         ScheduleAckTo(SourceOf(buffer));
         SendPendingAcksTo(SourceOf(buffer))
         end

      end; { while }
      ReleaseLocks
   end { if AquireLocks }

end; { InterruptHandler }
```

Figure 5.4 shows the per-processor send and receive queues as well as the queue serviced by the NIC. Recovery entry points are indicated in the figure with dotted arrows. Since the interrupt handler queues incoming messages

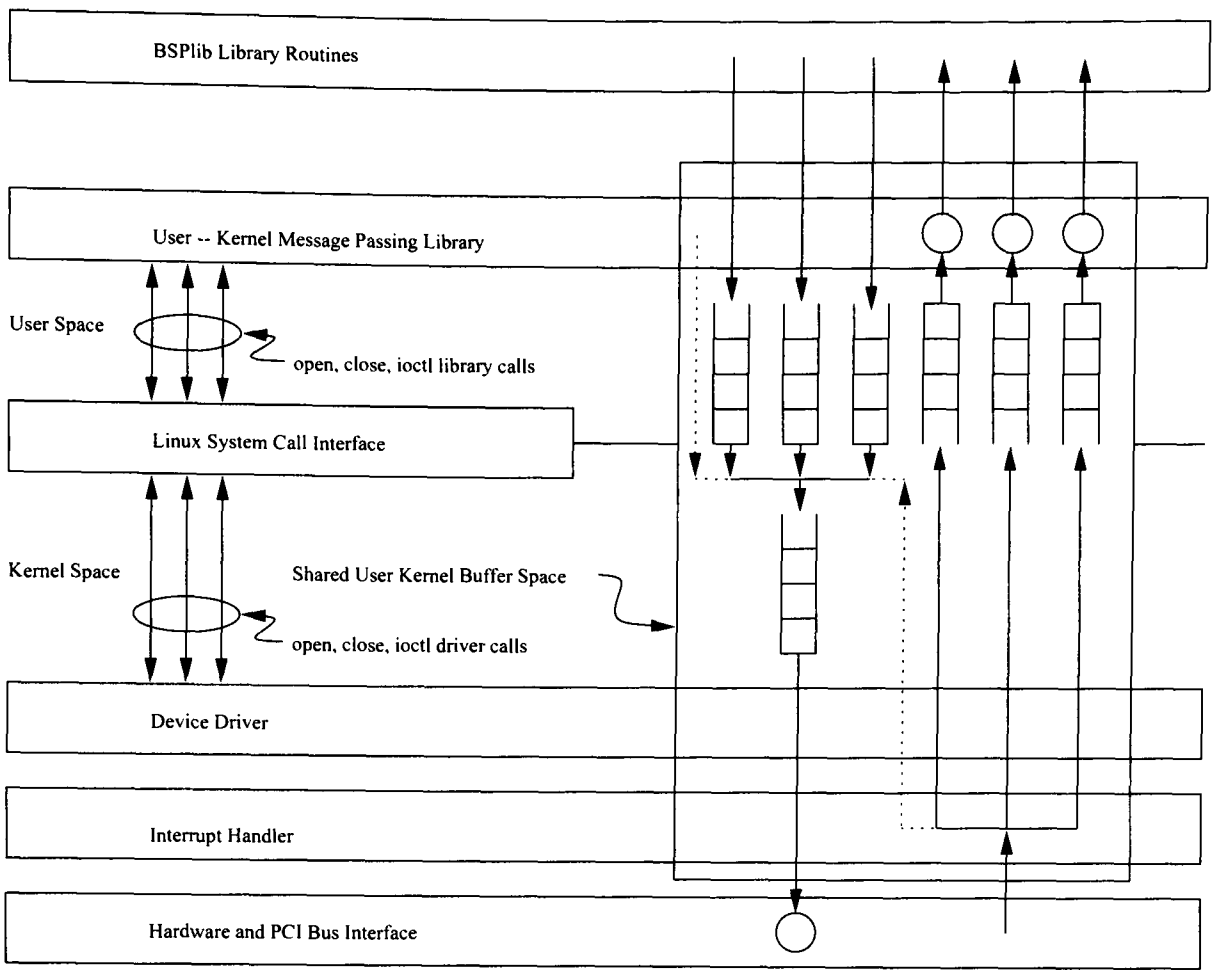


Figure 5.4: Hardware and software context of the *BSPlib*/NIC protocol stack

according to their originating processor, message reception is accomplished as outlined by:

```
procedure BlockingReceive(request : requests);

var buffer : buffers;

begin { BlockingReceive }

   if SourceRequested(request) = AnyPeer then
      ProbeFor(request)
   else
      if UserTypeOf(HeadOf(ReceiveQueue[SourceRequested(request)]))
            <> UserTypeRequested(request)
         ProbeFor(request);

   buffer := DeQueue(ReceiveQueue[SourceRequested(request)]);
   UnPackBuffer(buffer,request);
   EnQueue(FreePool,buffer)

end; { BlockingReceive }
```

5.3 Protocol properties

In the preceding sections of this chapter certain claims have been made about properties of the BSP protocol. While these claims have been made about the concrete implementations of the protocol, they are hard to demonstrate formally. The popularity of the *BSPlib*/UDP version of the protocol over *BSPlib*/TCPand *BSPlib*/NIC is due mainly to its relative independence of esoteric protocol options and of hardware. For example, the *BSPlib*/TCP version requires certain low-water mark options and/or behaviour to work correctly. Unfortunately, these options are not universally implemented. Also, the *BSPlib*/NIC version is specific to the 3COM 3C905B-TX and the Linux 2.0 kernel.

The popularity of *BSPlib*/UDP could also be attributed to its having the properties claimed. In the concrete implementations, these can be verified empirically, and over time confidence in the protocols can be achieved. As it is, the concrete implementations are used for quantitative aspects of the protocols and these are discussed in Chapter 6.

There exist many approaches to the qualitative analysis of protocols, some of which could have been performed on the (near) concrete implementations presented here. For example, Estelle [14] or SDL [16] could have been used as the programming language to implement the protocol, immediately making available a number of quantitative and qualitative analysis tools. There also exist tools to convert programs in either of these specification languages to C. However, the requirement to demonstrate that the protocol behaved properly arose only after the concrete implementations of the protocol had been written in C. In any case, a greater burden on the programming effort may have arisen as a result of using such tools by the need to map the interfaces available to the interfaces assumed by such tools.

During the development of the protocol, the techniques developed to ensure proper behaviour were tested using purpose-built benchmarks. Consequently, any formal technique applied to the protocol was not for engineering purposes, but to validate decisions taken during the design process. This was fortunate as the aspects of the protocol required for proper behaviour had become salient during its development. It was possible therefore to capture the essence of the protocol for purposes of validation quite succinctly and to not have to rewrite the protocol in an imperative language such as Estelle or SDL.

In the paper by Simpson *et al.* [107] the essence of the messaging protocol is described using CSP [70] and verified using the tool FDR2 [103]. The work described in the paper verifies that two safety harnesses at the interfaces of the protocol are indeed responsible for ensuring that the system is free from

deadlock. Although described earlier, their description is worth repeating. The first mechanism operates above the protocol. If a send fails by a higher protocol entity (above the messaging protocol stack) for lack of buffer space, then that upper layer enters a phase in which it checks for incoming messages that can be consumed.

The second mechanism operates at the lower interface of the protocol. If the number of free buffers is running low, then only the next in-sequence messages are accepted. Out-of-sequence messages arriving in this state are discarded, but not before their protocol state information has been taken into account.

Removing one of these conditions results in a livelock in which the lower layer embarks on an infinite uninterrupted sequence of events without communicating with the upper layer. From the point of view of the upper layer, the system is deadlocked. Removing both of these conditions results in deadlock.

By creating a faulty buffer process which can nondeterministically create, drop, or resequence a message at any time and provided this behaviour is bounded, the fault tolerance of the protocol is verified.

5.4 Conclusion

Clusters represent an extremely cost-effective approach to high-performance computing, as long as the potential of their communication mechanisms can be realized. There has been a great deal of interest in designing protocols that provide low latency and high bandwidth. However, a protocol that is designed to be general-purpose can make only the weakest assumptions about how it will be used.

The design of a protocol stack specifically arranged to make a cluster into a high-performance BSP machine is described. Better design decisions can be made both because the context is known, and because BSP's structure makes a novel kind of implicit global knowledge available to processors.

Earlier work has shown how *BSPlib*'s strategy of delaying communication enables messages to be packaged into optimally-sized pieces, and the choice of ordering of message sends to be made to reduce contention at destinations. A more subtle use of global information shown in Chapter 4, slotting, has been used to control the local insertion rate of data to meet global bandwidth limitations.

It has been shown here how both implicit and explicit global information can be used to:

- design an improved error recovery technique (hole-filling);
- reduce acknowledgement packet traffic by using expected traffic information;
- reduce acknowledgement packet traffic further by using BSP's superstep structure; and
- manage buffers on a per-processor basis.

All of these features result in a transport protocol that performs well (as demonstrated in the Chapter 6), especially as it provides reliability, and requires cheap hardware interfaces.

Clusters have attracted a great deal of attention, partly because of their low cost. These performance results show that it is their cost-effectiveness that should be the main attraction.

Chapter 6

BSP Protocol Performance

The performance benefits of the protocol introduced in Chapter 5 are demonstrated at the application level, unlike much of the other low-latency protocol literature. For comparison purposes implementation is also benchmarked at various other levels. Using low-level micro-benchmarks the raw efficiency of the communication equipment that can be achieved is shown and, at a higher application level, benchmarking compares *BSPlib*/NIC cluster with popular commercial parallel computers.

With a modest budget a cluster can be configured to deliver parallel performance that is comparable to large parallel computers that are at least an order of magnitude more expensive. Table 6.1 provides a summary of the results on the NAS parallel benchmarks. The *BSPlib* implementations of these benchmarks use similar code to the standard MPI implementations, except for the substitution of *BSPlib* calls for MPI calls. They show the aggregate performance improvement of *BSPlib*'s careful implementation and the NIC based protocol used as the transport layer on a cluster.

Section 6.1 gives the results of benchmarks of *BSPlib* and `mpich` on a cluster and compares performance with several commercial parallel computers. Results for both low-level (point-to-point sustained bandwidth) and high-level (NAS benchmarks) applications are given. Section 6.2 compares the *BSPlib* messaging layer with other low-level protocols designed for low latency.

Material in this chapter is based upon the paper "BSP Clusters: high performance, reliable and very low cost" by Donaldson *et al.* [33]. Some of the material also appeared in "Performance Results for a Reliable Low-Latency Cluster Communication Protocol" by Donaldson *et al.* [31].

	p	SP2 (MPI)	BSP Cluster (*BSPlib*)	Origin 2000 (MPI)
BT	4	31.20	56.18	51.10
SP	4	24.89	42.36	56.22
MG	8	36.33	39.62	36.16
LU	8	38.18	63.09	87.34

Table 6.1: Summary of the NAS benchmarks, where entries are in megaflops/sec *per process*

6.1 Benchmarks

The work has been benchmarked at various levels: at the lowest level, micro-benchmarks show the raw efficiency of the communication equipment that can be achieved. At a higher, application level, the cluster is compared to other popular computers. For these benchmarks a *BSPlib* port of the NAS Parallel Benchmarks 2.1 [4] provided by Oxford Parallel [74] is used. To compare the efficiency of the cluster and *BSPlib*, with other parallel machines, results from the MPI versions of the benchmarks are also included. On the IBM SP2 and the Origin 2000, results are for proprietary implementations of MPI; on the *BSPlib*/NIC cluster they are for `mpich` [53]. At an intermediate level of benchmarking results for the BSP parameters of the cluster are provided.

6.1.1 Cluster configuration

The prototype *BSPlib*/NIC system is a cluster of eight 400MHz Pentium II PC systems each with 128MB of 10ns SDRAM on 100MHz motherboards. Two distinct types of communication are required: slow(er) I/O communication and fast computation data. It simplifies coding and debugging to have two separate networks; a control network to deal with I/O (NFS for disks, and multiplexing terminal traffic back to the initiating console), and a network reserved for interprocess application data (i.e. the data transferred between processes initiated with *BSPlib* primitives). The control network uses the standard TCP/IP protocol suite. This structure is described in Chapter 3. Emphasis is placed on the data network.

Each of the processors runs the Linux 2.0 Kernel and the driver software is written according to the interfaces documented in Rubini [105] and Beck *et al.* [7]. Other than reserving memory for communication, no kernel changes are required. The communication devices used were eight 3COM

3C905B-TX NICs running at 100Mbps [1] and a 100Mbps Cisco 2916XL fast Ethernet switch [22].

6.1.2 Micro-Benchmarks

The micro-benchmarks are typical benchmarks used in low-level communication measurement, i.e. they measure the raw bandwidth and latency that can be achieved 'on the wire' (excluding protocol data). Three classes of micro-benchmarks are considered: measurement of the round-trip delay between two processors for various message size (Figure 6.1); link bandwidth between two processors for various message size (Figure 6.2); and per-packet latency for half-round-trip packets. In this last class, results for short (4 byte) (Figure 6.3), medium (256 byte) (Figure 6.4) and large (1400 byte) (Figure 6.5) packet sizes, and for various number of messages are provided. These benchmarks use the following configurations:

PII-BSPlib-NIC-100mbit-wire: PII Cluster, *BSPlib*, NIC driver messaging layer, 100BASE-TX, and cross-over wire.

PII-BSPlib-NIC-100mbit-2916XL: PII Cluster, *BSPlib*, NIC driver messaging layer, 100BASE-TX, and Cisco 2916XL.

PII-BSPlib-UDP-100mbit-2916XL: PII Cluster, *BSPlib*, UDP/IP messaging layer, 100BASE-TX, and Cisco 2916XL.

PII-BSPlib-TCP-100mbit-2916XL: PII Cluster, *BSPlib*, TCP/IP messaging layer, 100BASE-TX, and Cisco 2916XL.

PII-MPI-ch_p4-100mbit-2916XL: PII Cluster, mpich, ch_p4, 100BASE-TX, and Cisco 2916XL.

SP2-MPI-IBM-320mbit-vulcan: thin node, 66Mhz IBM SP2, IBM's implementation of MPI, and 320 Mbps Vulcan switch.

SP2-MPL-IBM-320mbit-vulcan: IBM SP2, IBM's proprietary messaging protocol MPL [75], and 320 Mbps Vulcan switch.

O2K-MPI-SGI-700mbit-ccnuma: SGI Origin 2000, and SGI's implementation of MPI.

Round-trip time

Figure 6.1 shows the round-trip time between two processors for increasing message sizes. The experiment performed thousands of samples, and the *minimum* round trip delay time was recorded (benchmarks that observe the variance in communication performance are described in section 6.1.3).

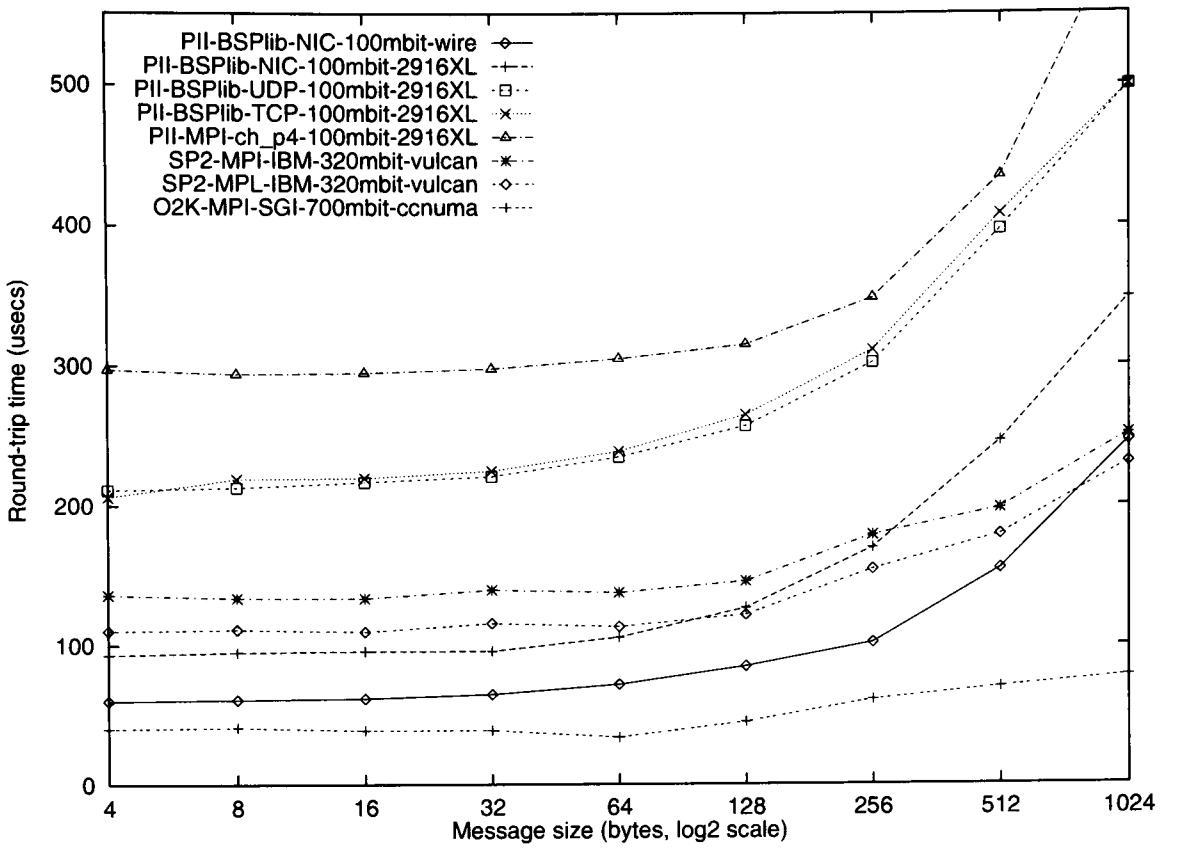


Figure 6.1: Round-trip time between two processors as a function of message size

This choice of minimum value accounts for the TCP/IP and UDP/IP results being identical since, when no packets are dropped and error recovery is not initiated, the two protocols behave similarly. The difference of approximately $100\mu s$ for small messages between the NIC and UDP/IP results demonstrates the shallowness of the protocol stack of Figure 5.3(a) compared to Figure 5.3(b) (page 66). The `mpich` results are consistently worse than any other protocol running on the cluster.

As well as benchmarking the NIC based protocol through a 100Mbps switch, two machines were connected back-to-back using a crossover wire. The purpose of this experiment was to identify the software latencies in the NIC, ignoring the effect of the switch. For 4-byte user messages (which will be 40 bytes including headers, but 60 bytes on the wire due to a 60-byte minimum frame size), the experiment PII-BSPlib-NIC-100mbit-wire achieved a round-trip time of $58\mu s$. This compares to $93\mu s$ for the PII-BSPlib-NIC-100mbit-2916XL experiment, the difference being entirely accounted for by the Cisco 2916XL switch latency [91] of $2 \times 17.2 = 34.4\mu s \approx 93 - 58 = 35\mu s$.

From the figure, the latency of the NIC based protocol running on the cluster is significantly smaller than any other protocol on the cluster, and outperforms the IBM SP2 for messages less than 256 bytes. The loss in latency for larger messages is entirely due to the increasing latency through

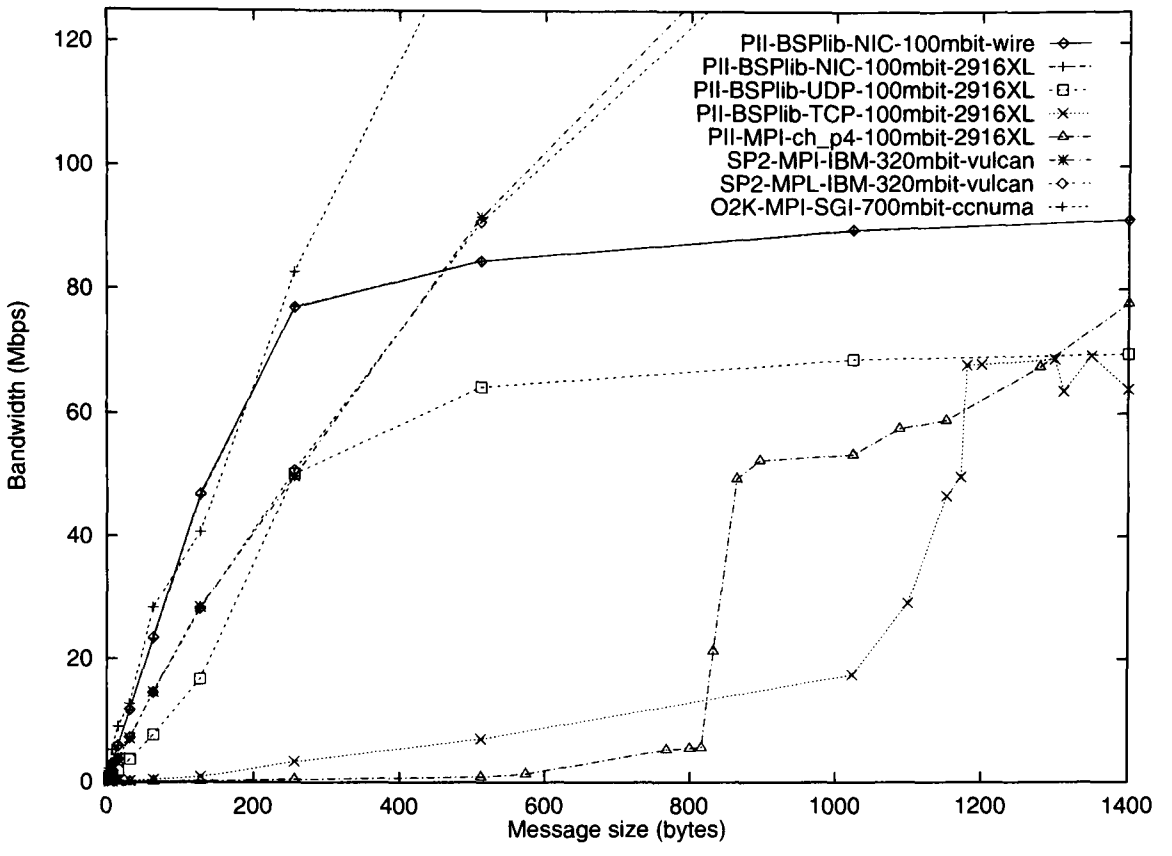


Figure 6.2: Link bandwidth between two processors as a function of message size

the switch. These results are consistent, though slightly better, than the results of U-Net over 100Mbps Ethernet [126]. However, unlike the U-Net results, error recovery is also performed in the *BSPlib*/NIC implementation.

Bandwidth

Figure 6.2 shows the sustained bandwidth between two processors for increasing message sizes. The benchmark arranges that one process sends a large number of packets (with their sizes shown on the horizontal axis) to another process, which returns a single small packet after all the packets have been received. The bandwidth on the vertical axis is calculated from the amount of data communicated, and the time between the first packet sent and the small control packet returned. The return latency can be ignored as there are many packets sent by the first processor. However, the traffic includes any reverse communication required to acknowledge the received packets or to recover from errors.

Considering that the peak bandwidths of the Origin 2000 and the SP2 are considerably higher than that of a 100Mbps switch, it is surprising that the cluster can outperform the SP2 for user messages up to 400 bytes, and can match the Origin 2000 for messages up to 200 bytes. After these two points,

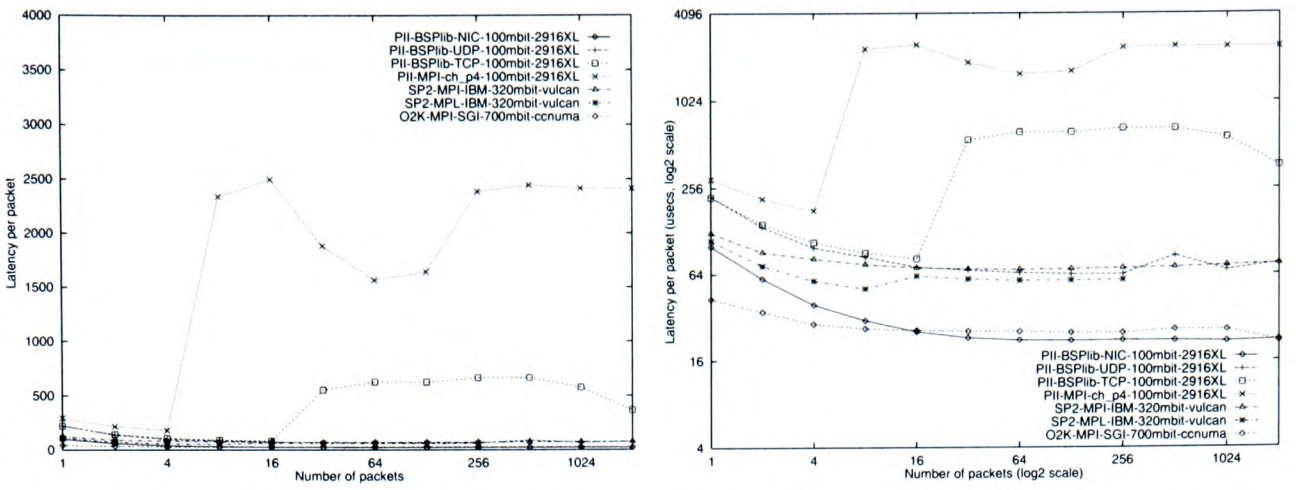


Figure 6.3: Latency per packet (half roundtrip) for 4-byte messages as a function of number of packets. The graph on the right is a log-log scale of the one on the left.

these more-expensive machines perform considerably better.

The *BSPlib*/UDP and *BSPlib*/NIC transport layers, which both use the error recovery and acknowledgement protocol described in Chapter 5, have smooth, predictable bandwidth curves that rise to their asymptotic levels quite early. In contrast, the *BSPlib*/TCP transport layer and `mpich` show erratic and late-rising curves. This demonstrates the earlier assertion of the unsuitability of TCP/IP for high-performance computation due to its inappropriate acknowledgement and error-recovery mechanisms.

Although the peak bandwidth of TCP/IP is greater than UDP/IP in the graphs, the *BSPlib*/UDP implementation can achieve higher bandwidth utilisation (not shown here), although at the expense of a later rising curve.

The observed software latency at the *BSPlib* layer for 1400 bytes of user data is 11.6μs for packet download. This includes up to 7.8μs spent in copying the application data structure into a user/kernel space buffer with `memcpy`. For packet upload (excluding memory copy into the application), the time for invoking the interrupt handler, doing any necessary error recovery, and uploading the packet is, on average, 3.6μs per packet. Due to the lightweight nature of this protocol, the observed efficiency on the wire is 93.62% (i.e. 93.62Mbps) for the packet including protocol headers (i.e. 1436 bytes including headers).

Spray latency

Non-blocking communications can easily fail or suffer from deadlock, if two processes simultaneously push data into their protocol stacks in an attempt to send large amounts of data to each other. The point at which this dead-

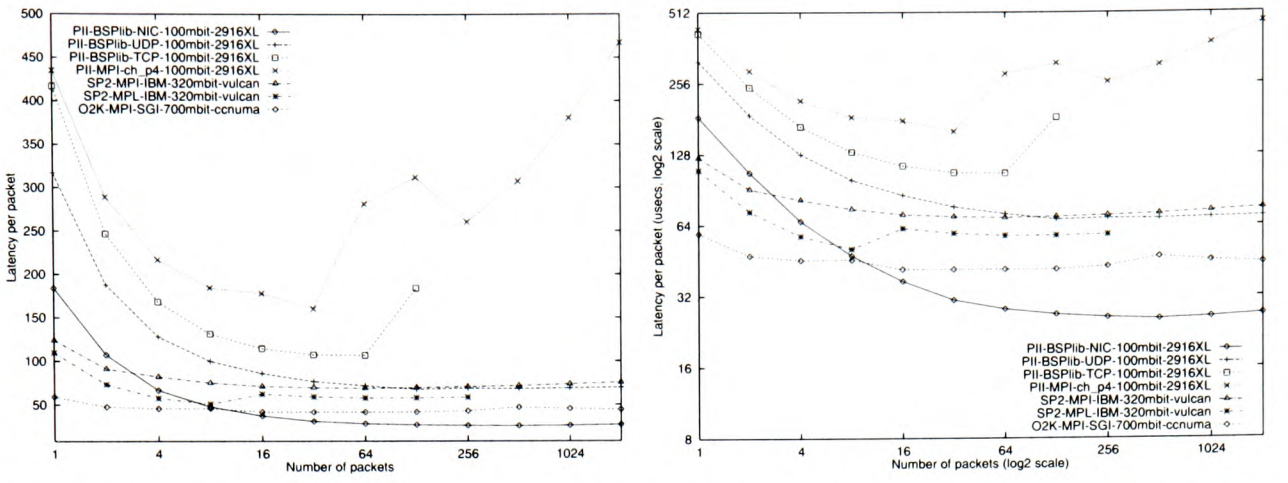


Figure 6.4: Latency per packet (half roundtrip) for 256-byte messages as a function of number of packets

lock occurs depends upon the buffering capacities of both sender and receiver. Quoting from the MPI report: "*... the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in 'pathological cases'.*" [90]. In MPI, if the error code of the send is ignored and data continues to be sent by a process, or the application fails to relieve the condition, deadlock soon occurs. Although well-defined MPI programs are supposed to check the error code, and users deserve everything they get if they ignore it, the benchmark described in this section is intended to quantify the effectiveness of the buffering capacity of the various messaging systems by continually sending data in an attempt to exhaust buffers.

Figures 6.3–6.5 show the gap between successive send calls in the steady state, for 4, 256 and 1400 byte messages. The benchmark combines the features of the round-trip and bandwidth benchmarks: a process sends i packets to a receiving process, which reflects them back. However, the application layer on the sender will not receive any of the reflected packets until all packets have been emitted. For i packets shown on the horizontal axis, the time shown on the vertical axis is the time between sending the first packet and waiting for the last packet to return, *divided by* i. In the limit, this benchmark measures the latency between two successive send calls. An alternative interpretation, assuming deadlock does not occur, is that it measures the level of pipelining of communication, i.e. at $i = 1$ it is equivalent to the round-trip benchmark; for $i > 1$ it measures the degree of communication overlap due to packets in flight.

Before describing the results of this benchmark, it is worth repeating that the *BSPlib*/UDP and *BSPlib*/NIC messaging layers are used by *BSPlib* in such

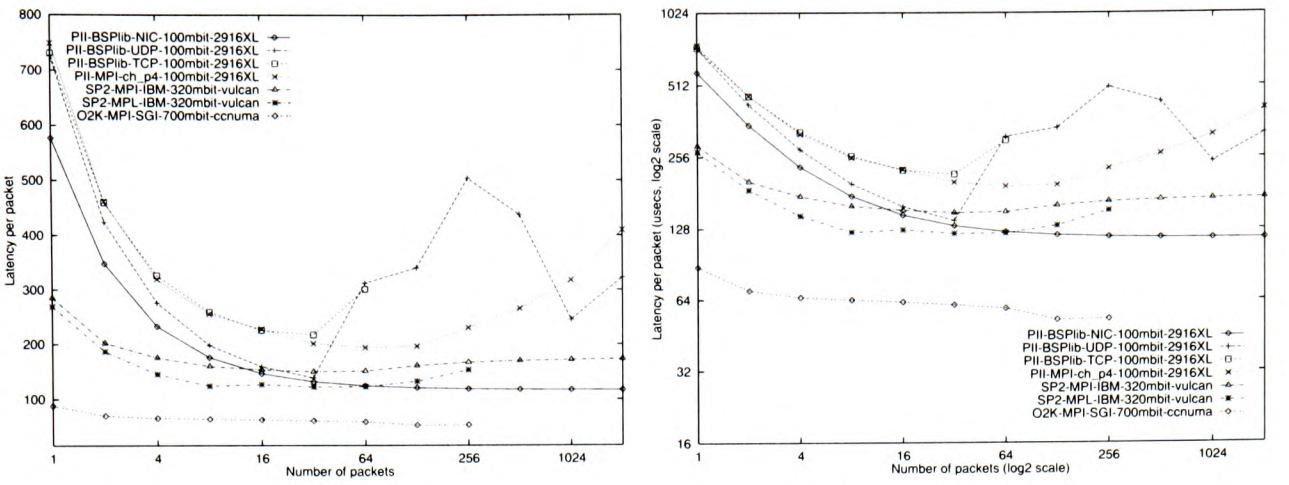


Figure 6.5: Latency per packet (half roundtrip) for 1400-byte (approximately full-frame) messages as a function of number of packets

a way *that this form of deadlock cannot occur.* This is achieved by ensuring that, if the protocol stack is unable to accept a send operation, then *BSPlib* will always attempt to either: (1) consume a packet if there is one available; or (2) those links that have a large number of unacknowledged send buffers will be requested to return an acknowledgement. A formal demonstration that the usage of the messaging layer in this manner is *BSPlib* required for proper protocol was demonstrated by Simpson *et al.* [107] and was discussed in Chapter 5. In the *BSPlib*/TCP implementation, deadlock is avoided by using global flow control via slotting (Donaldson *et al.* [32]; also Chapter 4).

All the figures show that for large packet sizes, for a large number of messages in flight (i.e. bulk communication as in BSP) the *BSPlib*/NIC implementation on the cluster outperforms all other configurations, including the SP2 and Origin 2000. All MPI implementations on the Cluster, SP2, and Origin 2000 suffer from deadlock when more than 256 packets are injected into the protocol stack.

While a process has data in the protocol stack to consume, it is possible that up to one time slice (more in the presence of higher priority work) can elapse before the process gets control to consume the packets. Because this time is bounded, the maximum number of packets that can fit into the protocol stack without packet loss is also bounded. This packet loss results in retransmissions which show in the figure as sudden increases in the per-packet latency (this is the per-packet latency observed by the application and includes the time required for any retransmissions). Several of the protocols exhibit this problem at quite small packet sizes. Even the *BSPlib*/UDP protocol has it, although not until quite large packet sizes and transmission rates. The NIC protocol does not show the problem because it has both a large number of buffers and keeps them in a shared pool, but the determination of the size of this buffer pool together with the injection

rate are still determined by this time slicing effect.

Although the round-trip time on the cluster is $95\mu s$ for a single 4-byte packet, multiple packets can be injected into the protocol stack every $26\mu s$. While it is possible to do so faster ($8.2\mu s$ is the minimum), the protocol will not settle to a steady state, as the receiver cannot keep up with incoming packets, and overruns start to occur. Therefore $26\mu s$ was achieved by pacing the sender. The hardware itself imposes a limit on the rate at which frames can be injected into the medium. Ethernet frames are required to have a minimum length of 512 bits or 64 bytes which corresponds to $5.12\mu s$ to transmit or receive a frame by the NIC over 100Mbps media. This does not imply that the software overhead is approximately only $3.1\mu s$, as the NIC operates concurrently with the processor.

With the *BSPlib* message packing scheme short messages are rare and a benchmark of sustained short messages is artificial. During normal operation with packed messages, the message lengths impose a minimum time for packet upload. At $26\mu s$ it is only frames shorter than 325 bytes that can be uploaded in a sustained manner quicker than once every $26\mu s$.

The $26\mu s$ is determined by the number of buffers and the length of the time slice interval (at least for the Linux 2.0 kernel scheduler's algorithm). By default, at most a time slice will elapse before a process is scheduled after becoming ready (provided that there are no other processes contending for the processor), so the risk of a user process locking a communication data structure and becoming descheduled because it has exhausted its time slice, is significant. In the *BSPlib*/NIC implementation, locked communication data structures mean that the servicing of uploaded frames is deferred until the data structures are unlocked. As this happens when the process is next scheduled, as much as a time slice could have elapsed. For the Linux 2.0 kernel on the Intel architecture this is $10ms$ which means that with 400 receive buffers made available (the default of the corresponding *BSPlib* runtime parameter) all will be exhausted if frames arrive at a rate of once every $10ms/400 = 25\mu s$ or faster. Therefore the figure of $26\mu s$ protects against such overruns caused by sustained short messages.

Figure 6.3 shows that the TCP/IP implementations (both `mpich` and the TCP/IP variant of *BSPlib*) all have a poor spray latency for large numbers of packets injected into the protocol stack. This is a consequence of the windowing flow control used by TCP where, if communication is run in a state where the window is always exhausted, throughput is considerably reduced.

6.1.3 *BSPlib* Benchmarks

Micro-benchmarks are an important way of showing the basic properties of a link: the minimum latency and the mean bandwidth. But optimising these is not enough to ensure proper performance in a parallel context. In Chapter 7 it is shown that low variance for the bandwidth is also important, as it has a strong effect that quickly limits scaling to larger configurations. The micro-benchmarks say nothing of the performance after integration, yet these are precisely the figures that concern the customer: how well the parallel computer performs under a realistic workload. There are two ways of looking at this problem: the BSP way, and the NAS parallel benchmark way. The BSP model asserts that the interconnect can be parameterised by using values of l and g. Using the BSP cost model, these can accurately predict application performance in a way that users can understand [59, 63, 101, 34]. The other approach is to run, as a series of benchmarks, applications similar to typical users' workloads. The advantage of the BSP approach is that it makes it possible for a potential user to compare machines without being required to run codes similar to their applications. This potentially allows a large number of machines to be considered at no extra cost. The downside is that in order for these figures to be translatable into something useful for the potential user, cost analysis must be performed on the user's code. If the code is written in a BSP style, then this cost analysis is not difficult and need be done only once for each application.

Another important aspect of benchmarking from a BSP point of view, is that it measures the *integration* of the communication protocols and hardware which are the most significant characteristics of a parallel computer. One can think of the micro-benchmarks as measuring individual components in isolation. However, the purpose of *BSPlib* is to provide a BSP computer with reasonable parameters. These parameters are global parameters and are a measure of the entire computing system under some non-trivial workload. The parameter g for example, is a measure of the global bandwidth and as such, when benchmarks are run to determine its value it is more than the point-to-point bandwidth between two processors that is being measured. What is being measured is the combined effect of the interconnect, the protocol stacks at the processors, the interaction of traffic in the interconnect, and the interaction of traffic from multiple processors. Software cannot change the physical characteristics of an interconnect, but protocol decisions which take the global picture into account can be made. For example, use of delayed communication, Latin squares, and pacing injection have already been mentioned. An obvious global protocol decision that improves bandwidth utilisation is the hole-filling protocol: without considering the interconnect globally, it seems sensible to use the full-duplex link for a go-back-n protocol. Similarly, for coarse-grained implicit acknowledgements, point-to-point

Machine	$mega - flops/s$	p	Barrier (μs)	Bandwidth ($Mbps$)
SGI Origin 2K	101	2	8.6	399.1
		4	22.7	355.3
		8	39.1	212.7
		16	121.1	97.1
		32	386.6	48.5
IBM SP2	26	2	73.2	106.6
		4	137.8	104.0
		8	208.2	73.0
BSP Cluster (NIC)	88	2	64.2	84.1
		4	133.6	89.4
		8	207.2	91.1

Table 6.2: Barrier latency and link bandwidth

protocols cannot take into account the state of other links. This leads to another benefit of thinking of protocols at a global level: in a point-to-point protocol such as TCP/IP, buffer resources are usually reserved for the circuit and often make a distinction between a receive buffer pool and a send buffer pool. This means that, for example, skewed communication patterns have to cater for the worst case of skewing in order for buffer starvation to be prevented on all links. In the *BSPlib* messaging protocol a single large buffer pool per processor is maintained, with the number of buffers remaining in the buffer pool used to make protocol decisions. This makes the protocol much more tolerant of skewed communication patterns. This is in support of the BSP model which ignores the skewness of the communications patterns and asserts that the cost of communication at a processor is the sum (or the maximum) of all traffic entering and leaving a processor during any particular superstep. Apart from this large degree of sharing of buffer resources the protocol realizes that there should be a large amount of slackness in the buffer pool given the asynchronous nature of the communication and the different processing elements.

The benchmarks in this section show that despite the advantage of raw high-performance interconnects, there is some limit on the scalability of this performance under a BSP workload (at least for the present *BSPlib* implementations) and that, with some effort, a more modest interconnect shows some good scalability characteristics.

In Table 6.2, sample mean barrier latencies (equal to the BSP parameters l/s) and link bandwidths (equal to the BSP parameters $32s/g$) are given. Projecting the results for the cluster and the Origin 2000, as the cluster

Protocol	p	$\tilde{Mbps}$	σ_{Mbps}
TCP	2	71.92	8.32
	8	59.90	7.53
UDP	2	81.44	18.90
	8	79.66	11.30
NIC	2	90.54	8.79
	8	86.67	8.66

Table 6.3: Mean and SD of the global bandwidth

has a constant global bandwidth of $\approx 90Mbps$ as p increases, and the Origin's performance halves as the number of processors is doubled, it is quite likely that at 16 processors in the cluster, the cluster's global bandwidth parameter will be approximately the same as the Origin 2000's. Independent performance reports for the Cisco 2916XL switch [91] show that the backplane performance of these switches will not start to flatten out until about 16 processors under these loads. To get above this number of processors, switches can be cascaded in a butterfly, and the peak bandwidth of each link can be increased above 100Mbps by multiplexing traffic over two or even three NICs (the PCI bus capacity should not be a problem for this number of NICs). Even at eight processors, the *BSPlib*/NIC implementation shows a better global bandwidth than the IBM SP2 communicating over a Vulcan switch. The barrier latencies of this implementation and the IBM SP2 are also very similar. As would be expected, the latencies of the CC-NUMA Origin 2000 are quite good. It should be taken into account that these are not only measurements of characteristics of the machine, but also of the *BSPlib* implementation providing the approximation of the BSP computer. As such, these are as much measurements of the hardware as they are of the protocol decisions, both good and bad.

The all-to-all communication patterns used in the above benchmark are very regular and balanced. In contrast, the results shown in Table 6.3 are from a benchmark that performs random, often skewed, communication patterns. The hypothesis according to the BSP cost model is that the cost of communication should be directly proportional to the maximum amount of traffic entering or leaving any single process—implying that the most heavily loaded NIC is the bottleneck in the system, and the interconnect is sufficiently rich. The results in Table 6.3 show the sample mean and sample standard deviation of the bandwidths, in terms of g/s, for random traffic [62].

The results for the mean bandwidth correlate well with the micro-benchmarks and show that the lower the protocol level, the better the mean band-

	p	SP2		Cluster				Origin 2000
		MPI	*BSPlib*	MPI	*BSPlib*			MPI
		IBM	MPL	`mpich`	TCP	UDP	NIC	SGI
BT	4	31.20	36.44	51.57	50.96	36.52	56.18	51.10
SP	4	24.89	27.31	33.15	40.86	18.37	42.36	56.22
MG	8	36.33	36.78	21.02	36.05	31.11	39.62	36.16
LU	8	38.18	36.50	46.34	55.50	50.15	63.09	87.34

Table 6.4: Results in mega-flops/sec *per process* for NAS parallel benchmarks (class A) v2.1

width. The standard deviations of the TCP/IP and NIC based protocols are similar. This is to be expected as they both perform error recovery in the kernel, whereas the *BSPlib*/UDP implementation executes this part of the protocol in user-space and has a high variance as a result. Although the NIC and TCP/IP standard deviations are similar, it is harder to achieve low variance at a high bandwidth, and therefore the NIC's standard deviation is more impressive. This can be highlighted by considering that it is always possible to achieve a good standard deviation at the expense of a low mean, by slowing down the protocol.

6.1.4 NAS Parallel Benchmark Performance

The low-level performance figures can be fully exploited at the application level. To see this, the NAS Parallel Benchmarks 2.1 [4] are used and compared. For the *BSPlib* benchmarks, the codes ported by Oxford Parallel [74] are used. In each case, the results from the MPI [89] versions of the code are also included. On the IBM SP2 and the Origin 2000 these are proprietary MPI libraries, whilst `mpich` is used on the cluster.

A BSP implementation [74] of version 2.1 of the NAS parallel benchmarks was used to compare the performance of the *BSPlib*/TCP, *BSPlib*/UDP, and the *BSPlib*/NIC implementations of *BSPlib* on four and eight processors. Also, a standard MPI implementation of the benchmarks was used to compare the relative performance of the cluster with an 8-processor thin-node 66Mhz IBM SP2, and a 72-processor 195Mhz Origin 2000. Class-A-sized benchmarks were used. Due to memory limitations of the cluster, results from the FT benchmark are not given. In the following sections, profiles from each of the benchmarks are analysed, and their resulting performance is discussed. In this section a brief summary of the results for each of the four benchmarks is given:

BT is an application that solves systems of blocked-tridiagonal linear equations.

SP is an application that solves systems of scalar-pentadiagonal linear equations.

MG is a multigrid benchmark.

LU solves a system of linear equations using LU decomposition. The benchmark performs large amounts of small (5 word) communications, and therefore provides a good measure of the communication latency of a system.

Table 6.4 summarises the results of the NAS benchmarks for the three *BSPlib* protocols running on the cluster, comparing MPI variants of the benchmarks running on the cluster, SP2, and Origin 2000. The *BSPlib*/NIC implementation on the cluster outperforms `mpich` and IBM's proprietary implementation of MPI on the IBM SP2 for all the benchmarks. On the Origin 2000, the *BSPlib*/NIC implementation performs better on half of the benchmarks.

As the NAS benchmarks are mostly compute bound, large improvements in communication performance will only be realized as small improvements in the total mega-flop rating of each of the benchmarks. In order to show the improvements of the *BSPlib*/NIC communication protocol on the cluster compared to MPI, Table 6.5 gives a breakdown of time spent in computation and communication for each of the protocols running on the cluster. The communication time is calculated using a BSP trace profiling tool described in Section 6.1.4 from data gathered using the *BSPlib*/NIC implementation of *BSPlib*. As all the benchmarks on the cluster use the same Fortran compiler (Absoft) and compiler options, and the computational part of the NAS benchmarks were not changed in any of the benchmarks, it is assumed that the computation time spent in all the benchmarks is independent of the protocol (i.e. MPI, *BSPlib*/NIC, *BSPlib*/UDP, and *BSPlib*/TCP) used for communication. This isn't necessarily true, and it will be argued that the results for the *BSPlib*/UDP implementation are due to poor computational performance caused by a scheduling problem. However, in this table, any slowdown in a benchmark's execution is attributed solely to the communication protocol.

Table 6.5 shows that the *BSPlib*/NIC implementation of *BSPlib* produces communication that is approximately 4 times better than `mpich` on all the NAS parallel benchmarks on the cluster. The improvements of the NIC protocol compared to the *BSPlib*/TCP implementation are not as marked as the `mpich` results, as some of the improvements are due to global optimisations that are applied in all implementations of *BSPlib* (i.e. packing and scheduling). As the NAS benchmarks communicate large amounts of data,

	p	compu-tation time	communication time and slowdowns						
			BSPlib NIC	`mpich`		*BSPlib*/TCP		*BSPlib*/UDP	
BT	4	724.6s	24.2s	91.1s	3.8	100.9s	4.2	427.5s	17.7
SP	4	458.9s	42.8s	182.2s	4.3	61.2s	1.4	697.8s	16.3
MG	8	11.1s	1.1s	4.6s	4.2	2.4s	2.2	4.5s	4.1
LU	8	205.9s	30.5s	115.9s	3.8	62.8s	2.1	91.4s	3.0

Table 6.5: Slowdown factor of each protocol compared to the *BSPlib*/NIC protocol on the cluster

		Data received	Explicit acks	Data dropped	Duplicate received	Implicit acks
BT	UDP	893922	7.32%	38964	14913	0.7%
$p = 4$	NIC	820233	2.53%	1	0	0.3%
SP	UDP	1561491	6.58%	69076	21882	1.0%
$p = 4$	NIC	1422927	1.00%	14	0	0.4%
MG	UDP	148023	4.21%	2339	1559	8.1%
$p = 8$	NIC	140113	1.51%	4	0	6.8%
LU	UDP	1968612	2.99%	33681	3819	32.0%
$p = 8$	NIC	2026933	6.23%	93	1900	30.5%

Table 6.6: Packet statistics for the NAS parallel benchmarks

the latency improvements of the *BSPlib*/NIC implementation over *BSPlib*/TCP will not be exercised to the fullest, and this accounts only for the approximately two times speed-up in communication for these benchmarks.

The *BSPlib*/UDP results are particularly interesting due to the extremely poor performance of the computation bound benchmarks SP and BT. This illustrates one of two problems: Having the error recovery in user-space can often result in it being sluggish due to the height of the UDP/IP protocol stack, and the time required to deliver a `SIGIO` signal. Another problem concerns process scheduling: since a larger portion of the protocol executes in user-space, the useful quanta of the program are diluted. For the benchmarks which are communication-bound, the overall results are better than `mpich`, but the scheduling problem still results in a poor flop rate compared to the TCP/IP implementation of *BSPlib*. These results show that the widespread trend towards purely user-space communication may be misguided, as it is at the expense of the computation rate.

Table 6.6 shows the number of extra error-recovery packets generated in the *BSPlib*/UDP and *BSPlib*/NIC implementations of the same protocol.

The high number of duplicates and data dropped in the UDP/IP implementation is because packets are being dropped by the kernel, somewhere between their reception at the processor and the `SIGIO` signal being scheduled to the user process. This highlights the importance of slackness in the buffer pool *at the lowest level of the protocol.* Designing buffer management techniques at a higher level is irrelevant if the device driver on the NIC or the UDP/IP protocol stack, is unable to handle the incoming data. The data in the "Explicit acks" column shows that the NIC implementation generates far fewer fine-grained acknowledgements than the *BSPlib*/UDP implementation. This is due to a large number of buffers. Also, some of the explicit acknowledgements are caused by 'prodding' messages for data that has been dropped (note that the *BSPlib*/UDP drops far more packets). From this data it seems that LU is generating twice as many acknowledgements in the *BSPlib*/NIC implementation. This is due to a high idling rate of some of the processors, which frequently 'prod' a partner process to check if anything has been dropped. The rate at which the communication occurs is proportional to the round-trip time (even though an exponential back-off is also used) and, because of the smaller round-trip of the NIC implementation, far more 'prodding' acknowledgements are generated. The column labelled "Implicit acks" identifies the percentage of packets that were reclaimed from the send queue due to the coarse-grained acknowledgement scheme. In most of the NAS benchmarks, piggy-backed acknowledgements and explicit fine-grained acknowledgements are used to remove most of the packets from the send queue. Only in LU is there enough communication to exploit the coarse-grained scheme. This results in the highest number of explicit acknowledgements generated by all the benchmarks, as well as 30.5% of all packets sent being acknowledged using the coarse-grained acknowledgement scheme of Section 5.2.1.

BT: Blocked Tridiagonal

Figure 6.6 shows a portion of the superstep communication trace [59] for BT. The figure shows two graphs, the top shows the volume of communication leaving each process, and the bottom shows the volume of communication arriving at each process. Each bar in the graph identifies the amount of data communicated for a particular superstep (the key at the top of the bar can be used to locate the superstep in the program). The width of the bar is determined by the amount of time spent in communication for that superstep. The white space between consecutive bars represents the amount of time spent in computation by the slowest running process. The figure shows that the computation in the problem overshadows communication, and that the communication is balanced amongst the processors (although not among the supersteps). As the *BSPlib*/NIC implementation of *BSPlib* outperforms the

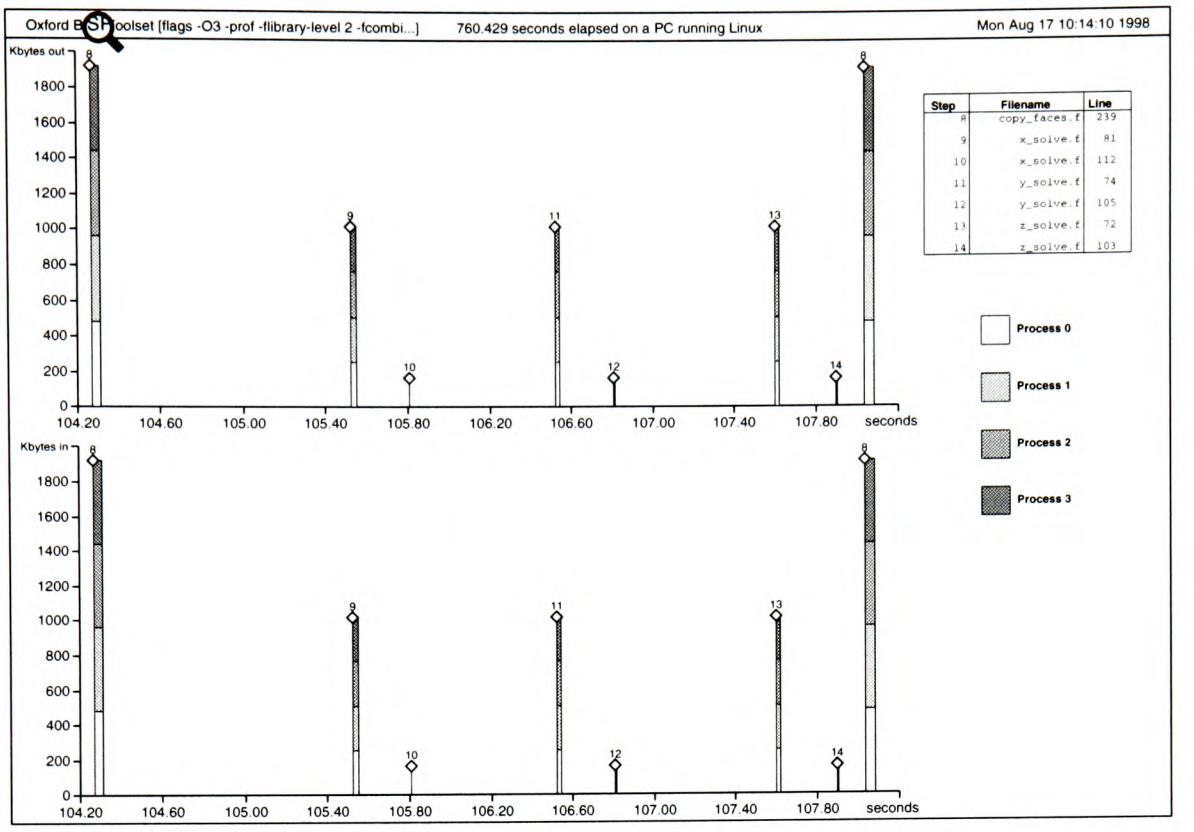


Figure 6.6: Superstep trace for BT

Origin 2000 in this benchmark, and the benchmark is computation-bound, the PII processors and memory organisation are slightly better than the R10000, *for this particular benchmark.*

SP: Scalar Pentadiagonal

The communication trace for SP (not shown) has a similar profile to BT. However, the two differ in the time required for a single iteration of the algorithm to complete. From Figure 6.6, an iteration takes approximately four seconds. In contrast, SP performs a similar amount of communication, although each iteration takes only 1.4 seconds. Therefore the ratio of communication to computation is higher in this benchmark, and accounts for the lower mega-flop rate (Table 6.4) on the cluster by comparison with the Origin 2000. This is due to large amounts of data being communicated in each superstep and, as was shown in Figure 6.2, the Origin has larger asymptotic bandwidth than the cluster.

MG: MultiGrid

MG has a more interesting communication profile than the previous benchmarks, in that it has multiple components to the algorithm, each of which

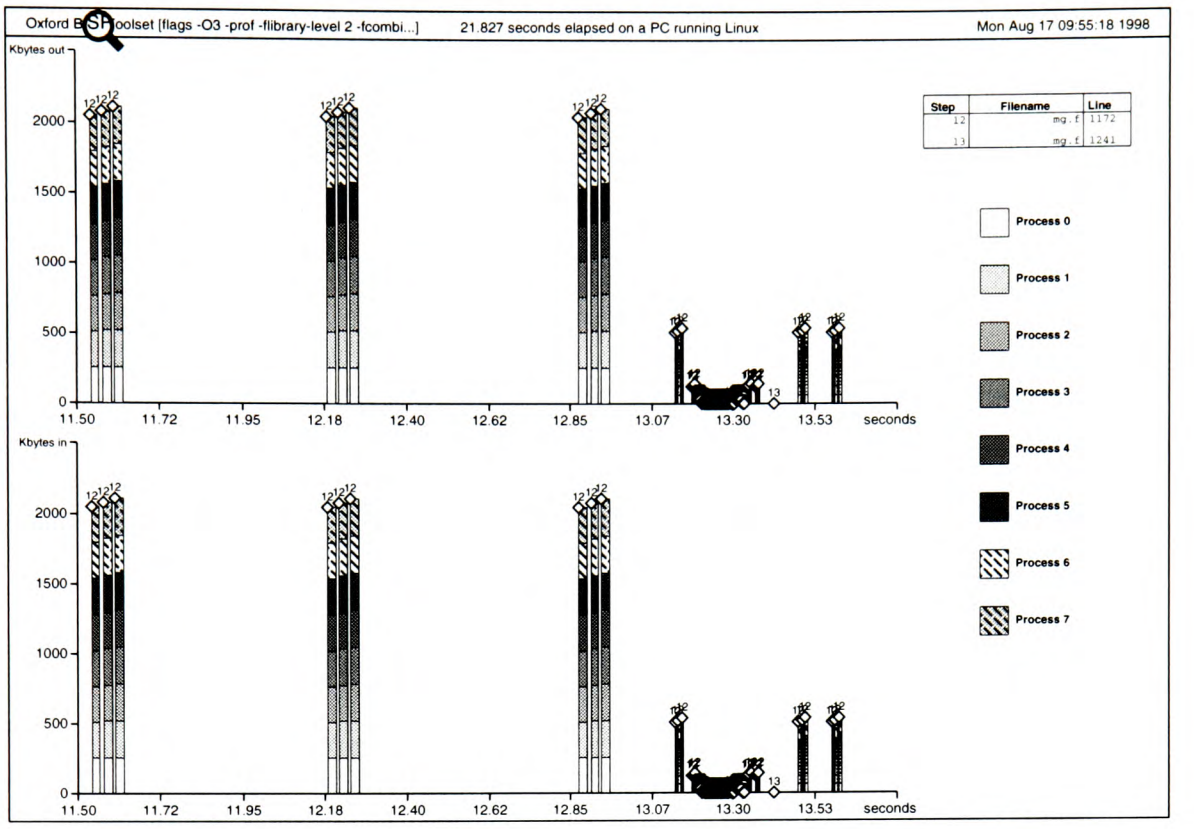


Figure 6.7: Superstep trace for MG

have a different computation and communication profile. Figure 6.7 shows that there are initially three smoothing iterations on a fine-grained grid that result in a large amount of computation and communication. Next, a larger number of iterations are performed on coarser grids. Although the size of communication on these grids is significantly smaller than the fine grids, the ratio of communication to computation is significantly higher. Surprisingly, the cluster outperforms the Origin 2000 on this benchmark. This can be attributed to the higher flop rate during the fine grid calculations in conjunction with a better sustained communication latency for smaller packets associated with the coarser grids (see Figure 6.4).

LU decomposition

Figure 6.8 shows a call-graph profile [65, 64] of LU running on eight processors of the cluster. The graph shows a call-trace of the procedures active during program execution, where the leaves of the graph identify supersteps that were encountered during program execution. For example, the bottom-left node identifies that the function `main` called `SSOR`, which contained a superstep at line 200 of the file `ssor.f` and which was executed 15500 times.

The root node `main` in the figure shows that both the computation and communication within LU is highly unbalanced. Of 281 seconds of computation

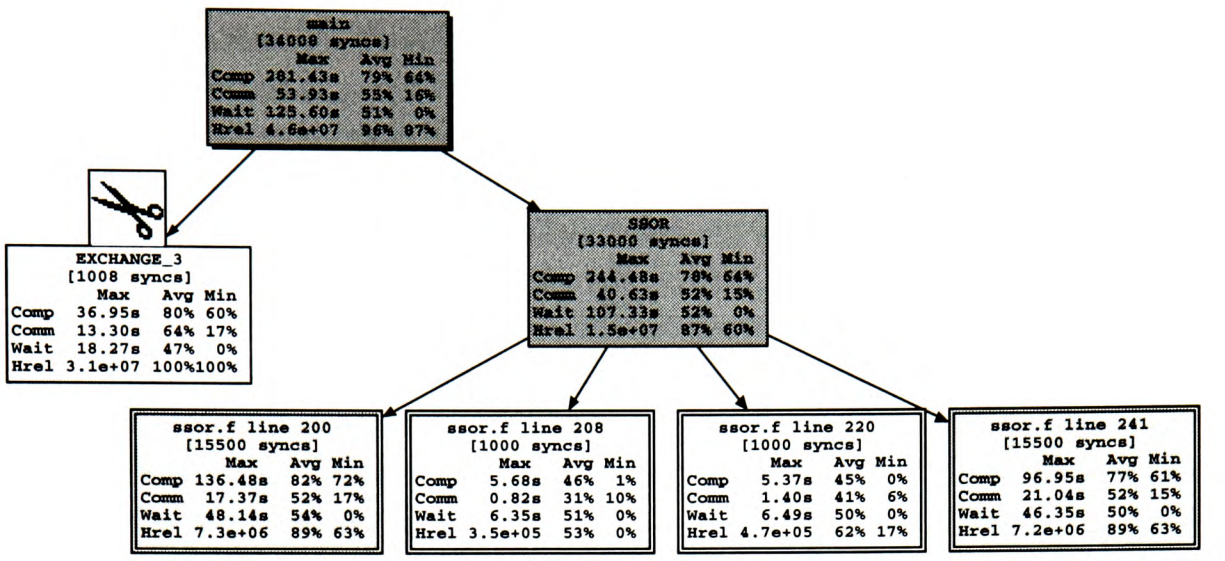
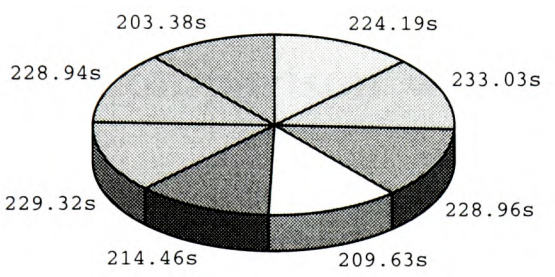
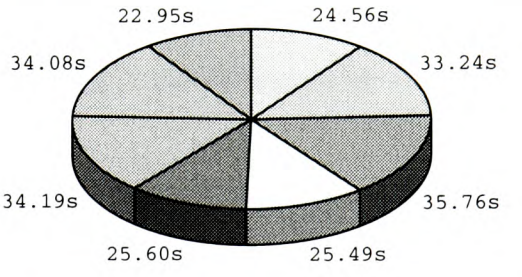


Figure 6.8: Call-graph profile of LU at $p = 8$



(a) Computation



(b) Communication

Figure 6.9: Computation and communication breakdown per process for the node `main`

time, the average utilisation of the processors in each superstep was only 79%. Similarly, the total communication time was 53 seconds yet, due to data skew, each processor on average utilised 55% of communication available during that superstep. Unlike the other NAS benchmarks, which have an obvious BSP structure and result in balanced computation and communication, this implementation of LU is not entirely suited to BSP (the BSP version of the benchmarks changed only the communication calls within the code). Algorithmically-better implementations of LU have been developed for BSP [71]. The problem here is that LU performs iterations down the diagonal of a matrix, and the BSP implementation performs an explicit barrier synchronisation on each iteration. Therefore, during the startup and close-down iterations only one process is active (while the others idle at barriers), and hence LU has a high waiting time (Figure 6.8 shows that the total idle time of at least one process is 125 out of 334 seconds). Only during the 'middle' iterations are all the processors active.

Figures 6.9(a) and (b) show the computation and communication (respec-

tively) breakdown by processor over the entire execution of the program. Unlike the call-graph profiles that calculate the sum of the maximum computation occurring in each processor for each superstep in the program, the pie charts show the maximum of the sum of computation times. As can be seen from the charts, LU is *globally* balanced in terms of computation and communication, but the call-graphs show that this is not exploited by the BSP implementation. The difference between the largest segments in the computation (229 seconds) and communication (35 seconds) pie charts, and the actual computation and communication rates observed of 281 and 53 seconds, illustrates this skew. If an implementation of MPI could exploit all the protocol techniques described here in the *BSPlib*/NIC messaging system, LU would require only $229+35 = 264$ seconds to execute. This equates to a mega-flop rate of 79.8 mega-flops/second/process. This figure is closer to that achieved by the Origin 2000.

6.2 Comparison with other NIC protocols

This section compares the work presented here with other low-latency and high-bandwidth network implementations for parallel or distributed computing in cluster environments. The low-latency and high-bandwidth implementation described in this chapter can be considered both as a vehicle and as a consequence of the work described in this thesis: a vehicle because clusters are desirable for parallel applications, and a consequence because many of the novel design features depend on properties of the BSP environment. There exist protocols whose hardware requirements are as modest, whose bandwidth is as high, or whose reliability is as great as the *BSPlib*/NIC prototype cluster's, but there are very few that achieve all three simultaneously. However, all three aspects are critical to building clusters that scale and can be used to execute applications.

It should be clear that a major benefit of BSP and the *BSPlib* implementation is that it allows an unstructured set of communication requests to be grouped and structured into actual communications in an efficient manner. Obviously, the amount of computation to derive these schedules must be small. In the implementation, the options that are considered to achieve this 'good' BSP behaviour include packing individual requests, pacing the communication, and ordering the point-to-point despatch of messages. By allowing unstructured requests, and a corresponding semantics that asserts that the point of request for a communication is the point of scheduling communication, it is difficult to schedule the actual communication in a more efficient manner. Nevertheless, techniques that deliver low-latency and efficient bandwidth utilisation in more standard settings are still prerequisites to the work described here and a comparison is useful.

The early approaches to low-latency and high-bandwidth communication recognised the redundancy in the protocol stack and hence the necessity of simplifying it, eliminating buffer copies by integration of kernel-space with user-space buffer management, and collapsing a number of network layers into one. This is the approach used in the protocol of Brustolini and Bershad [13], based on ATM, which achieves a reliable low-latency, high-bandwidth close to that which is allowed by the hardware. However, in their implementation the overhead of sending must be quite high as the sending process blocks until the message is placed on the network.

An approach that bypasses the need for buffer management is the Active Messages of von Eicken *et al.* [124]. In the active message scheme, messages contain the address of a routine in a receiving process which handles the message by sending a response and/or populating the data structures of the computation with the data in the payload. A prototype implementation for a SparcStation over ATM is described by von Eicken *et al.* [123] in which message delivery is reliable. Because the message handler routine runs in the user's address space, there has to be some mechanism to make sure that the receiving process is scheduled when a message arrives. In general, this requires some kernel changes. The prototype for the SparcStation solves this scheduling problem by having the receiving process poll for messages. Although active messages are intended to be one-sided, this polling activity is a compromise between two-sided and one-sided communication. This is not really an application programming issue as the intention is that the interface be used within a library or in code generated by a parallel-language compiler.

BSP communication is also one-sided. Apart from the obvious programming benefits of not having to match send and receive requests, as long as a reasonable number of buffers are deployed there is no necessity to schedule the distributed processes involved in a computation in synchrony, as message disposal is detached from message receipt in *BSPlib*/NIC. There appears to be no detrimental effect from this loose coordination of the processes, as very high percentage bandwidths are being realized. This is attributed to the number of buffers made available and the fact that data can be packed to fill them.

Active Messages have also been used at various other levels in implementing transport mechanisms: GAMMA [21] is an implementation of Active Messages over Fast Ethernet and achieves an impressive $12.7\mu s$ one-way latency and a full frame bandwidth of approximately 85% of the medium. This bandwidth rises to an asymptotic bandwidth of 98% of the medium for two processors sending frames back-to-back. However, the assumption is that the medium is reliable and that frames arrive in order. A later improvement in which a credit based flow control mechanism introduced reliability on

similar configurations is described by Chiola and Ciaccio [20]. In this implementation the one-way latency rises to $14\mu s$ and the asymptotic bandwidth drops slightly to 96% of the medium. A port of the `mpich` MPI implementation (described in the same paper) achieves the same bandwidth and a one-way latency of $17.7\mu s$.

The sender based protocols described by Swanson and Stoller [115] provide a restricted service compared to Active Messages. This restriction allows a simpler implementation at the receiver and is more efficient than the general receipt of messages in a sender based protocol. This is because the handler in Active Messages is user-specified, whereas in the sender based protocols, the user specifies only the buffer address. Hence, the protocol provides a one-sided direct remote memory access (DRMA) write primitive. Depending on the implementation, at least one less copy of the data is needed as the user-specified buffer could be used instead of a network buffer (but this depends on the implementation). As for Active Messages, it is the sender who specifies the user-level component that is to receive the message, in the form of a buffer address. The problem of this address in the send based protocol packet is the same as the handler address in the Active Message's packet (i.e. local, and possibly dynamic, information has to be managed globally). In the implementation, it is the receiving entity that verifies that the specified address refers to a valid user buffer (by bounds checking the address against known user buffers).

Table 6.7 provides a summary of the performances of various research groups' low-latency and high-bandwidth clusters. The U/Net protocol [126] has similar properties to *BSPlib*/NIC as they both use fast Ethernet networks, and use similar techniques for allocating shared user-kernel memory buffers. As can be seen from the results, the bandwidth and latency of the two are similar. Pupa [122] is also a low-latency communication system over Fast Ethernet. It concentrates on buffer management and provides a reliable transport protocol but does not make assumptions about the network except that packets arrive in order. A missing packet triggers the receiver to initiate recovery. Thus there is an implicit assumption that there is only one route between a pair of nodes. Pupa achieves a one-way latency of $198\mu s$ and achieves a bandwidth of 62Mbps for very large messages (10000 bytes) but achieves less than 60% efficiency for full frame-sized messages.

There are a number of protocols operating over Myrinet [9], for example BIP [100]. BIP (Basic Interface for Parallelism) has similar design objectives to Pupa: to support a message-passing parallel-computing environment (there is an MPI implementation for BIP). No error recovery is supported, but each packet receives a sequence number and contains a checksum, and hence errors can be detected. BIP achieves a good one-way latency of $4.3\mu s$ and an asymptotic bandwidth of 1008Mbps (96% of the available 1056Mbps).

Table 6.7: Half roundtrip latency (μs) and Link Bandwidth (Mbps)

Group	Name	Network	μs	Mbps	Reliable
Genova[a]	GAMMA (Active Messages)	100Mbps hub	13	98	×
Genova[b]	GAMMA (Flow Control)	100Mbps hub	14	96	√
Oxford	PII-BSPlib-NIC-100mbit-wire	100Mbps wire	29	91	√
Cornell[c]	U-Net/FE (P133, hub)	100Mbps hub	30	97	×
Cornell[c]	U-Net/FE (P133, switch)	100Mbps switch	40	97	×
Oxford	PII-BSPlib-NIC-100mbit-2916XL	100Mbps switch	46	91	√
Oxford	PII-BSPlib-TCP-100mbit-2916XL	100Mbps switch	103	70	√
Oxford	PII-BSPlib-UDP-100mbit-2916XL	100Mbps switch	105	79	√
ANL[d]	PII-MPI-ch_p4-100mbit-2916XL	100Mbps switch	147	78	√
SUNY-SB[e]	Pupa	100Mbps switch	198	62	√
Utah[f]	Sender Based Protocols	FDDI ring	22	82	√
	FDDI (theoretical peak)[g]	FDDI ring	510	100	√
NASA[h]	MPI/davinci cluster FDDI	FDDI ring	818	73	√
Cornell[i]	Active Messages (write, SS20)	ATM switch	22	44	√
Cornell[i]	Active Messages (read, SS20)	ATM switch	32	44	√
Cornell[j]	U-Net/ATM (P133)	ATM switch	42	120	×
CMU[k]	Simple Protocol Processing	ATM switch	75	48	√
NASA[l]	MPI/davinci cluster Fore ATM	ATM switch	1005	98	?
NASA[l]	MPI/davinci cluster SGI ATM	ATM switch	1262	85	?
Lyon[m]	BIP	Myrinet	4	1008	×
Princeton[n]	VMMC (Myrinet 2ndG, P166)	Myrinet	10	867	?
UCB[o]	VIA	Myrinet	25	230	√
ETHZ[p]	The Swiss-Tx Project	Swiss-Tx	5	368	?
Princeton[q]	VMMC (SHRIMP)	Intel Paragon	5	184	?
SGI	O2K-MPI-SGI-700mbit-ccnuma	CC-NUMA	17	325	√
Karlsruhe[r]	PULC (port-M PVM)	ParaStation	27	92	√
IBM[s]	SP2 (theoretical peak)	Vulcan switch	40	320	√
IBM	SP2-MPL-IBM-320mbit-vulcan	Vulcan switch	55	173	√
IBM	SP2-MPI-IBM-320mbit-vulcan	Vulcan switch	67	173	√
	Ethernet (theoretical peak)[s]	10Mbps hub	465	10	√
NASA[l]	MPI/davinci cluster Ethernet	10Mbps hub	900	8	?
NASA[l]	MPI/davinci cluster HIPPI	HIPPI	851	265	√

[a]Ciacco [21]
[b]Chiola and Ciaccio [20]
[c]Welsh *et al.* [126]
[d]Gropp and Lusk [53]
[e]Verma and Chiueh [122]
[f]Swanson and Stoller [115]
[g]Hennessy and Patterson [57]
[h]National Aeronautics and Space Administration [95]
[i]von Eicken *et al.* [123]
[j]Brustolini and Bershad [13]
[k]Brustolini and Bershad [13]
[l]National Aeronautics and Space Administration [95]
[m]Prylli and Tourancheau [100]
[n]Dubnicki *et al.* [35]
[o]Buonadonna *et al.* [15]
[p]Brauss *et al.* [11]
[q]Dubnicki *et al.* [36]
[r]Blum *et al.* [8]
[s]Hennessy and Patterson [57]

The performance of BIP under the NAS benchmarks has been investigated by Geoffrey *et al.* [43].

PULC [8] is a user-level communication library for systems communicating on the ParaStation. Alpha 31164 and Pentium 166MHz processor implementations have been produced. The Pentium implementation also used the Linux 2.0 kernel. PULC supports a socket interface as well as an interface called port-M upon which a PVM implementation has been built. PULC allows more than one process to have access to the device and there can be many parallel jobs in the cluster executing at the same time. When messages arrive they are demultiplexed and each application code receives its data when it executes a receive call. Hence, at this level, the communication is not one-sided. As a high-level library, the port-M version of PVM achieves a latency of $27.2\mu s$ and a bandwidth of 92Mbps when two processors are communicating.

The Beowulf project [112] tackles a slightly different problem to that of reducing latency, but improves bandwidth by enriching the interconnect. They provide programming libraries such as MPI, PVM, and BSP to deliver a low cost parallel computing environment.

The Swiss-Tx effort is an ambitious project whose eventual aim is to build a machine of 504 processors using custom devices for communication. A fast communication library is supported as well as the necessary abstract device interface to support MPI. Communication is based on a one-sided remote memory write operation. Brauss *et al.* [11] present performance results for an early prototype. The peak bandwidth achievable by the communication hardware is 384Mbps, but messages of approximately 1000 bytes achieve only 50% of this. Very large messages of over 10000 bytes achieve 368Mbps for simple benchmarks, whilst an impressive one-way latency of $5\mu s$ is recorded.

The suitability of some of the network media mentioned in Table 6.7 for high-performance parallel computing may not be immediately obvious. Certainly, a single wire between two processors is not at all scalable and cannot be taken seriously (such configurations are included only to illustrate the contribution of the switch to the latency). Also, all of the bus based schemes are not scalable; this includes the Fast Ethernet implementations that use an Ethernet hub, since the hub merely acts as an an extension of the bus and its bandwidth is divided amongst the communicating processors. Hub implementations imply the use of the CSMA/CD protocol, which makes no promise for reliable delivery, and hence this must be built into some higher protocol. For scalability purposes, the FDDI ring can also be considered a bus because the bandwidth is diluted amongst the communication processes. However, a single FDDI ring can be used in such a way that the higher-level protocols do not have to provide recovery.

	Message size	Latin square	Destination order	Improvement factor
Cluster(NIC)	16k words	39468 μs	76597 μs	1.9
	32k words	78683 μs	14043686 μs	178

Table 6.8: Time for an 8-processor total exchange (μs).

Reliability is not orthogonal to performance, and it cannot be designed in later. The performance benefit of scheduling communication to avoid contention at destinations has already been commented on. Performance data are shown in Table 6.8 for *BSPlib*/NIC, with destination scheduling turned on and off. The table shows that poor destination scheduling can decrease performance by a factor of about 2 for a total exchange in which each processor sends a message of 16k words to every other processor. However, as messages get bigger, the performance penalty for a bad choice of destination schedule is a reduction in performance of a factor of 178.

Clearly, destination scheduling is a good thing. But the more general point is that the poor performance is caused by buffer overflows, triggering resend requests, which in turn require more buffers. This can happen to any protocol, but a poor design for the recovery mechanism makes it much worse. To reiterate, the factor of 178 is small compared to what other protocols might manage, because *BSPlib*/NIC implementation is careful about the cost of recovery.

Treating error recovery as optional, based on the underlying reliability of a point-to-point benchmark is a false argument. Unless some back-pressure notification is made available to the sender, it will always be possible to overwhelm a component of the switch in the presence of unstructured communication patterns. Switches are starting to appear which address this problem, either by augmenting the protocol to send a flow-control packet back along the link, or by holding the cable busy so that the NIC CSMA/CD engine will hold off transmitting the next packet. However, this does not solve edge problems when multiple links are used.

Communication primitives that require the receiver to poll on a device, for example variants of active messages such as GAMMA [21] and U/NET [123], are not one-sided and the measurement of latency may hide a potentially large loss in CPU cycles. This loss can arise as a result of the sender running ahead of the receiver. If the send is truly synchronous, then the sender waits for the corresponding receive; if the sender continues, it may re-invoke a send later and cause an overrun (or an overrun due to another processor sending data to the same destination). If the receiver runs ahead, then time will be wasted polling for the incoming message. Of course depending on the

model, such wastage of CPU time may be inevitable, but it does not occur for well-designed BSP computations. Having tightly-coupled communication is difficult in the loosely-coupled environment of clusters and a buffer-slackness approach to accommodate this loose coupling has been adopted.

The switched solutions are the only ones that provide scalability (Myrinet, ATM and switched Fast Ethernet). Of the Fast Ethernet solutions, results presented here are best in terms of scalability, reliability, latency and bandwidth, and are competitive with the much more expensive Myrinet solutions.

6.3 Conclusion

The performance of the protocol stack described in Chapter 5 has been discussed, demonstrating that the aggregated design decisions of the recovery protocol (hole-filling); using global knowledge of expected packet traffic; implicit acknowledgement; and buffer management result in a transport protocol that performs well, especially as it provides reliability, and requires only inexpensive hardware interfaces.

Also explored was the issue of whether protocols should execute in user or in kernel space. The trend in high-performance protocols for clusters has been to increase the amount of protocol work that takes place in user space. The results suggest that this may not be a good thing to do because of the impact on computation performance. Also, executing in kernel space need not increase latency, especially in the presence of large frames created by delaying and packing messages analogous to the exploitation of parallel slackness.

The performance behaviour of this new protocol stack has been documented, using low-level benchmarks to explore single-link behaviour, and the NAS parallel benchmarks to demonstrate that the single-link performance scales to applications.

In Chapter 7, the implementations of the protocol in *BSPlib*/UDP and *BSPlib*/NIC will be analysed in terms of the edge effects and the behaviour of the protocol as the number of processors increases. It will be shown in Section 7.2 that the point-to-point performance of the protocol, although important, is not sufficient to ensure proper performance when scaling to a larger number of processors. The two implementations of the protocol for $p = 2$ achieve approximately the same bandwidth, but the global bandwidth achieved by the two implementations varies greatly as p increases. Neither is it sufficient that the mean value of the point-to-point case exhibits a good mean independent of the variance. Finally, the analysis shows that *BSPlib*/-UDP implementation demonstrates a strong interference among the various

circuits. On the other hand, the observed degrading of the time to realize the fixed size h-relation as p increases in the *BSPlib*/NIC implementation is largely explained by the observed variance in the protocol. In terms of scalability, this would indicate that as p increases the independent $p(p-1)/2$ circuits do not interfere with each other and the remaining impedance to scalability is explained in terms of this observed variance.

Chapter 7

Predictability and the Limits of Scalability

In high-performance computing the effective performance of an architecture is never as good as its specifications and simple benchmarks would suggest. A partial explanation for this phenomenon in the domain of communication is given in this chapter. It will be shown that the mean performance of an ensemble depends not only upon the mean performances of the components, but also *on the standard deviations of these performances.* In other words, it is important to provide high throughput (or alternatively low latency) *on average*, but it is also critical to make the standard deviation of these measures as small as possible.

It is not clear to what extent manufacturers of high-performance interconnects are aware of this point. It is certainly possible to point to design decisions that suggest that it has not been widely appreciated. On the other hand, variability in communication performance depends on a number of features beyond their control; for example, design of protocol stacks, choices of buffer size, operating system sophistication, network interfaces and the systems interaction with the workload.

In systems based on point-to-point messaging the issue of variability in delivery times due to communication subsystem design cannot be discussed sensibly without considering the nature of the usage of the subsystem by all other communicating processors. It has been simply assumed that the only important phenomenon is the first-order delay caused by the other messages that each message encounters *en route.* There is no way to capture second-order phenomena since no sequence of events in the communication system

Part of this chapter is based upon the paper "Communication performance optimisation requires minimising variance" by Donaldson *et al.* [29].

is repeatable, and hence there is no straightforward way to talk about the distribution of delays.

In contrast, the BSP cost model treats communication as an aggregate operation of the entire executing architecture, and models the cost of delivery using a single architectural parameter, the permeability, g. A more convenient measure of the communication permeability for the present argument is the term g/s (in units of microseconds per word) as it factors out any dependence on processor speed. Communication properties are bulk properties, and hence accessible for measurement and study. The conclusions drawn in this chapter could have been explored only within a framework in which communication is treated globally. However, they apply in any setting, since the phenomena described affect all forms of communication.

If processor i sends or receives h_i data items during a communication phase, then the BSP cost model asserts that the delivery time for the entire global communication is bounded by $\max\{h_i g/s \mid i \in \langle p \rangle\}$. The value of g/s for a particular parallel computer is obtained by measuring delivery times as a function of applied communication patterns chosen from a standard benchmark. Although the cost model asserts that g/s should remain constant over all applied communication patterns for which $\max\{h_i \mid i \in \langle p \rangle\}$ remains the same, in practice a distribution of values for g/s is obtained.

This is not surprising as the BSP cost model charges the same for a communication pattern in which one processor sends data of size h and a communication pattern in which p processors send data of size h. The total volume of data moved varies between h and ph over this range of patterns. The cost model is implicitly asserting that the dominating cost is the cost of crossing the boundary of the network, rather than the cost of internal transit. This is largely true for today's highly-connected networks.

Figure 7.1 shows distributions of g/s under a uniform randomly applied communication pattern for a randomly-applied load on four widely-different parallel architectures (for a description of the benchmark used to obtain this data see Hill *et al.* [62]). Also shown for each of the machines are the normal and log-normal distributions suggested by the means and standard deviations of the sampled distributions. The figure demonstrates that in practice, the observed distribution of g/s closely approximates a normal distribution and it is only in the distribution of g/s for the 10Mbps Ethernet that the approximation is poor. The poor Ethernet fit may be attributable to the increasingly longer delays that a frame is subject to on each successive collision. As can be seen from the discussion of the media access protocol in Chapter 4, the mean value of this delay is exponential in the number of collisions. That these distributions are approximately normal is reassuring, since it suggests that variation in the effective value of g/s is related to small random events within the communication subsystem, rather than

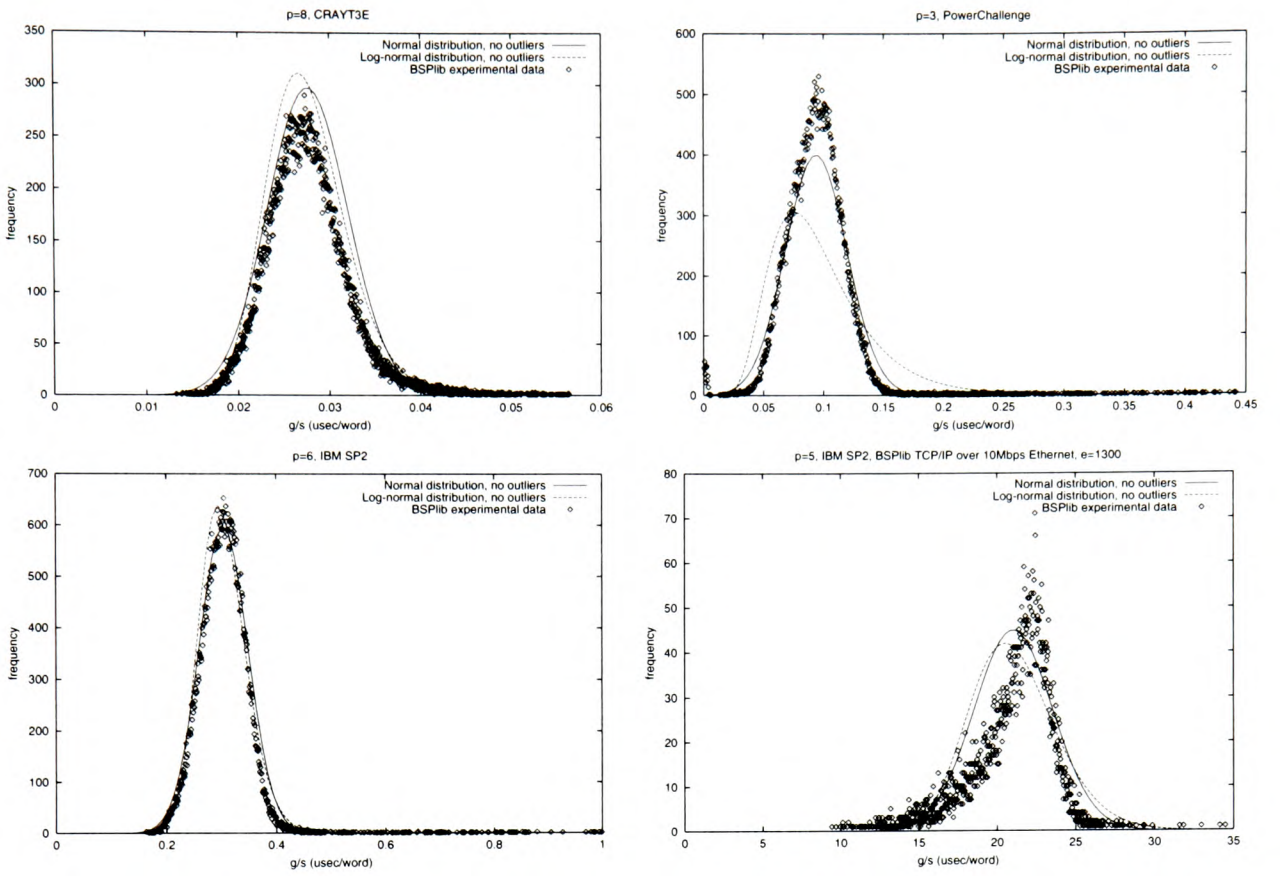


Figure 7.1: Distributions of g/s for the Cray T3E, SGI PowerChallenge, IBM SP2, and a RS6000 cluster connected by dedicated 10Mbps Ethernet

systematic, unmodelled effects. The fact that the distribution of g/s can be approximated by such distributions means that the stability of an architecture with respect to communication can be captured by the standard deviation of this distribution (and the value of g/s by its mean). The normal distribution is used in the sequel because of its proximity, but can be substituted for the appropriate distribution or by the frequencies extracted from experimentation as is the case in Section 7.2.

This standard deviation is a measure of the quality of the communication system in the sense of how accurately the BSP cost model will describe it. All of the BSP parameters capture important properties of architectures independently of whether BSP is used as the programming model. This is because the BSP parameters measure the performance of systems under heavy, but not unreasonable, load. When a system has good BSP parameters, then its performance under less demanding loads is also likely to be good. For example, the g/s parameter measures the ability of a system to communicate continuous traffic *without* any exploitable locality. Hence it stresses the communication system maximally. Similarly, the l parameter measures the ability of a system to reach a globally-consistent state.

The standard deviation of g/s is similarly a useful datum, independently

of BSP. If it is small, the system will communicate with predictable cost. However, more importantly, this parameter predicts how well an architecture will scale.

Section 7.1 shows how the the standard deviation experienced within the links of an interconnect affects the performance of the interconnect. Section 7.2 shows, independent of the topology and hence the number of links, how the edge effects impact the performance of the communication. By 'edge' is meant the processing nodes including runtime libraries and protocol stacks. Whilst the analysis uses the BSP model, the effects are generally applicable. In the first instance, the superstep structure is implicitly assumed by the use of the BSP parameter g as it pertains to the cost of realizing h-relations. The choice of topology, a fully-connected interconnect, coincides with the worst possible effects on communication performance even though the interconnect is idealistically the best choice. In the second case, the analysis is based solely on the BSP cost model and assumes a behaviour borne out by experimentation as documented by, for example, Juurlink and Wijshoff [77] and Hill *et al.* [62].

7.1 Link Effects of Variance in BSP Computation

Consider a connection between two processors. Its performance can be described by its g/s value and, as mentioned above, this value is typically approximated by a normal distribution. Let the mean and standard deviation of this distribution be μ and σ. Then the corresponding probability density function $f(x)$ and probability distribution function $F(y) = P\{g/s \leq y\}$ are given by:

$$\begin{aligned} f(x) &= \frac{\mathrm{e}^{-(x-\mu)^2/2\sigma^2}}{\sigma\sqrt{2\pi}} \\ F(y) &= \int_{-\infty}^{y} f(x)\mathrm{d}x \end{aligned}$$

Suppose that this single connection were used to build a larger architecture (with the same basic properties, i.e. protocols etc.). For example, a four-processor system that uses a fully-connected communications topology would require $p(p-1)/2 = 6$ independent links. The probability density function for g_p/s of a system containing p processors will be the joint probabilities of each of the independent links. Informally, what is being asserted here is that the timing of the communication ends when the last bit has arrived at its destination processor and that all the links are independent. This tends to skew the distribution towards increasingly larger values of g/s

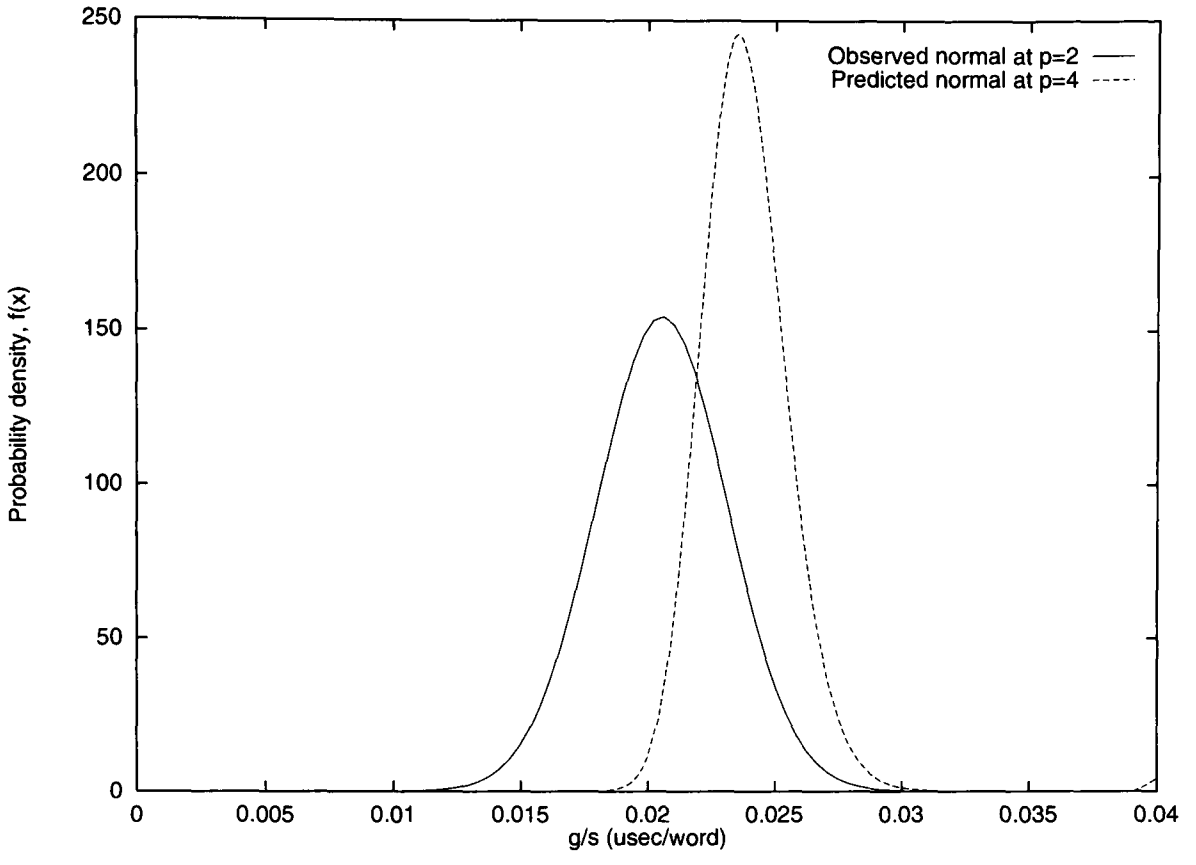


Figure 7.2: Probability density function at $p = 4$ (6 links) predicted from $p = 2$ experimental data for the Cray T3E

as p increases. In particular, the mean value of g/s increases, although the standard deviation decreases as shown in Figure 7.2.

Since the links are independent, for particular p, μ and σ the probability distribution of g_p/s, $F_p(y) = P\{g_p/s \leq y\}$ can be calculated, based on a random variable g/s (in effect g_2/s) for the links, with probability distribution of $F(y) = P\{g/s \leq y\}$:

$$\begin{aligned} F_p(y) &= (P\{g/s \leq y\})^{p(p-1)/2} \\ &= F(y)^{p(p-1)/2} \end{aligned}$$

The corresponding density can be obtained by taking the derivative with respect to y. The expected value, g_p/s, can then be taken from this density function:

$$g_p/s = \int_{-\infty}^{\infty} y \cdot \frac{\mathrm{d}}{\mathrm{d}y} [F_p(y)] \, \mathrm{d}y \tag{7.1}$$

Equation (7.1) gives the approximate predicted value for the scaled machine in terms of the distribution of the original g/s. Therefore, by considering that the distribution of g/s is normal with parameters μ and σ an expression for g_p/s in terms of the parameters of the original distribution is obtained.

This shows how g_p/s is sensitive to the original distribution of g/s, in particular to the original standard deviation. Equation (7.1) is used in Figure 7.5 to obtain predicted values of g/s for the Cray T3E under *BSPlib* for $p = 4$ as the standard deviation varies. The predicted values are based on measurements of g/s for $p = 2$ and includes an experimental value of g/s for $p = 4$ shown at a point corresponding to the sampled standard deviation for $p = 2$.

This analysis is based on a fully-connected topology, which is too expensive to be used in scalable parallel computers. However, any sparser interconnection topology will have an effective value of g/s larger (i.e. worse) than that of a fully-connected network, so this result defines an upper bound on the throughput of communication performance. The critical point is that a large value for the standard deviation of g/s for a small ensemble means a large value for the mean of any larger ensemble. Thus the design of high performance scalable interconnects requires small mean delivery times *and* small variation in those times.

7.2 Edge Effects of Variance in BSP Computation

The argument above illustrates the effect of variance on the scalability of an interconnect. In this section the related problem of different sized computations is considered in a BSP sense on a given fixed architecture. For example, the four processor Cray T3E used in Figure 7.2 is not a four processor computer built from six links with the same characteristics as the link used in a two processor machine. Both the two processor case and the four processor case are measurements taken from a particular benchmark on a 32 processor Cray T3E. The Cray T3E interconnect, a 3D Torus, has no assignment of four processors in a fully-connected configuration even though each node has degree six.

This section presents a similar analysis from a BSP perspective and not the idealised fully-connected interconnect assumed in the previous section. As mentioned in Chapter 2, the BSP model encourages balanced computation and balanced communication. If one assumes this then $w = w_i$ and $h = h_i$ for $i \in \langle p \rangle$. Also, as observed in the beginning of this chapter, the model implies that the cost of communication is incurred at the edges of the interconnect and that internal congestion in the interconnect is negligible. If this is also true of the contribution to the variance, then the cost of crossing a particular boundary i is the random variable hg_i/s. Furthermore, this random variable is independent of the costs incurred crossing each of the other boundaries.

From the BSP cost model the cost of such a superstep is

$$C \quad = \quad \max\{w_i \mid i \in \langle p \rangle\} + g \max\{h_i \mid i \in \langle p \rangle\} + l.$$

For the experiments in this section, g/s is benchmarked in an application which repeatedly performs a total exchange with a fixed and full h-relation and in which no local computation is performed. Hence what is actually being measured is

$$C' \quad = \quad h' \max\{g'_i/s \mid i \in \langle p \rangle\} + l/s.$$

By setting $h' = 0$ the value of l/s can be estimated and for non-zero values of h' a sample of $g_{h'}/s$ can be obtained as

$$g_{h'}/s \quad = \quad C'/h'.$$

Thus $g_{h'}/s = \max\{g'_i/s \mid i \in \langle p \rangle\}$ where $g'_i/s = g'_e/s$ are identical independent random variables. If $g_{h'}/s \leq g/s$, then $g'_i/s \leq g/s$ for $i \in \langle p \rangle$ and hence $P\{g_{h'}/s \leq g/s\} = \prod_{i=0}^{p-1} P\{g'_e/s \leq g/s\}$. That is

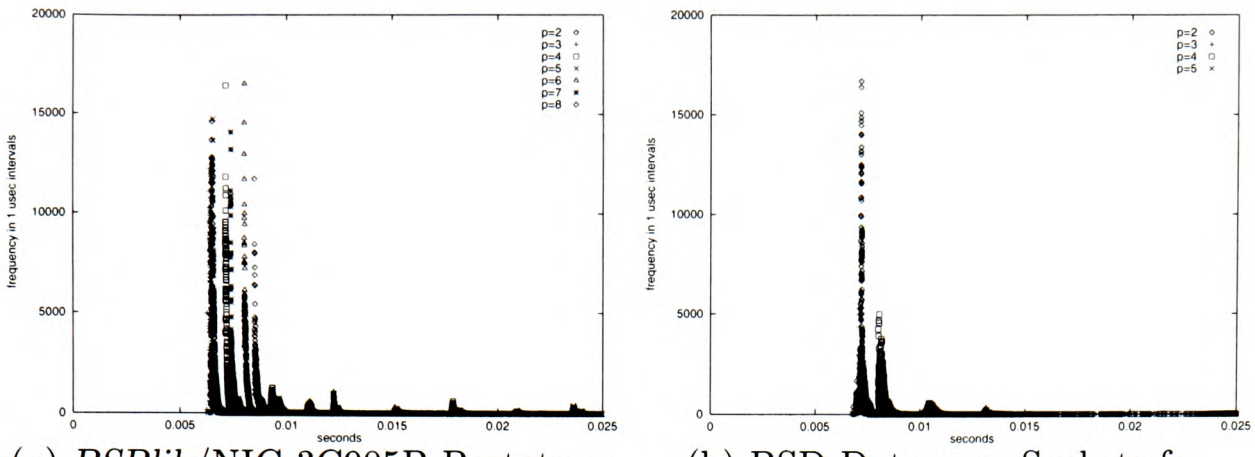
$$P\{g_{h'}/s \leq g/s\} \quad = \quad P\{g'_e/s \leq g/s\}^p. \tag{7.2}$$

This is very similar to the probability distribution function of the scaled architecture in the previous section. The only difference is the number of joint probabilities which is reduced from $p(p-1)/2$ to p.

The assumption in this case is slightly stronger than that of the previous section in that the model itself implies that these effects are edge effects and one does not have to assume anything as strong as a fully-connected interconnect. Equation (7.2) asserts that there is an *edge* distribution, $P\{g'_e/s \leq g/s\}$, that can be computed from the sampled distribution of communication throughput on a p processor BSP computer and that this distribution should be independent of the number of processors, p.

In practice, some dependence on p is expected, either because there are some internal congestion effects or because of the overheads in maintaining, for example, $p-1$ circuits or links at each of the nodes. In any case, the experiments reveal a class of distributions indexed by p. The closer the distributions the more 'BSP-like' the implementation in that the edge effects are dependent on the size of the h-relation and not the number of parallel partners involved in the communication. Also, similar distributions imply that the interconnect does not suffer from internal congestion and that it is scalable (at least over the sampled range of p) in the traditional sense, which ignores the compositionality issues highlighted by this consideration of the variance.

BSPlib implemented on parallel, SMP and distributed computers provides an approximation to a BSP computer which, in general, makes do with the



(a) *BSPlib*/NIC 3C905B Prototype for $p = 2, \ldots, 8$.

(b) BSD Datagram Sockets for $p = 2, \ldots, 5$.

Figure 7.3: Edge distributions computed from sampled fixed 16384 32-bit word h-relations timed over a total exchange

'transport' layer provided. In doing so, and because of the nature of communication in BSP computation, it is generally possible to infer and take into account the global state when using the underlying transport and interconnect. Indeed, this is precisely what is done in the 'global' transport protocol for clusters described in this thesis. Examples of transport mechanisms are the SYSV shared memory library on SMPs [49], Cray SHMEM library on the 3D-Torus interconnect of the Cray T3E [6], MPL on the Vulcan in the IBM SP2 [113], and BSD Sockets [88] on distributed computers.

Part of that motivation for this global transport protocol can be observed in the computed distributions $P\{g'_e/s \leq g/s\}$ for *BSPlib*/NIC shown in Figure 7.3(a), derived from sample distributions for the time to realize a total exchange communication pattern with a fixed h-relation size of 16384 32-bit words and for $p = 2, \ldots, 8$. Figure 7.3(b) shows the corresponding experiment using *BSPlib*/UDP over BSD Datagram Sockets. These are implementations of the same protocol over the same hardware configuration except that the distributions corresponding to $p = 6, 7, 8$ have been omitted from Figure 7.3(b) as they do not show on the chosen scale.

Figure 7.4 shows the mean time in seconds to realize an h-relation size of 16384 32-bit words comparing the transport layers for $p = 2, \ldots, 8$. The figure shows that for $p = 2$ there is very little difference between the throughput achieved for the two protocols. From Figure 7.3 it appears that for $p = 2$ the distribution for *BSPlib*/UDP is better than the prototype *BSPlib*/NIC cluster protocol. Yet it is clear from both Figures 7.3 and 7.4 that for $p > 2$ this is not the case. This demonstrates that optimising protocols for the point-to-point case alone, as is typically the case for TCP/IP and UDP/IP implementations, is generally not sufficient to achieve global optimisation.

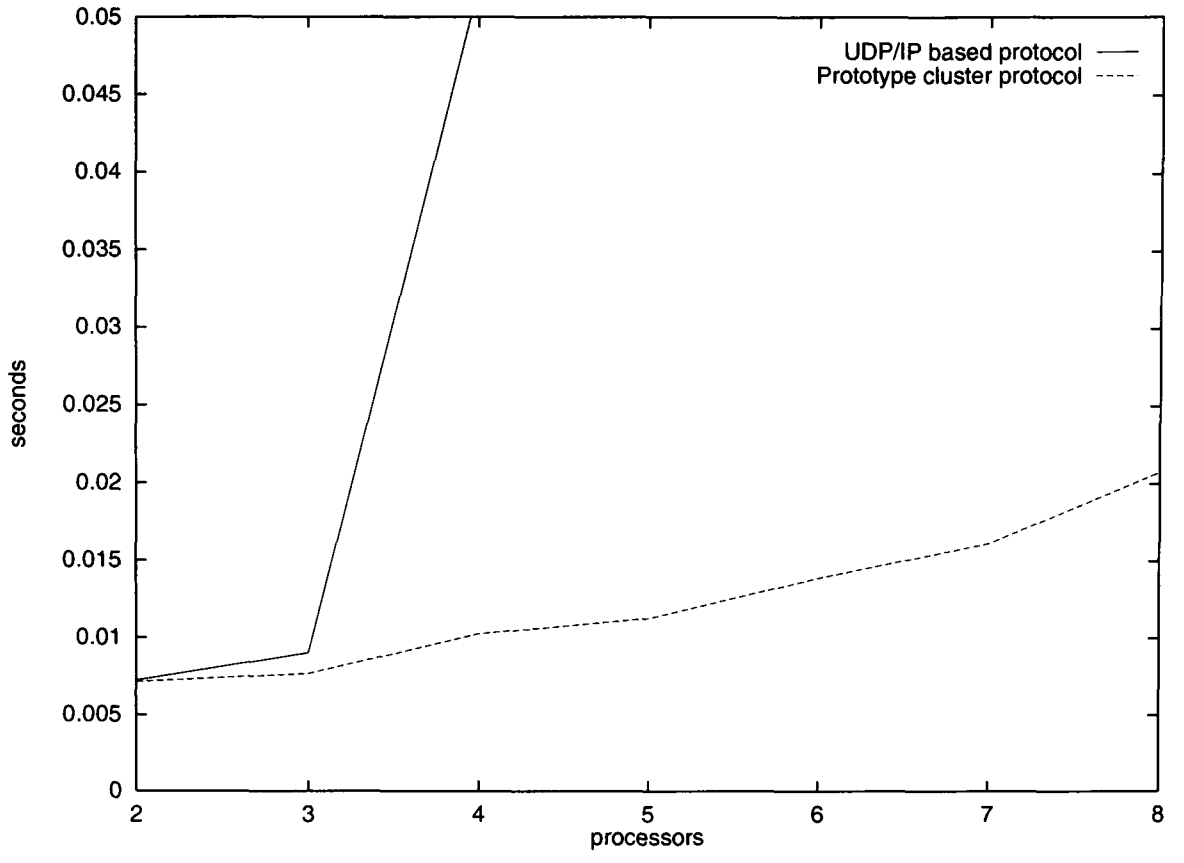


Figure 7.4: Time to realize an h-relation of size 16384 32-bit words for $p = 2, \ldots, 8$

7.3 Variance as an Architecture Parameter

This result is primarily of interest because it allows us to derive bounds on the g parameter or g/s, and hence on the scalability of communication in general. It may also be used to predict applications that do not scale well, because their particular traffic patterns induce high variance in the underlying communication network.

For example, Figure 7.5 shows how varying the standard deviation on delivery times for a 2-processor Cray T3E affects the performance of the network of a 4-processor system, using Equation (7.1) under the assumptions of a normal distribution of delivery times and a fully-connected topology. It can be clearly seen from the figure that increasing the standard deviation increases the mean. This contradicts previous theoretical analysis, which assumes that if fully-connected graphs were feasible, then they are necessarily scalable because there is no internal contention, and the network has constant diameter. This work shows that variance limits scalability or compositionality even in such ideal interconnects. Also shown in Figure 7.5 is the sampled mean for the 4-processor case taken from the experiments described in Hill *et al.* [62]. The graph shows that the effects modelled by the distribution are largely responsible for the observed mean in this 4-processor

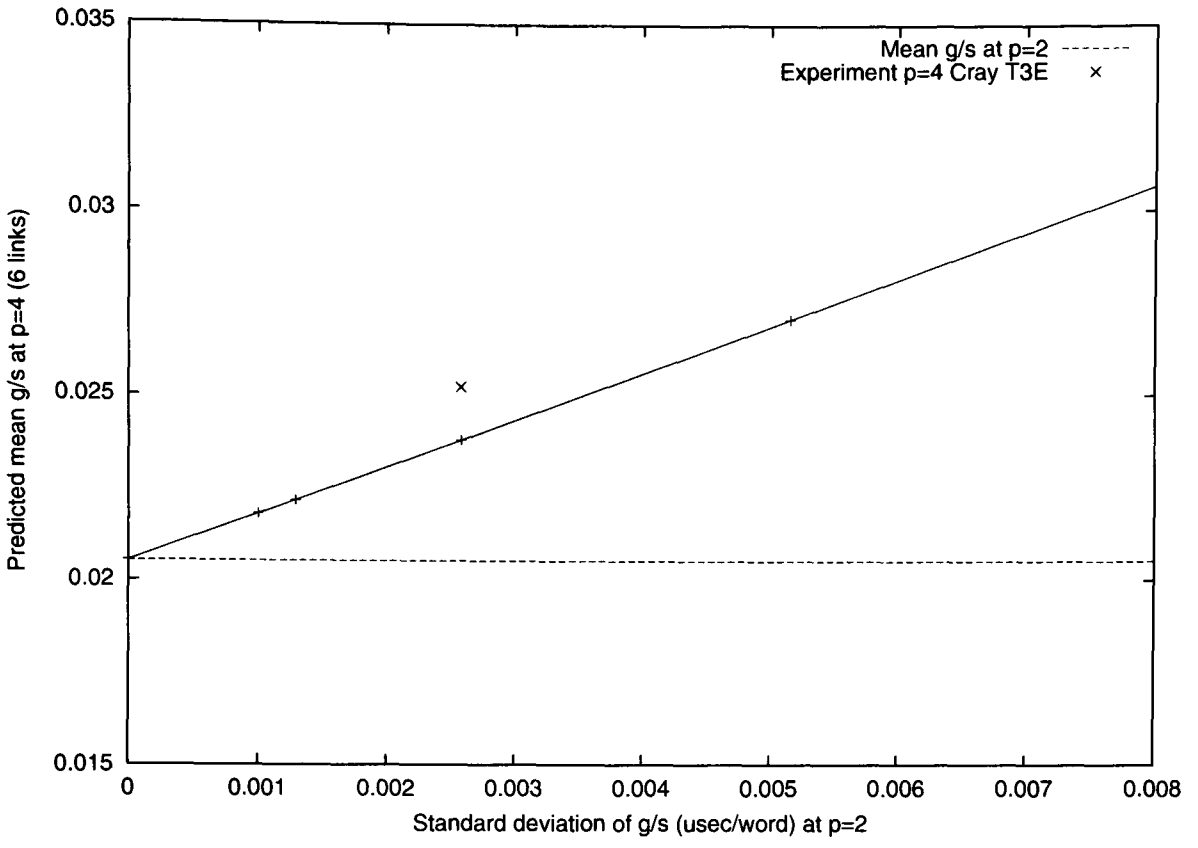


Figure 7.5: Predicted mean at $p = 4$ as a function of standard deviation and mean at $p = 2$

case.

Clearly, reducing the variation in delivery times for communication is important if sustained performance of scalable systems is to match the expectations set by single-link peak performance. This requires manufacturers to explicitly address, not just peak performance achievable on links, but also the standard deviation of such performance. However, this is not enough since the effective performance across even a single link depends on operating system design and the design of the runtime system supporting communication. The kind of optimisations needed to reduce variation in delivery times are not possible within point-to-point messaging systems, since they have no global knowledge on which to base them. A thesis of this dissertation is that such global knowledge is readily available in the BSP model assumed here and can be exploited to improve communication performance. *BSPlib* expends a great deal of effort both to minimise the mean of the delivery rate, and its standard deviation. It uses techniques such as message packing to reduce start-up overheads, and delivery scheduling to reduce contention at processors [67]. In switched LAN and shared-media interconnects such as Ethernet where the protocol described in Chapter 5 is deployed, it uses the following techniques to reduce variations in communications performance: sophisticated flow control, pacing message injection into the interconnect,

early recovery, resource based rather than timer based demand acknowledgements (to reduce non-piggy backed acknowledgements which are further reduced with implicit acknowledgements), and large buffer-slackness.

7.4 Causes of Variance

Previous work in the field has investigated the causes of variance in communications systems as a whole, usually concentrating on the protocols used in point-to-point communication. Problems causing variance at the processor-network interface have been studied by Mraz [94]. He identifies how activities such as interrupts, scheduling, daemon activity and priority settings affect the variance of point-to-point communication between workstations. To minimise this variance his solution is to tightly control the interaction between the operating system and the communications subsystem by a suitable assignment of process priorities and deferring the processing of long-running interrupt handlers.

The general network problem of quality of service in which pairs of applications communicate over a WAN/LAN/SAN with arbitrary types of data (such as video, audio, or data) has been studied by Chien and Kim [18]. In their work, quality of service is defined in terms of guarantees on maximum bandwidths and minimum latencies. Consequently, global scheduling is difficult in the absence of tight coupling. In contrast, the *BSPlib* setting defines quality of service in terms of the mean performance and its variance. Also, scheduling problems in this framework are more tractable as the computation consists of a closed collection of processors all executing the same SPMD BSP program. The global scheduling of BSP communication is discussed in Hill and Skillicorn [67] and Hill *et al.* [62].

In the implementations of *BSPlib* on networks of workstations and clusters (Chapters 4, 5 and 6), variance is addressed at a number of levels. The use of the local network by the computation at a global level is taken into account. This is particularly important in networks which display a peak in their successful versus offered load curves such as Ethernet or networks where internal and edge congestion can occur. However, there are a number of operating system effects which also contribute to a large variance. For example, in the UDP/IP implementation message arrivals are asynchronous with respect to the executing process and hence are dependent on the dispatcher and signal delivery. On the Linux 2.0 kernel using Pentium processors, this means that as much as $10ms$ could elapse before arriving messages are processed. Since this implementation of *BSPlib* performs its own error recovery, this means that the start of recovery is also subject to these delays. Significantly contributing to the observed variance, but still

better than the implementation using TCP/IP (details in Chapters 5 and 6; and Donaldson *et al.* [32]).

In the *BSPlib*/NIC cluster implementation the problems encountered in the *BSPlib*/UDP implementation are addressed and a further reduction of the variance was achieved by providing a suitable recovery mechanism in the driver and ensuring sufficient buffering such that the scheduling problem does not result in dropped packets. Thus the approach is three pronged, the number of times recovery is entered and the cost of recovery is reduced, as well as the delay before recovery is entered. As in the UDP/IP case, the usage of the network by the computation as a whole is still taken into account by the mechanism described in Chapter 4.

7.5 Conclusion

Message-passing systems treat each message as independent. Measuring the cost of message-passing programs is notoriously difficult because of the detail required to obtain even fair approximations to observed data. In contrast, the BSP g/s parameter measures the delivery capacity of an architecture in a highly-accurate way. Furthermore, because of BSP's treatment of communication as a bulk activity, it is possible to define a distribution of values for the g/s parameter as a function of applied communication patterns.

Examining such distributions leads to a number of conclusions:

- They are approximately normal, suggesting that no significant property is unmodelled (this is discussed at much greater length in Hill *et al.* [62]).
- The standard deviation of such normal distributions can be regarded as a quality parameter for the underlying architecture.
- The standard deviation of the cost of communication over a single link, or over a cluster, can be used to derive lower bounds for the time per unit transfer of communication of larger ensembles. This suggests that high-performance architectures must concentrate on reducing variability in performance, as well as improving it in absolute terms. This point seems not to be widely appreciated; which is not surprising since the issue can hardly be formulated outside an approach that treats communication globally. Nevertheless, the result is broadly applicable.

Furthermore, one can demonstrate that protocols optimised for point-to-point communication do not scale well in a parallel setting where many circuits have to be maintained. Comparing such a protocol with the pro-

totype cluster protocol shows that a significant improvement is possible by considering the communication as a bulk global activity, and that this is in line with the expectation of the BSP cost model.

Chapter 8

Process Migration and Fault Tolerance in Networks of Workstations

In the BSP model the globally consistent state that is available after the global barrier not only helps when reasoning about parallel programs, but also suggests a programming discipline in which computation (and communication) is balanced across all the processes. As balance is so important to BSP, profiling tools have concentrated upon exposing imbalances to the programmer so that they can be eliminated, for example, the *BSPlib* call-graph tool used in Chapter 6 [65, 64]. However, not all imbalances that arise during program execution are caused by the program. In an environment where processors are not dedicated resources, the BSP computation proceeds at the speed of the slowest processor. This would suggest that the synchronous nature of BSP is a disadvantage compared to the more lax synchronisation regime of message passing systems such as MPI. However, most programs written using collective communications, or scientific applications such as the NAS parallel benchmarks [4] are highly synchronous in nature, and are therefore limited to the performance of the slowest running process. This is the case in either BSP or MPI. Therefore, if a network user logs on to a machine that is part of a BSP job, this may have a significant effect on the entire job. This chapter describes a technique that ensures a p process BSP job continually adapts itself to run on the p least loaded processors in a network consisting of P machines ($p \leq P$).

Dedicated parallel machines can impose a global scheduling policy upon their user community such that parallel jobs do not interfere with each other in

This chapter is based on the paper "Process migration and fault tolerance of *BSPlib* programs running on Networks of Workstations" by Hill *et al.* [60].

a detrimental manner. The environment described is one in which it is not possible to impose some schedule on the user community. The components of the parallel computation are invariably guests on other people's machines and should not impose any restrictions on them for hosting the computation. Furthermore, precisely because of this arrangement, the availability of the nodes and the available resources at these nodes is quite erratic and unpredictable. The philosophy is that in such a situation it is reasonable to expect the parallel job to look after itself.

The steps involved in migrating a BSP job that has been written using the *BSPlib* communications library among a set of machines, is briefly described as are the strategies used in making check-pointing and migration decisions across all machines. The first technical challenge (Section 8.1) describes how the state of a UNIX process is captured and restarted in the same state on another machine of the same type and operating system. The simplicity of the superstep structure of BSP programs provides a convenient point at which local checkpoints capture the global state of the entire BSP computation. This therefore enables process migration and check-pointing to be achieved without any changes to the user's program. Next, a strategy whereby all processes simultaneously decide that a different set of machines would provide a better service is described (Section 8.2). When the BSP job decides that processes should be migrated, all processes perform a global checkpoint, they are then terminated and restarted on the least loaded machines from that checkpoint. Section 8.3 describes a technique for determining the global load of a system. Various algorithms are analysed and compared using Markov theory (see, for example, Kleinrock [82]) and reinforced by simulations and experimental data. Section 8.4 describes an experiment using an industrial electro-magnetic application on a network of workstations. This demonstrates how the scientist or engineer is allowed to concentrate on the application and not on maintaining or worrying about the choice of processors in the network. Section 8.5 describes some related work.

8.1 Check-pointing and restarting single processes

BSPlib provides a simple API for inter-processor communication in the context of the BSP model. This simple interface has been implemented on four classes of machine: (1) distributed memory machines where the implementation uses either proprietary message passing libraries or MPI; (2) Distributed memory machines where the implementation uses primitive one-sided communication, for example the Cray SHMEM library of the T3E; (3) shared memory multi-processors where the implementation uses either proprietary concurrency primitives or System V semaphores; and (4) Networks

of workstations or clusters where the implementation uses TCP/IP, UDP/IP or custom sequenced packet protocols such as that described in Chapter 5. In this chapter the check-pointing of programs running on the network of workstations version of the library, *BSPlib*/UDP or *BSPlib*/TCP, is considered. By choosing to checkpoint at the barrier synchronisation that delimits supersteps, because there is a globally consistent state upon exiting the barrier (where all communication is quiesced), the task of performing a global checkpoint reduces to the task of check-pointing all the processes at the local process level. This is unlike other check-pointing schemes for message passing systems (See Section 8.5).

All that is required to perform a local checkpoint is to save all program data that are active. Unfortunately, because data (i.e. the state) can be arbitrarily dispersed amongst the stack, heap and program text, capturing the state of a running program is not as simple as it would first appear. A relatively straight-forward solution in a UNIX environment is to capture an image of the running process and create an executable which contains the state of the modified data section (including any allocated heap storage) and a copy of the stack. When the check-pointed executable is restarted the original context is restored and all *BSPlib* supporting IPC connections (pipes and sockets) are re-established before the computation is allowed to proceed. All this activity is transparent to the programmer as it is performed as part of the *BSPlib* primitives. Furthermore, by restricting the program to the semantics of *BSPlib*, no program changes are required.

The process of taking a checkpoint involves making a copy of the stack on the heap, saving the current stack pointer and frame pointer registers, and saving any additional state information (for example, on the SPARC the register windows need to be flushed onto the stack before it is saved, and the subroutine return address needs to be saved as it is stored in a register; in contrast, on the Intel X86 architecture, all necessary information is already stored on the stack).

The executable that captures this state information is built using the `unexec` function which is distributed as part of Emacs [110]. The use of `unexec` is similar to its use (or the use of `undump`) in Emacs, LaTeX (which builds executables containing initialised data structures) and Condor which also performs check-pointing [38]. However, the state saving in Condor captures the point of execution and the stack height using the standard C function `setjmp` and restores it with `longjmp` which only guarantees far jumping into activation records already on the stack and within the same process instance. Instead, the concrete semantics of the processor are used to capture the additional state. To restart a process and restore its context, the restart routine adjusts the stack pointer to create enough space on the stack so that the saved stack can be copied to its original address, and restores any

saved registers.

8.2 Determining when to migrate processes

As mentioned above, the philosophy is that the executing job be sensitive to the environment in which it is executing and that it is the job, not an external scheduler, that makes appropriate scheduling decisions. For the job to make an informed decision, some global resource information needs to be maintained. The solution adopted is to have a daemon running on each machine in the network which maintains local approximations to the global load. The accuracy of these approximations is discussed in the next section. Here it is assumed that each machine contains information on the entire network which is no more than a few minutes old with a high probability.

When a BSP job requests a number of processes, the local daemon is queried for a suitable list of machines on which to execute the job (the daemon responds so that the job may be run on the least loaded machines). In order that the decisions are not too fickle, the five minute load averages are used. Also, it is a requirement that not too much network traffic be generated to maintain a reasonable global state. Since the five minute load averages are being used, it is not too important that entries in the load table become slightly out of date as the wildest swings in the load averages take a couple of minutes to register in the five minute load average figures.

Let G_i be the approximation to the global load on machine i. Then given P machines, the true global load is $G = G_1 \sqcup \cdots \sqcup G_P$; where $\sqcup$ is used to merge the approximations from two machines. Given a BSP job running on p processors with machine names[1] j in the set $\{i_1, \ldots, i_p\}$, the approximation $G' = G_{i_1} \sqcup \cdots \sqcup G_{i_p}$ is used, which is better than any of the individual approximations with a high probability. G' is a sequence of machine names sorted in decreasing priority order (based on load averages, number of CPUs and their speeds). If the top set of p entries of G' is not $\{i_1, \ldots, i_p\}$ then an alternate and better assignment of processes to machines exists (call this predicate *fn*). In order not to cause processes to thrash between machines, a measure x of the load of the job (where $0 \leq x \leq 1$) is added to the load averages of all machines not involved in the BSP computation before the predicate *fn* is applied. This anticipates the maximum increase in the load of a machine when a process is migrated to it. Any observed increase in load greater than x is therefore caused by additional external load.

The aim is to ensure that the only overhead in process migration is the time taken to write p instances of the check-pointed program to disk. Therefore,

[1] A distinction is made between machines and processors as each machine may contain a number of processors, each of which runs multiple processes.

it is required that the calculation that determines when to perform process migration does not unduly impact the computation or communication performance of *BSPlib*. The equation $fn(G') = fn(G_{i_1} \sqcup \cdots \sqcup G_{i_p})$ needs to be solved either on a superstep by superstep basis or every N supersteps. However, the result can be obtained without first merging the global load approximations. This can be done by each processor independently computing its load approximations G_i and checking that it is amongst the top p after adding x_j to the loads of machines not involved in the current BSP job; where $0 \leq x_j \leq 1$ is the contribution of the BSP process on machine j, to the load average on that machine. This calculation can be performed on entry to the superstep T seconds after the last checkpoint (i.e. this checking does not have to be performed in synchrony). The Boolean result from each of the processors is reduced with the or-operator to ensure that all processors agree to checkpoint during the same superstep. In the implementations of *BSPlib* described in this thesis, this is piggy-backed onto a bitwise or-reduction that is used to globally optimise communication because

$$cost(2 \times reduction(|p|)) \quad \gg \quad cost(reduction(2 \times |p|))$$

as the network latency is considerable. Reduction of the resulting $|p|$-bit vector using the logical-and operator indicates whether or not all processors agree to checkpoint. Hence, if a checkpoint is not necessary, there is no substantial impact on the performance of *BSPlib*.

8.3 Determining the global load of a system

The load daemons use a protocol in which the locally held global load states are periodically sent to k randomly chosen daemons running on other machines. This update period is uniformly and randomly distributed with each machine being independent. When a load daemon receives an update, it responds by merging the locally held load table and sending the resultant table back to the sender. The sender then merges the entries of the returned table with the entries of the locally held table. For purposes of simplifying the analysis, the updates are assumed to happen at fixed intervals and in a lockstep fashion. It is also assumed that the processors do not respond with the merged table, but merely update their tables by merging in the update requests. The analysis that follows always provides an upper bound for the actual protocol used.

If each processor sent out a message at the end of each interval to all the other processors, this would require P^2 messages to maintain the global state. A job requiring $p \leq P$ processes could contact P machines and arrive at an optimal choice of processors with considerably fewer messages provided that

jobs did not start very often. However, with a large network of machines, the rate of jobs starting and the number of machines P would quickly lengthen the delay in scheduling a BSP job. By maintaining a global processor utilisation state at each of the machines, starting a job only involves contacting the local daemon when choosing a set of p processors and thus need not contribute to network traffic. Even once the ordering of processors has been selected, the problem of over assigning work to the least loaded machines can be avoided by having those machines reject the workload request based on knowledge built up locally. The algorithm then simply tries the machine with the next highest priority.

The quality of the decision for the optimal set of machines depends on the ages of the entries in the distributed load averages tables. The way in which the age of an entry varies over time can be considered as a semi-Markov process, in which the embedded Markov chain changes states as entries are rejuvenated by being updated or are allowed to age in time intervals. By considering the embedded Markov chain, the effectiveness of the traffic in keeping the information current under various algorithms can be compared. The Markov property holds in the random processes presented here because the algorithms do not change their message policy over time and hence the transition probabilities are independent of time (see, for example, Kleinrock [82] or Ross [104]).

If each machine uniformly and randomly chooses a partner machine at the end of each interval and sends its load average value to that machine, then the mean age of each entry in a distributed load average table can be calculated by considering the discrete time Markov chain shown in Figure 8.1. The existence of the stationary distribution $\{\pi_i\}$ can be established by considering the states of this Markov chain. It is clear that state 0 is aperiodic and positive recurrent ($P_{0,0} \neq 0$) and because all states communicate, the process is an irreducible ergodic Markov chain [104]. Hence, the stationary or limiting probability distribution $\{\pi_i\}$ exists and is the unique solution of

$$\pi_i = \sum_{j=0}^{\infty} \pi_j P_{j,i} \; i \in \mathbb{N} \tag{8.1}$$

$$\sum_{i=0}^{\infty} \pi_i = 1. \tag{8.2}$$

The transition probabilities ($P_{j,i}$) can be obtained by considering that each of the processors choose their message destinations uniformly at random amongst the other $p-1$ processors. Using this algorithm there would only be p messages at the end of each interval and the age distribution $\{\pi_i : i \in \mathbb{N}\}$ and the mean age μ can be found using the transition probabilities and

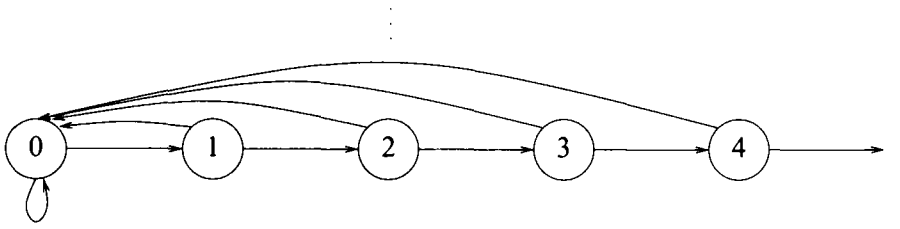


Figure 8.1: Markov chain considering only single direct updates

solving Equations (8.1) and (8.2):

$$\begin{aligned} \pi_i &= \frac{1}{p-1}\left(\frac{p-2}{p-1}\right)^i \\ \mu &= \sum_{i=1}^{\infty} i\pi_i = p-2. \end{aligned}$$

By sending out messages more frequently, or sending out k messages at the end of each interval, the mean age can be reduced to $\mathcal{O}(p/k)$, but this increases the traffic and required number of virtual circuits to pk.

By allowing each machine to exchange *all* their load average information with k other machines at the end of each interval, a significant reduction in the mean age of the entries can be achieved with pk circuits. This scheme allows machines to choose between the most current load average figures, even if they were indirectly obtained from other machines. Figure 8.3 shows the corresponding Markov chain which is similarly irreducible and ergodic.

The transition probabilities can be computed by considering that the processors send messages uniformly at random to the other processors with the proviso that no destination is chosen more than once. The resulting transition probabilities can be used to solve Equations (8.1) and (8.2) for the distribution, $\{\pi'_i : i \in \mathbb{N}\}$, of the ages of the entries in the load average tables. In this stochastic process, transitions into state 0 arise only out of direct updates, that is, a machine directly updating its entry in a remote table and a closed form solution for π'_0 can be obtained:

$$\begin{aligned} \pi'_0 &= \sum_{i=0}^{\infty} \pi'_i \left(1 - \frac{p-2}{p-1} \times \frac{p-3}{p-2} \times \cdots \times \frac{p-1-k}{p-k}\right) \\ &= \frac{k}{p-1} \end{aligned}$$

The stationary probabilities π'_i for $i > 0$ from Equations 8.2 and considering Figure 8.3 are given by the recurrence

$$\pi'_i = \pi'_{i-1} P\{\textit{no useful updates}\}$$

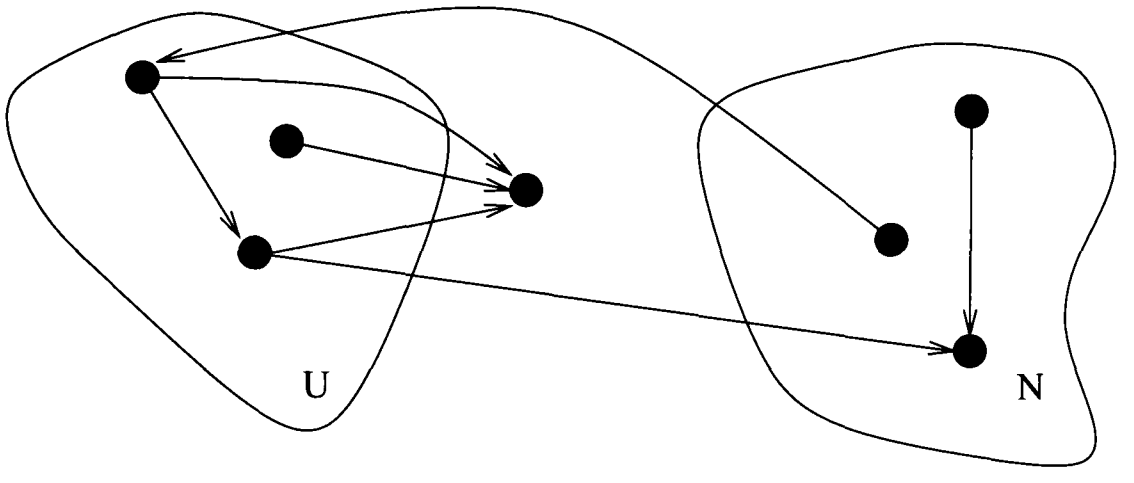


Figure 8.2: Partition of $p-2$ processors into those that send updates, U, and those that do not, N

$$+ \left(1 - \sum_{j=0}^{i-1} \pi'_j\right) P\{min\ age\ is\ i\}.$$

In determining the probability of receiving no useful updates at the end of an interval only indirect updates need be considered with

$$P\{no\ direct\ update\} = \frac{p-1-k}{p-1}.$$

The $p-2$ other processors must not send useful updates. Consider a partition of the processors into those that send an update, U, and those that do not, N, as depicted in Figure 8.2. If $|U| = u$ then $|N| = p-2-u$. The probability that no useful updates are received is obtained by considering all such partitions and the fact that the processors act independently:

$$\sum_{u=0}^{p-2} \binom{p-2}{u} \left(\frac{k}{p-2}\right)^u \left(\frac{p-2-k}{p-2}\right)^{p-2-u} \left(\sum_{w=i-1}^{\infty} \pi'_w\right)^u$$

Since the destination processors are chosen independent of message content, conditioning on no direct updates gives

$$P\{no\ useful\ updates\}$$
$$= \frac{p-1-k}{p-1} \sum_{u=0}^{p-2} \binom{p-2}{u} \left(\frac{k}{p-2}\right)^u \left(\frac{p-2-k}{p-2}\right)^{p-2-u} \left(\sum_{w=i-1}^{\infty} \pi'_w\right)^u.$$

Direct updates are also excluded when computing the probability that the minimum age of messages updating an entry is $i > 0$. Consider the set

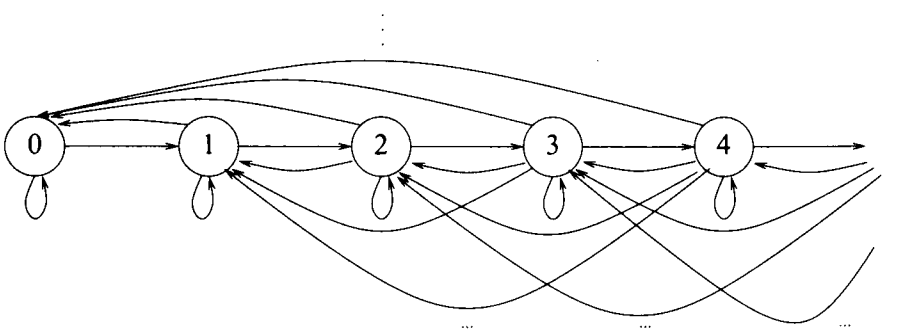


Figure 8.3: Markov chain when indirect updates are allowed

U split into two groups of size x and $u - x$. Since these messages are from processors under the same conditions and exactly one time period has elapsed since their last possible update, the ages of the entries in these messages have a distribution of $\{\pi'_{i-1}\}$. Hence, the probability that the best of u updates is i is

$$B(i,u) = \sum_{x=1}^{u} \binom{u}{x} \left(\pi'_{i-1}\right)^x \left(\sum_{t=i}^{\infty} \pi'_t\right)^{u-x}.$$

Conditioning on no direct updates, the independence of the processors and taking all possible partitions into account gives

$$\begin{aligned} & P\{min\ age\ is\ i\} \\ &= \frac{p-1-k}{p-1} \sum_{u=1}^{p-2} \binom{p-2}{u} B(i,u) \left(\frac{k}{p-2}\right)^u \left(\frac{p-2-k}{p-2}\right)^{p-2-u} \\ &= (p-1-k) \sum_{u=1}^{p-2} \binom{p-2}{u} \left(\frac{k}{p-2}\right)^u \left(\frac{p-2-k}{p-2}\right)^{p-2-u} \\ &\quad \times \left[\frac{-\left(1-\sum_{t=0}^{i-1}\pi_t\right)^u + \left(1+\frac{\pi_{i-1}}{1-\sum_{t=0}^{i-1}\pi_t}\right)^u \left(1-\sum_{t=0}^{i-1}\pi_t\right)^u}{(p-1)} \right]. \end{aligned}$$

The actual protocol implemented re-uses the established k circuits to send the merged tables back to the sender daemon. Simulation results show that this strategy is better than using $2k$ update circuits as it results in lower mean ages of the table entries.

Figure 8.4 shows the mean table entry ages of the three strategies when $k \in \{1, 3, 6\}$ against P. As P is given as a log scale, it is clear from the figure that while the first two strategies give a mean age μ as $\mathcal{O}(P)$, the third strategy (allowing indirect updates) gives a mean age of μ' as $\mathcal{O}(\log P)$. The figure also shows simulation results for $k \in \{3, 6\}$ for the indirect update strategy

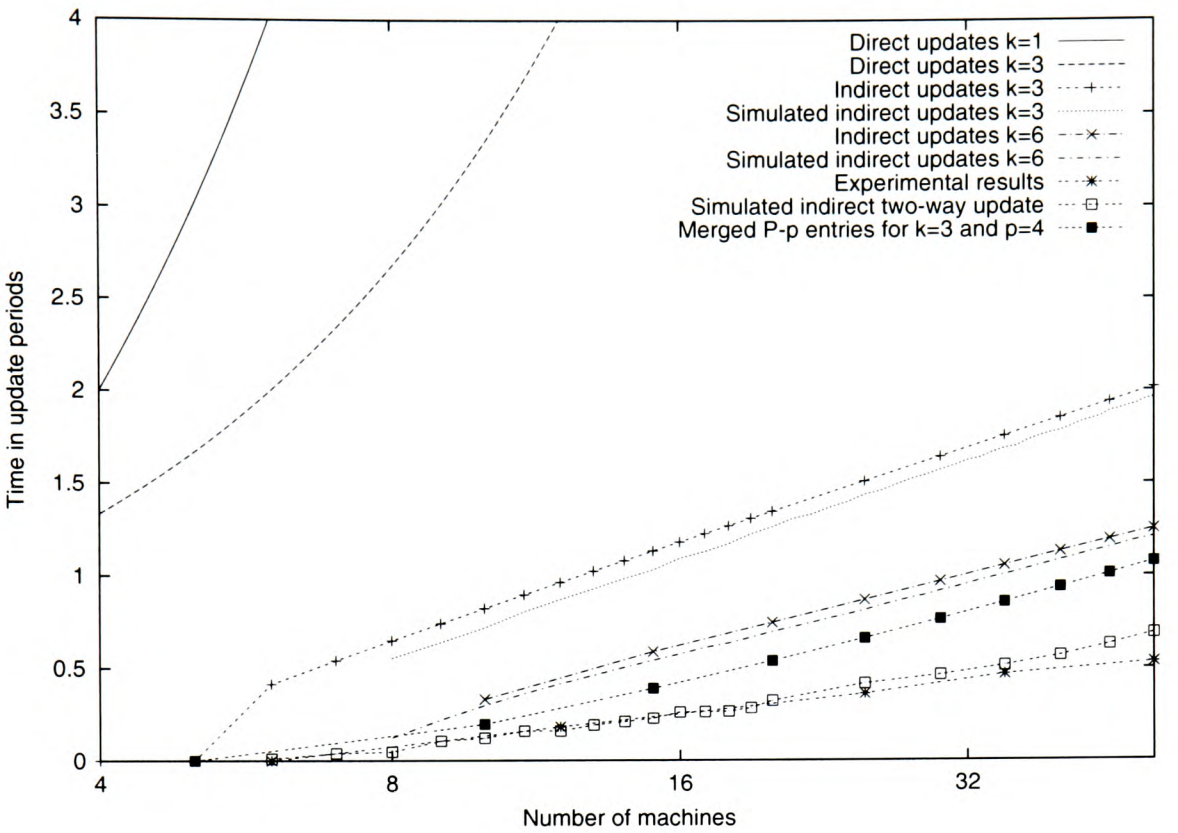


Figure 8.4: Experimental, simulation and Markov analysis of the mean ages of entries in global load tables under the various strategies considered, as well as the mean ages of the load entries for the $P - p$ processors computed by the p processors

supporting the analytic results of the Markov chain shown in Figure 8.3. For the implemented strategy, the experimental results were converted to update intervals by dividing by the mean update period of 240 seconds and subtracting a half as the states in the Markov chains represent time. These experimental results compare favourably with a simulation of the process in which updates occur in both directions. They show that the lock-step assumption is a reasonable approximation making it possible to compare all strategies on the same graph.

If the discrete times of the Markov chains are replaced with update intervals, t, the distributions above give the mean age at the beginning of each of the intervals. The figure shows that in order to bound the mean age to, say, five minutes one must ensure that

$$\begin{aligned} t(\mu + \frac{1}{2}) &\leq 5 \text{ minutes, or} \\ t &\leq \frac{10}{2\mu + 1}. \end{aligned}$$

Therefore when $p = 32$, t should be less than or equal to $3\frac{1}{3}$ minutes using the indirect update strategy with $k = 6$. For the simulation of the indirect

p	`ypcat` order	daemon order	daemon order + forced migration	daemon order + selective migration
2	3255	623	841	658
4	3855	1155	1916	1092
8	2968	1161		

Table 8.1: Execution time in seconds for the electro-magnetics simulation

two-way update strategy with $k = 3$ (where $\mu \approx \frac{1}{2}$ at $p = 32$), t should be less than or equal to 5 minutes. The line marked "experimental results" shows mean ages achieved (normalised to update periods for comparison) where t, in seconds, is drawn uniformly and randomly in $[180, 300]$.

As described in Section 8.2, by having the p processes involved in a BSP computation merge their load average tables before choosing where to migrate the processes, the *current* five minute load averages for the p processors executing the job is obtained, and the ages of the load average table entries for the other $P - p$ machines have a distribution $\{\pi''_i : i \in \mathbb{N}\}$ where

$$\pi''_i = \sum_{x=1}^{p} \binom{p}{x} (\pi'_i)^x \left(1 - \sum_{t=0}^{i} \pi'_t\right)^{p-x}.$$

Figure 8.4 includes the resulting mean age from this distribution for $p = 4$ against the total number of machines P, and compares it with the mean derived from the original distribution $\{\pi'_i : i \in \mathbb{N}\}$. It is clear from the figure that as P decreases to p, the mean age of the data from the merged tables decreases until the mean age is zero when $p = P$.

8.4 Experimental results for an electro-magnetics application

Table 8.1 shows the results from an electro-magnetics application[2] running on various numbers of processors. The four columns of execution time show the following: (1) a job running on a random choice of machines. These may be highly loaded (in terms of the current number of jobs and the average number of ready to run jobs) or may not have fast processors; (2) a job that is initiated on the p best machines (i.e. the fastest least loaded machines); (3) a job that is started on the best p machines, but is check-pointed every

[2]The application is EMMA T:FE3D and is part of the British Aerospace EMMA electro-magnetic software suite.

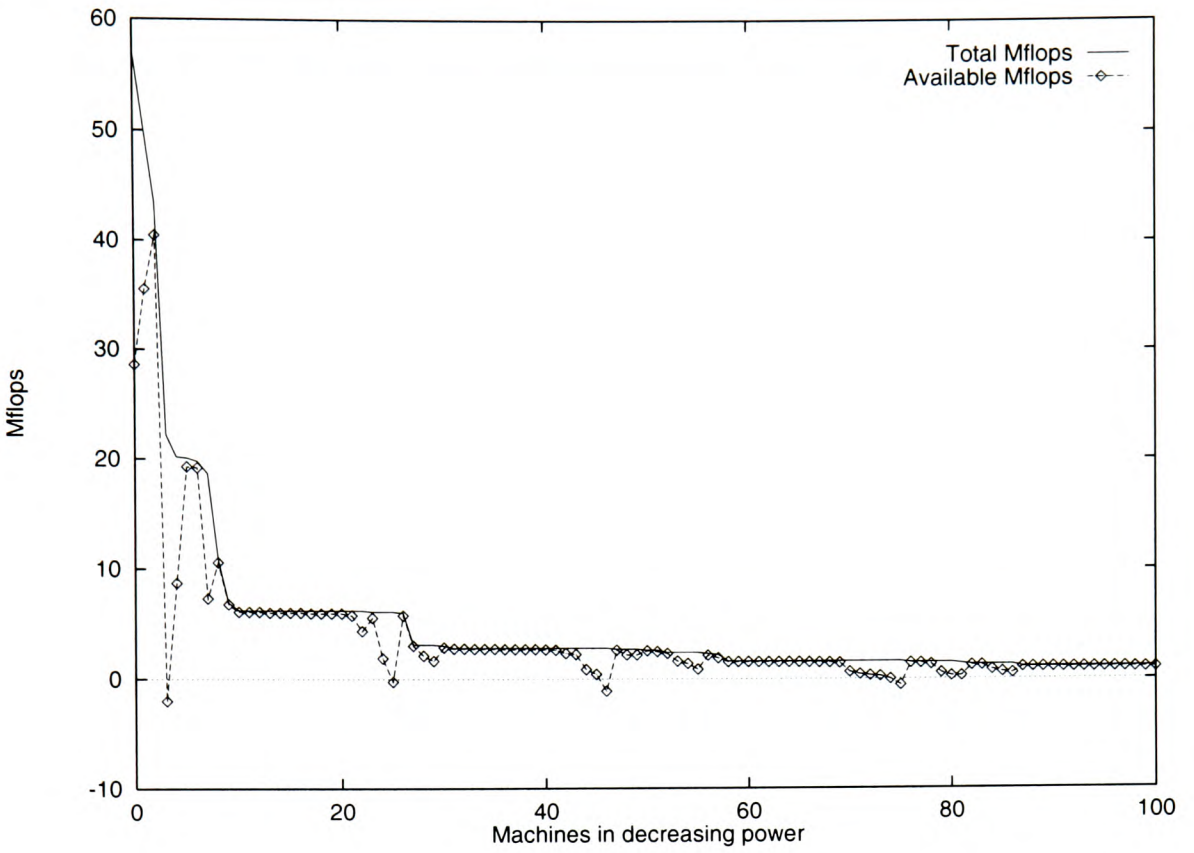


Figure 8.5: Profile of the computational power of a collection of workstations

two minutes; and (4) a job that is started on the best p machines and checks every two minutes whether it is beneficial to check-point and migrate.

The first experiment is an approximation to a job that encounters poor service during execution. As can be seen from the dramatic decrease in execution times in the second experiment it is always beneficial to start a job on the most powerful unloaded machines. In the situation where the chosen processors have the same power, and the job is long running, then the second experiment will degrade to the performance of the first. The third experiment quantifies the overhead in check-pointing. As ten check-points were performed at $p = 4$ the increase in execution time shows that a local checkpoint takes 19 seconds to write a seven megabyte image to disc. The fourth experiment shows that there is little overhead in checking whether a check-point is necessary.

As it costs $19p$ seconds to perform a migration, it is only beneficial to migrate in situations where the processor usage is not too erratic, thus allowing the job to recoup the cost of the migration on the set of processors that were migrated to. If jobs are long running and compute bound then there is a lot of potential for regaining lost ground due to having to migrate from a loaded machine.

The results shown in Table 8.1 are from an electro-magnetics experiment

that simulates a field around a sphere. As can be seen from the table, no parallel speed-up was achieved when increasing the size of the NOW; this was due to the dominance of communication over computation in this small test case. Larger realistic test cases at BAe have shown linear speed-up up-to sixteen processes as computation begins to dominate.

This work is based on the assumption that the available cycles on the machines change continuously over time. One might expect that in the absence of any prior knowledge that the probability of available mega-flops is uniform over all the processors. One situation where this might not be the case is when the workstations are not homogeneous in available power. For example, Figure 8.5 shows a graph of available computing power on each workstation in Oxford University Computing Laboratory against available power at a particular instance in time. The available power is expressed by the formula $(n - L)s$; where n is the number of processors in a machine, L is the load average on that machine and s is the mega-flop rating of a single processor. In this case the shape of the graph can be attributed to Moore's law as the workstations were purchased over time with the low powered aging workstations to the right and the high-performance machines procured more recently to the left.

When choosing to schedule a p process parallel job, either a homogeneous set of unloaded (slow) machines can be used, or the best p. The policy adopted is that although the unloaded machines ensure little interference, they provide only a fraction of the power of the best machines. However, running on the popular powerful machines, can also make a job susceptible to low throughput as the available cycles at a node can vary dramatically over time. For example, the figure shows that the fourth machine from the left has a peak performance of 50 mega-flop per second, yet it is so loaded that there are no free cycles for parallel jobs. In summary, the erratic, but powerful machines, to the left of the graph are most suited to computational intensive parallel jobs, yet they are the very machines which would benefit from process migration.

8.5 Related work

The process migration work described in this chapter also provides fault tolerance if the mean time between failure is greater than the rate at which processes migrate/checkpoint. Existing work in this area has tended to concentrate on fault tolerance using redundant computation. For example, Nibhanupudi and Szymanski [96] minimise the slowdown of a BSP job when external loads are applied to machines in a network by replicating computation on a number of machines. At the end of a superstep, the results of the

fastest of the replicated jobs is used to form the global state of the system. Although their system allows the mean time between failure to be less than that in the scheme described here due to the replicated jobs, their prototype system requires user annotation of the data structures to be included in a checkpoint, and they assume that there won't be a machine failure during communication. In contrast, the fault tolerance in the system described here is transparent to the user, and has no restrictions on when faults can occur. Also, this approach doesn't suffer from the considerable overhead that would be incurred to implement a process replication scheme. As already noted, the only overhead incurred is the time taken to write the p checkpoints to disk.

A similar approach is that of Kaashoek *et al.* [78] where fault tolerance of Orca programs is provided on top of the Amoeba distributed operating system. Their approach is complicated by the fact that they have to determine locally when communication is quiescent so that a stable checkpoint can be taken. Other parallel check-pointing systems for MPI [111] and PVM [114] also suffer from this problem, as a checkpoint can be performed only if there is no communication in transit when each process performs a local checkpoint to capture the global state [5]. Fortunately, the check-pointing regime described here is far simpler than any of the approaches used in message passing systems, as opportunities for a global checkpoint naturally arise out of the superstep structure of BSP programs. The process migration facilities provided for MPI [111] and PVM have usually been developed on top of the check-pointing and batch scheduling facilities provided by Condor [38] and LSF [125].

Other schemes have recently been presented which perform task, process or thread migration for purposes of load balancing. Tan *et al.* [116] describe task migratable PVM or tmPVM which exploits the homogeneity of MPPs to achieve load balancing among the processors of a PVM job. As their approach requires the co-existence of both the source and destination processes during the migration, their approach does not address the problem of fault tolerance. Load monitors on each of the nodes are responsible for collecting load information on the local node and forwarding to a centralised resource monitor. Thus unlike the approach presented in this chapter, the global load state is held centrally. Antonui *et al.* [3] show that by splitting static data from dynamic thread data and transferring only the dynamic data as part of the migration process, thread migrations of less than $75\mu s$ are possible for trivial applications over BIP [100] using Myrinet [9] in the PM2 multi-threaded runtime system. Whilst the scheme achieves a finer control of load balancing than the *BSPlib* scheme (as work is migrated at the thread level) the set of processors hosting these threads and thus the application, do so from the start of the job. As this set of processors is constant, fault tolerance is not supported.

8.6 Conclusion

It has been shown that it is possible to perform fault tolerance and process migration of BSP programs in a transparent way on a network of workstations. By paying careful attention to the design of a distributed load manager, it is possible to determine the global load of a system with minimal impact on network traffic. This, in conjunction with the active manner in which BSP jobs migrate between machines, produces a system that is unobtrusive to non-BSP users, whilst providing good usage of available resources.

Chapter 9

Conclusions and Future Work

A theme throughout this thesis has been the exploitation of the global state amongst the processes of a BSP computation. *BSPlib*, the particular model of BSP computation which is somewhat relaxed from that proposed by Valiant [120], decouples not only communication and synchronisation, but also the requests for communication and the realization of that communication.

This gap has been exploited before by Hill and Skillicorn [67] to combine messages destined for a particular process in a BSP computation, and the choosing of a schedule which avoids congestion at the destination nodes. This exploitation is extended here by a scheme which modulates injection without requiring exact timing or token passing to achieve a global bandwidth close to the peak bandwidth of the network medium.

All networks that suffer from some loss of efficiency as the applied load increases, can benefit from this technique. For example, Stallings [109] shows applied load against successful load for packet switching, frame relay and ATM or cell relay networks, which have similar characteristics to that shown for CSMA/CD in Figure 4.2.

In order to apply this technique to BSP computation, some notion of the applied global load on the communication subsystem has to be known. The arrangement of BSP computation according to the *BSPlib* model makes such information available at practically no extra cost. A general assumption about network performance analysis is that the stations can be thought of as operating independently. This is obviously not true for the processes involved in a parallel computation, and in particular a BSP style of computation represents a synchronised assault of traffic onto the network. One

would expect then that for all parallel computation, particularly a BSP style would cause the types of network mentioned above to operate in a state of severe congestion. This technique demonstrated with both TCP/IP and UDP/IP over Ethernet, re-introduces a measure of independence of the stations or processes on the network. Ironically, while BSP style computations are most vulnerable to such network characteristics, it is the global structure of BSP computation that is exploited in order to solve the problem. The performance that can be achieved in this way is better than can be obtained in systems whose semantics impose a tighter synchrony in their communication such as MPI.

Two popular measures of network performance are bandwidth and latency. However, in BSP these measures are of the global bandwidth of the interconnect, and the latency across the interconnect which corresponds roughly with the cost of synchronisation. In this work it is shown that the variance of the achieved bandwidth is also important. Again, it is not only BSP computation that can suffer from a high variance in bandwidth utilisation, but the BSP model can be used in the analysis of the phenomenon.

Two approaches have been taken. One can think of the communication interconnect as providing $p(p-1)/2$ bi-directional circuits or links that connect the p processors and span the interconnect. Alternatively, since the BSP cost model implicitly asserts that the cost of communication is encountered at the edges of the interconnect, the value of g achieved for a particular communication pattern can be thought of as the g value made up from the participating processors independently.

In the first case, it is demonstrated that the bandwidths achieved over a link and the variance of this bandwidth can be used to place a bound on the performance of an interconnect built up of such links. That this bound is an upper bound follows from the fact that the interconnect is idealised and fully connected with no internal congestion contributing to the degradation of performance as p increases. The observation that this second order effect is an important architectural parameter as it affects the stability of the BSP cost model, forms the basis of an empirical study of interconnect behaviours under the *BSPlib* programming library [62].

The second case, from the BSP cost model point of view, would require that locally there be little dependence on the $p(p-1)/2$ circuits and that the protocol stacks themselves be largely independent of the number of processes involved. One would expect this to be the case if the protocol stacks are considered scalable. This assumption is stronger than that made above when considering the interconnect as comprising $p(p-1)/2$ independent links, in that it assumes that the BSP machine is idealised (rather than the interconnect topology). All that is required of the interconnect topology is that there be negligible congestion contributing to the degradation of the achieved

bandwidth and its variance. Clearly this latter behaviour is a requirement for a solution to be considered scalable and the analysis gives a means of determining whether the experimental results demonstrate this scalability notwithstanding the observed variance. The families of distributions shown in Figures 7.3(a) and (b) and the h-relations timings plotted in Figure 7.4 show the superiority of the *BSPlib*/NIC implementation compared to the *BSPlib*/UDP implementation.

The shift in emphasis from circuits to protocol stacks is parallelled in the development of the messaging layers of *BSPlib* on networks of workstations. The *BSPlib*/TCP implementation uses BSD sockets as an interface to TCP/IP and as such each process manages $p-1$ circuit endpoints. This proved to be less than ideal and easily demonstrated by the BSP model. Not only because each process had to manage $p-1$ independent circuits, each with a separate pool of buffer resources, but also because the point-to-point transport protocol on each of the links takes no account of the communication load applied on the $p(p-1)/2-1$ other links. Neither would one expect such consideration of a protocol designed to provide end-to-end reliability over a datagram service over the expanse of the Internet. On the other hand, operating at the level of the datagram service of UDP/IP provides a convenient setting for developing and implementing protocols more structured toward the requirements of the BSP model.

While it was possible to implement the pacing on top of the TCP/IP based protocol, the time based rather than resource usage based explicit acknowledgements of the TCP/IP protocol, meant that extra traffic on the medium had to be taken into account. This is so because as p increases fewer circuits can have traffic placed on them in the reverse direction within $200ms$ of receiving a message to piggy-back acknowledgements and to avoid the resulting timeout of TCP/IP which generates an explicit acknowledgement. In the limit the circuits behave in such a way that every frame is explicitly acknowledged.

Hence in the UDP/IP based protocol upon which the *BSPlib*/UDP messaging layer was built, the explicit acknowledgements are generated by taking resource usage into account and timing them in such a way that acknowledgements are received just before buffer starvation can occur. As these are buffers held by the sender, the requesting of such acknowledgements is generated by the sender explicitly, or by piggy-backing the request on traffic in the required direction. This allows the protocol to operate with very few explicit acknowledgements (Table 6.6).

For patterns of communication which do not allow all data to be acknowledged by piggy-backing acknowledgements onto traffic in the reverse direction, a significant number of explicit acknowledgements can be avoided by using implicit acknowledgements. This is a further exploitation of BSP

computation and its bulk communication property (Table 6.6).

Another advantage of forgoing the $p(p-1)/2$ point-to-point circuit based protocol of TCP/IP in favour of the p end-point protocol described in this thesis, is that for skewed h-relations a single buffer pool provides a far better buffer utilisation and is better suited to the BSP model.

Relative to the time required to send a message at the physical layer, the mean time to dispatch a waiting process after the arrival of a message can be significantly large (Section 6.1.2). Furthermore, the unreliable component is the physical layer. Consequently, by providing recovery across the user space end-points as in the case of *BSPlib*/UDP, buffers are held for longer than they need to be, as acknowledgements will get generated only once the message has reached the user process. Also, when recovery needs to be triggered because of the loss of a frame, this can be done only once the receiving process has been dispatched. Hence, and contrary to recent trends, the *BSPlib*/NIC implementation of the protocol executes recovery as soon as possible after the interrupt handler has been invoked and the error has been detected. Similarly, any updated protocol information, such as the highest in-sequence sequence number and the sequence numbers that record the extent of any missing frames in the frame sequence, is updated and made available immediately for messages returning on that circuit. Taking account of the received acknowledgement numbers at this level makes sure that buffers held for retransmission are freed as the incoming message acknowledges receipt of the frame. The implementation of the protocol is not entirely kernel based and like many user space protocols, buffer pools are mapped into the user space to avoid excessive copying and to minimise the number of system calls.

The resulting protocol compares favourably with a number of contemporary proposed protocols, but with two major benefits: Firstly, by concentrating on global issues, especially those exposed by the BSP model, a protocol with latency and bandwidth utilisation characteristics has been devised which scales reasonably well with respect to the backplane bandwidth of the hardware over which it has been benchmarked (Table 6.7). Secondly, by targeting *BSPlib*, all the implementations are directly usable by existing *BSPlib* programs without requiring any changes.

A requirement of process migration or fault tolerance achieved by restarting processes is that the restart point be consistent and reflect the global state. This is something which is hard to achieve in a general distributed setting involving inter-process communication as observed by Epema *et al.* [38]. In a BSP setting one can capture a globally consistent state by realizing that after a global synchronisation the global state is reduced to the collection of local states of the individual processes. Hence, by deploying techniques similar to Condor (Epema *et al.* [38]) to capture the local states of the

processes, it is possible to capture a globally consistent state of the entire BSP computation.

Capturing the global state of a BSP computation enables BSP computation to be both fault tolerant (at least to node failures) and facilitates process migration. Of course, process migration is implied by this type of fault tolerance, but it is also applicable in networks of workstations where spare cycles can be put to good use. For example, by choosing a set of processors to host the processes of a BSP computation based on the loads of the processors, may not be optimal given that the low loads may simply represent a temporary lull in activity on those nodes, and could have a significant impact on the completion time of the job.

Because BSP encourages balanced computation among the processes of a BSP job, the entire job is likely to progress at a speed dictated by the processor with the fewest available cycles. This processor may be a slow processor or it may be committed to other work. This leads to the motivation for migration amongst the processors of networks of workstations in an environment similar to the power and workload distribution depicted in Figure 8.5. The combined curves in the figure demonstrates that the probability of the processors' cycles being available increases as the power of the processors decreases, making the lower powered and less popular processors more attractive for stability but offering very low computational power. On the other hand, the more popular higher powered processors have the potential to offer greater computational power, but both the probabilities of this power being available and remaining so, is considerably less than that offered by the lower powered processors.

This problem has been considered in an open network of workstations environment where it is politically not possible to control the assignment of work amongst the workstations centrally. Consequently, the workloads have to be attained by polling the machines. In this way, the machines offering the most available cycles can be chosen as candidate nodes for the computation. Also, because additional work arrives from outside this system, the nodes offering the most available power at the start of the computation may not remain the optimal nodes as the computation proceeds.

A scheme for maintaining an approximation of the global load on the system amongst all the nodes has been described, as well as for making consistent checkpoint decisions, and for migrating processes amongst the processors should a better candidate set of processors become available. In doing so, the various algorithms for computing a distributed global load across all the participating workstations is analysed in terms of how old the values are from their true sampled values, and the amount of traffic required to maintain the approximations.

9.1 Future work

There are many aspects of this work that can be expanded on and applied in other settings. Some of the remarks below about expansions of the work are purely speculative whilst others are already being attempted.

Certain solutions or techniques described are hard to implement in a more general setting in their present form. While the general principle of global control can be expanded upon and applied in other settings, many of the implementations have to be considered only as prototypes and are not portable. For example, fault tolerant and process migration supporting code in *BSPlib* breaks a certain amount of abstraction of the C Standard Library implementation. This was necessary in order that sufficient state pertaining to the computation could be captured and restored with modification to the application. Support for such facilities are thus improperly placed within *BSPlib* and should be implemented closer to the native runtime system.

A similar, but less severe problem exists with the *BSPlib*/NIC implementation of the protocol. In this instance there is not sufficient abstraction of the network device, and porting of the protocol to another device would not be as trivial an effort as it ought to be. The biggest problem is that certain native features of the device, on which the protocol relies, are not generally available on other network interface cards. In order to port the protocol, these missing features would have to be provided by augmenting the physical device with a thin software layer. There would be some performance penalty, particularly if the large and open descriptor rings or the work polling features, had to be simulated.

In some respects the protocol represents reversion to more classic protocol stacks. For example, reliability is provided at the lowest layer, whilst segmentation and reassembly are performed at a higher level. In the consideration of an extension where multiple network interface cards are used to increase the bandwidth, one could perform recovery at the NIC level; in effect providing multiple recovery agents between pairs of processors each responsible for an independent fallible 'circuit'. It would still make sense to provide packing, segmentation and reassembly at a higher level over the aggregated and reliable circuits, to provide a single logical stream between the pair of processors. A motivation for such an arrangement would be that it is at the hardware level and link level that errors tend to occur (provided enough buffer slackness is made available at the end-points). Since each hardware adaptor potentially represents a different end-point to a set of paths, recovery over all the adaptors results in a large number of duplicate frames thus depleting the available bandwidth. Another advantage of this arrangement would be in the consumption of messages. Since all the reliable circuits deliver subsequences that are a partition of the main sequence, the

reconstitution of the main sequence is a simple merging operation that is performed as the frames are consumed by the higher layer entity or application, and no sorting is required.

At present the *BSPlib*/NIC implementation supports only a single application at a time (this is generally the case where high-performance computing is the focus). Moving to the more general setting of a multiprogramming environment poses an enormous problem if the performance of *BSPlib*/NIC is to be achieved in a manner that is fair to the various jobs in the mix. A major problem would be the maintenance of system integrity and fault isolation amongst the various users. Such system integrity and application isolation would be hard to provide while maintaining good values for the performance measures, not only for *BSPlib* but for other systems particularly where the good performance is partly obtained by the use of busy-waiting techniques.

Whilst the protocol and its implementation are restricted to local area networks of closed collections of processors, certain aspects of the work could be applied to the wider and open Internet setting (not in a high-performance computation setting, but possibly in a server and routing setting). For example, the protocol for the maintenance of load averages could maintain an approximation of load averages for a large number of servers, or the approximation of link utilizations between a large number of servers and their front-end processors, in order that reasonable routing and load balancing decisions are made. It may also be possible to manipulate the point-to-point TCP/IP circuits to achieve a fair and globally better utilisation of resources over a large number of server machines connected to the Internet.

List of Figures

List of Tables

Bibliography

[1] *3C90x Network Interface Cards Technical Reference.* 3COM, December 1997. Part Number:09-1163000.

[2] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995. ISSN 0272-1732.

[3] Gabriel Antonui, Luc Bougé, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *International Parallel Processing Symposium (IPPS'99): Workshop on Run-Time Systems for Prallel Programming*, volume 1586 of *Lecture Notes in Computer Science*, pages 496–510. Springer-Verlag, 1999.

[4] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Report NAS-95-020, NASA, December 1995.

[5] R. Baldoni, J. M. Hélary, A. Mostefaoui, and M. Raynal. A communication-induced checkpointing protocol that ensures the rollback-dependency trackability property. In *Proc. of the 27th IEEE Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 68–77, Seattle, WA, June 1997. IEEE.

[6] Ray Barriuso and Allan Knies. SHMEM user's guide, revision 2.0. Technical report, Cray Research Inc., Mendota Heights, MN, May 1994.

[7] M. Beck, H. Böhme, M. Dziaddzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals.* Addison-Wesley, 2nd edition, 1998.

[8] Joachim M. Blum, Thomas M. Warschko, and Walter F. Tichy. PULC: Parastation user-level communication. Design and Overview. In *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 498–509. Springer-Verlag, 1998.

[9] Nanette J Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet

— A gigabit-per-second local-area network. `http://www.myri.com`, November 1994.

[10] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library — design, implementation and performance. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Ditsributed Processing IPPS/SPDP 1999: April 12–16, 1999, San Juan, Puerto Rico*, pages 99–104, 1999.

[11] Stephan Brauss, Martin Frey, Anton Gunzinger, Martin Lienhard, and Josef Nemecek. Swiss-Tx communication libraries. In *High Perfomance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 623–632, Amsterdam, April 1999. Springer-Verlag.

[12] Gordon Brebner. *Computers in Communication.* McGraw-Hill, 1997.

[13] José C. Brustolini and Briand N. Bershad. Simple protocol processing for high-bandwidth low-latency networking. Technical Report CMU-CS-93-132, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1992.

[14] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems, North-Holland*, 14:3–23, 1987.

[15] Philip Buonadonna, Andrew Geweke, and David E. Culler. Implementation and analysis of the Virtual Interface Architecture. In *SuperComputing'98*, 1998.

[16] *Recommendation Z.100: Specification and Description Language SDL.* CCITT, blue book, volume x.1 edition, 1988.

[17] T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant. Bulk synchronous parallel computing - A paradigm for transportable software. In *Proc. 28th Hawaii International Conference on System Science*, volume II, pages 268–275. IEEE Computer Society Press, January 1995.

[18] Andrew A. Chien and Jae H. Kim. Approaches to quality of service in high performance networks. In *Proceedings of the Parallel Computer Routing and Communications Workshop*, volume 1417 of *Lecture Notes in Computer Science*, Atlanta, Georgia, July 1997. Springer-Verlag.

[19] Andrew Chin. Complexity models for all-purpose parallel computation. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel*

Computation, volume 4 of *Cambridge International Series on Parallel Computation*, pages 393–404. Cambridge University Press, Cambridge, 1993.

[20] G. Chiola and G. Ciaccio. Porting MPICH ADI on GAMMA with flow control. In *Proceedings of MWPP'99 (1999 Midwest Workshop on Parallel Processing)*, Kent, Ohio, August 1999.

[21] Giuseppe Ciacco. Optimal communication performance on Fast Ethernet with GAMMA. In *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 534–548. Springer-Verlag, 1998.

[22] *Catalyst 2900 series XL installation and configuration guide.* Cisco Systems, 1997. Part Number: 78-4417-01.

[23] Computer Systems Research Group, University of California at Berkeley. *4.4BSD Programmer's Reference Manual.* USENIX Association and O'Reilly & Associates, Inc., 1994.

[24] Computer Systems Research Group, University of California at Berkeley. *4.4BSD Programmer's Supplementary Documents.* USENIX Association and O'Reilly & Associates, Inc., 1994.

[25] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7:354–375, 1973.

[26] Paul I. Crumpton and Mike B. Giles. Multigrid aircraft computations using the OPlus parallel library. In *Parallel Computational Fluid Dynamics: Implementation and Results using Parallel Computers. Proceedings Parallel CFD '95*, pages 339–346, Pasadena, CA, USA, June 1995. Elsevier/North-Holland.

[27] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[28] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Scalable Parallel Computation.* Morgan Kaufmann, 1999.

[29] Stephen R. Donaldson, Jonathan M. D. Hill, and David B. Skillicorn. Communication performance optimisation requires minimising variance. *Future Generation Computer Systems*, 15(3):453–459, 1999.

[30] Stephen R. Donaldson, Jonathan M. D. Hill, and David B. Skillicorn. Exploiting global structure for performance on clusters. In *Proceedings*

13th International Parallel Processing Symposium and 10th Symposium on Parallel and Ditsributed Processing IPPS/SPDP 1999: April 12–16, 1999, San Juan, Puerto Rico, pages 176–182, 1999.

[31] Stephen R. Donaldson, Jonathan M. D. Hill, and David B. Skillicorn. Performance results for a reliable low-latency cluster communication protocol. In *International Parallel Processing Symposium (IPPS'99): 2nd Workshop on Personal Computer Based Networks of Workstations (PC-NOW'99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1097–1114. Springer-Verlag, 1999.

[32] Stephen R. Donaldson, Jonathan M. D. Hill, and David B. Skillicorn. Predictable communication on unpredictable networks: Implementing BSP over TCP/IP and UDP/IP. *Concurreny Practice and Experience*, 11(11):687–700, 1999.

[33] Stephen R. Donaldson, Jonathan M. D. Hill, and David B. Skillicorn. BSP clusters: high performance, reliable and very low cost. *Parallel Computing: Special Issue on Cluster Computing*, 26(2–3):199–242, 2000.

[34] M. C. Dracopoulos, C. Glasgow, K. Parrott, and J. Simkin. Bulk synchronous parallelisation of industrial electromagnetic software. *International Journal of Supercomputer Application of High Performance Computing*, 1996.

[35] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 388–396. IEEE, IEEE Press, 1997.

[36] Cezary Dubnicki, Livin Iftode, Edward W. Felton, and Kai Li. Software support for virtual memory mapped communication. In *Proceedings of the 10th International Parallel Processing Symposium*. IEEE, IEEE Press, 1996.

[37] Calvin C. Elgot and Abraham Robinson. Random-access stored-program machines, an approach to programming languages. *Journal of the Association of Computing Machinery*, 11(4):365–399, 1964.

[38] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generations of Computer Systems*, 12, 1996.

[39] Michaeil J. Flynn. Very high speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1902–1909, 1966.

[40] Steven Fortune and James Wyllie. Parallelism in random access ma-

chines. In *Proceedings of the 10th Annual ACM Conference on Theory of Computing*, pages 114–118. ACM, 1978.

[41] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496–507, 1970.

[42] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[43] Patrick Geoffrey, Laurent Lefèvre, CongDuc Pham, Loïc Prylli, Olivier Reymann, Bernard Tourancheau, and Roland Westrelin. High-speed LANs: New environments for parallel and distributed applications. In *EuroPar'99)*, volume 1685 of *Lecture Notes in Computer Science*, pages 633–642, Toulouse, August/September 1999. Springer-Verlag.

[44] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan. PRAM programming: Theory vs. practice. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, page unknown, 1998.

[45] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, August 1994.

[46] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[47] Leslie M. Goldschlager. A universal interconnection pattern for parallel computation. *Journal of the Association for Computing Machinery*, 29(3):1073–1086, July 1982.

[48] Herman H. Goldstine. *The Computer: from Pascal to von Neumann.* Princeton University Press, 1972.

[49] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4.* Prentice Hall, 1994.

[50] Mark W. Goudreau, Kevin Lang, Satish B. Rao, and Thanasis Tsantilas. The Green BSP Library. Technical Report 95–11, University of Central Florida, August, 1995.

[51] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, New York, 1995.

[52] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level message passing inter-

face. Technical Report MCS-P342-1193, Argonne National Laboratory, 1993.

[53] William Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Argonne National Laboratory, 1994.

[54] William Gropp and Ewing Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997. ISSN 0167-8191.

[55] Joseph L. Hammond and Peter J. P. O'Reilly. *Performance Analysis of Local Computer Networks.* Addison Wesley, 1987.

[56] Jifeng He, Quentin Miller, and Lei Chen. Algebraic laws for BSP programming. In *EuroPar'96 Parallel Processing: Volume II*, volume 1124 of *Lecture Notes in Computer Science*, pages 359–368. Springer-Verlag, 1996.

[57] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann, second edition, 1996.

[58] Jonathan M. D. Hill. The Oxford BSP toolset machine parameters. `http://www.bsp-worldwide.org/implmnts/oxtool/params_frame.html`, 1999.

[59] Jonathan M. D. Hill, Paul I. Crumpton, and David A. Burgess. Theory, practice, and a tool for BSP performance prediction. In *EuroPar'96*, volume 1124 of *Lecture Notes in Computer Science*, pages 697–705, Lyon, France, August 1996. Springer-Verlag.

[60] Jonathan M. D. Hill, Stephen R. Donaldson, and Tim Lanfear. Process migration and fault tolerance of *BSPlib* programs running on networks of workstations (distinguished paper). In *EuroPar'98*, volume 1470 of *Lecture Notes in Computer Science*, Southampton, UK, September 1998. Springer-Verlag.

[61] Jonathan M. D. Hill, Stephen R. Donaldson, and Alistair McEwan. Installation and user guide for the Oxford BSP toolset (v1.4) implementation of *BSPlib*. Technical report included with the *BSPlib* disttribution (`http://www.comlab.ox.ac.uk/oucl/oxpara/bsp/bspmodel.htm`), 1998.

[62] Jonathan M. D. Hill, Stephen R. Donaldson, and David Skillicorn. Stability of communication performance in practice: from the Cray T3E to networks of workstations. Technical Report PRG-TR-33-97, Programming Research Group, Oxford University Computing Laboratory, October 1997.

[63] Jonathan M. D. Hill, Stephen R. Donaldson, and David B. Skillicorn.

Portability of performance with the *BSPlib* communications library. In *Programming Models for Massively Parallel Computers, (MPPM'97)*, pages 33–42, London, November 1997. IEEE Computer Society Press.

[64] Jonathan M. D. Hill, Stephen Jarvis, Constantinos Siniolakis, and Vasil P. Vasilev. Analysing an SQL application with a *BSPlib* call-graph profiling tool. In *EuroPar'98*, volume 1470 of *Lecture Notes in Computer Science*, Southampton, UK, September 1998. Springer-Verlag.

[65] Jonathan M. D. Hill, Stephen Jarvis, Constantinos Siniolakis, and Vasil P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, pages 286–292. IEEE Computer Society Press, January 1998.

[66] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998. ISSN 0167-8191.

[67] Jonathan M. D. Hill and David Skillicorn. Lessons learned from implementing BSP. *Journal of Future Generation Computer Systems*, 13 (4–5):327–335, April 1998.

[68] Jonathan M. D. Hill and David B. Skillicorn. Practical barrier synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, pages 438–444. IEEE Computer Society Press, January 1998.

[69] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[70] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[71] Guy Horvitz and Rob H. Bisseling. Designing a BSP version of ScaLAPACK. Preprint 1074, Department of Mathematics, Utrecht University, Utrecht, the Netherlands, July 1998.

[72] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *IEEE Computer Society's Seventh International Computer Software & Applications Conference (COMPSAC'83)*, pages 627–631, November 1983.

[73] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computation.* McGraw-Hill, 1998.

[74] Antoine Le Hyaric. Converting the NAS benchmarks from MPI to

BSP. Technical report, Oxford University Computing Laboratory, 1997. Available from `ftp://ftp.comlab.ox.ac.uk/pub/Packages/BSP/NASfromMPItoBSP.tar`.

[75] *IBM AIX parallel environment: Parallel Programming Subroutine Reference.* IBM, 2 edition, June 1994. Order Number: SH26-7228-01.

[76] Joseph Jájá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[77] B. H. H. Juurlink and H. A. G. Wijshoff. Experiences with a model for prallel computation. In *12th ACM Symposium on Principles of Distributed Systems*, pages 87–96. ACM, ACM Press, 1993.

[78] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S Tanenbaum. Transparent fault-tolerance in parallel orca programs. In *Proc. Symp. on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, 1992.

[79] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 17, pages 869–941. The MIT Press/Elsevier, 1990.

[80] Tom Kilburn. The Manchester University digital computing machine. In *Report on a Conference on High Speed Automatic Calculating Machines 22–35 June 1949*, Volume 14 in the Charles Babbage Institute Reprint Series for the History of Computing, pages 138–146 (in the reprint). MIT and Tomash, 1950.

[81] Peter J. B. King. *Computer and Communication Systems Performance Modelling.* International series in Computer Science. Prentice Hall, 1990.

[82] Leonard Kleinrock. *Queueing Systems, Volume I: Theory.* Wiley-Interscience, New York, 1975.

[83] Simon Knee. Program development and performance prediction on BSP machines using Opal. Technical Report PRG-TR-18-94, Oxford University Computing Laboratory, August 1994.

[84] F. Thomsom Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

[85] W. F. McColl. General purpose parallel computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, volume 4 of *Cambridge International Series on Parallel Computation*, pages 337–391. Cambridge University Press, Cambridge, 1993.

[86] W. F. McColl. Special purpose parallel computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, volume 4 of *Cambridge International Series on Parallel Computation*, pages 261–336. Cambridge University Press, Cambridge, 1993.

[87] W. F. McColl and Q. Miller. The GPL language: Reference manual. Technical report, ESPRIT GEPPCOM Project, Oxford university Computing Laboratory, October 1995.

[88] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison Wesley, 1996.

[89] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[90] Message Passing Interface Forum. *MPI A Message-Passing Interface Standard*, May 1994.

[91] Mier Communucations Inc. Product lab testing comparison: 10/100BASET switches, April 1998.

[92] Richard Miller. *Two approaches to architecture-independent parallel computation.* D.Phil thesis, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, Michaelmas Term 1994.

[93] Marvin L. Minsky. *Computation: Finite and Infinite Machines.* Series in Automatic Computation. Prentice Hall, 1967.

[94] Ronald Mraz. Reducing the variance of point-to-point transfers for parallel real-time programs. *IEEE Parallel and Distributed Technology*, 24:20–31, Winter 1994.

[95] National Aeronautics and Space Administration. Summary of recent network performance on davinci cluster. `http://science.nas.nasa.gov/Groups/LAN/cluster/latresults/sumtab.recent.html`, 1996.

[96] Mohan V. Nibhanupudi and Boleslaw K. Szymanski. Adaptive parallelism in the bulk synchronous parallel model. In *EuroPar'96*, volume 1124 of *Lecture Notes in Computer Science*, pages 311–318, Lyon, France, August 1996. Springer-Verlag.

[97] Gregory F Pfister. *In Search of Clusters.* Prentice Hall, 1998.

[98] Jon Postel. Internet protocol. RFC 791, Information Sciences Institue, University of Southern California, 4676 Admiralty Way, Ma-

rina del Rey, California 90291, USA, September 1981. Available via anonymous ftp from `ds.internic.net`.

[99] Jon Postel. Transmission control protocol. RFC 793, Information Sciences Institue, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, USA, September 1981. Available via anonymous ftp from `ds.internic.net`.

[100] Loic Prylli and Bernard Tourancheau. BIP: A new protocol for high performance networking on Myrinet. In *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 472–485. Springer-Verlag, 1998.

[101] Joy Reed, Kevin Parrott, and Tim Lanfear. Portability, predictability and performance for parallel computing: BSP in practice. *Concurrency: Practice and Experience*, 8(10):799–812, December 1996.

[102] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[103] A. W. Roscoe. Model checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in honour of C.A.R. Hoare*, pages 353–378. Prentice Hall, 1994.

[104] Sheldon M. Ross. *Introduction to Probability Models.* Academic Press, Inc., sixth edition, 1997.

[105] Alessandro Rubini. *Linux Device Drivers.* O'Reilly and Associates, 1998.

[106] John E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison Wesley, 1998.

[107] Andrew Simpson, Jonathan M. D. Hill, and Stephen R. Donaldson. BSP in CSP: easy as ABC. In *International Parallel Processing Symposium (IPPS'99): Workshop on Formal Methods for Parallel Programming: Theory and Applications*, volume 1586 of *Lecture Notes in Computer Science*, pages 1299–1313. Springer-Verlag, 1999.

[108] David Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.

[109] William Stallings. *Data and Computer Communications.* Prentice Hall, fifth edition, 1997.

[110] Richard M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. AI memo 519A, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), 1979.

[111] G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press. ISBN 0-8186-7255-2.

[112] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *ICPP'95: International Conference on Parallel Processing*, August, 1995.

[113] C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Rice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker. The SP2 high-performance switch. *IBM Systems Journal*, 34(2):185–204, 1995.

[114] Kasidit Chanchio Xian-He Sun. Efficient process migration for parallel processing on non-dedicated networks of workstations. Technical Report TR-96-74, Institute for Computer Applications in Science and Engineering, December 1996.

[115] Mark R. Swanson and Leigh B. Stoller. Low latency workstation cluster communications using sender-based protocols. Technical Report UUCS-96-001, Department of Computer Science, University of Utah, Salt Lake City, UT 84112, USA, 1996.

[116] C. P. Tan, W. F. Wong, and C. K. Yuen. tmPVM — task migratable PVM. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Ditsributed Processing IPPS/SPDP 1999: April 12–16, 1999, San Juan, Puerto Rico*, pages 196–202a, 1999.

[117] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Upper Saddle River, NJ 07458, third edition, 1996.

[118] Shuji Tasaka. *Performance Analysis of Multiple Access Protocols.* Computer Systems Series. MIT Press, 1986.

[119] Alan M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[120] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[121] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 18, pages 943–971. The MIT Press/Elsevier, 1990.

[122] Manish Verma and Tzi-cker Chiueh. Pupa: A low-latency communication system for Fast Ethernet, April 1998. Workshop on Personnel Computer Based Network of Workstations held at the 12th International Parallel Processing Symposium and the 9th Symposium on Parallel and Distributed Processing.

[123] Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch. Low-latency communication over ATM networks using Active Messages. Technical report, Department of Computer Science, Cornell University, Ithaca, NY 14850, 1995.

[124] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, volume 20(2) of *ACM SIGARCH Computer Architecture News*. ACM Press, May 1992.

[125] Jingwen Wang, Songnian Zhou, Khalid Ahmed, and Weihong Long. LSBATCH: A distributed load sharing batch system. Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, April 1993.

[126] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Low-latency communication over Fast Ethernet. In *EuroPar'96 Parallel Processing: Volume I*, volume 1123 of *Lecture Notes in Computer Science*, pages 187–194. Springer-Verlag, 1996.

[127] Gary R. Wright and W. Richard Stephens. *TCP/IP Illustrated, Volume 2.* Addison-Wesley, 1995.