# Contents

1 Introduction ................................................................. 5  
   1.1 Sequential Computing ............................................ 6  
   1.2 Universal Computing ............................................. 8  

2 The BSP Model ................................................................... 11  
   2.1 Supersteps ............................................................ 11  
   2.2 Cost Modelling ...................................................... 12  
   2.3 BSP - Some Frequently Asked Questions ..................... 13  

3 Parallel Programming .......................................................... 21  
   3.1 BSP Programming .................................................... 21  
   3.2 Message Passing ..................................................... 22  
   3.3 Data Parallelism ..................................................... 23  

4 BSPlib - A Standard BSP Programming Library ...................... 24  
   4.1 Programming Primitives ........................................... 25  
       4.1.1 Creating BSP Processes .................................. 26  
       4.1.2 Superstep Synchronisation ............................... 27  
       4.1.3 Bulk Synchronous Remote Memory Access ............. 27  
       4.1.4 Bulk Synchronous Message Passing ................... 31  
       4.1.5 Raising an Error and Halting ......................... 34  
       4.1.6 Timing Routine ............................................. 34  
   4.2 Commentary on BSPlib ............................................. 34  
       4.2.1 The BSP Communication Spectrum .................. 34  
       4.2.2 Buffering Semantics of DRMA Operations .......... 35  
       4.2.3 Bulk Synchronous Message Passing ................. 38  
       4.2.4 Collective Communications ............................ 39  
       4.2.5 Subset Synchronisation and BSP ................... 40
4.3 Implementations ........................................... 41
4.4 Benchmarking ............................................ 41

5 Networks and Routing Methods 44
5.1 Interconnection Networks ............................... 44
5.2 Comparison Networks .................................... 48
  5.2.1 Merging ........................................... 49
  5.2.2 Sorting ........................................... 50
5.3 Routing Methods ......................................... 51
5.4 Oblivious Routing ....................................... 51
5.5 Randomised Routing .................................... 52
5.6 Routing on Random Networks ......................... 53
5.7 Routing on Sorting Networks .......................... 53
5.8 Optical Communication .................................. 54
  5.8.1 Solving The VLSI Wiring Problem ............... 54
  5.8.2 Randomised Optical Routing ...................... 55
5.9 Networks, Routing and BSP ............................ 57

6 Structured Communications 59
6.1 Broadcasting ............................................ 59
6.2 Combining ............................................... 59
6.3 Prefix Sums ............................................. 60

7 BSP Scheduling of Parallel Computations 61
7.1 Diamond DAG ........................................... 61
  7.1.1 Solution of Triangular Systems ................. 62
  7.1.2 String Comparison ................................ 62
7.2 Cube DAG ............................................... 63
  7.2.1 Matrix Multiplication ............................ 64
  7.2.2 LU Decomposition ............................... 64
  7.2.3 Algebraic Path Problem ......................... 66

8 Design and Analysis of BSP Algorithms 69
8.1 Dense Matrix-Vector Multiplication ................. 69
8.2 Matrix Multiplication ................................ 70
8.3 Strassen’s Algorithm .................................. 71
8.4 Fast Fourier Transform ............................... 74
9 Scientific Computing

9.1 Linear algebra in Scientific Computing ......................... 81
9.1.1 Iterative Methods .............................................. 83
9.1.2 Partial Differential Equations ............................... 83
9.1.3 Molecular Modelling .......................................... 84
9.2 The MLIB Test Set of Sparse Matrices ......................... 86
9.3 Sparse Matrix-Vector Multiplication ........................... 90
9.4 Results for Structure Independent Distributions ............... 95
9.5 Results for Structure Dependent Distributions ................ 100
9.6 Asymptotic Analysis .............................................. 106
9.7 Conclusions .......................................................... 107

10 The PRAM Model and Virtual Shared Memory ................. 109

10.1 The PRAM Model ................................................. 109
10.2 Circuits ............................................................. 111
10.3 Shared Memory Algorithms ................................... 112
  10.3.1 Addition ..................................................... 112
  10.3.2 Polynomial Evaluation .................................... 113
  10.3.3 Prefix Sums ................................................. 115
  10.3.4 Matrix Multiplication ..................................... 116
  10.3.5 Linear Recurrences ........................................ 117
  10.3.6 Algebraic Path Problem .................................. 118
10.4 The Class $\mathcal{NC}$: Parallelisable Problems ............... 119
10.5 $P$-Completeness: Inherently Sequential Problems? .......... 121
10.6 Work Efficient PRAM Algorithms ............................... 123
10.7 PRAM Simulation .................................................. 125
  10.7.1 Hashing ...................................................... 126
  10.7.2 Combining .................................................... 128
10.8 BSP and the PRAM Model ....................................... 129
Preface

Scalable computing will, over the next few years, become the normal form of computing. In this course we present a unified framework, based on the BSP model, which aims to serve as a foundation for this evolutionary development. A number of important techniques, tools and methodologies for the design of sequential algorithms and programs have been developed over the past few decades. In the transition from sequential to scalable computing we will find that new requirements such as universality and predictable performance will necessitate significant changes of emphasis in these areas. Programs for scalable computing, in addition to being fully portable, will have to be efficiently universal, offering high performance, in a predictable way, on any general purpose parallel architecture. The BSP model provides a discipline for the design of scalable programs of this kind.
Chapter 1

Introduction

Over the last thirty years or so we have seen the emergence and development of a huge global software industry. A large part of that industry is concerned with the production of portable applications software for the wide variety of sequential computers which are currently in use, from personal computers to large mainframes. The roots of this success story can be traced back to the work of von Neumann in the 1940s and to the developments in the areas of high level languages and compilers which followed on from that work in the 1950s and early 1960s. The model of a general purpose sequential computer proposed by von Neumann has served as the basic framework for almost all of the sequential computers which have been produced from the late 1940s to the present time. The stability which the model has provided has been crucial to the growth of the software industry over the years.

Since the earliest days of computing it has been clear that, sooner or later, sequential computing would be superseded by parallel computing. This has not yet happened, despite the availability of numerous parallel machines and the insatiable demand for increased computing power. For parallel computing to become the normal form of computing we require a model which can play a similar role to that which the von Neumann model has played in sequential computing. The emergence of such a model would stimulate the development of a new parallel software industry, and provide a clear focus for future hardware developments. For a model to succeed in this role it must offer three fundamental properties:

scalability - the performance of software and hardware must be scalable from a single processor to several hundreds or thousands of processors.
**portability** - software must be able to run unchanged, with high performance, on any general purpose parallel architecture.

**predictability** - the performance of software on different architectures must be predictable in a straightforward way.

It should also, ideally, permit the correctness of parallel programs to be determined in a way which is not much more difficult than for sequential programs.

During the last thirty years, a large number of parallel models have been proposed. They include: SIMD parallelism, synchronous message passing, logic programming, graph reduction, dataflow, and various forms of cache-based virtual shared memory. None of these approaches has, however, achieved all three main requirements.

The BSP model, which we describe in this course, is quite different from many of the approaches which have been proposed in the past. It decouples the two fundamental aspects of parallel computation - communication and synchronisation. This decoupling is the key to achieving universal applicability across the whole range of parallel architectures. Recent research on BSP algorithms, architectures and languages has demonstrated convincingly that the BSP model can achieve all three of the requirements mentioned above.

### 1.1 Sequential Computing

For fifty years, sequential computing has been the normal form of computing. From the early proposal of von Neumann [17] to the present day, sequential computing has grown relentlessly, embracing new technologies as they have emerged and discarding them whenever something better came along. Today it is a huge global industry. The two parts of that industry, hardware and software, are quite different. Most sequential hardware has a life span of a few years. New technologies continually offer new ways of realising the same basic design in a form which is cheaper, more powerful, or both. In contrast, sequential software systems often take many years to develop, and are expected to last for a very long time, preferably for ever.

The key reason for the spectacular success of sequential computing has been the widespread, almost universal, adoption of the basic model proposed by von Neumann. To see why this happened it is necessary to go back further, to the work of Turing [86]. In his theoretical studies, Turing demonstrated
that a single general purpose sequential machine could be designed which would be capable of efficiently performing any computation which could be performed by a special purpose sequential machine. Stated more concisely, he demonstrated that efficient universality was achievable for sequential computations. The importance of this universality result for subsequent practical developments in computing cannot be overstated. It showed that, for a given computational problem, the additional efficiency advantages which could be gained by designing a special purpose sequential machine for that problem would not be great. Around 1944, von Neumann produced his proposal for a general purpose sequential computer which captured the principles of Turing’s work in a practical design. The design, which has come to be known as the “von Neumann computer”, has served as the basic model for almost all sequential computers produced from the late 1940s to the present time. As noted in [40], “The paper by Burks, Goldstine and von Neumann ([17]) was incredible for the period. Reading it today, one would never guess this landmark paper was written more than 40 years ago, as most of the architectural concepts seen in modern computers are described there.” For an account of the principles of modern general purpose sequential (i.e. von Neumann) computer design, see [40]. The stability and universality provided by the von Neumann model has permitted and encouraged the development of high level languages and compilers. These, in turn, have created a large and diverse software industry producing portable applications software which will run with high performance on any sequential computer.

For sequential computing, the von Neumann model has given the two parts of the computing industry what they require to succeed. The hardware industry is provided with a focus for its technological innovation, and the confidence that, since the model is stable, software companies will find that the time and effort involved in developing software can be justified. The software industry meanwhile sees the stability of the model as providing a basis for the high level, cost-effective development of applications software, and also providing a guarantee that the software thus produced will not become obsolete with the next technological change. The stability of the underlying model has also allowed the development of a robust complexity theory for sequential computation, and a set of algorithm design and software development techniques of wide applicability.
1.2 Universal Computing

General purpose parallel computing systems come in a variety of forms. We have various kinds of distributed memory architectures, shared memory multiprocessors, and clusters of workstations. New technologies may increase this range still further. Can one hope to design portable and scalable parallel software in the face of such architectural diversity? In this course we show that it is indeed possible to produce fully portable parallel software which will run with highly efficient, scalable and predictable performance on any general purpose parallel architecture. In this chapter we briefly consider the past, present and future of parallel architectures.

In recent years, the pursuit of higher performance from computers has forced the introduction of parallel computing in many areas, particularly in scientific and engineering applications and in database systems and transaction processing. The transition from sequential to parallel has not, however, been without its problems. On most of the early commercial parallel machines produced in the 1980s, scalable parallel performance could only be achieved by carefully exploiting the particular architectural details of the machine. Besides being extremely tedious and time consuming in many cases, this usually resulted in parallel applications software which could not be easily adapted to run on other machines. In a world of rapidly changing and diverse parallel architectures, this architecture dependence of the parallel software was a major weakness and seriously inhibited the growth of the field.

Over the last few years the situation has improved somewhat. For a variety of technological and economic reasons, the various classes of parallel computer in use (distributed memory machines, shared memory multiprocessors, clusters of workstations) have been steadily becoming more and more alike. The economic advantages of using standard commodity components has been a major factor in this convergence. Other influential factors have been the need to efficiently support a single address space on distributed memory machines for ease of programming, and the need to replace buses by networks to achieve scalability in shared memory multiprocessors. These various pressures have acted to produce a rapid and significant evolutionary restructuring of the parallel computing industry.

There is now a growing consensus that for a combination of technological, commercial, and software reasons, we will see a steady evolution over the next few years towards a “standard” architectural model for scalable
parallel computing. It is likely to consist of a collection of (workstation-like) processor-memory pairs connected by a communications network which can be used to efficiently support a global address space. As with all such successful models, there will be plenty of scope for the use of different designs and technologies to realise such systems in different forms depending on the cost and performance requirements sought.

The simplest, cheapest, and probably the most common architectures will be based on clusters of personal computers. The Intel Pentium Pro microprocessor is an example of a high volume, commodity microprocessor for the personal computer market which provides hardware support for multiprocessing. Using such commodity microprocessor components, various companies are now producing very low cost multiprocessor systems for use as parallel servers. Off-the-shelf networking technologies such as ATM and new scalable software systems for communications will allow these to be assembled into clusters which will provide a very low cost option for many high performance commercial, scientific and internet applications.

At the other end of the spectrum, there will continue to be a small group of companies producing very large, very powerful and very expensive parallel supercomputer systems for those with applications which require those computing resources. The CRAY T3D is a good example of such a system. It is a very powerful distributed memory architecture based on the DEC Alpha microprocessor. In addition to offering high bandwidth global communications, it has several specialised hardware mechanisms which enable it to efficiently support parallel programs which execute in a global address space [6]. The mechanisms include hardware barrier synchronisation and direct remote memory access. The latter permits each processor to get a value directly from any remote memory location in the machine, and to put a value directly in any remote memory location. This is done in a way which avoids the performance penalties normally incurred in executing such operations on a distributed memory architecture, due to processor synchronisation and other unnecessary activities in the low level systems software. The companies which compete at the top end of the market will, as today, focus their attention not only on highly optimised architectural design, but also on new, and perhaps expensive, technologies which can offer increased performance. For example, some might use optical technologies to achieve more efficient global communications than could be achieved with VLSI systems.

The implementation of a global address space on a distributed memory architecture requires an efficient mechanism for the distributed routing of
data communications through the network of processors. A number of efficient routing and memory management techniques have been developed for this problem, see e.g. [61, 88, 89]. Consider the problem of packet routing on a \( p \)-processor network. Let an \( h \)-relation denote a routing problem where each processor has at most \( h \) packets to send to various processors in the network, and where each processor is also due to receive at most \( h \) packets from other processors. Here, a packet is one word of information, such as e.g. a real number or an integer. Using two-phase randomised routing one can, for example, show that every \((\log p)\)-relation can be realised on a \( p \) processor hypercube in \( O(\log p) \) steps. In [30] a simple and practical randomised method of routing \( h \)-relations on an optical communications system is described. The optical system is physically realistic and the method requires only \( O(h + \log p \log \log p) \) steps. In [74] a simple and very efficient protocol for routing \( h \)-relations using only the total-exchange primitive is described.

We will discuss networks and routing methods in Chapter 5.

The process of architectural convergence which has been described brings with it the hope that we can, over the next few years, establish parallel computing as the standard method of computing, and begin to see the growth of a large and diverse global parallel software industry similar to that which currently exists for sequential computing. The main goal of that industry will be to produce scalable programs which, in addition to being fully portable, will offer high performance, in a predictable way, on any general purpose parallel architecture. The BSP model provides a discipline for the design of universal programs of this kind.
Chapter 2

The BSP Model

2.1 Supersteps

In architectural terms, the BSP model is essentially the standard model described above. A bulk synchronous parallel (BSP) computer [63, 88] consists of a set of processor-memory pairs, a global communications network, and a mechanism for the efficient barrier synchronisation of the processors. A BSP computer operates in the following way. A computation consists of a sequence of parallel supersteps, where each superstep consists of a sequence of steps, followed by a barrier synchronisation at which point all data communications will be completed. During a superstep, each processor can perform a number of computation steps on values held locally at the start of the superstep, send and receive a number of messages, and handle various remote read and write requests.

Although we have described the BSP computer as an architectural model, one can also view bulk synchrony as a programming model or, indeed, as a kind of programming methodology. The essence of the BSP approach to parallel programming is the notion of the superstep, in which communication and synchronisation are completely decoupled. A “BSP program” is simply one which proceeds in phases, with the necessary global communications taking place between the phases. This approach to parallel programming is applicable to all kinds of parallel architecture: distributed memory architectures, shared memory multiprocessors, and networks of workstations. It provides a consistent, and very general, framework within which to develop portable parallel software for scalable parallel architectures.
Since communication and synchronisation are decoupled in a BSP program, the programmer does not have to worry about problems such as deadlock, which can occur with synchronous message passing. Debugging a BSP program is also made much easier by this decoupling. The barrier at the end of a superstep provides an appropriate breakpoint at which the global state of the parallel computation is well defined and can be interrogated. Debugging and reasoning about the correctness of a BSP program are, therefore, not much more difficult than for a sequential program.

The BSP computer is a two-level memory model, i.e. each processor has its own physically local memory module; all other memory is non-local, and is accessible in a uniformly efficient way. By uniformly efficient, we mean that the time taken for a processor to read from, or write to, a non-local memory element in another processor-memory pair should be independent of which physical memory module the value is held in. The algorithm designer and the programmer should not be aware of any hierarchical memory organisation based on network locality corresponding to the particular structure of the communications network.

2.2 Cost Modelling

If we define a time step to be the time required for a single local operation, i.e. a basic operation (such as addition or multiplication) on locally held data values, then the performance of any BSP computer can be characterised by three parameters: \( p = \) number of processors; \( l = \) number of time steps for barrier synchronisation; \( g = \) \((\text{total number of local operations performed by all processors in one second})/\text{(total number of words delivered by the communications network in one second, in a situation of continuous traffic)}\). There is also, of course, a fourth parameter \( s \), the number of time steps per second. However, since the other parameters are normalised with respect to that one, it can be ignored in the design of algorithms and programs. The parameter \( g \) corresponds to the frequency with which non-local memory accesses can be made; in a machine with a higher value of \( g \) one must make non-local memory accesses less frequently. More formally, \( g \) is related to the time required to realise \( h \)-relations in a situation of continuous message traffic; \( g \) is the value such that an \( h \)-relation can be performed in \( h \cdot g \) time steps. Any parallel computing system can be regarded as a BSP computer, and can be benchmarked accordingly to determine its BSP parameters \( l \) and
g. The BSP model is therefore not prescriptive in terms of the physical architectures to which it applies. Every general purpose parallel architecture can be viewed by an algorithm designer or programmer as simply a point \((p, l, g)\) in the space of all BSP machines.

The time for a superstep \(S\) is determined as follows. Let the work \(w\) be the maximum number of local computation steps executed by any processor during \(S\). Let \(h_s\) be the maximum number of messages sent by any processor during \(S\), and \(h_r\) be the maximum number of messages received by any processor during \(S\). The time for \(S\) is then at most \(w + g \cdot \max\{h_s, h_r\} + l\) steps. The total time required for a BSP computation is easily obtained by adding the times for each superstep. Analysing and predicting the cost of a BSP program is, therefore, no more difficult than analysing and predicting the cost of a sequential program.

By adding the time for each superstep we obtain an expression of the form \(W + H \cdot g + S \cdot l\) where \(W, H, S\) will typically be functions of \(n\) and \(p\). In designing an efficient BSP algorithm or program for a problem which can be solved sequentially in time \(T(n)\) our goal will, in general, be to produce an algorithm requiring total time \(W + H \cdot g + S \cdot l\) where \(W(n, p) = T(n)/p, H(n, p)\) and \(S(n, p)\) are as small as possible, and the range of values for \(p\) is as large as possible. In many cases, this will require that we carefully arrange the data distribution so as to minimise the frequency of remote memory references. Another property of interest in BSP algorithm design is the space (or memory) efficiency of the computation. We will use \(M(n, p)\) to denote the maximum number of values which any one processor has to store at any point during the computation.

2.3 BSP - Some Frequently Asked Questions

What is Bulk Synchronous Parallelism?

Bulk Synchronous Parallelism, or BSP, is a style of parallel programming developed for general-purpose parallelism, that is parallelism across all application areas and a wide range of architectures [61]. Its goals are more ambitious than most parallel programming systems which are aimed at particular kinds of applications, or work well only on particular classes of parallel architectures [62]. BSP’s most fundamental properties are:
• *It is simple to write.* BSP programs are much the same as sequential programs. Only a bare minimum of extra information needs to be supplied to describe the use of parallelism.

• *Independent of target architectures.* Unlike many parallel programming systems, BSP is designed to be architecture-independent, so that programs run unchanged when they are moved from one architecture to another. Thus BSP programs are portable in a strong sense.

• *The performance of a program on a given architecture is predictable.* The execution time of a BSP program can be computed from the text of the program and a few simple parameters of the target architecture. This makes design possible, since the effect of a decision on performance can be determined at the time it is made.

BSP achieves these properties by raising the level of abstraction at which programs are written and implementation decisions made. Rather than considering individual threads of execution and individual communication actions, BSP considers computation and communication at the level of the entire program and executing computer. Determining the *bulk* properties of a program, and the *bulk* ability of a particular computer to satisfy them makes it possible to design with new clarity. One way in which BSP is able to achieve this abstraction is by renouncing locality as a performance optimisation. This simplifies many aspects of both program and implementation design, and in the end does not adversely affect performance for most application domains. It does mean that parallel programming systems that do exploit locality will outperform BSP on architectures that are similarly able to exploit locality. This happens much less often than might be expected naively.

**What does the BSP programming style look like?**

BSP programs have both a horizontal structure and a vertical structure. The horizontal structure arises from concurrency, and consists of a fixed number of virtual threads. These are each associated, at run-time, with a physical processor. The vertical structure arises from the progress of a computation through time. For BSP, this is a sequential composition of global *supersteps*, which conceptually occupy the full width of the executing architecture. Each superstep is further subdivided into three ordered phases consisting of:
A superstep is shown in Figure 2.1. We will use $p$ to denote the virtual parallelism of a program, that is the number of threads it uses. If the target parallel computer has fewer processors than the virtual parallelism, an extension of Brent’s theorem [16] can be used to transform the BSP program into a slimmer version.

**How does communication work?**

Most parallel programming systems handle communication, both conceptually and in terms of implementation, at the level of individual communication actions: memory-to-memory transfers, send/receive messages, or active messages. However, this level is difficult to work at because there are many
simultaneous communication actions in a parallel program, and their inter-
actions make it hard to say much about the time each single one will take to
complete. Considering communication actions \emph{en masse} both simplifies their
treatment and makes it possible to bound the time it will take to deliver a
whole set of data. BSP does this by considering all of the communication
actions of a superstep as a unit. Each processor has some set of outgoing mes-
sages and receives a set of incoming messages. The BSP cost model charges
time $h \cdot g$ for completing an $h$-relation, where $g$ is a (bulk) parameter of an
architecture. It measures the ability of the architecture’s interconnection net-
work and routing techniques to deliver data under conditions of continuous
network traffic. It is a measure of the permeability of the communication
mechanism to data. The parameter $g$ is almost always a function of the
number of processors of the target machine. Unsurprisingly it typically takes
longer to deliver an $h$-relation on a larger machine.

\textbf{Surely this isn’t a very precise measure of how long
communication takes? Don’t hot spots and congestion
make it very inaccurate?}

One of the most difficult problems of determining the performance of conven-
tional messaging systems is precisely that congestion makes upper bounds
hard to determine and quite pessimistic. BSP avoids this difficulty in a cou-
ple of ways. First, notice that the definition of an $h$-relation and the linear
dependence of the cost of communication on $h$ mean that balanced com-
munication will be much faster than unbalanced. A communication pattern
in which each processor sends a message to a single distinct processor is a
1-relation, and takes time $g$. A communication pattern in which every pro-
cessor sends a message to processor 0 is a $p$-relation, and hence much more
expensive, taking time $p \cdot g$. Thus the cost model does take into account
congestion phenomena arising from the limits on each processor’s capacity
to send and receive data.

However, it is still possible that an apparently-balanced communication
pattern might generate hot spots in some region of the interconnection topol-
ogy. BSP avoids this in several ways. First, the placement of threads in
processors is random, so that patterns arising from the problem domain tend
to be broken up. Second, routing techniques that avoid localised congestion
are preferred. These include randomised routing, in which particular kinds
of randomness are introduced into the choice of route for each communication action, and adaptive routing, in which data are diverted from their normal route in a controlled way to avoid congestion. Even if congestion occurs, $g$ captures its effect on continuous message traffic unless it is extremely sensitive to patterns.

It is also the case that today’s problems have much more virtual parallelism than the size of today’s parallel computers. When a problem with locality is mapped onto a much smaller number of processors, the contribution of locality to the overall traffic pattern becomes small in comparison to that of the total volume of data being moved. Experiments have shown that $g$ is an accurate measure of the cost of moving large amounts of data on a wide range of existing parallel computers.

**Most parallel computers have a considerable cost associated with starting up communication.** Doesn’t this mean that the cost model is inaccurate for small messages, since $g$ doesn’t account for start-up costs?

Yes, but it only matters when the total volume of data in a superstep is small. The size of individual messages doesn’t matter. A superstep in which a particular thread sends 10000 10-byte messages is counted as costing the same as if it has sent two 50000-byte messages — they both cost $100000 \cdot g$. The implementation has only to guarantee that the messages will have been delivered by the end of the superstep. Therefore it can, and does, buffer them in time so that a single 100000-byte message is sent. So either description of the communication to be done in the program results in exactly the same communication taking place. Therefore both forms of program will have the same cost. Nevertheless, a superstep that only requires a small amount of data in total to be moved, that is an $h$-relations with very small $h$, will be slightly more expensive than expected. $h$ really does have to be very small. A program in which a thread sends 10000 10-byte messages in independent supersteps will be vastly more expensive than one in which a single 100000-byte message is sent in a single superstep. This is because there is a cost to ensuring that all of the program’s threads have reached a consistent state at the end of each superstep, which is done by requiring each superstep to end with a barrier synchronisation. The cost of executing a barrier synchronisation is given by an architecture parameter $l$, which again
is a function of the size of the executing architecture. Clearly a BSP program should always use as few supersteps as possible to avoid the cost of barrier synchronisations.

**How do these parameters allow the cost of programs to be determined?**

The cost of a single superstep is the sum of three terms: the cost of the local computations on each processor, the cost of the global communication of an $h$-relation, and the cost of the barrier synchronisation at the end of the superstep. Thus the cost is given by

$$\text{cost of a superstep} = w + h \cdot g + l$$

where $w$ is the maximum of the execution time of thread computations. To make this sum meaningful, and to allow comparisons between different parallel computers, the parameters $w$, $g$, and $l$ are expressed in terms of the basic instruction execution rate of the target architecture. The cost of an entire BSP program is just the sum of the cost of each superstep. This cost model makes it clear what strategies should be adopted to write efficient BSP programs:

- balance the computation between threads, since $w$ is a maximum over computation times;
- balance the communication between threads, since $h$ is a maximum over fan-in and fan-out of data; and
- minimise the number of supersteps, since this determines the number of times $l$ appears in the final cost.

The cost model also shows how the cost model can be used to predict performance on a particular target architecture. The values of $w$ and $h$ for each superstep, and the number of supersteps can be determined by inspection of the program code. Values of $p$, $g$, and $l$ can then be inserted to give execution time before the program is executed. The cost model can be

- used as part of the design process for BSP programs;
• used to predict the performance of programs ported to new parallel computers; and

• used to guide buying decisions for parallel computers if the BSP program characteristics of typical workloads are known.

Other cost models for BSP have been proposed. For example, communication and computation could conceivably be overlapped, giving a superstep cost of the form

$$\text{MAX}(w, gh) + l$$

It is also sometimes argued that the cost of an h-relation is limited by the time taken to send h messages and then receive h messages, so that the communication term should be of the form

$$g(h_{\text{in}} + h_{\text{out}})$$

All of these variations alter costs by no more than small constant factors, so we will use the original form of cost model in the interests of simplicity and clarity.

A more important omission from the original cost model was any restriction on the amount of memory required at each processor. While the original cost model encourages balance in communication and limited barrier synchronisation, it encourages profligate use of memory. In this course, we extend the original cost model to include memory requirements.

The BSP cost model makes it possible to use BSP to design algorithms, not just programs. Here the goal is to build solutions that are optimal with respect to total computation, total communication, and total number of supersteps over the widest possible range of values of p. Designing a particular program involves choosing algorithms that are optimal for the range of machine sizes envisaged for the application. For example, two BSP algorithms for matrix multiplication are described in Chapters 7 and 8. The first has BSP complexity

$$\frac{n^3}{p} + \left(\frac{n^2}{p^{1/2}}\right)g + p^{1/2}l$$

requiring memory at each processor of size $n^2/p$. This is optimal in time and memory requirement. The second has BSP complexity

$$\frac{n^3}{p} + \left(\frac{n^2}{p^{2/3}}\right)g + l$$
requiring memory at each processor of size $n^2/p^{2/3}$. This is optimal in time, communication, and supersteps, but requires more memory at each processor. Therefore the choice between these two algorithms in an implementation may well depend on the relationship between the size of problem instances and the memory available on processor of the target architecture.

**Is BSP a programming discipline, or a programming language, or something else?**

BSP is a model of parallel computation. It can be expressed by a wide variety of programming languages and systems. For example, BSP programs could be written using existing communication libraries such as PVM [29] or MPI [38]. All that is required is that they provide communication mechanisms and a way to implement barrier synchronisation. However, the values of $g$ and $l$ depend not only on the hardware performance of the target architecture but also on the amount of software overhead required to achieve the necessary behaviour, so systems not designed with BSP in mind may not provide good values of $g$ and $l$, i.e. they will give poor performance. BSP is concerned with high-level structure of computations. Therefore it does not prescribe the way in which local computations are carried out, nor how communication actions are expressed. All existing BSP languages are imperative, but there is no intrinsic reason why this need be so. The most common approach to BSP programming is SPMD imperative programming using Fortran or C, with BSP functionality provided by library calls. Two BSP libraries have been in use for some years: the Oxford BSP Library [67] and the Green BSP Library [34]. A standard has recently been agreed for a library called BSPLib [35]. BSPLib contains operations for delimiting supersteps, and two variants of communication, one based on direct remote memory access, and the other on buffered message passing. Other BSP languages have been developed, including GPL [64] and Opal [47].
Chapter 3
Parallel Programming

3.1 BSP Programming

In this section we briefly describe the main characteristics of BSP programming. We also compare the BSP approach with two other approaches to parallel programming - data parallelism and message passing.

As was noted earlier, the essence of the BSP approach to parallel programming is the notion of the superstep, in which communication and synchronisation are completely decoupled. A “BSP program” is simply one which proceeds in phases, with the necessary global communications taking place between the phases. One simple way of specifying the data communications in a BSP program is to use remote memory access primitives. The operation \texttt{put} deposits locally held data into a remote memory area on another process. The \texttt{get} operation reaches into the local memory of another process to copy values held there into a data structure in its own local memory. The \texttt{put} and \texttt{get} operations are both one-sided communication primitives. They do not require the active participation of the other process. In accordance with BSP superstep semantics, they are also both non-blocking. All put and get operations initiated during a superstep will be completed before the start of the next superstep.

Bulk synchronous remote memory access is a very convenient style of programming for BSP computations which can be statically analysed in a straightforward way. It is less convenient for computations where the volumes of data being communicated between supersteps is irregular and data dependent, and where the computation to be performed in a superstep de-
pends on the quantity and form of data received at the start of that superstep. A more appropriate style of programming in such cases is bulk synchronous message passing. In bulk synchronous message passing, a non-blocking send operation is used to transfer values held locally into a buffer on the destination process. The values are guaranteed to be in the remote buffer before the start of the next superstep, and can be safely inspected and manipulated by the receiving process at that time.

3.2 Message Passing

Since the early 1980s, message passing has been the dominant programming approach in the area of parallel computing.

In recent years, the PVM message passing library [29] has been widely implemented and widely used. In that respect, the goal of source code portability in parallel computing has already been achieved by PVM. What then, are the advantages of BSP programming, if any, over a message passing framework such as PVM? On shared memory architectures and on modern distributed memory architectures with powerful global communications, message passing models such as PVM are likely to be less efficient than the BSP model, where communication and synchronisation are decoupled. This will be especially true on those modern distributed memory architectures which have hardware support for direct remote memory access (or one-sided communications). PVM and all other message passing systems based on pairwise, rather than barrier, synchronisation also suffer from having no simple analytic cost model for performance prediction, and no simple means of examining the global state of a computation for debugging.

MPI [38] has been proposed as a new standard for those who want to write portable message passing programs in Fortran and C. At the level of point-to-point communications (send, receive etc.), MPI is similar to PVM, and the same comparisons apply. [The MPI standard is very general and appears to be very complex relative to the BSP model. However, one could use some carefully chosen combination of the various non-blocking communication primitives available in MPI, together with its barrier synchronisation primitive, to produce an MPI based BSP programming model.] At the higher level of collective communications, MPI provides support for various specialised communication patterns which arise frequently in message passing programs. These include broadcast, scatter, gather, total exchange, reduc-
tion, scan etc. These standard communication patterns also arise frequently in the design of BSP algorithms. It is important that such structured patterns can be conveniently expressed and efficiently implemented in any BSP programming language, in addition to the more primitive operations such as put and get which generate arbitrary and unstructured communication patterns. There has been one attempt to compare BSP performance with MPI [69] on a network of workstations. The results show that performance differences are very small, of the order of a few percent.

Comparing it to PVM and MPI, it might be argued that the BSP approach offers (a) a simple programming discipline (based on supersteps) which makes it easier to determine the correctness of programs, (b) a cost model for performance analysis and prediction which is simpler and compositional, and (c) more efficient implementations on many machines.

3.3 Data Parallelism

Data parallelism is an important niche within the field of scalable parallel computing. A number of interesting programming languages and elegant theories have been developed in support of the data parallel style of programming, see e.g. [82]. High Performance Fortran [49] is a good example of a practical data parallel language. The BSP approach, as outlined in this paper, aims to offer a more flexible and general style of programming than is provided by data parallelism. The two approaches are not, however, incompatible in any fundamental way. For some applications, the increased flexibility provided by the BSP approach may not be required and the more limited data parallel style may offer a more attractive and productive setting for parallel software development, since it frees the programmer from having to provide an explicit specification of the various processor scheduling, communication and memory management aspects of the parallel computation. In such a situation, the BSP cost model can still play an extremely important role in terms of providing an analytic framework for performance prediction of the data parallel program.
Chapter 4

BSPlib - A Standard BSP Programming Library

The Cray T3D SHMEM library provides primitives for direct remote memory access which can be used for BSP programming. The Oxford BSP Library [67] and the Oxford BSP Toolset [42] both provide a similar set of programming primitives for bulk synchronous remote memory access. The Green BSP Library [34] provides a set of bulk synchronous message passing primitives based on fixed sized packets. Considerable experience of BSP programming has been gained through the use of these various libraries. A number of major projects in universities and in industry are now using them to develop parallel applications, see e.g. [42, 69]. The experience gained in these practical projects would appear to confirm the various claims made above, regarding BSP and its advantages over message passing.

In December 1995, the inaugural meeting of BSP Worldwide was held in Oxford. BSP Worldwide is a new global organisation to coordinate research and development activities in the area of BSP computing, and to work on the standardisation of programming tools for the growing number of software developers who are now adopting this approach. BSP Worldwide has recently launched an initiative to produce a standard low level BSP programming library for use with sequential languages such as Fortran and C. An initial proposal for this library is given in [35]. Its main characteristics are as follows:

- Single Program Multiple Data (SPMD) parallelism.
- Primitives for buffered and unbuffered bulk synchronous remote memory access.
• Primitives for buffered bulk synchronous message passing with tagged messages.

• Primitives for address registration to (a) support data communications into static, stack and heap allocated data structures, and (b) support BSP programming in heterogeneous environments.

A number of features which are semantically well defined, such as nested parallelism and subset synchronisation, have been excluded from the initial version of the library since they can have an adverse effect on the predictability of performance.

4.1 Programming Primitives

BSPLib provides the operations shown in Table 4.1.

<table>
<thead>
<tr>
<th>Class</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation</td>
<td>bsp_init</td>
<td>Simulate dynamic processes</td>
</tr>
<tr>
<td></td>
<td>bsp_begin</td>
<td>Start of SPMD code</td>
</tr>
<tr>
<td></td>
<td>bsp_end</td>
<td>End of SPMD code</td>
</tr>
<tr>
<td>Enquiry</td>
<td>bsp_pid</td>
<td>Find my process id</td>
</tr>
<tr>
<td></td>
<td>bsp_nprocs</td>
<td>Number of threads</td>
</tr>
<tr>
<td></td>
<td>bsp_time</td>
<td>Local time</td>
</tr>
<tr>
<td>Synchronisation</td>
<td>bsp_sync</td>
<td>Barrier synchronisation</td>
</tr>
<tr>
<td>DRMA</td>
<td>bsp_pushregister</td>
<td>Make region globally visible</td>
</tr>
<tr>
<td></td>
<td>bsp_proprgister</td>
<td>remove global visibility</td>
</tr>
<tr>
<td></td>
<td>bsp_put</td>
<td>Push to remote memory</td>
</tr>
<tr>
<td></td>
<td>bsp_pop</td>
<td>Pull from remote memory</td>
</tr>
<tr>
<td>BSMP</td>
<td>bsp_set_tag_size</td>
<td>Choose tag size</td>
</tr>
<tr>
<td></td>
<td>bsp_send</td>
<td>Send to remote queue</td>
</tr>
<tr>
<td></td>
<td>bsp_get_tag</td>
<td>Match tag with message</td>
</tr>
<tr>
<td></td>
<td>bsp_move</td>
<td>Fetch from queue</td>
</tr>
<tr>
<td>Halt</td>
<td>bsp_abort</td>
<td>One process halts all</td>
</tr>
<tr>
<td>High Performance</td>
<td>bsp_hpput</td>
<td>Unbuffered versions</td>
</tr>
<tr>
<td></td>
<td>bsp_hppget</td>
<td>of communication</td>
</tr>
<tr>
<td></td>
<td>bsp_hppmove</td>
<td>primitives</td>
</tr>
</tbody>
</table>

Table 4.1: Core BSP operations

There are operations to:

• set up a BSP program;

• discover properties of the environment in which each process is executing;
• participate in a barrier synchronisation;
• communicate, either directly into or out of a remote memory, or using a message queue;
• abort a computation from anywhere inside it; and
• communicate in a high-performance unbuffered mode.

BSPLib can be used with Fortran and C. Several simpler interfaces, which are mapped to BSPLib calls but which hide some of the detail for specialised application domains, are under development. Another higher-level library provides specialised collective communication operations. These are not considered as part of the core library, but they can be easily realised in terms of the core. These include operations for broadcast, scatter, gather, and total exchange.

4.1.1 Creating BSP Processes

Processes are created in a BSPLib program by the operations \texttt{bsp} \texttt{begin} and \texttt{bsp} \texttt{end}. There can only be one instance of a \texttt{bsp} \texttt{begin}/\texttt{bsp} \texttt{end} pair within a program, although there are two different ways to start a BSPLib program: If \texttt{bsp} \texttt{begin} and \texttt{bsp} \texttt{end} are the first and last statements in a program, then the entire BSPLib computation is SPMD.

In an alternative mode a single process starts execution and determines the number of parallel processes required for the calculation. It then spawns the required number of processes using \texttt{bsp} \texttt{begin}. Execution of the spawned processes then continue in an SPMD manner, until \texttt{bsp} \texttt{end} is encountered by all the processes. At that point, all but process zero is terminated, and process zero is left to continue the execution of the rest of the program sequentially. One problem with providing this mode is that some parallel machines available today, for example almost all distributed-memory machines, e.g. IBM SP2, Cray T3D, Meiko CS-2, Parsytec GC, Hitachi SR2001, do not provide dynamic process creation. Therefore we \textit{simulate} dynamic spawning using an operation \texttt{bsp} \texttt{init} which takes as its argument a procedure name. The procedure named in \texttt{bsp} \texttt{init} must contain \texttt{bsp} \texttt{begin} and \texttt{bsp} \texttt{end} as its first and last statements. The C interface for these library operations is

26
void bsp_init(void (*startproc)(void));
void bsp_begin(int maxprocs);
void bsp_end()

where

maxprocs is the number of processes requested by the user.

startproc is the name of a procedure that contains bsp_begin and bsp_end as its first and last statements.

There are also operations to determine total number of processes and for each process to identify which it is. The interface for these operations is:

    int bsp_nprocs();
    int bsp_pid();

If the function bsp_nprocs is called before bsp_begin, then it returns the number of processors which are available. If it is called after bsp_begin it returns \( n \), the actual number of processes allocated to the program, where \( 1 \leq n \leq maxprocs \), and maxprocs is the number of processes requested in bsp_begin. Each of the \( n \) processes created by bsp_begin has a unique associated value \( m \) in the range \( 0 \leq m \leq n - 1 \). The function bsp_pid returns the associated value of the process executing the function call.

### 4.1.2 Superstep Synchronisation

A BSPLib calculation consists of a sequence of supersteps. The end of one superstep and the start of the next is identified by a call to the library procedure bsp_sync with interface

    void bsp_sync();

### 4.1.3 Bulk Synchronous Remote Memory Access

There are two ways of communicating between threads: one using direct remote memory access (DRMA), and the other using a BSP version of message passing. The DRMA communication operations are defined for stack and heap allocated data structures as well as for static data. This is achieved
by only allowing a process to reference only certain *registered* areas of a remote memory. In this registration procedure, processes use the operation `bsp_pushregister` to announce the address of the start of their local area which is available for global remote use. The operation `bsp_put` pushes locally held data into a registered remote memory area on a target process, without the active participation of the target process.

```c
void bsp_[hp]put(
    int pid,
    const void *src,
    void *dst,
    int offset,
    int nbytes);
```

where

- **pid** is the identifier of the process where data is to be stored.

- **src** is the location of the first byte to be transferred by the put operation. The calculation of `src` is performed on the process that initiates the put.

- **dst** is the base address of the area where data is to be stored. It must be a previously registered data area.

- **offset** is the displacement in bytes from `dst` to which `src` will copy. The calculation of `offset` is performed by the process that initiates the put.

- **nbytes** is the number of bytes to be transferred from `src` into `dst`. It is assumed that `src` and `dst` are addresses of data structures that are at least `nbytes` in size.

The operation `bsp_get` reaches into the registered local memory of another process to copy data values held there into a data structure in its own local memory.
void bsp_[hp]get(
    int pid,
    const void *src,
    int offset,
    void *dst,
    int nbytes);

where

pid is the identifier of the process where data is to be obtained from.

src is the base address of the area from which data will be obtained. src must be a previously registered data-structure.

offset is an offset from src. The calculation of offset is performed by the process that initiates the get.

dst is the location of the first byte where the data obtained is to be placed. The calculation of dst is performed by the process that initiates the get.

nbytes is the number of bytes to be transferred from src into dst. It is assumed that src and dst are addresses of data structures that are at least nbytes in size.

A BSPLib program consists of p processes with identical code but different local memory. Thus there are p instances of each variable, sharing the same name but not, in general, having the same physical address because of differences in heap allocation. To allow BSPLib programs to execute correctly we require a mechanism for relating these various addresses. The method used is chosen to enable procedures called within supersteps to be written in a modular way, able to register their own common locations without being aware of regions registered in the calling environment. Registration takes effect at the next barrier synchronisation.

void bsp_pushregister (void *region, int nbytes);
void bsp_popregister (void *region);

where
region is the starting address of the region to be registered or unregistered.

nbytes is the size of the region (used for range checking).

Each processor maintains a stack of registration slots. Each bsp_pushregister associates a region with the next available slot, and each bsp_popregister invalidates the association at the top of the stack. Variable names associated with the same slot on their local registration stacks are associated with each other. A reference to a registered name in a put or get is translated to the remote variable whose name is in the same slot on the remote processor. Here is an example:

Process 0:

```c
int x;
bsp_pushregister(&x, sizeof(int));
bsp_sync;
x := 3;
bsp_put(1, x, x, 0, sizeof(int));
bsp_sync;
```

Process 1:

```c
int y;
bsp_pushregister(&y, sizeof(int));
bsp_sync;
```

Process 0 registers x in the first slot, while Process 1 registers y in the first slot. When Process 0 executes a put, using x as the destination region name, this is mapped to the region whose address is associated with the first slot in Process 1. Therefore, the variable y has the value 3 placed in it as the result of the put. The same, or overlapping, regions may be registered in more than one slot. Because the slots are a stack, processes must unregister regions in the reverse order to that in which they were registered.

The semantics adopted for BSPLib bsp_put communication is buffered locally/buffered remotely. While the semantics is clean and safety is maximized, puts may unduly tax the memory resources of an implementation, thus preventing large transports of data. Consequently, BSPLib also provides
a high performance put operation bsp_hpput whose semantics is unbuffered locally/unbuffered remotely. The use of this operation requires care as correct data delivery is only guaranteed if neither communication nor local/remote computations modify either the source or the destination areas during the superstep. The main advantage of this operation is its economical use of memory. It is therefore particularly useful for applications which repeatedly transfer large data sets. The bsp_get and bsp_hpget operations reach into the local memory of another process and copy previously-registered remote data held there into a data structure in the local memory of the process that initiated them.

4.1.4 Bulk Synchronous Message Passing

In BSPLib, bulk synchronous message passing is based on the idea of two-part messages, a fixed-length part carrying tagging information that will help the receiver to interpret the message, and a variable-length part containing the main data payload. We will call the fixed-length portion the tag and the variable-length portion the payload. In C programs, either part could be a complicated structure. The length of the tag is required to be fixed during any particular superstep, but can vary between supersteps. The buffering mode of the BSMP operations is buffered locally/buffered remotely. A collection of messages sent to the same process has no implied ordering at the receiving end. However, since each message may be tagged, the programmer can identify messages by their tag.

The procedure to set tag size must be called collectively by all processes. Moreover, in any superstep where bsp_set_tag_size is called, it must be called before sending any messages.

```c
void bsp_set_tag_size (int *tag_bytes);
```

where

`tag_bytes`, on entry to the procedure, specifies the size of the fixed-length portion of every message from the current superstep until it is updated; the default tag size is zero. On return from the procedure, `tag_bytes` is changed to reflect the previous value of the tag size.

The tag size of incoming messages is prescribed by the outgoing tag size of
the previous step.

The \texttt{bsp\_send} operation is used to send a message that consists of a tag and a payload to a specified destination process. The destination process will be able to access the message during the subsequent superstep. Its interface is

\begin{verbatim}
void bsp_send(int pid,
              const void *tag,
              const void *payload,
              int payload_bytes);
\end{verbatim}

where

- \texttt{pid} is the identifier of the process where data is to be sent.
- \texttt{tag} is a token that can be used to identify the message. Its size is determined by the value specified in \texttt{bsp\_set\_size\_tag}.
- \texttt{payload} is the location of the first byte of the payload to be communicated.
- \texttt{payload\_bytes} is the size of the payload.

It copies both the tag and the payload of the message out of user space into the system before returning. The \texttt{tag} and \texttt{payload} inputs are allowed to be changed by the user immediately after the \texttt{bsp\_send}.

To receive a message, the operations \texttt{bsp\_get\_tag} and \texttt{bsp\_move} are used. The operation \texttt{bsp\_get\_tag} returns the tag of the first message in the buffer.

\begin{verbatim}
void bsp_get_tag(int *status,
                 void *tag);
\end{verbatim}

where

- \texttt{status} returns -1 if the system buffer is empty. Otherwise it returns the length of the payload of the first message in the buffer. This length can be used to allocate an appropriately sized data structure for copying the payload using \texttt{bsp\_move}.
- \texttt{tag} is unchanged if the system buffer is empty. Otherwise it is assigned the tag of the first message in the buffer. The operation \texttt{bsp\_move} copies the payload of the first message in the buffer.
buffer into payload, and removes that message from the buffer. Its interface is

```c
void bsp_move(void *payload,
               int reception_nbytes);
```

where

- `payload` is an address to which the message payload will be copied. The buffer is then advanced to the next message.
- `reception_nbytes` specifies the size of the reception area where the `payload` will be copied into. At most `reception_nbytes` will be copied into `payload`. Note that `bsp_move` serves to flush the corresponding message from the buffer, while `bsp_get_tag` does not. This allows a program to get the tag of a message (as well as the payload size in bytes) before obtaining the payload of the message. It does, however, require that even if a program only uses the fixed-length tag of incoming messages the program must call `bsp_move` to get successive message tags. `bsp_get_tag` can be called repeatedly and will always copy out the same tag until a call to `bsp_move`.

```c
int bsp_hpmove(void **tag_ptr_buf, void **payload_ptr_buf);
```

where

- `bsp_hpmove` is a function which returns -1, if the system buffer is empty. Otherwise it returns the length of the payload of the first message in the buffer and (a) places a pointer to the tag in `tag_ptr_buf`; (b) places a pointer to the payload in `payload_ptr_buf`; and (c) removes the message (by advancing a pointer representing the head of the buffer).

We note that since messages are referenced directly they must be properly aligned and contiguous. This puts additional requirements on the library implementation that would not be there without this feature. The storage referenced by these pointers remains valid until the end of the current superstep.
4.1.5 Raising an Error and Halting

The function `bsp_abort` can be used to print an error message followed by a halt of the entire BSPLib program. The routine is designed *not to* require a barrier synchronisation of all processes. A single process in a potentially unique thread of control can therefore halt the entire BSPLib program.

```c
void bsp_abort(char* format,...);
```

where

`format` is a C-style format string as used by `printf`. Any other arguments are interpreted in the same way as the variable number of arguments to `printf`.

4.1.6 Timing Routine

The function `bsp_time` provides access to a high-precision timer—the accuracy of the timer is implementation specific. The function is a local operation of each process, and can be issued at any point after `bsp_begin`. The result of the timer is the time in seconds since `bsp_begin`. The semantics of `bsp_time` is as though there were `bsp_nprocs` timers, one per process. BSPLib does *not impose any synchronisation requirements between the timers in each process*.

```c
double bsp_time();
```

4.2 Commentary on BSPLib

4.2.1 The BSP Communication Spectrum

A wide variety of communication operations can be used in a BSP computation. There is a *spectrum* between the leanest and most efficient mechanisms, and those which are more convenient for the programmer. At one end of the spectrum, unbuffered bulk synchronous direct remote memory access (DRMA) provides the highest performance. However it requires the programmer to consider the existence and location of data structures on the destination process where data is communicated into, and to ensure that the data on the process initiating the DRMA is not changed during the lifetime of the superstep (see §4.2.2). At the other end of the spectrum, buffered
bulk synchronous message passing (BSMP) is more flexible, as it relieves the programmer from both of these requirements. There is no need to consider the existence or placement of data at the destination of the communication, as BSMP communicates into a system buffer from where messages are copied from by the user at the end of the superstep. There is also no need to consider if the data on the initiating process is changed during the lifetime of the superstep as it is buffered.

How large are the efficiency differences across the BSP spectrum from unbuffered DRMA to buffered BSMP? Any overheads with BSMP will be due to the buffering of communication at the sending and destination processes. This can be quantified by considering that if $g$ is the asymptotic communication bandwidth for a DRMA operation, then $g + 2m$ is the asymptotic communication bandwidth for a BSMP; where $m$ is the cost of writing a single word into main memory. Since $m$ will never be larger than $g$, the ratio between the costs of BSMP and DRMA will never be more than three. For most machines, the ratio will be much less than this.

4.2.2 Buffering Semantics of DRMA Operations

There are four forms of buffering with respect to the DRMA put operations:

**Buffered remotely:** Data communication into registered areas will only take effect at the end of the superstep. Therefore, the registered remote area may be safely operated on during a superstep, although any changes will be overwritten at the end of a superstep if data is communicated there.

**Unbuffered remotely:** Data communication into registered areas can take effect at any time during the superstep.

**Buffered locally:** The data communicated from a process will contain the values which are held at the time the operation was called\(^1\). Therefore, the process may reuse its storage area during the superstep.

**Unbuffered locally:** The data transfer resulting from a call to a communication operation may occur at any time between the time of the call and the end of the superstep. Therefore, for safety, no process

\(^1\)more precisely, this is the time when the communication operation is apparent to the process holding the data.
should change the data structures used in this communication during
the course of the superstep.

The various buffering choices are crucial in determining the safety of the
communication operation, i.e., the conditions which guarantee correct data
delivery as well as its effects on the computation taking place on the pro-
cessors involved in the operation. However, it should be noted that even
the most cautious choice of buffering mode does not completely remove the
effects of non-determinism from a remote process. For example, if more than
one process transfers data into overlapping memory locations, then the result
at the overlapping region will be nondeterministically chosen; it is implement-
tion dependent which one of the many “colliding” communications should
be written into the remote memory area.

Buffered DRMA operations maximise safety, ease the debugging process
and simplify reasoning about programs. Indeed, a DRMA communication
operation which is unbuffered remotely is potentially dangerous due to the
added nondeterminism as data delivery can take effect at any time during
a superstep. This puts a major burden on the programmer to determine
when DRMA operations do not interfere with data that is being manipulated
locally within a superstep. Furthermore, if a communication operation is
unbuffered locally, the local data may be changed by remote communications
or by local data manipulations, before being sent. Thus, if BSP programs do
not adhere to separating remote accesses from local manipulation of data, the
added nondeterminism will complicate the programming and the debugging
processes. Buffering does not, however, come without cost. Each buffered
operation requires twice the amount of memory of it’s unbuffered alternative,
and thus it limits the size of data that can be used in a given application. In
turn, this may limit the usefulness of the library.

The solution adopted by BSPLib is to provide a set of maximally safe
primitives, bsp.put and bsp.get, which can be used to develop and test
BSPLib programs. These operations have a clean, simple semantics and
allow a convenient entry point for new BSP programmers. For high per-
formance programming, BSPLib offers a set of fully unbuffered primitives,
bsp_hpput and bsp_hppget, which have the potential of performing commu-
nication very efficiently, both in terms of time and space. Users can employ
these operations from the outset or as replacements, after they have devel-
oped and tested their programs. In either case, their use requires careful
programming in order to avoid undesirable effects due to nondeterminism.
int *good_total_exchange(int x) {
  int i, *result;
  result=calloc(bsp_nprocs(),
                sizeof(int));

  bsp_pushregister(result, bsp_nprocs()*sizeof(int));
  bsp_sync();

  for(i=0;i<bsp_nprocs();i++) {
    bsp_hpput(i,
              &x,,
              result,
              bsp_pid()*sizeof(int),
              sizeof(int));
  }
  bsp_sync();
  bsp_popregister(result);
  return result;
}

Figure 4.1: Well-defined BSPLib program

When using the high-performance communication primitives bsp_hpget and bsp_hpput that are not buffered locally, it is important that those data structures are available at the end of the superstep. The lifetime of a superstep may span the scope of many procedures. It is therefore quite possible to initiate a communication from a stack allocated data-structure, in a situation where the data structure is out of scope at the end of the superstep. For example, consider the procedure in Figure 4.1 that performs an all-to-all communication. The array result is first allocated and then registered at the start of the procedure. A for-loop is then used to copy the value x local to each process into the registered arrays result in every process at the bsp_pid()th position. The similar procedure shown in Figure 4.2 is however ill-defined. As the value of y communicated by bsp_hpput only has scope local to the procedure do_store because of the call-by-value semantics of C-procedure invocation, the variable will not be in scope when the communication actually occurs at the second bsp_sync within bad_total_exchange. This example shows that the user has to be careful when communicating from
int *bad_total_exchange(int x) {
    int i, *result;
    result = calloc(bsp_nprocs(), sizeof(int));

    bsp_pushregister(result, bsp_nprocs() * sizeof(int));
    bsp_sync();

    for(i=0;i<bsp_nprocs();i++) {
        do_store(i, x, result);
    }
    bsp_sync();
    bsp_popregister(result);
    return result;
}

void do_store(int pid, int y, int *result){
    bsp_hpput(pid, &y,
              &result,
              bsp_pid() * sizeof(int),
              sizeof(int));
}

Figure 4.2: Ill-defined BSPLib program

data structures that do not persist for the lifetime of the program. This is because communication may be delayed until the barrier synchronisation at the end of each superstep.

4.2.3 Bulk Synchronous Message Passing

In the BSMP system, the primary design decisions were as follows:

- A message consists of two parts: a fixed-length tag and a variable-length payload.
- The length of the tag can vary between supersteps, but all messages sent in a superstep must have the same tag size.
• The messages are buffered locally: after the tag and payload of a message are sent, the user can safely change the memory that stored the tag and payload.

We address the two-part message decision first. Tags are associated with messages for identification purposes. Users are allowed to select a tag size and format that accommodates their application. The programmer can conveniently label each message to identify the order in which the messages were sent, the source process, the data type of the payload, or whatever information is appropriate for the identification of each message. It is in general convenient to use tags that are of a different data type than the payload. Since FORTRAN does not support mixed-type data structures, multi-part messages are required. For C, it would not have been necessary to treat a tag as a structure that is separate from the payload of the message. However, even for C the use of a separate tag is advantageous when transporting large arrays. We allow the tag size to change to allow for greater modularity in BSP programs. That is, each subroutine or even each superstep can set the tag as is appropriate for the information that is communicated during the superstep. The fact that a subroutine may not know the tag size of the calling procedure required that \texttt{bsp\_set\_tag\_size} return the old tag size so that a subroutine can reset the tag size upon exit as necessary.

We now discuss some issues concerning implementation. The semantics of the BSMP primitives are classified as \textit{buffered locally/buffered remotely}, but the underlying implementation can vary greatly for different systems. The natural implementation is to buffer messages at the destination. Some implementations may choose to buffer messages at the source as well in order to bundle messages and make better use of network bandwidth. Other implementations will immediately send messages through the network and avoid physical source buffering.

### 4.2.4 Collective Communications

Some message passing systems, such as MPI [38], provide primitives for various specialised communication patterns which arise frequently in message passing programs. These include broadcast, scatter, gather, total exchange, reduction, prefix sums (scan), etc. These standard communication patterns also arise frequently in the design of BSP algorithms. It is important that such structured patterns can be conveniently expressed
and efficiently implemented in a BSP programming system, in addition to the more primitive operations such as put and get which generate arbitrary and unstructured communication patterns. The library we have described can easily be extended to support such structured communications by adding \texttt{bsp\_broadcast}, \texttt{bsp\_combine}, \texttt{bsp\_scatter}, \texttt{bsp\_gather}, \texttt{bsp\_scan}, \texttt{bsp\_exchange}, etc. as higher level operations. These could be implemented in terms of the core operations, or directly on the architecture if that was more efficient.

4.2.5 Subset Synchronisation and BSP

Although it may sometimes appear to be convenient to have the possibility of groups of processes synchronising independently, we believe that this can significantly increase the difficulty of BSP cost analysis. The essence of BSP cost modeling is that the cost of a series of supersteps is simply the sum of the costs of each separate superstep. If subset synchronisation is allowed, then the cost of each group should probably be calculated independently (using its own \textit{p}, \textit{g} and \textit{l}?). The total cost of the computation will then be determined as some function of the costs of the groups (and their various \textit{p}, \textit{g} and \textit{l} parameters?).

Over the last few years, various research projects on BSP languages have explored more flexible forms of synchronisation in BSP computations. The experience thus far suggests that in most cases, the advantages are modest or nonexistent, while the disadvantages (particularly in terms of cost analysis) are usually very considerable. We therefore feel that inclusion of subset synchronisation in this basic library would be unwise. The wide variety of possible forms of subset synchronisation seems to provide another compelling reason for avoiding it at the core level. Should subset synchronisation be: static or dynamic?, if dynamic then on a superstep by superstep basis?, if static then one level of groups?, two levels?, many levels?, recursion and nested parallelism?, synchronisation across groups?, etc.

Synchronizing a subset of executing processes is a complex issue because the ability of an architecture to synchronize is not necessarily a bulk property of an architecture in the sense that its processing power and communication resources are. Certain architecture provide a special hardware mechanism for barrier synchronisation across all of the processors. For example the Cray T3D provides an add and broadcast tree, and work at Purdue University has created generic, fast, and cheap barrier synchronisation hardware for a wide
range of architectures. Sharing this single synchronisation resource among several concurrent subsets that may wish to use it at any time seems difficult. We are currently exploring this issue.

Architectures in which barrier synchronisation is implemented in software do not have any difficulty in implementing barriers for subsets of the processors. The remaining difficulty here is a language design one – it is not yet clear what an MIMD, subset synchronising language, and its associated cost model, should be like.

### 4.3 Implementations

During 1996, BSPLib will be implemented on a large number of machines. The implementations will include generic versions which run on any UNIX machine with TCP/IP, or System V Shared Memory primitives. Highly optimised native implementations will been produced for the IBM SP2, the SGI Power Challenge, the CRAY T3D and other machines.

### 4.4 Benchmarking

Using the Oxford BSP toolset implementation of BSPLib, some preliminary benchmarking studies have been carried out by Jon Hill to estimate the values of $l$ and $g$ which a programmer using the Toolset should expect. The following table lists these values.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Mflops</th>
<th>p</th>
<th>l (bops)</th>
<th>l (µs)</th>
<th>g (local) (bops)</th>
<th>g (local) (µs)</th>
<th>g (all-to-all) (bops)</th>
<th>g (all-to-all) (µs)</th>
<th>$N_2$ words</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGI Power Challenge</td>
<td>53</td>
<td>1</td>
<td>226</td>
<td>3.1</td>
<td>0.5</td>
<td>0.007</td>
<td>0.5</td>
<td>0.007</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1132</td>
<td>15.3</td>
<td>9.8</td>
<td>0.13</td>
<td>10.2</td>
<td>0.14</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1496</td>
<td>20.2</td>
<td>8.9</td>
<td>0.12</td>
<td>9.5</td>
<td>0.13</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1960</td>
<td>25.7</td>
<td>9.8</td>
<td>0.13</td>
<td>9.3</td>
<td>0.13</td>
<td>12</td>
</tr>
<tr>
<td>Sun</td>
<td>3.8</td>
<td>1</td>
<td>24</td>
<td>2.4</td>
<td>0.4</td>
<td>0.04</td>
<td>0.4</td>
<td>0.04</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>54</td>
<td>5.3</td>
<td>3.0</td>
<td>0.29</td>
<td>3.4</td>
<td>0.34</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>74</td>
<td>7.4</td>
<td>2.9</td>
<td>0.29</td>
<td>4.1</td>
<td>0.41</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>118</td>
<td>11.7</td>
<td>3.3</td>
<td>0.32</td>
<td>4.1</td>
<td>0.41</td>
<td>11</td>
</tr>
<tr>
<td>IBM SP2 (switch)</td>
<td>25</td>
<td>1</td>
<td>244</td>
<td>9.4</td>
<td>1.3</td>
<td>0.05</td>
<td>1.3</td>
<td>0.05</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1903</td>
<td>73.2</td>
<td>6.3</td>
<td>0.24</td>
<td>7.8</td>
<td>0.30</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3583</td>
<td>137.8</td>
<td>6.4</td>
<td>0.25</td>
<td>8.0</td>
<td>0.31</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>5412</td>
<td>208.2</td>
<td>6.9</td>
<td>0.27</td>
<td>11.4</td>
<td>0.43</td>
<td>6</td>
</tr>
<tr>
<td>Hitachi SR2001</td>
<td>2.3</td>
<td>1</td>
<td>31</td>
<td>5.6</td>
<td>0.2</td>
<td>0.05</td>
<td>0.2</td>
<td>0.05</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1165</td>
<td>216.1</td>
<td>2.6</td>
<td>0.50</td>
<td>3.0</td>
<td>0.54</td>
<td>8</td>
</tr>
</tbody>
</table>

*continued on next page*
Table 4.2: Values of $l$ and $g$ for current machines.

1. Three values are given for $s$:  
   $\lfloor s \rfloor$ is the megaflop rating for a dot-product on two $1,048,576$ element single precision floating point arrays.  
   $\lceil s \rceil$ is the megaflop rating for a dense matrix-matrix multiply on two $300 \times 300$ single-precision floating point matrices.  
   $s$ is the average of the above two benchmarks.

2. All values for $g$ are for communications of 32-bit words.

3. The values of $g$ are reported for two benchmarks: (1) a local neighbour communication where processes communicate to their right in a cyclic manner; (2) a global all-to-all communication. On most machines values for $g$ should scale with the first of these. The latter shows if the machine is truly scalable.

4. All benchmarks were performed at the -O3 optimisation level.

5. The SGI PowerChallenge, IBM SP2, Parsytec GC, and Hitachi SR20001 used native implementations of the toolset.

<table>
<thead>
<tr>
<th>Machine</th>
<th>MFlops</th>
<th>$p$</th>
<th>$l$</th>
<th>$g$ (local)</th>
<th>$g$ (all-to-all)</th>
<th>$N_2$ words</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[s]</td>
<td>s</td>
<td>[s]</td>
<td>[s]</td>
<td>[s]</td>
<td>[s]</td>
</tr>
<tr>
<td>Convex</td>
<td>10.5</td>
<td>1</td>
<td>60</td>
<td>5.8</td>
<td>0.16</td>
<td>0.02</td>
</tr>
<tr>
<td>Exemplar</td>
<td>10.1</td>
<td>1</td>
<td>29</td>
<td>2.9</td>
<td>0.3</td>
<td>0.03</td>
</tr>
<tr>
<td>Digital</td>
<td></td>
<td>2</td>
<td>17202</td>
<td>1703.1</td>
<td>81.1</td>
<td>8.0</td>
</tr>
<tr>
<td>Alpha</td>
<td></td>
<td>2</td>
<td>34356</td>
<td>3401.6</td>
<td>83.0</td>
<td>8.2</td>
</tr>
<tr>
<td>Farm</td>
<td></td>
<td>4</td>
<td>47109</td>
<td>4664.3</td>
<td>81.3</td>
<td>8.1</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>25</td>
<td>27</td>
<td>26</td>
<td>1</td>
<td>241</td>
<td>9.3</td>
</tr>
<tr>
<td>(ethernet)</td>
<td></td>
<td>2</td>
<td>18759</td>
<td>721.5</td>
<td>182.1</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>39025</td>
<td>1500.9</td>
<td>388.2</td>
<td>14.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>88795</td>
<td>3415.2</td>
<td>1246.6</td>
<td>47.3</td>
</tr>
<tr>
<td>Parsytec GC</td>
<td>19.3</td>
<td>1</td>
<td>98</td>
<td>5.1</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>6309</td>
<td>325</td>
<td>109</td>
<td>5.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>23538</td>
<td>1219</td>
<td>190</td>
<td>9.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>29080</td>
<td>1506</td>
<td>252</td>
<td>13.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>224977</td>
<td>11600</td>
<td>253</td>
<td>13.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>123527</td>
<td>6700</td>
<td>272</td>
<td>14.1</td>
</tr>
</tbody>
</table>

... continued from previous page
6. The toolset used on the multiprocessor Sun was built using generic System V shared memory facilities.

7. The Digital Alpha Farm consists of a cluster of Alpha workstations connected via FDDI and a giga-switch. The toolset implementation was built on top of a generic version of MPI (mpich).
Chapter 5

Networks and Routing Methods

5.1 Interconnection Networks

A distributed memory architecture can be thought of as having $p$ processor-memory pairs located at distinct nodes of a $p$-node graph. Each processor can send packets to, and receive packets from, processors at adjacent nodes in the graph. Each edge of the graph can transmit one packet of information in unit time, and has a queue for storing packets that have to be transmitted along it.

A large number of graphs have been proposed as interconnection networks for such multicomputers. Two important parameters of any such graph are its degree, i.e. the maximum number of edges incident at any vertex, and its diameter, i.e. the maximum distance between any pair of vertices, where the distance between two vertices is the length of a shortest path between them. If implemented using conventional VLSI technology, a graph with low degree is likely to have advantages in terms of physical packaging. The advantage of using a graph with small diameter is, of course, that it will permit a packet to be sent quickly between any two vertices in the network. Another important property of any such graph is its bisection width. The bisection width of a graph is the minimum number of edges that have to be removed in order to partition the graph into two parts where the numbers of vertices differ by at most one. The bisection width of a network is often a critical factor in determining the speed with which a multicomputer can
perform a computation. This is due to the fact that for many problems, the data contained in, or computed by, one half of the machine may be needed by the other half before the computation can be completed. A network with low bisection width will suffer from problems of congestion in situations where there is a large amount of data traffic. A fourth property of any such graph, closely related to the bisection width, is the area required for its layout in the VLSI model.

**Theorem 5.1.1 (Thompson [85])** Any VLSI layout of a graph with bisection width $B$ and degree at most four requires area $\Omega(B^2)$.

**Proof.** Suppose the graph can be laid out in a rectangle of height $h$ and width $w$. We can show that there is a line which partitions the layout into two parts where the numbers of vertices differ by at most one, and which runs vertically between the cells, except possibly for a single jog of one cell width. By the definition of the bisection width $B$, the number of edges crossing the line is at least $B - 1$ (at most one edge may cross the horizontal segment). Thus, the height of the layout must be at least $B - 1$. Similarly, the width of the graph must be at least $B - 1$. Therefore, the area must be at least $(B - 1)^2$. ☐

We have now listed four important properties of any graph proposed for the interconnection network of a multicomputer: degree, diameter, bisection width, and area. For our fifth, and final, property we note that ease of programming and avoidance of bottlenecks both suggest that a useful feature for any such graph is that it should look isomorphic from any vertex, i.e. it should be vertex transitive [8]. Let $G = (V, E)$ be an undirected graph, where $V$ is the set of vertices and $E$ is the set of edges. An automorphism of $G$ is a permutation $\pi$ of $V$ such that $(v_1, v_2) \in E$ holds if and only if $(\pi(v_1), \pi(v_2)) \in E$. The set of all automorphisms with the operation of composition forms a group acting on $V$. For example, for the complete graph $K_p$ on $p$ vertices, the group of automorphisms is the symmetric group $S_p$ since any permutation of vertices preserves adjacency. (The empty graph on $p$ vertices also admits $S_p$.) A graph is vertex transitive if it admits a group of automorphisms acting transitively on the vertices [8]. (A permutation group $(H, V)$ is transitive if there is only one orbit in the action of $H$ on $V$. We can check this by picking an element $v \in V$ and looking for elements of $H$ taking $v$ to every other element in $V$.) There is also a related property of edge transitivity but, for simplicity, we will not pursue that notion here. Various
graphs on \( p \) vertices which might be considered as interconnection networks, are listed in the following table, together with information on their degree, diameter, bisection width, area, and vertex transitivity.

<table>
<thead>
<tr>
<th>Name</th>
<th>Degree</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Area</th>
<th>VT?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D array (ring)</td>
<td>2</td>
<td>( p/2 )</td>
<td>2</td>
<td>( \Theta(p) )</td>
<td>yes</td>
</tr>
<tr>
<td>Complete binary tree</td>
<td>3</td>
<td>( 2 \log p )</td>
<td>1</td>
<td>( \Theta(p) )</td>
<td>no</td>
</tr>
<tr>
<td>Shuffle-exchange</td>
<td>3</td>
<td>( 2 \log p )</td>
<td>( \Theta(p/\log p) )</td>
<td>( \Theta(p^{2/3}/(\log p)^{2/3}) )</td>
<td>no</td>
</tr>
<tr>
<td>Cube-connected-cycles</td>
<td>3</td>
<td>( \lfloor 3/2 \rfloor \log p )</td>
<td>( \Theta(p/\log p) )</td>
<td>( \Theta(p^{2/3}/(\log p)^{2/3}) )</td>
<td>no</td>
</tr>
<tr>
<td>2D mesh of trees</td>
<td>3</td>
<td>( 2 \log p )</td>
<td>( \Theta(p^{1/2}) )</td>
<td>( \Theta(p^{2/3}/(\log p)^{2/3}) )</td>
<td>no</td>
</tr>
<tr>
<td>3D mesh of trees</td>
<td>3</td>
<td>( 2 \log p )</td>
<td>( \Theta(p^{2/3}) )</td>
<td>( \Theta(p^{2/3}) )</td>
<td>yes</td>
</tr>
<tr>
<td>1D multigrid</td>
<td>4</td>
<td>( 4 \log p )</td>
<td>( \Theta(\log p) )</td>
<td>( \Theta(p) )</td>
<td>no</td>
</tr>
<tr>
<td>2D array (toroidal)</td>
<td>4</td>
<td>( \Theta(p^{1/2}) )</td>
<td>( \Theta(p^{1/2}) )</td>
<td>( \Theta(p) )</td>
<td>yes</td>
</tr>
<tr>
<td>Tree of meshes</td>
<td>4</td>
<td>( \Theta(p^{1/2}/(\log p)^{2/3}) )</td>
<td>( \Theta(p^{1/2}/(\log p)^{2/3}) )</td>
<td>( \Theta(p \log p) )</td>
<td>no</td>
</tr>
<tr>
<td>Fat tree</td>
<td>4</td>
<td>( \Theta(\log p) )</td>
<td>( \Theta(p^{1/2}/(\log p)^{2/3}) )</td>
<td>( \Theta(p) )</td>
<td>no</td>
</tr>
<tr>
<td>Butterfly (wrapped)</td>
<td>4</td>
<td>( 2 \log p )</td>
<td>( \Theta(p/\log p) )</td>
<td>( \Theta(p^{2/3}/(\log p)^{2/3}) )</td>
<td>yes</td>
</tr>
<tr>
<td>de Bruijn graph</td>
<td>4</td>
<td>( \log p )</td>
<td>( \Theta(p/\log p) )</td>
<td>( \Theta(p^{2/3}/(\log p)^{2/3}) )</td>
<td>no</td>
</tr>
<tr>
<td>Complete quadtree</td>
<td>5</td>
<td>( \log p )</td>
<td>2</td>
<td>( \Theta(p) )</td>
<td>no</td>
</tr>
<tr>
<td>X-Tree</td>
<td>5</td>
<td>( 2 \log p )</td>
<td>( \Theta(\log p) )</td>
<td>( \Theta(p) )</td>
<td>no</td>
</tr>
<tr>
<td>2D multigrid</td>
<td>6</td>
<td>( 3 \log p )</td>
<td>( \Theta(p^{1/2}) )</td>
<td>( \Theta(p) )</td>
<td>no</td>
</tr>
<tr>
<td>3D array (toroidal)</td>
<td>6</td>
<td>( \Theta(p^{1/2}) )</td>
<td>( \Theta(p^{2/3}) )</td>
<td>( \Theta(p^{2/3}) )</td>
<td>yes</td>
</tr>
<tr>
<td>Twin butterfly (wrapped)</td>
<td>8</td>
<td>( 2 \log p )</td>
<td>( \Theta(p/\log p) )</td>
<td>( \Theta(p^{2/3}/(\log p)^{2/3}) )</td>
<td>no</td>
</tr>
<tr>
<td>Pyramid</td>
<td>9</td>
<td>( \log p )</td>
<td>( \Theta(p^{1/2}) )</td>
<td>( \Theta(p) )</td>
<td>no</td>
</tr>
<tr>
<td>Complete 2(k)-ary tree</td>
<td>( 2^k+1 )</td>
<td>( 2 \log_{p/k}(p) )</td>
<td>( 2^k-1 )</td>
<td>( \Theta(4^k p) )</td>
<td>no</td>
</tr>
<tr>
<td>Hypercube</td>
<td>( \log p )</td>
<td>( \log p )</td>
<td>( p/2 )</td>
<td>( \Theta(p^* \log p) )</td>
<td>no</td>
</tr>
<tr>
<td>(p(^{1/2}))-deg. random graph</td>
<td>( p^{1/2} )</td>
<td>2</td>
<td>( \Theta(p^{3/2}) )</td>
<td>( \Theta(p^3) )</td>
<td>no</td>
</tr>
<tr>
<td>Complete graph ((K_p))</td>
<td>( p )</td>
<td>1</td>
<td>( \Theta(p^*) )</td>
<td>( \Theta(p^*) )</td>
<td>yes</td>
</tr>
</tbody>
</table>

If our interconnection network is to be implemented using VLSI technology then, ideally, we would like it to have low degree, low diameter, high bisection width and low area, and also to be vertex transitive. The reason so many networks have been proposed in the last decade is that, of course, these desirable features, to some extent, conflict with one another. The constant bisection width of rings and trees tends to rule them out of serious consideration. Graphs with degree higher than about \( \log p \) tend to be ruled out for multicomputer networks on grounds of packaging. The multigrids, X-Tree and pyramid have been suggested as appropriate for applications in numerical computation, image processing and computer vision. They do, indeed, appear to be well suited to such applications. However, for a broader class of computations, some of which require a large amount of simultaneous non-local communication, the exponentially decreasing bisection width as we move up from the base of the 2D multigrid or pyramid to the top level, is likely to be a severely limiting factor. Arrays have been widely proposed as the best general structure for multicomputers, and a number of practical designs are based on arrays. The main advantage of the 2D and 3D arrays
are their simplicity, and the fact that they can be easily implemented as physical VLSI systems. 2D arrays have optimal $\Theta(p)$ VLSI area and 3D arrays have optimal $\Theta(p)$ VLSI volume. The main disadvantage of arrays, for computations requiring a large amount of non-local communication, is their high diameter. We now briefly discuss some of the properties of the remaining graphs in the table. The shuffle-exchange, cube-connected-cycles, butterfly and de Bruijn graphs have very similar properties. We will refer only to the butterfly. Various fat tree designs have been proposed. They are all essentially improvements of Leighton’s original tree of meshes graph [54]. For that reason, we will not explicitly consider the original tree of meshes graph further here. The 2D and 3D mesh of trees differ only in terms of their dimension. We will consider only the 2D mesh of trees. We thus have reduced our list of candidates to the following.

<table>
<thead>
<tr>
<th>Name</th>
<th>Degree</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Area</th>
<th>V.T.?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D mesh of trees</td>
<td>3</td>
<td>$2 \log p$</td>
<td>$\Theta(p^{1/2})$</td>
<td>$\Theta(p \log^2 p)$</td>
<td>no</td>
</tr>
<tr>
<td>Fat tree</td>
<td>4</td>
<td>$\Theta(\log p)$</td>
<td>$O(p^{1/2}/\log p)$</td>
<td>$\Theta(p)$</td>
<td>no</td>
</tr>
<tr>
<td>Butterfly (wrapped)</td>
<td>4</td>
<td>$2 \log p$</td>
<td>$\Theta(p/\log p)$</td>
<td>$\Theta(p^2/\log^2 p)$</td>
<td>yes</td>
</tr>
<tr>
<td>Twin butterfly (wrapped)</td>
<td>8</td>
<td>$2 \log p$</td>
<td>$\Theta(p/\log p)$</td>
<td>$\Theta(p^2/\log^2 p)$</td>
<td>no</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$\log p$</td>
<td>$\log p$</td>
<td>$p/2$</td>
<td>$\Theta(p^2)$</td>
<td>yes</td>
</tr>
</tbody>
</table>

Leiserson [57] has argued that, in the design of interconnection networks, a major goal should be to achieve area universality in the two dimensional VLSI model, or volume universality in the three dimensional VLSI model. An area universal network is one which, for a given area $A$, can simulate any network of comparable area, with only $O(\log A)$ slowdown (at least with high probability.) It is well known that a $p$-processor hypercube can efficiently simulate any fixed degree $p$-processor network. We need only realise an $h$-relation, where $h$ is the degree of the network. This can be done in $O(\log p)$ steps on a hypercube for any $h \leq \log p$ [89]. If, however, we normalise by area instead of by number of processors, we see that an area $A$ hypercube cannot simulate all area $A$ networks efficiently. For example, since an area $A$ hypercube (butterfly) has only $\Theta(A^{1/2})$ ($\Theta(A^{1/2} \log A)$ respectively) processors (see above table), it cannot simulate a 2D array with area $A$, which has $\Theta(A)$ processors (see table) in any polylogarithmic time. Leiserson has established the following theorem.

**Theorem 5.1.2** The fat tree is an area universal network for the two dimensional VLSI model, and a volume universal network for the three dimensional VLSI model.
For details of the proof of Theorem 5.1.2, see [57]. The design and analysis of efficient algorithms for the fat tree is considered in [58]. Theorem 5.1.2 suggests that if one plans to implement a $p$-processor interconnection network in VLSI, then, for large $p$, one should implement a fat tree rather than a butterfly or a hypercube, despite the fact that it lacks the attractive property of being vertex transitive. (Note. The interconnection networks for the CM5 and for the Meiko CS2 architecture are both based on fat tree designs.)

We have seen that butterflies and hypercubes cannot be efficiently implemented using VLSI technology, i.e. without having long wires for some of the edges in the network. However, optical communication systems, see later, offer the prospect of a dramatic improvement in the efficiency with which such networks can be implemented.

## 5.2 Comparison Networks

It is well known that $O(n \log n)$ binary comparisons are necessary and sufficient to sort $n$ elements drawn from an arbitrary totally ordered set. The lower bound follows from a simple information theoretic argument. The matching upper bound can be obtained, e.g. by a simple recursive mergesort algorithm. A convenient model for the investigation of the parallel complexity of comparison problems such as sorting, merging and selection is the comparison network [7, 48]. Comparison networks have the attractive property that they are oblivious. An oblivious algorithm is one in which the sequence of operations performed is independent of the input data.

A comparison element is a two-input two-output device which computes the minimum $\min(x, y)$ and the maximum $\max(x, y)$ of its inputs $x, y$. In an $n$-line comparison network, the $n$ inputs $<x_1, x_2, \ldots, x_n>$ are presented on the $n$ lines and at each successive level of the network at most $n/2$ disjoint pairs of lines are put through comparison elements. After each level, the $n$ lines carry the inputs in some permuted order. The size of a comparison network is the number of elements.

If we let $l_i$ denote level $i$ and $(j, k)$ denote a comparison element connecting lines $j, k$, then the following is a comparison network of size five which
sorts four elements

\[ L_1 : \{ (1, 2), (3, 4) \} \]
\[ L_2 : \{ (1, 3), (2, 4) \} \]
\[ L_3 : \{ (2, 3) \} \]

i.e. if we present the inputs \( < x_1, x_2, x_3, x_4 > \) on the four lines then after these three levels of comparison elements the values will appear on the lines in sorted order.

The parallel complexity of a comparison network is simply the *depth* of the network, i.e. the number of levels.

### 5.2.1 Merging

Let \( < x_1, x_2, \ldots, x_m > \) and \( < y_1, y_2, \ldots, y_n > \) be two sorted sequences. The *merging problem* is to produce a single sorted sequence consisting of the \( m + n \) elements. This can, of course, be performed by a simple sequential algorithm which uses at most \( m + n - 1 \) binary comparisons. We will describe two efficient parallel comparison networks for merging, both due to Batcher \[7, 48\]. The two techniques are known as odd-even merging and bitonic sorting.

In odd-even merging, we merge the “odd sequences” \( < x_1, x_3, x_5, \ldots > \) and \( < y_1, y_3, y_5, \ldots > \), obtaining \( < v_1, v_3, v_5, \ldots > \); and merge the “even sequences” \( < x_2, x_4, x_6, \ldots > \) and \( < y_2, y_4, y_6, \ldots > \), obtaining \( < w_1, w_2, w_3, \ldots > \). These two merges are performed in parallel. Finally, we apply comparison elements to the pairs \( (w_1, v_2), (w_2, v_3), (w_3, v_4), \ldots \) to complete the merging.

This recursive method yields the following upper bounds:

\[
\text{size}(n) \leq 2 \ast \text{size}(n/2) + O(n) = O(n \log n)
\]

\[
\text{depth}(n) \leq \text{depth}(n/2) + 1 \leq \lceil \log_2 n \rceil
\]

A sequence \( < z_1, z_2, \ldots, z_p > \) is *bitonic* if and only if \( z_1 \geq \cdots \geq z_k \leq \cdots \leq z_p \) for some \( 1 \leq k \leq p \). An *n-line bitonic sorter* is a comparison network which will sort any bitonic sequence of length \( n \). Merging can be performed by sorting the bitonic sequence \( < x_m, x_{m-1}, \ldots, x_1, y_1, y_2, \ldots, y_n > \). Noting that any subsequence of a bitonic sequence is bitonic, it follows that we
can construct an \( n \)-line bitonic sorter by first sorting the two bitonic subsequences \(< z_1, z_3, \ldots >\) and \(< z_2, z_4, \ldots >\) in parallel, and then applying comparison elements to the pairs \((z_1, z_2), (z_3, z_4), \ldots\) to complete the sort. This alternative method yields essentially the same upper bounds on size and depth as odd-even merging. It does, however, have some advantages in terms of simplicity of description. A bitonic sorter with \( 2^n \) lines numbered \( 0, 1, 2, \ldots, 2^n - 1 \) can be defined (nonrecursively) in the following way: Lines \( i, j \) are compared on level \( k \) if and only if \( i, j \) differ only in their \( k \)th most significant bit.

It is quite easy to prove that both the size and depth of these merging networks are optimal to within a constant factor \([48]\). This shows that, for the problem of merging, comparison networks are much less powerful than general (adaptive) algorithms, at least when one compares the total number of comparisons performed. As we shall see in the next section, this does not apply in the case of the related problem of sorting.

### 5.2.2 Sorting

An \( n \)-line sorting network can be constructed recursively using either of the efficient merging networks described above \([7, 48]\). To sort the set \( x_1, x_2, \ldots, x_n \) we first (recursively) sort the two subsets \( x_1, x_2, \ldots, x_{n/2} \) and \( x_{(n/2)+1}, x_{(n/2)+2}, \ldots, x_n \) in parallel. The two sorted sequences can then be combined using one of the above merging networks of size \( O(n \log n) \) and depth \( O(\log n) \). This yields the following upper bounds for sorting \( n \) elements by a comparison network.

\[
\begin{align*}
\text{size}(n) &\leq 2 \times \text{size}(n/2) + O(n \log n) \\
&= O(n \log^2 n) \\
\text{depth}(n) &\leq \text{depth}(n/2) + \log n \\
&= O(\log^2 n)
\end{align*}
\]

From 1968 until 1983 this was the best known oblivious sorting algorithm. In 1983, Ajtai, Komlos and Szemeredi \([3]\) succeeded in producing a remarkable \( n \)-line sorting network of size \( O(n \log n) \) and depth \( O(\log n) \), both of which are of course asymptotically optimal. A more efficient version, which improves the constant factors involved, has since been produced by Paterson \([72]\). Although the constant factors are still too large to make the networks
competitive with those obtained from, say, odd-even merging, this is a result of major theoretical significance as it shows that for the important problem of sorting, adaptive algorithms are not (asymptotically) more powerful than oblivious ones.

5.3 Routing Methods

A large amount of work has been done in recent years on the development of efficient routing methods, on the efficient embedding of one network in another, and on the demonstration of work-preserving emulations of one network by another. We will focus our attention here on the routing problem.

We consider the problem of routing $h$-relations on a $p$-processor network. We are interested in the development of distributed routing methods in which the routing decisions made at a node at some point in time are based only on information concerning the packets that have already passed through the node at that time. In the nondistributed case where global information is available everywhere, the problem of routing is easier and well understood.

5.4 Oblivious Routing

Let us first consider deterministic methods for distributed routing. We define a routing method to be oblivious if the path taken by each packet is entirely determined by its source and destination. (Note that the use of the term oblivious here is slightly different from its use in the context of comparison networks.) It is known [15] that, for a $1$-relation no deterministic oblivious routing method can do better than $\Omega(p^{1/2}/d)$ time steps, in the worst case, for any degree $d$ graph. The most obvious examples of deterministic oblivious approaches are greedy methods in which one sends all packets to their destination by a shortest path through the network. For $1$-relations, the performance of greedy routing on a butterfly can be summarised as follows. All $1$-relations can be realised in $O(p^{1/2})$ steps, which, as we have observed, is an optimal worst case bound for any such fixed degree network. A large number of $1$-relations which arise in practical parallel computation, e.g. the bit-reversal permutation and the transpose permutation, provably require $\Theta(p^{1/2})$ steps. What about the “average case”? Define a random $1$-mapping to be the routing problem where each processor has a single packet
which is to be sent to a random destination. Greedy routing of a random 1-mapping on a butterfly will terminate in $O(\log p)$ steps. Moreover, the fraction of all random 1-mappings which do not finish in $O(\log p)$ steps is incredibly small, despite the fact that most of the 1-relations which seem to arise in practice do not finish in this time. We can probably conclude from these results that “typical” routing problems, in a practical sense, is a rather different concept from “typical” routing problems in a mathematical sense. The performance of greedy routing on a hypercube is very similar to the case of the butterfly. All 1-relations can be realised in $O(p^{1/2}/\log p)$ steps, which is an optimal worst case bound for any $\log p$ degree network. For the average case, where each packet has a random destination, greedy routing will terminate in $O(\log p)$ steps. In the case of the hypercube, there are exponentially many shortest paths for a greedy method to choose from, but even randomising among these choices still gives no better than $O(p^\alpha)$, $\alpha > 0$, steps for many 1-relations.

5.5 Randomised Routing

We have seen that for the butterfly and hypercube, the performance of greedy routing on random 1-mappings is much better than on “worst case 1-relations”, such as the bit-reversal permutation in the case of the butterfly. Around 1980, Valiant made the simple and striking observation that one could achieve efficient distributed routing, in terms of worst case performance, if one could reduce a 1-relation to something like the composition of two random 1-mappings. The resulting technique which emerged from this observation has come to be known as two-phase randomised routing [89]. Using this approach, a 1-relation is realised by initially sending each packet to a random node in the network, using a greedy method. From there it is forwarded to the desired destination, again by a greedy method. Both phases of the routing correspond closely to the realisation of a random 1-mapping. Extensive investigation of this method, in terms of the number of steps required, size of buffers required etc., has shown that it performs extremely well, both in theory and in practice. The main theoretical results which follow from the use of randomised routing are summarised in the following two theorems.

**Theorem 5.5.1** With high probability, every 1-relation can be realised on a $p$ processor cube-connected-cycles, butterfly, 2D array and hypercube in a
number of steps proportional to the diameter of the network.

For the fixed degree networks in Theorem 5.5.1, this result is essentially optimal. For the $(\log p)$-degree hypercube, the following stronger result can be obtained.

**Theorem 5.5.2** With high probability, every $(\log p)$-relation can be realised on a $p$ processor hypercube in $O(\log p)$ steps.

Proofs of Theorems 5.5.1 and 5.5.2 can be found in [89]. Randomised routing can also be used to achieve good worst case performance on networks such as the shuffle-exchange graph and fat trees.

## 5.6 Routing on Random Networks

An interesting alternative to using randomised routing on a standard, well defined network such as a butterfly, is to use *deterministic routing on a randomly wired network*. In [56, 87] it is shown that a simple deterministic routing algorithm can be used to realise a 1-relation in $O(\log p)$ steps on a randomly wired, bounded degree network known as a multibutterfly. An important feature of multibutterflies is that they have powerful expansion properties. In addition to permitting fast deterministic routing, such expander graphs also have very strong fault tolerance properties.

## 5.7 Routing on Sorting Networks

Another possible approach to deterministic routing is to use a sorting network of low depth. For example, from Batcher’s odd-even merge sorting network we can obtain an interconnection network by associating each line in the sorting network with a node in the interconnection network, and each comparison element in the sorting network with an edge in the interconnection network. In this way, we can obtain a $p$-node graph of degree $O(\log^2 p)$ and diameter $O(\log^2 p)$. Any 1-relation routing problem can be realised in $O(\log^2 p)$ steps by using the associated sorting network as a means of “sorting” the packet addresses. Note that this method is deterministic and requires no queueing. By using the sorting network of Ajtai, Komlos and Szemeredi [3], or its refinement by Paterson [72], we can improve this result to obtain degree, diameter, and number of time steps $O(\log p)$. However,
the complex structure of the networks involved, and the very large constant factors hidden in the $O(\log p)$ bounds, rule out this approach as a practical option, at least for the present time.

5.8 Optical Communication

Simple arguments can be used to show that various low diameter networks, such as the butterfly and the hypercube, cannot be implemented using VLSI technology without having long wires for some of the edges in the network. This has led some to conclude that such networks should be replaced by networks such as fat trees [57, 58] which are more efficient, in the VLSI model, in terms of their use of area or volume.

In this section we show that optical communication systems offer the prospect of a dramatic improvement in the efficiency with which non-local communication can be achieved. We show that a simple (and possibly cheap) optical interconnection architecture based on wavelength division multiplexing can be used (a) to solve the above mentioned VLSI wiring problem, and (b) to implement an extremely simple and efficient form of randomised optical routing. The results presented in this section are due, in this form, to Rao [75], although most of them are reinterpretations, in terms of optical communication, of known results in the theory of parallel computation.

Rao [75] considers various routing problems on a $p^{1/2} \times p^{1/2}$ 2D array of processors, where the processors on each row of the array are connected by an optical bus, and the processors on each column of the array are also connected by such a bus. We thus have $2p^{1/2}$ buses, each of length $p^{1/2}$. Each bus uses wavelength division multiplexing (WDM) to support simultaneous communication between many disjoint pairs of processors on the same bus. This communications architecture will be referred to as the $p$ processor mesh of buses (MOB).

5.8.1 Solving The VLSI Wiring Problem

To solve the VLSI wiring problem we need only consider the simplest MOB in which the optical buses have fixed transmitters and receivers at each processor, i.e. where the transmitters and receivers are initially set (off-line) to achieve a certain communication pattern, e.g. a hypercube, and then remain fixed as that pattern is used. The first result shows that all networks can be
emulated on the MOB with an efficiency related to the degree of the network.

**Theorem 5.8.1** Any $p$ processor network $N$ of degree $d$ can be emulated on a $p$ processor MOB so that (i) there are $O(d)$ transmitters/receivers per processor, and (ii) each edge in $N$ is realised by a path of length at most three in the MOB.

By an edge in the MOB we mean a channel on one of the buses, and by a path we mean a sequence of such edges. Theorem 5.8.1 is a consequence of the following well known results: (a) any such degree $d$ network has $O(d)$ perfect matchings, (b) each perfect matching corresponds to a 1-relation, and (c) by Hall’s Theorem, all 1-relations can be routed (off-line) in a 2D array by permuting the packets of the rows, then permuting the packets of the columns, and then finally permuting the packets of the rows again. For details of the proof of part (c), see [55]. For networks such as the cube-connected-cycles and the hypercube we can do even better.

**Theorem 5.8.2** A $p$ processor cube-connected-cycles network can be emulated on a $p$ processor MOB so that (i) there are $O(1)$ transmitters/receivers per processor, and (ii) each edge in the cube-connected-cycles is realised by a single edge in the MOB.

**Theorem 5.8.3** A $p$ processor hypercube network can be emulated on a $p$ processor MOB so that (i) there are $O(\log p)$ transmitters/receivers per processor, and (ii) each edge in the hypercube is realised by a single edge in the MOB.

Theorems 5.8.2 and 5.8.3 follow directly from standard VLSI layouts of those networks.

### 5.8.2 Randomised Optical Routing

We now consider a model of optical communication corresponding to the case where the buses in the MOB have tunable transmitters at each processor. Unlike the previous model, we are now able to dynamically change (on-line) the destination on the bus to which a processor can send a message. The model of computation is as follows. At any step, a processor can send a message directly to any other processor on the same bus. A message is successfully received if it was the only message sent to that destination in
the step. A processor which successfully receives a message sends back an acknowledgement. We now describe a very simple randomised method for routing 1-relations on a MOB with tunable transmitters, which Rao [75] credits to Leighton and Maggs. The method proceeds in rounds, where each round consists of the following sequence of steps.

1. Each processor with a message sends it to a randomly selected position in its row bus.

2. Each processor that successfully received a message in step 1 forwards it using its column bus to the correct destination row for that message.

3. Each processor that successfully receives a message from step 2 forwards it using its row bus to the correct destination for that message.

4. For each message that was successfully received in step 3, an acknowledgement is sent to its source along the path it took. (It is easy to show that an acknowledgement gets back to any processor whose message was successfully sent to its destination.)

5. When a processor receives an acknowledgement, it does not send the message in later rounds.

**Theorem 5.8.4** With high probability, the above method will route any 1-relation in $O(\log \log p)$ steps on a MOB.

The $O(\log \log p)$ upper bound is easily established as follows. Each round takes six communication steps. The probability of a given message colliding with another in the first round is less than $1/e$. If we have $p/l$ messages left at the start of some round, then no more than about $p/l^2$ of them will be unsuccessful in that round. Therefore, after $k$ rounds we will have no more than about $p/e^{2k}$ of them left, and thus $O(\log \log p)$ rounds will be sufficient. On a completely connected optical network with the same collision rules, the problem of routing a 1-relation can be trivially completed in one step, whereas the above method requires $O(\log \log p)$ steps on the MOB. For the problem of routing $h$-relations, where $h$ is at least logarithmic in $p$, Rao [75] has established the following powerful result.

**Theorem 5.8.5** With high probability, any $h$-relation, with $h \geq \log p$, can be routed in $O(h)$ steps on a MOB.

Therefore, for such larger $h$-relations, the MOB is as powerful as a completely connected network!
5.9 Networks, Routing and BSP

The use of the parameters $l$ and $g$ to characterise the communications performance of the BSP computer contrasts sharply with the way in which communications performance is described for most distributed memory architectures on the market today. A major feature of the BSP model is that it lifts considerations of network performance from the local level to the global level. We are thus no longer particularly interested in whether the network is a 2D array, a butterfly or a hypercube, or whether it is implemented in VLSI or in some optical technology. Our interest is in global parameters of the network, such as $l$ and $g$, which describe its ability to support data communications in a uniformly efficient manner.

In the design and implementation of a BSP computer, the values of $l$ and $g$ which can be achieved will depend on the capabilities of the available technology and the amount of money that one is willing to spend on the communications network. As the computational performance of machines continues to grow, we will find that to keep $l$ and $g$ low it will be necessary to continually increase our investment in the communications hardware as a percentage of the total cost of the machine. In asymptotic terms, the values of $l$ and $g$ one might expect for various $p$ processor networks are: ring [$l = O(p), g = O(p)$], 2D array [$l = O(p^{1/2}), g = O(p^{1/2})$], butterfly [$l = O(\log p), g = O(\log p)$], hypercube [$l = O(\log p), g = O(1)$]. These asymptotic estimates are based entirely on the degree and diameter properties of the corresponding graph. In a practical setting, the channel capacities, routing methods used, VLSI implementation etc. would also have a significant impact on the actual values of $l$ and $g$ which could be achieved on a given machine. New optical technologies may offer the prospect of further reductions in the values of $l$ and $g$ which can be achieved, by providing a more efficient means of non-local communication than is possible with VLSI.

If we are interested in the problem of designing improved networks and routing methods which reduce $g$ then perhaps the most obvious approach is to concentrate instead on the alternative problem of reducing $l$. This strategy is suggested by the following simple reasoning: If messages are in the network for a shorter period of time then, given that the network capacity is fixed, it will be possible to insert messages into the network more frequently. In studying BSP algorithms we see that, in many cases, the performance of a BSP computation is limited much more by $g$ than by $l$. This suggests that in future, when designing networks and routing methods it may be advanta-
geous to accept a significant increase in $l$ in order to secure even a modest decrease in $g$. This raises a number of interesting architectural questions which have not yet been fully explored. The work in [74] contains some interesting initial ideas in this direction. It is also interesting to note that the characteristics of many modern communication systems (slow switching, very high bandwidth) may be very compatible with this alternative approach.
Chapter 6
Structured Communications

In this chapter we show how various structured communication patterns which arise frequently in parallel programs can be efficiently realised in a BSP architecture.

6.1 Broadcasting

To broadcast a single value from processor $i$ to all $p$ processors we can use a balanced $k$-ary tree of height $s$, where $s = \log_k p$. This translates into a BSP algorithm with $s$ supersteps, each costing $k \cdot g + l$. The total cost is therefore $k \cdot s \cdot g + s \cdot l$ and this is clearly minimised when $k = l/g$. To efficiently broadcast an array of $n$ values from processor $i$ to all $p$ processors, where $n >> p$, we use a different approach. In the first superstep, processor $i$ sends each of the processors a subarray of size $n/p$. The cost of this step is $n \cdot g + l$. In the second superstep, each processor sends its subarray to all of the processors. The cost of this step is again $n \cdot g + l$.

6.2 Combining

Many parallel algorithms and programs involve global reductions or combine operations. A global reduction combines a sequence of $p$ values held on the processors, using some basic function $\oplus$, and stores the result on some designated processor. By “inverting” the methods for broadcasting described above, we can obtain BSP algorithms for combining. Let $t$ denote the time
required to perform the $\oplus$ operation. For the problem of combining a set of
$p$ single values using $\oplus$ we can obtain upper bounds such as
\[(\log p) \cdot (g + l + t)\]
or
\[p \cdot g + l + p \cdot t\]

### 6.3 Prefix Sums

Let $x_0, x_1, \ldots, x_{p-1}$ be a set of values and $\oplus$ be an associative operation on
that set. Given $x_i$ initially on processor $i$, the prefix sums problem is, to
compute $s_i = x_0 \oplus x_1 \oplus \cdots \oplus x_i$, for all $0 \leq i < p$, and store $s_i$ on processor $i$.
Several well known parallel methods can be used to show that the prefix sums
problem can be solved in $O(\log p)$ parallel steps, using only $O(p)$ operations
in total. The following program, while using more operations, is very simple
and can be used to obtain a BSP algorithm.

```plaintext
k := \lceil\log_2 p\rceil;
seq for i ← 0 .. (k - 1)
    par for j ← 0 .. (p - 1)
        if j \geq 2^i then x_j := x_{j-2^i} \oplus x_j;
```

This yields a BSP algorithm of complexity $(\log p) \cdot (g + l + t)$ where $t$ denotes
the time required to perform the $\oplus$ operation. An alternative method is to
send each of the $p$ values to all processors and then compute the prefix sums
locally. This yields an upper bound of $p \cdot g + l + p \cdot t$
Chapter 7

BSP Scheduling of Parallel Computations

Many static computations can be conveniently modelled by directed acyclic graphs, where each node corresponds to some simple operation, and the arcs correspond to inputs and outputs.

7.1 Diamond DAG

Let $D_n$ denote the directed acyclic graph which has $n^2$ nodes $v_{i,j}$, $0 \leq i, j < n$, and arcs from $v_{i,j}$ to $v_{i+1,j}$ and $v_{i,j+1}$ where those nodes exist. The graph $D_n$ can be scheduled for a $p$ processor BSP computer by partitioning $D_n$ into $p^2$ subgraphs, each of which is isomorphic to $D_{n/p}$.

Let $s = n/p$ and $C^{\hat{i},\hat{j}}$, $0 \leq \hat{i}, \hat{j} < p$, denote the set of nodes $v_{i,j}$ in $D_n$ where $i \text{ div } s = \hat{i}$ and $j \text{ div } s = \hat{j}$. The following simple schedule for $D_n$ requires $2p - 1$ supersteps: During superstep $s(t)$, each $C^{\hat{i},\hat{j}}$ for which $\hat{i} + \hat{j} = t$ is computed by one of the $p$ processors, with no two of them computed by the same processor. From the structure of $D_n$ it is clear that during a superstep, each processor will receive $n/p$ values, send $n/p$ values, and perform $n^2/p^2$ computation steps. The total time required for the BSP implementation of any computation which can be modelled by $D_n$ is therefore at most $n^2/p + n \cdot g + p \cdot l$. [Throughout this paper we will omit the various small constant factors in such formulae.]

In some situations, using more processors in a BSP architecture will actually increase the runtime. For example, consider a BSP architecture based
on a ring, with \( l = g = p \). The runtime of our BSP schedule for \( D_n \) will be 
\[ \frac{n^2}{p} + np + p^2. \] 
This runtime is minimised when \( p = n^{1/2} \). Increasing the number of processors beyond this value will increase the runtime.

### 7.1.1 Solution of Triangular Systems

Let \( A \) be an \( n \times n \) non-singular, lower triangular matrix, and \( b \) be an \( n \)-element vector. The linear system \( A \cdot x = b \) can be solved by back substitution using the recurrence 
\[ x_i = \left( b_i - \sum_{1 \leq j < i} a_{i,j} \cdot x_j \right) / a_{i,i} \]
for \( 0 \leq i < n \). This back substitution recurrence can be reformulated into the following set of definitions: For all \( 0 \leq j \leq i < n \),
\[
\begin{align*}
t_{i,i} &= \left( b_i - t_{i-1,i} \right) / a_{i,i} \\
t_{i,j} &= a_{i,j} \cdot r_{i,j} + t_{i,j-1} \text{ if } i > j \\
r_{i,i} &= t_{i,i} \\
r_{i,j} &= r_{i-1,j} \text{ if } i > j \\
x_i &= t_{i,i}
\end{align*}
\]
where \( t_{i,-1} = 0 \). These definitions can be directly translated into a directed acyclic graph which is a subgraph of \( D_n \).

### 7.1.2 String Comparison

Let \( X_m = x_1x_2\ldots x_m \) and \( Y_n = y_1y_2\ldots y_n \) be two strings. The string comparison problem is the problem of determining the minimum number of insertions and deletions required to change \( X_m \) into \( Y_n \). We assume that each insertion or deletion costs 1. Very large instances of this problem arise in the area of molecular biology where the strings correspond to nucleic acid sequences [78]. The edit distance is a measure of the similarity of the two sequences. When \( m = 0 \) we have \( \text{Cost}(X_m, Y_n) = n \). Similarly, \( n = 0 \) gives \( \text{Cost}(X_m, Y_n) = m \). If \( m > 0 \), \( n > 0 \) then we have
\[
\begin{align*}
\text{Cost}(X_m, Y_n) & \leq \text{Cost}(X_{m-1}, Y_{n-1}) & \text{if } x_m = y_n \\
\text{Cost}(X_m, Y_n) & \leq 1 + \text{Cost}(X_{m-1}, Y_n) & \text{[delete } x_m \text{]} \\
\text{Cost}(X_m, Y_n) & \leq 1 + \text{Cost}(X_m, Y_{n-1}) & \text{[insert } y_n \text{]}
\end{align*}
\]
Using dynamic programming we can tabulate the values of Cost in a straightforward way. As an example, consider the computation of Cost(D\textsc{efine}, D\textsc{esign}).

<table>
<thead>
<tr>
<th></th>
<th>∅</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>I</th>
<th>N</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>I</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>N</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The number below xᵢ and to the right of yⱼ is Cost(Xᵢ, Yⱼ). The chain of bracketed numbers indicate that we should perform the following sequence of edits to change DEFINE into DESIGN: Delete F, insert S, insert G, delete E. The sequence comparison problem can, therefore, be solved by the following parallel program, where cᵢ,ⱼ corresponds to Cost(Xᵢ, Yⱼ).

\[
\begin{align*}
\text{par for } & i \leftarrow 0..n \\
& c_{0,i} := i \\
\text{par for } & i \leftarrow 0..m \\
& c_{i,0} := i \\
\text{seq for } & d \leftarrow 0..(m+n) \\
& \text{par for } i \leftarrow \min\{0,d-m\}..\min\{d,n\} \\
& \quad c_{i,d-i} := \min\{c_{i,d-i-1},c_{i-1,d-i}\} + 1; \\
& \quad \text{if } x_i = y_{d-i} \text{ then } c_{i,d-i} := \min\{c_{i,d-i},c_{i-1,d-i-1}\}
\end{align*}
\]

This program can be mapped directly onto Dₙ.

### 7.2 Cube DAG

Let Cₙ denote the directed acyclic graph which has n³ nodes νᵢ,j,k, 0 ≤ i, j, k < n, and arcs from νᵢ,j,k to νᵢ+1,j,k, νᵢ,j+1,k and νᵢ,j,k+1 where those nodes exist. The graph Cₙ can be scheduled for a p processor BSP computer by partitioning Cₙ into p³/2 subgraphs, each of which is isomorphic to Cₙ/p₁/2. Let s = n/p₁/2 and Cᵢ,j,k, 0 ≤ i, j, k < p₁/2, denote the subset of s³ nodes νᵢ,j,k in Cₙ where i \text{ div } s = i, j \text{ div } s = j and k \text{ div } s = k. The following
simple schedule for \( C_n \) requires \( 3p^{1/2} - 2 \) supersteps: During superstep \( s(t) \), each \( C^\hat{i},\hat{j},\hat{k} \) for which \( \hat{i} + \hat{j} + \hat{k} = t \) is computed by one of the \( p \) processors, with no two of them computed by the same processor. From the structure of \( C_n \) it is clear that during a superstep, each processor will receive \( n^2/p \) values, send \( n^2/p \) values, and perform \( n^3/p^{3/2} \) computation steps. The total time required for the BSP implementation of any computation which can be modelled by \( C_n \) is therefore at most \( n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l \).

### 7.2.1 Matrix Multiplication

Consider the problem of multiplying two \( n \times n \) dense matrices \( A, B \) to produce \( C \). In [62] it is shown that the product \( C \) can be computed using the following set of definitions: For all \( 0 < i, j, k \leq n \),

\[
\begin{align*}
    a_{i,j,k} &= a_{i,j-1,k} \\
    b_{i,j,k} &= b_{i-1,i,j,k} \\
    c_{i,j,k} &= c_{i,j,k-1} + (a_{i,j,k} \cdot b_{i,j,k})
\end{align*}
\]

where \( a_{i,0,k} = a_{i,k}, b_{0,j,k} = b_{k,j} \) and \( c_{i,j,0} = 0 \). These definitions can be directly translated into a labelled version of the directed acyclic graph \( C_n \). The BSP time complexity of matrix multiplication is therefore at most \( n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l \). For the standard \( n^3 \) sequential matrix multiplication algorithm, this BSP schedule is optimal in terms of its computation cost \( W(n,p) = n^3/p \). It is also optimal in terms of its space complexity. The matrices \( A \) and \( B \) can be uniformly distributed across the \( p \) processors, with each one holding an \( n/p^{1/2} \times n/p^{1/2} \) block of the matrix. Given this uniform input distribution, we can schedule the reuse of memory locations in a straightforward way to ensure that no processor will be required to store more than \( n^2/p \) values in any superstep. Therefore we have \( M(n,p) = n^2/p \).

### 7.2.2 LU Decomposition

Another very important problem in linear algebra is LU decomposition. We begin by describing the standard recursive algorithm for the LU decomposition of an \( n \times n \) nonsingular matrix \( A \) (without pivoting) [23]. For \( n = 1 \) the problem is trivial since we can take \( L = I \) and \( U = A \). For \( n > 1 \), we
break $A$ into four parts:

$$A = \begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
$$

\[= \begin{pmatrix}
a_{11}w^T \\
v
da_n
\end{pmatrix} \begin{pmatrix}
1 & 0 \\
v/a_{11} & I_{n-1}
\end{pmatrix} \begin{pmatrix}
a_{11} & w^T \\
0 & \hat{A} - vw^T/a_{11}
\end{pmatrix}
\]

where $v$ is an $(n-1)$-column vector, $w^T$ is an $(n-1)$-row vector, and $\hat{A}$ is an $(n-1) \times (n-1)$ matrix. Then, using elementary matrix algebra, we can factor $A$ as

$$A = \begin{pmatrix}
1 & 0 \\
v/a_{11} & \hat{L}
\end{pmatrix} \begin{pmatrix}
a_{11} & w^T \\
0 & \hat{U}
\end{pmatrix} = LU$$

The 0's in the first and second matrices of the factorisation are row and column vectors, respectively, of size $n - 1$. The term $vw^T/a_{11}$, formed by taking the outer product of $v$ and $w$ and dividing each element of the result by $a_{11}$, is an $(n-1) \times (n-1)$ matrix, which conforms in size to the matrix $\hat{A}$ from which it is subtracted. The resulting $(n-1) \times (n-1)$ matrix $\hat{A} - vw^T/a_{11}$ is called the Schur complement of $A$ with respect to $a_{11}$. We can now apply the technique recursively to find the LU decomposition of the Schur complement $\hat{A} - vw^T/a_{11}$. Let $\hat{A} - vw^T/a_{11} = \hat{L}\hat{U}$ where $\hat{L}$ is lower triangular and $\hat{U}$ is upper triangular. Then we have

$$A = \begin{pmatrix}
1 & 0 \\
v/a_{11} & \hat{L}
\end{pmatrix} \begin{pmatrix}
a_{11} & w^T \\
0 & \hat{U}
\end{pmatrix} = LU$$

thereby providing our LU decomposition. (Note that because $\hat{L}$ is lower triangular, so is $L$, and because $\hat{U}$ is lower triangular, so is $U$.) By transforming this tail recursion to an iteration we can obtain a parallel program, which after localising broadcasts, is described by the following set of definitions: For all $0 \leq k \leq i, j < n$,

\[
\begin{align*}
    u_{k,k,k} &= a_{k,k,k-1} \\
l_{i,j,k} &= a_{i,j,k-1}/u_{i,j,k} \text{ if } j = k, \text{ and } l_{i,j-1,k} \text{ otherwise.} \\
u_{i,j,k} &= a_{i,j,k-1} \text{ if } i = k, \text{ and } u_{i-1,j,k} \text{ otherwise.} \\
a_{i,j,k} &= a_{i,j,k-1} - (l_{i,j,k} \cdot u_{i,j,k})
\end{align*}
\]
where $a_{i,j,-1} = a_{i,j}$. These definitions can be directly translated into a directed acyclic graph which is a subgraph of $C_n$. The BSP time complexity of LU decomposition is therefore at most $n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l$. In terms of space complexity, we have $M(n, p) = n^2/p$.

### 7.2.3 Algebraic Path Problem

A **closed semiring** is an algebraic structure $(S, \oplus, \otimes, I_\oplus, I_\otimes)$ with the following properties:

- $\oplus$ is a commutative monoid ($\oplus$ satisfies the closure, associative, commutative properties, and has identity element $I_\oplus$).
- $\otimes$ is a monoid ($\otimes$ satisfies the closure, associative properties, and has identity element $I_\otimes$).
- $\otimes$ is right and left distributive over $\oplus$.
- For all $s \in S$, $s \otimes I_\oplus = I_\oplus$.

Let $G = (V, A)$ be a directed graph on $|V| = n$ vertices, in which each $<i,j> \in A$ has an associated weight $s_{ij} \in S$. Define an $n \times n$ matrix $M$ of weights $m_{ij}$ corresponding to the arcs of $G$ : $m_{ij} = s_{ij}$ if $<i,j> \in A$, $I_\oplus$ otherwise. The **Algebraic Path Problem (APP)** is to compute $M^* = \bigoplus_{k=0}^{\infty} M^k$ where matrix product is defined in terms of the two operations $\oplus$ and $\otimes$. ($M^0$ is the identity matrix with diagonal elements $I_\otimes$). $M^*_i$ gives the “sum” of the weights of all directed paths from $i$ to $j$ where the weight of a path is the “product” of the weights of the arcs.

The APP is a problem of major importance in a wide variety of areas and has been extensively studied in recent years. Some examples of instances of the APP are the following:

<table>
<thead>
<tr>
<th>Problem</th>
<th>$S$</th>
<th>$\oplus$</th>
<th>$\otimes$</th>
<th>$I_\oplus$</th>
<th>$I_\otimes$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity (trans. closure) [94]</td>
<td>{false, true}</td>
<td>or</td>
<td>and</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>Generation of regular lang.</td>
<td>{words}</td>
<td>$\cup$</td>
<td>$-$</td>
<td>$\emptyset$</td>
<td>empty word</td>
</tr>
<tr>
<td>Max. capacity path</td>
<td>$\mathbb{R}^+ \cup \infty$</td>
<td>max</td>
<td>min</td>
<td>$0$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>Path with min. number of arcs</td>
<td>$\mathbb{N} \cup \infty$</td>
<td>min</td>
<td>$+$</td>
<td>$\infty$</td>
<td>$0$</td>
</tr>
<tr>
<td>Shortest paths [28]</td>
<td>$\mathbb{R} \cup \infty$</td>
<td>min</td>
<td>$+$</td>
<td>$\infty$</td>
<td>$0$</td>
</tr>
<tr>
<td>Max. reliability path</td>
<td>${a \mid 0 \leq a \leq 1}$</td>
<td>max</td>
<td>$\ast$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>Min. cost spanning tree [59]</td>
<td>$\mathbb{R}^+ \cup \infty$</td>
<td>min</td>
<td>max</td>
<td>$\infty$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

The APP also finds application in areas such as parsing and logic programming, and can be used as the basis of fast parallel algorithms for matrix
inversion. Like the prefix sums computation, it is a remarkably versatile second-order function.

The APP can be solved using the algorithm of Floyd [28]. Translated into our notation, Floyd’s algorithm corresponds to the following parallel program.

```plaintext
seq for k ← 1 .. n
  par for (i ← 1 .. n ; j ← 1 .. n)
    m_{ij} := m_{ij} ⊕ (m_{ik} ⊗ m_{kj})
```

Replacing the assignments by equalities we obtain the following definitions.

For all \(1 \leq i, j, k \leq n\),

\[
\begin{align*}
m_{ij}^{(0)} &= m_{ij} \\
m_{ij}^{(k)} &= m_{ij}^{(k-1)} ⊕ (m_{ik}^{(k-1)} ⊗ m_{kj}^{(k-1)})
\end{align*}
\]

By localising the broadcasts we obtain the following modified set of definitions.

For all \(1 \leq i, j, k \leq n\),

\[
\begin{align*}
m_{i,j,0} &= m_{ij} \\
r_{i,j,k} &= m_{i,j,k-1} \text{ if } j = k, r_{i,j+1,k} \text{ if } j < k, r_{i,j+1,k} \text{ otherwise} \\
c_{i,j,k} &= m_{i,j,k-1} \text{ if } i = k, c_{i+1,j,k} \text{ if } i < k, c_{i-1,j,k} \text{ otherwise} \\
m_{i,j,k} &= m_{i,j,k-1} \oplus (r_{i,j,k} \otimes c_{i,j,k})
\end{align*}
\]

These definitions do not map directly onto the cube DAG \(C_n\). However, they do map onto a closely related DAG. Let \(C_n^+\) denote the directed acyclic graph which has \(n^3\) nodes \(v_{i,j,k}, 0 \leq i, j, k < n\), and arcs from \(v_{i,j,k}\) to \(v_{i+1,j,k}\) if \(i \geq k\), \(v_{i-1,j,k}\) if \(i \leq k\), \(v_{i,j+1,k}\) and \(v_{i,j,k+1}\) where those nodes exist. Let \(C_n^{++}\) denote the directed acyclic graph which has \(n^3\) nodes \(v_{i,j,k}, 0 \leq i, j, k < n\), and arcs from \(v_{i,j,k}\) to \(v_{i+1,j,k}\) if \(i \geq k\), \(v_{i-1,j,k}\) if \(i \leq k\), \(v_{i,j+1,k}\) if \(j \geq k\), \(v_{i,j-1,k}\) if \(j \leq k\), and \(v_{i,j,k+1}\) where those nodes exist. The graph \(C_n^{++}\) can be scheduled for a \(p\) processor BSP computer by partitioning \(C_n^{++}\) into \(p^{3/2}\) subgraphs, each of which is isomorphic to \(C_{n/p}^{++}\), to \(C_{n/p}^{+}\), or to \(C_{n/p}^{1/2}\).
The resulting BSP algorithm gives the same bounds as for $C_n$, to within a small constant factor.
Chapter 8

Design and Analysis of BSP Algorithms

8.1 Dense Matrix-Vector Multiplication

In this section we consider the problem of multiplying an $n \times n$ matrix $M$ by an $n$-element vector $v$ on $p$ processors, where $M, v$ are both dense. The BSP algorithm which we describe has complexity $n^2/p + (n/p^{1/2}) \cdot g + l$.

For dense $M$ and $p \leq n$, the standard $n^2$ sequential algorithm can be adapted to run on a $p$ processor BSP machine as follows. The matrix elements are initially distributed uniformly across the $p$ processors, with each processor holding an $n/p^{1/2} \times n/p^{1/2}$ submatrix of $M$. [In Chapter 9 this is referred to as a “block-block” distribution.] The vectors $u$ and $v$ are also uniformly distributed across the machine, with $n/p$ elements of $u$, and $n/p$ elements of $v$, allocated to each processor. In the first superstep, each processor gets the set of all $n/p^{1/2}$ vector elements $v_i$ for which it holds an $m_{ij}$. The cost of this step is $(n/p^{1/2}) \cdot g + l$. In the second superstep, each processor $k$ computes $u_i^k = \sum_{a \leq j < a + n/p^{1/2}} m_{ij} \cdot v_i$ for each of the $n/p^{1/2}$ subrows of $M$ which it holds, and sends the partial sum $u_i^k$ to the processor which holds $u_i$. The time required for this step is $n^2/p + (n/p^{1/2}) \cdot g + l$. In the third and final superstep, each processor computes the value of each of its vector elements $u_i$ by adding the $p^{1/2}$ values $u_i^k$ which it receives. The time required for this final step is $n/p^{1/2} + l$.

An input-output complexity argument, similar to those in [2, 43], can be used to show that for any BSP algorithm which computes $u = M \cdot v$, if
\( W(n, p) = n^2/p \) then \( H(n, p) \geq n/p^{1/2} \). Noting that the \( n^2 \) sequential computation cost is itself optimal we see that the above method is in a strong sense a best possible BSP algorithm for this problem. It simultaneously achieves the optimal computation cost \( W(n, p) = n^2/p \), the optimal communication cost \( H(n, p) \cdot g = (n/p^{1/2}) \cdot g \) and the optimal synchronisation cost \( S(n, p) \cdot l = l \). Other matrix distributions, such as the “block-grid” distribution, see Chapter 9, also give these bounds. [The block-grid distribution is defined as follows. Let \( PROC(r,c) \), \( 0 \leq r, c < p^{1/2} \), denote the \( p \) processors. The matrix elements \( m_{ij} \), \( 0 \leq i, j < n \), are uniformly distributed across the processors, with \( m_{ij} \) allocated to \( PROC(i \text{ div } (n/p^{1/2}), j \text{ mod } p^{1/2}) \).

In Figure 8.1 we give a pseudocode version of the BSP algorithm for dense matrix-vector multiplication, where the data distribution is defined to be block-grid. For simplicity we assume that \( n \) is a multiple of \( p \), and that \( p \) is a perfect square. The assumption of such an ideal match between problem size and machine size is, of course, unrealistic in practice. It does however permit us to give a clear and concise description of the main points of the BSP algorithm and its implementation, without having to give all of the technical details required for the general case. The various non-standard constructs used in the pseudocode are informally described in Figure 8.2.

### 8.2 Matrix Multiplication

We now consider the problem of multiplying two \( n \times n \) dense matrices \( A, B \) on \( p \) processors. For \( p \leq n^2 \), the standard \( n^3 \) sequential algorithm can be adapted to run on a \( p \) processor BSP machine as follows. Each processor computes an \( (n/\sqrt{p}) \times (n/\sqrt{p}) \) submatrix of \( C = A \cdot B \). To do so it will require \( n^2/\sqrt{p} \) elements from \( A \) and the same number from \( B \). If \( A \) and \( B \) are both distributed uniformly across the \( p \) processors, with each processor holding \( n^2/p \) of the elements from each matrix, then the total time required for this algorithm will be \( n^3/p + gn^2/\sqrt{p} + l \). In Figure 8.3 we give a pseudocode version of this BSP algorithm. For simplicity we have assumed that \( n \) is a multiple of \( \sqrt{p} \), and that \( p \) is a perfect square.

We now describe a more efficient BSP realisation of the standard \( n^3 \) algorithm, due to McColl and Valiant. Its BSP time complexity is \( n^3/p + (n^2/p^{2/3}) \cdot g + l \). As in the previous algorithm, we begin with \( A, B \) distributed uniformly but arbitrarily across the \( p \) processors. At the end of
the computation, the $n^2$ elements of $C$ should also be distributed uniformly across the $p$ processors. Let $s = n/p^{1/3}$ and $A[i,j]$ denote the $s \times s$ submatrix of $A$ consisting of the elements $a_{ij}$ where $i \div s = i$ and $j \div s = j$. Define $B[i,j]$ and $C[i,j]$ similarly. Then we have $C[i,j] = \sum_{0 \leq k<p^{1/3}} A[i,k] \cdot B[k,j]$.

Let $\text{PROC}(i,j,k)$, $0 \leq i, j, k < p^{1/3}$, denote the $p$ processors. In the first superstep each processor $\text{PROC}(i,j,k)$ gets the set of elements in $A[i,k]$ and those in $B[k,j]$. The cost of this step is $(n^2/p^{2/3}) \cdot g + l$. In the second superstep $\text{PROC}(i,j,k)$ computes $A[i,k] \cdot B[k,j]$ and sends each one of the $n^2/p^{2/3}$ resulting values to the unique processor which is responsible for computing the corresponding value in $C$. The cost of this step is $n^3/p + (n^2/p^{2/3}) \cdot g + l$. In the final superstep, each processor computes each of its $n^2/p$ elements of $C$ by adding the $p^{1/3}$ values received for that element. The cost of this step is $n^2/p^{2/3} + l$.

An input-output complexity argument can be used to show that for any BSP implementation of the standard $n^3$ sequential algorithm, if $W(n,p) = n^3/p$ then $H(n,p) \geq n^2/p^{2/3}$. This second BSP schedule for matrix multiplication therefore provides a realisation of the standard $n^3$ method which simultaneously achieves the optimal values for computation cost $W(n,p)$, communication cost $H(n,p)$ and synchronisation cost $S(n,p)$. The memory requirement of this algorithm is, however, inferior to the first algorithm. Its memory complexity $M(n,p)$ is $n^2/p^{2/3}$.

### 8.3 Strassen’s Algorithm

Let $A$ and $B$ be two $n \times n$ matrices and consider the problem of computing $C = A \cdot B$. We can regard the matrices $A, B, C$ as each composed of four $n/2 \times n/2$ submatrices. For example,

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

If the submatrices of $B$ and $C$ are described in the same way then we have

$$C_{ij} = A_{i0} \cdot B_{0j} + A_{i1} \cdot B_{1j}$$

for all $i, j$. This yields a recursive sequential algorithm of complexity $n^3$ for matrix multiplication. Strassen [84] observed that the eight submatrix multiplications in this recursive algorithm can be reduced to seven, by using
matrix additions and subtractions in a more complex way. The following method is a simple variant of the original algorithm presented by Strassen. It appeared in [71]. Let

\[
L_0 = A_{00} \quad R_0 = B_{00} \\
L_1 = A_{01} \quad R_1 = B_{10} \\
L_2 = A_{10} + A_{11} \quad R_2 = B_{01} - B_{00} \\
L_3 = A_{00} - A_{10} \quad R_3 = B_{11} - B_{01} \\
L_4 = L_2 - A_{00} \quad R_4 = R_3 + B_{00} \\
L_5 = A_{01} - L_4 \quad R_5 = B_{11} \\
L_6 = A_{11} \quad R_6 = B_{10} - R_4
\]

and

\[
M_i = L_i \cdot R_i \text{ for all } 0 \leq i \leq 6.
\]

If we now let

\[
T_0 = M_0 + M_4 \\
T_1 = T_0 + M_3
\]

then we have

\[
C_{00} = M_0 + M_1 \\
C_{01} = T_0 + M_2 + M_5 \\
C_{10} = T_1 + M_6 \\
C_{11} = T_1 + M_2
\]

If \( \text{MM}(n) \) denotes the cost of \( n \times n \) matrix multiplication, and \( \text{MA}(n) \) denotes the cost of \( n \times n \) matrix addition/subtraction, then the above algorithm shows that \( \text{MM}(n) \leq 7 \cdot \text{MM}(n/2) + 15 \cdot \text{MA}(n/2) \). Noting that \( \text{MA}(n) = n^2 \), we obtain \( \text{MM}(n) = n^{\log_2 7} \).

If, in the BSP algorithm for matrix multiplication due to McColl and Valiant, we use Strassen's method for the matrix products computed in the second superstep, then we immediately obtain a BSP algorithm for matrix multiplication which has complexity \( (n^{\log_2 7}/(p^{(\log_2 7)/3})) + (n^2/p^{2/3}) \cdot g + l \). In the remainder of this section we will show that, asymptotically, it is possible to improve both the computation term \( n^{\log_2 7}/p^{(\log_2 7)/3} \) and the communication term \( (n^2/p^{2/3}) \cdot g \) in this upper bound, while keeping the synchronisation term linear in \( l \).
For simplicity, assume that $n = c \cdot 4^k \cdot 7^k$ and $p = 7^{2k}$. We can regard $A, B, C$ as each composed of $4^k$ submatrices, or blocks, of size $c \cdot 7^k \times c \cdot 7^k$. Each such submatrix can similarly be regarded as being composed of $7^{2k}$ subsubmatrices, or subblocks, of size $c \times c$.

The distribution of the elements of $A$ is as follows. Each processor is identified with one of the $7^{2k}$ subblock positions within each block. The processor initially holds all of the $4^k$ subblocks of $A$ which are in that position within their block. The distribution of the elements of $B$ and $C$ are the same as for $A$. The product $C = A \cdot B$ is computed as follows.

In the first superstep we initiate Strassen’s algorithm, halting each chain of recursive calls when the level of recursion reaches $2k$. [We halt it before the matrix product subcomputation is started.] For each processor, the computation cost of this superstep is given by $\sum_{l=1}^{2k} 7^{l-1} \cdot 8 \cdot MA(n/2^l) / p$ which is at most $n^2 \cdot (7/4)^{2k} / p$. For $n = c \cdot 4^k \cdot 7^k$ and $p = 7^{2k}$, this upper bound is no more than $n^{\log_2 7} / p$. There is no communication cost associated with this superstep, since each recursive level of addition and subtraction operations is carried out across the whole machine by simultaneously adding or subtracting the corresponding subblocks.

The first superstep produces $7^{2k}$ subcomputations, each of which is a matrix product involving a block $\hat{A}$ of values, each corresponding to a sum of elements of $A$, and a block $\hat{B}$ of values, each corresponding to a sum of elements of $B$. These $p$ matrix product subcomputations are carried out on the $p$ processors, with each processor computing a single product locally. The various subblocks of $\hat{A}$ and $\hat{B}$ are first obtained. The block product $\hat{A} \cdot \hat{B}$ is then computed locally using Strassen’s method as the sequential algorithm. Finally, each of the subblocks of the resulting product is sent to the processor which is responsible for subblocks in that position. The computation cost of this second superstep is $(c \cdot 7^k)^{\log_2 7}$ which is no more than $n^{\log_2 7} / p$. The communication cost is $c^2 \cdot 7^{2k} \cdot g$ which is at most $(n^2 / p^{2/\log_2 7}) \cdot g$.

In the final superstep, the additions to complete the recursive calls are performed in the same way as in the first superstep, i.e. by performing local additions on subblocks. The computation cost of the final superstep is given by $\sum_{l=1}^{2k} 7^{l-1} \cdot 7 \cdot MA(n/2^l) / p$ which is no more than the computation cost of the first superstep. As in the first superstep, there is no communication cost.

The total cost of this BSP realisation of Strassen’s algorithm is therefore

$$\left( n^{\log_2 7} / p \right) + \left( n^2 / p^{2/\log_2 7} \right) \cdot g + l$$
and it can be used for all \( p \leq n^\alpha \), where \( 7^{2k} = (4^k7^k)^\alpha \), i.e. \( \alpha = 2\log_2 7/(2 + \log_2 7) \).

### 8.4 Fast Fourier Transform

We now consider the computation of the discrete Fourier transform (DFT) of an \( n \)-element complex vector \( X = (x_0, x_1, \ldots, x_{n-1}) \) using the Fast Fourier Transform (FFT) algorithm. The vector \( (y_0, y_1, \ldots, y_{n-1}) \) is the DFT of \( X \) if for all \( k, 0 \leq k < n \), we have \( y_k = \sum_{j=0}^{n-1} \omega_n^{kj}x_j \) where \( \omega_n = e^{-2\pi i/n} \) is an \( n \)th root of unity, i.e. \( \omega_n^n = 1 \). The FFT algorithm [92] on \( n \) points has sequential complexity \( O(n \log n) \) and can be represented as a \( (\log n) \)-level butterfly graph with a bit reversal permutation applied to the inputs. It can be implemented on a \( p \) processor BSP architecture in \( (\log n)/(\log (n/p)) \) supersteps, in each of which each processor sequentially computes the next \( \log (n/p) \) levels of the butterfly graph on its \( n/p \) points in time \( (n/p) \log (n/p) \). The cost of each superstep is \( (n/p) \log (n/p) + gn/p + l \) and therefore the total time required is at most \( (n \log n)/p)(1 + g/\log (n/p) + l/(n/p) \log (n/p)) \).

In most practical situations, the number of processors, \( p \), would be no more than \( \sqrt{n} \). In such cases, the number of supersteps would be at most two, and the total time required would be \( (n \log n)/p + gn/p + l \). In Figure 8.4 we give a pseudocode version of this algorithm for the simple case where \( n = 2^k \), \( p = 2^{k-1} \) and \( k \) is a multiple of \( l \).
Given integers $ndivp$, $sqrtp$.

```plaintext
const $p = sqrtp^2$;
const $n = ndivp*p$;
const $b = n/sqrtp$;
const $side = 0 .. n-1$;
var $amem : array [side,side] of real with blocksize [b,b]$;
view $A[i,j] = amem[i,b*(j mod sqrtp)+(j div sqrtp)]$;
var $vmem : array [side,(0 .. sqrtp)] of real with blocksize [ndivp,sqrtp+1]$;
view $v[i] = vmem[i,0]$;
view $vsums[i,j] = vmem[i,j]$;
par for $(i ← 0, b .. n-1 ; j ← 0 .. sqrtp-1)$
  at $A[i,j]$;
  const $jgroup = j, j+sqrtp .. n-1$;
  var $t : real$;
  get $v[k]$ for $k ← jgroup$;
  seq for $bi ← i .. i+b-1$
    $t := 0.0$;
    seq for $bj ← jgroup$
      $t := t+A[bi,bj]*v[bj]$;
    put $t$ in $vsums[bi,j+1]$;
par for $k ← 0, ndivp .. n-1$
  at $v[k]$;
  seq for $w ← k .. k+ndivp-1$
    $v[w] := 0.0$;
    seq for $x ← 1 .. sqrtp$
      $v[w] := v[w]+vsums[w,x]$;
```

*Figure 8.1: BSP pseudocode for an $n \times n$ matrix vector product on $p$ processors.*
The ordered sequence of integers 

\[ a + (b - a)k \]

for \( k = 0, 1, \ldots \)

which lie in the closed interval between \( a \) and \( c \). The range 

\( a \ldots c \)

is equivalent to \( a, a+1 \ldots c \) if \( a \leq c \) and to \( a, a-1 \ldots c \) otherwise.

\textbf{var} \( A : \text{array} \[ [0 \ldots c*r-1],[0 \ldots d*s-1] \] \text{of real with blocksize} \ [r,s]; \)

Declares a two dimensional array \( A \) of size \( c*r \times d*s \), in which the elements of \( A \) are to be partitioned into \( c*d \) contiguous blocks, each of size \( r \times s \). For each \( 0 \leq i < c, 0 \leq j < d \), the block containing \( A[i*r,j*s] \) is allocated to processor \( (i*d+j) \mod p \). [The processors are numbered from 0 to \( p-1 \).] The case of a one-dimensional array of size \( b*n \), with blocksize \( b \), corresponds to having \( c=1, r=1, d=n \) and \( s=b \).

\textbf{seq for} \( (i \leftarrow 1 \ldots m ; j \leftarrow 1 \ldots n) \text{code}(i,j) \)

Sequentially execute the \( m \times n \) instances of \textit{code}(i,j) in the order

\( \text{code}(1,1); \text{code}(1,2); \ldots \text{code}(1,n); \text{code}(2,1); \text{code}(2,2); \ldots \text{code}(m,n); \)

\textbf{par for} \( (i \leftarrow 1 \ldots m ; j \leftarrow 1 \ldots n) \text{code}(i,j) \)

Execute the \( m \times n \) instances of \textit{code}(i,j) in parallel.

\textbf{on} \( m; \)

Execute this parallel thread on processor number \( m \).

\textbf{at} \( v; \)

Execute this parallel thread on the processor-memory pair to which the variable \( v \) has been allocated.

\textbf{get} \( v; \)

Get a local copy of the variable \( v \) for use in the operations of the next superstep. If \( v \) is already held locally then the operation has no effect. There is an implicit barrier synchronisation after any sequence of \textit{get} statements. Within a superstep, any attempt to access a value which is not held locally will result in a run-time error.

\textbf{put} \( u \text{ in} v; \)

Assign the value of the local variable \( u \) to the variable \( v \) (which may be non-local). This assignment will be performed at the end of the current superstep.

\textbf{sync; \)

Barrier synchronisation. End of superstep.

Figure 8.2: Notations used in BSP pseudocodes.
p := sqrt(p^2);
n := b*sqrt(p);
side := 0 .. n−1;

var A,B,C : array [side,side] of real with blocksize [b,b];
par for (i ← 0, b .. n−1 ; j ← 0, b .. n−1)
on i+(j/b);
iside := i .. i+b−1;
jside := j .. j+b−1;
get A[i,j] for (i ← iside ; j ← side);
get B[i,j] for (i ← side ; j ← jside);
seq for (x ← iside ; z ← jside)
  C[x,z] := 0.0;
  seq for y ← side
    C[x,z] := C[x,z]+A[x,y]*B[y,z];

Figure 8.3: BSP pseudocode for an n × n matrix product on p processors.
numbits := numplaces + numsteps; n := 2\lceil numbits; 
p := 2\lceil (numbits - numplaces); blsize := 2\lceil numplaces;

var X : array [0 .. n-1] of complex with blocksize [blsize];
var L : array [0 .. n-1] of integer with blocksize [blsize];

function bitrev(i : integer) : integer
is a function which computes the value obtained by reversing the numbits binary digits of i.

function swap(i : integer) : integer
Let <d_0, d_1, \ldots, d_{nubits-1}> denote the binary digits of i, with d_0 the least significant digit. swap is a function which computes the value obtained by swapping d_j and d_{step*numplaces+j} for all j, 0 ≤ j < numplaces.

procedure BUTTERFLY(pnum : integer)
locblsize := 1; z := 2\lceil((step-1)*numplaces);
seq for level ← 1 .. numplaces
    z := 2*z; locblsize := 2*locblsize;
    numlocblocks := blsize/locblsize; pairgap := locblsize/2;
    seq for position ← 0 .. pairgap-1
        line := L[pnum*blsize+position] mod z;
        w := omega(z)^line;
        seq for block ← 0 .. numlocblocks-1
            left := pnum*blsize+position+block*locblsize;
            right := left+pairgap;
            temp := w*X[right];
endBUTTERFLY

par for procsnum ← 0 .. p-1
    on procsnum;
    block := blsize*procsnum .. blsize*(procsnum+1)-1;
    L[i] := i for i ← block;
    put X[i] in X[bitrev(i)] for i ← block;
    sync;
    seq for step ← 1 .. numsteps
        BUTTERFLY(procsnum);
        if step<numsteps then
            put X[i] in X[swap(i)] for i ← block;
            put L[i] in L[swap(i)] for i ← block;
            sync;
        put X[i] in X[L[i]] for i ← block;
    sync;

Figure 8.4: BSP pseudocode for an n point FFT on p processors.
Chapter 9
Scientific Computing

Scientific computing is a vast area, and contains many computationally demanding problems for which efficient BSP algorithms are required. In this chapter, we theoretically and experimentally analyse the efficiency with which a wide range of important scientific computations can be performed on BSP architectures. The computations considered include the iterative solution of sparse linear systems, molecular dynamics, linear programming, and the solution of partial differential equations on a discrete grid. We analyse these computations in a uniform manner by formulating their basic procedures as a sparse matrix-vector multiplication. Most of the results in this chapter appeared in [11].

In our analysis, we give the normalised BSP cost of an algorithm as an expression of the form $a + b \cdot g + c \cdot l$, where $a$, $b$, and $c$ are scalar values which depend on the algorithm, on the number of processors, and on the chosen data distribution. An ideal parallel algorithm has the values $a = 1$, $b = 0$, and $c = 0$; an algorithm with load imbalance has a value $a > 1$; an algorithm with communication overhead has a value $b > 0$; and an algorithm with synchronisation overhead has a value $c > 0$.

As an example, consider the execution of a five-point Laplacian finite difference operator on a two-dimensional toroidal grid. This operator computes new values at a grid point using the old values at the grid point and its direct neighbours to the north, east, south, and west. Our BSP algorithm for this computation has a normalised cost on 100 processors of $1.0 + 0.022 \cdot g + 0.00056 \cdot l$ for a grid of size $200 \times 200$. This low cost is achieved by distributing the grid by orthogonal domain partitioning over the processors, assigning a square block of $20 \times 20$ grid points to each proces-
The resulting cost value implies that this computation can be performed efficiently on BSP computers with \( g \leq b^{-1} \approx 45 \) and \( l \leq c^{-1} \approx 1800 \).

In the design of efficient BSP algorithms, it is important to find a good data distribution. In fact, the choice of a data distribution is one of the main means of influencing the performance of the algorithm. In the BSP model, the partitioning of the data is a crucial issue, as opposed to the mapping of the resulting partitions to particular processors, which is irrelevant. This leads to an emphasis on problem dependent techniques of data partitioning, instead of on hardware dependent techniques that take network topologies into account. The algorithm designer who is liberated from such hardware considerations may concentrate on exploiting the essential features of the problem. In our analysis, this leads to the application of sphere packing techniques to reduce communication in molecular dynamics simulations and to the application of tiling techniques to reduce communication in discrete grid calculations.

We present experimental results for the multiplication \( u := Av \) of a sparse matrix \( A \) and a vector \( v \). The experiments are performed on the sparse matrix test library MLIB, which we developed with the aim of capturing the essence of a range of important scientific computations in the uniform format of a sparse matrix. The library contains matrices with a regular structure, such as the adjacency matrix of a multidimensional toroidal grid, and also matrices with an irregular structure, such as random sparse matrices. Furthermore, the library contains matrices with an underlying, but hidden structure (given as supplementary information), such as the matrices that describe the short-range interaction between particles in a molecular dynamics simulation.

Our BSP algorithm for sparse matrix-vector multiplication imposes the constraint that the vectors \( u \) and \( v \) and the diagonal of \( A \) are distributed in the same way and that the matrix \( A \) is distributed in a so-called Cartesian manner. This means that the \( p \) processors are numbered by two-dimensional Cartesian coordinates \((s, t)\), and that each matrix row is assigned to a set of processors with the same first coordinate \( s \), and each matrix column to a set of processors with the same second coordinate \( t \). This distribution leads to a simple sparse matrix-vector multiplication algorithm. Within this scheme, various choices are possible. For general sparse matrices, with no known structure, a good choice is to distribute the matrix diagonal randomly over the processors, taking care that each processor receives an equal number of diagonal elements, and using a square Cartesian processor numbering, i.e. with \( 0 \leq s, t < \sqrt{p} \). For matrices with a known structure, this method can
be greatly improved upon by using techniques such as spatial decomposition of the corresponding physical domain. We present several new techniques based on spatial decomposition and demonstrate their practical utility by numerical experiments.

9.1 Linear algebra in Scientific Computing

Linear algebra is of crucial importance to scientific computing. The main reason for this is the large amount of computing time consumed by linear algebra computations in a wide range of application areas. Often, applications require the solution of large linear systems or large eigensystems. This has lead to the extensive use of linear algebra libraries such as LINPACK, EISPACK, and their common successor LAPACK [5]. To achieve portability, many scientific computer programs rely on using the common Basic Linear Algebra Subprograms (BLAS) for their vector, matrix-vector, and matrix-matrix operations. Today, efficient BLAS implementations are available for most computer architectures. Another reason for the importance of linear algebra is that the language of linear algebra provides a powerful formalism for expressing scientific computations, including many computations that are not commonly thought of as linear algebra computations. A prime example of the benefit of this approach is the use of matrix-vector notation to formulate Fast Fourier Transform algorithms [92].

One important application of linear algebra occurs in the solution of partial differential equations (PDEs) by finite difference, finite element, or finite volume methods. These require the repeated solution of systems of linear equations $Ax = b$, where $A$ is an $n \times n$ nonsingular matrix, and $x$ and $b$ are vectors of length $n$. Usually, the matrix $A$ is sparse, i.e., only $O(n)$ of its $n^2$ elements are nonzero. The system can be solved by a direct algorithm, using Cholesky factorisation in the case of a symmetric positive definite matrix $A$, or LU decomposition in the general case, see [32]. An alternative approach is to use an iterative algorithm, based on successive improvements of approximate solution vectors $x^{(k)}$. Two important iterative algorithms are the conjugate gradient algorithm [41] for symmetric positive definite matrices $A$ and the generalised minimal residual algorithm [77] for general matrices. Iterative methods use the matrix $A$ mainly in a multiplicative manner, by computing matrix-vector products of the form $u := Av$. Iterative methods are increasingly becoming popular, because they enable the solution of very
large linear systems such as those originating in PDE solving on large threedimensional grids. Discretising a PDE on a grid of $100 \times 100 \times 100$ points with one variable per grid point already leads to linear systems of one million equations in one million variables. Such systems may arise for instance in the simulation of oil reservoirs, in the modelling of semiconductor devices, and in aerodynamics computations. Iterative methods may often solve these systems within reasonable time and with acceptable memory use, because the systems are sparse and remain so during their solution, whereas direct methods will break down, because they create too many new nonzero elements in the matrix.

Another application of linear algebra in scientific computing is in the solution of linear programming (LP) problems, such as the problem of minimising the cost $c^T x$ under the constraint $A x \leq b$ (to be interpreted component-wise), where $A$ is an $m \times n$ matrix, $c$ and $x$ are vectors of length $n$, and $b$ is a vector of length $m$. This optimisation problem can be solved by the simplex method, which involves a sequence of rank-one updates of the form $A := A + uv^T$, or by an interior-point method, which involves multiplying the matrix $A$ by its transpose and solving a symmetric positive definite linear system of the form $AA^T u = v$. This system is usually solved by Cholesky factorisation (see [10] for a parallel implementation), although currently much research is being done on the applicability of iterative methods.

Linear algebra is also important in the field of molecular quantum chemistry, where various properties of molecules are determined from first principles by solving the time-independent Schrödinger equation, for instance by the direct SCF method. Although the dominant part of this computation is the calculation of 2-electron integrals and their incorporation into a Hamiltonian matrix, other important parts are the computation of the eigenvalues and eigenvectors of this matrix, and the multiplication of matrices. Since the latter parts are more difficult to parallelise than the trivially parallel integral calculations, they may well dominate the computing time on a parallel computer.

The examples above suggest that a first approach to achieving general purpose parallel computing for scientific applications may be based on developing BSP algorithms for linear algebra computations. For scientific applications that are not based on linear algebra, we may still be able to capture the essence of the computation in linear algebra language, so that we can use BSP techniques developed for linear algebra to gain further insight into these applications as well.
This chapter focuses on one particular linear algebra operation, sparse matrix-vector multiplication. This simple problem has numerous applications and is, hence, of major importance.

9.1.1 Iterative Methods

Sparse matrix-vector multiplication is the basis of iterative methods for the solution of sparse linear systems $Ax = b$. At every iteration, the matrix $A$ (and in certain cases its transpose) is multiplied by a vector, and the resulting vector is used to update the best current approximate solution. Similarly, this multiplication is also the basis for the Lanczos method [53], an iterative method which can be used to find the extremal eigenvalues of a sparse symmetric matrix (see e.g. [32]).

9.1.2 Partial Differential Equations

Sparse matrix-vector multiplication represents the execution of the finite difference operator in certain PDE solvers. This even holds in the common case of matrix-free solvers which do not form the finite difference matrix explicitly, but instead apply the finite difference operator directly on the current approximate solution vector. An example is the five-point Laplacian finite difference operator used to solve a second-order elliptic PDE on a two-dimensional grid of size $r \times r$. This operator can be formulated in matrix terms by defining an $n \times n$ matrix $A$, with $n = r^2$, by

$$a_{ij} = \begin{cases} 
-4 & \text{if } i = j \\
1 & \text{if } i = j \pm 1, j \pm r \\
0 & \text{otherwise.} 
\end{cases} \quad (9.1)$$

The solution value of the PDE at a grid point $(k, l), 0 \leq k, l < r$ corresponds to a component $x_i, 0 \leq i < n$, of the solution vector $x$ of a linear system $Ax = b$, by the relation $i = kr + l$. In a matrix-free PDE solver based on an iterative linear system solver, the equivalent of the sparse matrix-vector multiplication $u := Av$ will simply be executed by summing the values of $v$ in the neighbouring grid points $(k+1, l), (k-1, l), (k, l+1)$, and $(k, l-1)$, and subtracting from the result four times the value of $v$ in the grid point $(k, l)$, to produce the value of $u$ in that grid point.
9.1.3 Molecular Modelling

Molecular modelling is one of the most important practical applications of parallel architectures. Ab initio methods are based on quantum-mechanical calculations of molecular structure. The computation times required for such calculations, even with highly parallel architectures, are at present prohibitive for all but the smallest molecules. However, as the power of parallel machines increases we can expect a much wider use of this approach.

At present, most calculations of the structure and dynamics of proteins, carbohydrates and nucleic acids are carried out using molecular mechanics, molecular dynamics and free energy techniques. These methods adopt a force field approach to calculating the potential energy of a molecule in a given conformation. The AMBER force field [81] is quite typical. Let \( i \leftrightarrow j \) denote the (symmetric) relation that atom \( i \) is bonded to atom \( j \). Then in AMBER the potential energy is calculated as a sum of the following types of term.

- Bond energies for all \( i, j, i \leftrightarrow j \)
- Angle energies for all \( i, j, k, (i \leftrightarrow k) \land (k \leftrightarrow j) \)
- Torsional energies for all \( i, j, k, l, (k \leftrightarrow i) \land (i \leftrightarrow j) \land (j \leftrightarrow l) \)
- Coulombic potential for all \( i, j, (i \not\leftrightarrow j) \)
- Lennard-Jones potential for all \( i, j, (i \not\leftrightarrow j) \)
- Hydrogen bonding term for all \( i, j, (i \not\leftrightarrow j) \)

At an abstract level, the calculation of the potential energy of a molecular conformation can be described as follows: For each pair of atoms \( i, j \), given the coordinate and bond information we execute the following program.

```plaintext
if \( i \leftrightarrow j \) then calculate bond term;
for all \( k, l, (k \leftrightarrow i) \land (i \leftrightarrow j) \land (j \leftrightarrow l) \),
calculate torsional terms;
elseif \( (i \leftrightarrow k) \land (k \leftrightarrow j) \) then calculate angle energy;
else calculate Coulombic energy;
calculate Lennard-Jones potential;
calculate hydrogen bonding term;
```

84
Let \( d_{ij} \) denote the distance between atoms \( i, j \). The contributions from the Lennard-Jones and hydrogen bonding terms rapidly decline with increasing \( d_{ij} \) and can safely be neglected beyond a certain predefined cut-off distance. Unfortunately, the Coulombic potential does not decline quickly, and it is the calculation of these Coulombic terms for all non-bonded pairs which dominates the energy calculation for large molecules.

The above “all pairs” computation, if implemented directly, would yield an \( O(n^2) \) sequential algorithm.

Sparse matrix-vector multiplication may be used to model two-particle interactions of this kind. As an example, consider an orthogonal three-dimensional molecular dynamics universe of size \( 1 \times 1 \times 1 \) with periodic boundaries. The universe is filled with \( n \) particles, numbered \( 0 \leq i < n \). Each particle moves under the influence of the forces caused by the other particles. Each force is determined by a potential, such as the Lennard-Jones potential for nonbonded particles. Let \( F_{ij} \) denote the force on particle \( i \) due to particle \( j \), so that the total force on particle \( i \) is \( F_i = \sum_{j=0}^{n-1} F_{ij} \). Note that \( F_{ii} = 0 \). The force \( F_{ij} \) is a function of the position \( \mathbf{r}_i \) of particle \( i \) and the position \( \mathbf{r}_j \) of particle \( j \), \( F_{ij} = F(\mathbf{r}_i, \mathbf{r}_j) \). Therefore, to compute the force on a particle \( i \), one needs, besides the position of the particle \( i \) itself, the positions of all the other particles with which it interacts. The need for information about particle positions can be expressed in an \( n \times n \) matrix \( A \), defined by

\[
a_{ij} = \begin{cases} 
1 & \text{if } i = j \text{ or particles } i \text{ and } j \text{ interact} \\
0 & \text{otherwise.}
\end{cases}
\] (9.2)

An analogy to the force computation from the positions of the particles is the sparse matrix-vector multiplication \( \mathbf{u} := \mathbf{A}\mathbf{v} \), where \( \mathbf{u} \) is a vector that models the force components and \( \mathbf{v} \) is a vector that models the particle positions. For short-range potentials, there exists a cut-off radius \( r_c > 0 \), such that \( a_{ij} = 0 \) if the distance between particle \( i \) and particle \( j \) is larger than or equal to \( r_c \). For \( r_c \ll 1 \) this leads to \( \mathbf{A} \) being very sparse. The movement of the particles will cause the sparsity pattern of the matrix to change during the course of the simulation. All efficient simulation methods exploit the sparsity to limit the total number of force computations. Furthermore, distributed memory parallel algorithms based on geometric parallelism also exploit the sparsity to reduce the number of communications of current particle positions.

Simplifying molecular dynamics simulations by modelling their essence in matrix terms may give remarkable new insights, and may even lead to new ways of performing these simulations. A recent example of this approach is
the work of Hendrickson and Plimpton [39] on parallel many-body simulations (such as molecular dynamics). They achieve a reduction in communication volume by an order of $\sqrt{p}$, compared to all-to-all communication, by using techniques from dense linear algebra and carefully translating them to the many-body context. The main idea in their method is to cluster the force computations in a particular way, and to replace the all-to-all communication of particle positions by partial broadcasts of these positions and partial combines of accumulated forces. No sparsity is used to reduce the communication. (It is used, of course, to reduce the total number of force computations.) Because of this, the method is most suited for long-range or medium-range potentials, with a break-even point that is much more favourable than that of conventional all-to-all methods.

9.2 The MLIB Test Set of Sparse Matrices

Our motivation for developing a new library of sparse matrices, MLIB, came from the desire to mimic various areas of scientific computing in one common format and to use this format to investigate parallel scientific computing. The essential properties of problems in a wide range of application areas can often be captured in one sparse matrix or one family of matrices with the same structure. An example is the solution of a PDE on a regular two-dimensional $r \times r$ grid using the five-point Laplacian finite-difference operator, see Section 9.1. Taking the grid points as vertices and their neighbour relations as directed edges, while assuming periodic boundary conditions, we obtain a directed $r$-ary, two-dimensional hypercube graph. In general, PDE-solvers on regular grids give rise to hypercube graphs with a large radix and a low dimension. The adjacency matrix $A$ of such a graph is sparse and its size grows rapidly with increasing dimension or increasing radix. On a distributed-memory parallel computer it would be efficient to distribute the matrix and the related vectors by using the knowledge of the underlying neighbour structure of the graph. This may eliminate unnecessary communication of grid variables.

At present, there exists a library of sparse matrices, the Harwell-Boeing (HB) library [25, 26], which is widely used to test sparse matrix algorithms. It contains many examples of matrices that occur in practical applications. We have included a small subset of eleven matrices of the HB library in MLIB, mainly to facilitate our experiments on such practical matrices. For
our specific purpose of mimicking scientific computation, the HB matrices are not well suited, and therefore we decided to design our own library. We do not claim in any way that the matrix library MLIB is complete or representative. We present it as a first attempt to capture some features of scientific computing in the common format of sparse matrices.

The matrix library MLIB consists of 34 sparse matrices and their generating programs. Each matrix is represented by a file which contains the nonzero elements of the matrix stored by the coordinate scheme (see [24]). The element \(a_{ij} \neq 0\) is stored as a triple \((i, j, x)\) where \(i\) is the row index, \(j\) the column index, and \(x = a_{ij}\) the numerical value. The numerical values of MLIB are dummies, except in the case of the HB subset, which retains the original numerical values. At this stage, our interest is in sparsity patterns and their implications for parallel computing, and not in numerical issues. Nevertheless, we decided to include numerical values in the format, to enable possible future use of such values. The format of a matrix file is: first, a line containing the matrix size \(m \times n\); then the nonzeros, one per line; after that, a terminator line “−1”; and, optionally, additional information on the matrix, such as particle positions in the case of molecular dynamics matrices.

The MLIB library is available upon request from the authors. More details can be found in the documentation of the generating programs.

The MLIB library contains the following classes of matrices:

- **hyp.r.d.D**, the hypercube matrix with radix \(r\), dimension \(d\), and distance \(D\), \(r, d, D \geq 1\). For \(D = 1\), this is the adjacency matrix of the directed \(r\)-ary, \(d\)-dimensional hypercube graph. The vertices of this graph form a \(d\)-dimensional grid of \(n = r^d\) points; they are numbered lexicographically. Each vertex has directed edges to itself and to its immediate neighbours in each direction. The size of the hypercube matrix is \(n \times n\). For \(D > 1\), the hypercube graph is obtained by connecting each vertex to those vertices that can by reached by a path of length \(\leq D\) in the original \(D = 1\) graph. This models certain higher-order finite difference operators.

- **dense.n**, the dense matrix of size \(n \times n\). All elements of this matrix are nonzero.

- **random.n.\(\rho^{-1}\)**, an \(n \times n\) matrix with a random sparsity structure and a nonzero density \(\rho\).
• **hb.x**, the matrix x from the HB collection [25]. For a description of the matrix, see [26]. The subset of the HB collection that is included in MLIB consists of eleven matrices from various application fields. It is the same subset as the one used in [9] to test a parallel iterative linear system solver.

• **md.n.r_c^{-1}**, an \( n \times n \) matrix which corresponds to \( n \) particles in a three-dimensional molecular dynamics simulation with short-range potentials, see Section 9.1. The particles \( i \) and \( j \) interact, i.e. \( a_{ij} \neq 0 \), if \( \|r_i - r_j\| \leq r_c \), where \( r_i \) is the position of particle \( i \) and \( r_c \) the cut-off radius. The positions \( r_i = (x_i, y_i, z_i) \), with \( 0 \leq x_i, y_i, z_i \leq 1 \), are given at the end of the file. The interactions assume periodic boundaries.

• **mdr.n.r_c^{-1}.\rho^{-1}**, an \( n \times n \) matrix which corresponds to \( n \) particles in a three-dimensional molecular dynamics simulation with short-range potentials and, additionally, an artificial long-range potential for certain randomly selected particle pairs. The sparsity pattern of this matrix is the union of the sparsity patterns of a short-range molecular dynamics matrix with cut-off radius \( r_c \) and a random sparse matrix with density \( \rho \). Here, long-range interactions between selected particles represent interactions between distant clusters of particles. The aim of this procedure is to mimic e.g. multipole expansions.

• **lp.n**, an \( n \times n \) matrix which resembles certain symmetric matrices that occur in the solution of LP problems by interior point methods. The matrix is constructed by placing dense square submatrices of random size at random places in the matrix, with bias towards small sizes. This captures a structural feature that we observed in certain LP problems. We would like to add a disclaimer about this particular matrix class: one may argue about whether this represents the typical structure of LP matrices. Therefore, we present this type of matrix as just a first and modest attempt to capture some of the common characteristics of LP matrices.

Table 9.1 presents the size and the number of nonzeros of the 34 matrices from MLIB.
<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Order ( n )</th>
<th>Nonzeros ( nz(A) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>hyp.2.10.1</td>
<td>1024</td>
<td>11264</td>
</tr>
<tr>
<td>hyp.2.10.2</td>
<td>1024</td>
<td>57344</td>
</tr>
<tr>
<td>hyp.2.10.3</td>
<td>1024</td>
<td>180224</td>
</tr>
<tr>
<td>hyp.3.10.1</td>
<td>59049</td>
<td>1240029</td>
</tr>
<tr>
<td>hyp.3.8.1</td>
<td>6561</td>
<td>111537</td>
</tr>
<tr>
<td>hyp.20.4.1</td>
<td>160000</td>
<td>1440000</td>
</tr>
<tr>
<td>hyp.30.3.1</td>
<td>27000</td>
<td>189000</td>
</tr>
<tr>
<td>hyp.50.3.1</td>
<td>125000</td>
<td>875000</td>
</tr>
<tr>
<td>hyp.50.2.1</td>
<td>2500</td>
<td>12500</td>
</tr>
<tr>
<td>hyp.100.2.1</td>
<td>10000</td>
<td>50000</td>
</tr>
<tr>
<td>hyp.200.2.1</td>
<td>40000</td>
<td>200000</td>
</tr>
<tr>
<td>dense.100</td>
<td>100</td>
<td>10000</td>
</tr>
<tr>
<td>dense.500</td>
<td>500</td>
<td>250000</td>
</tr>
<tr>
<td>random.1000.1000</td>
<td>1000</td>
<td>1002</td>
</tr>
<tr>
<td>random.1000.100</td>
<td>1000</td>
<td>10013</td>
</tr>
<tr>
<td>random.1000.10</td>
<td>1000</td>
<td>100000</td>
</tr>
<tr>
<td>hb.impcolb</td>
<td>59</td>
<td>312</td>
</tr>
<tr>
<td>hb.west0067</td>
<td>67</td>
<td>294</td>
</tr>
<tr>
<td>hb.fs5411</td>
<td>541</td>
<td>4285</td>
</tr>
<tr>
<td>hb.steam2</td>
<td>600</td>
<td>13760</td>
</tr>
<tr>
<td>hb.shl400</td>
<td>663</td>
<td>1712</td>
</tr>
<tr>
<td>hb.bp1600</td>
<td>822</td>
<td>4841</td>
</tr>
<tr>
<td>hb.jpwh991</td>
<td>991</td>
<td>6027</td>
</tr>
<tr>
<td>hb.sherman1</td>
<td>1000</td>
<td>3750</td>
</tr>
<tr>
<td>hb.sherman2</td>
<td>1080</td>
<td>23094</td>
</tr>
<tr>
<td>hb.lns3937</td>
<td>3937</td>
<td>25407</td>
</tr>
<tr>
<td>hb.gemat11</td>
<td>4929</td>
<td>33185</td>
</tr>
<tr>
<td>lp.1000</td>
<td>1000</td>
<td>66512</td>
</tr>
<tr>
<td>lp.6000</td>
<td>6000</td>
<td>321256</td>
</tr>
<tr>
<td>md.6000.20</td>
<td>6000</td>
<td>25054</td>
</tr>
<tr>
<td>md.6000.10</td>
<td>6000</td>
<td>155592</td>
</tr>
<tr>
<td>md.6000.8</td>
<td>6000</td>
<td>300928</td>
</tr>
<tr>
<td>mdr.6000.10.2000</td>
<td>6000</td>
<td>175176</td>
</tr>
<tr>
<td>mdr.6000.8.1000</td>
<td>6000</td>
<td>337380</td>
</tr>
</tbody>
</table>

Table 9.1: Matrix library MLIB
9.3 Sparse Matrix-Vector Multiplication

In this section, we present a parallel algorithm for the multiplication of a sparse matrix $A$ and a dense vector $v$,

$$u := Av,$$  \hspace{1cm} (9.3)

which produces a dense vector $u$. The matrix $A = (a_{ij}, 0 \leq i, j < n)$ has size $n \times n$ and the vectors $u = (u_i, 0 \leq i < n)$ and $v = (v_i, 0 \leq i < n)$ have length $n$. We assume that the matrix is distributed by a Cartesian distribution. This means that the processors are numbered by two-dimensional identifiers $(s, t)$, with $0 \leq s < q_0$ and $0 \leq t < q_1$, where $p = q_0 q_1$ is the number of processors, and that there are mappings $\phi_0 : \{0, 1, \ldots, n - 1\} \rightarrow \{0, 1, \ldots, q_0 - 1\}$ and $\phi_1 : \{0, 1, \ldots, n - 1\} \rightarrow \{0, 1, \ldots, q_1 - 1\}$ such that matrix elements are distributed according to

$$a_{ij} \mapsto \text{processor}(\phi_0(i), \phi_1(j)).$$  \hspace{1cm} (9.4)

Note that the elements of a matrix row are mapped to processors with the same first identifier coordinate and that the elements of a matrix column are mapped to processors with the same second coordinate. This reflects the row-wise and column-wise nature of many linear algebra algorithms and this often leads to reduced communication requirements in linear algebra computations on distributed-memory parallel computers. This two-dimensional numbering of processors originates in special purpose algorithms for mesh networks of processors [9, 12]. In the present work, however, the two-dimensional numbering reflects a property of the problem to be solved and not of any particular network topology: the BSP-model is topology-independent. We assume that vectors are distributed in the same manner as the diagonal of the matrix, i.e. according to

$$u_i \mapsto \text{processor}(\phi_0(i), \phi_1(i)).$$  \hspace{1cm} (9.5)

This particular distribution scheme is flexible enough to accommodate many commonly used distribution methods while it is also sufficiently restrictive to impose efficient communication patterns. The flexibility is illustrated by the following two examples. The first example concerns the two-dimensional Laplacian operator. One often uses domain partitioning to split the corresponding discrete grid into blocks of grid points with the aim
of allocating blocks to processors. Since each grid point corresponds to one vector component, this amounts to distributing the vector over the processors in a locality-preserving manner. The complete row $i$ of the Laplacian matrix is usually allocated to the same processor as vector component $i$. In our scheme, this can simply be achieved by taking $q_0 = p$ and $q_1 = 1$. Another example is the square grid distribution, which is the matrix distribution defined by

$$\phi_0(i) = \phi_1(i) = i \mod q_0,$$

where $q_0 = q_1 = \sqrt{p}$. As we have seen, this distribution is efficient for linear algebra computations such as dense LU decomposition. This distribution is known under various names, such as scattered square decomposition and cyclic storage. (The grid distribution of a matrix should not be confused with discrete grids used to model e.g. PDE’s.) Our general distribution scheme leaves much freedom in choosing particular mappings, and this can be exploited to achieve a good load balance and to reduce communication, see the next section. A detailed discussion and motivation of this distribution scheme in the context of sparse matrix-vector multiplication is given in [9].

Figure 9.1 presents a BSP sparse matrix-vector multiplication algorithm. The algorithm consists of four supersteps: a fan-out of vector components to the processors that need them; a multiplication of the local part of the sparse matrix by the corresponding part of the input vector; a fan-in of partial sums; and, finally, the computation of the local part of the output vector. The fan-out and the fan-in are $h$-relations; the other supersteps are local computations. The communication requirements are derived from the computations on the basis of the “need to know”. Matrix elements are not communicated. The only communication needed is that of vector components and of partial sums used to compute new vector components. The input and output vectors are required to be distributed in the same manner. This facilitates repeated application of the algorithm, e.g. in an iterative linear system solver. The sparsity of the matrix is exploited in two ways: first, computations are performed only for nonzero elements; second, communications are performed only if the matrix element that makes them necessary is nonzero. The text given is the program text for a processor $(s, t)$, with $0 \leq s < q_0$ and $0 \leq t < q_1$. The execution of the program depends on the parameters $s$ and $t$.

The BSP cost of the sparse matrix-vector multiplication algorithm is determined as follows. The first superstep is the fan-out, which is a communi-
\{ A : n \times n, \text{distr}(A) = \phi, \\
v : n, \text{distr}(v) = \text{distr}(\text{diag}(A)) \} \\

\{ \text{fan-out} \} \\
\text{for all } j : 0 \leq j < n \land \phi_0(j) = s \land \phi_1(j) = t \text{ do} \\
\quad \text{send } v_j \text{ to processors } \{(\phi_0(i), t) : 0 \leq i < n \land a_{ij} \neq 0\}; \\

\{ \text{local sparse matrix-vector multiplication} \} \\
\text{for all } i : 0 \leq i < n \land \phi_0(i) = s \land (\exists r : 0 \leq r < n \land \phi_1(r) = t \land a_{ir} \neq 0) \text{ do} \\
\quad \text{begin} \\
\quad \quad u_{it} := 0; \\
\quad \quad \text{for all } j : 0 \leq j < n \land \phi_1(j) = t \land a_{ij} \neq 0 \text{ do } u_{it} := u_{it} + a_{ij}v_j \\
\quad \text{end}; \\

\{ \text{fan-in} \} \\
\text{for all } i : 0 \leq i < n \land \phi_0(i) = s \land u_{it} \neq 0 \text{ do} \\
\quad \text{send } u_{it} \text{ to processor } (s, \phi_1(i)); \\

\{ \text{summation of partial sums} \} \\
\text{for all } i : 0 \leq i < n \land \phi_0(i) = s \land \phi_1(i) = t \text{ do} \\
\quad \text{begin} \\
\quad \quad u_i := 0; \\
\quad \quad \text{for all } k : 0 \leq k < q_1 \land u_{ik} \neq 0 \text{ do } u_i := u_i + u_{ik} \\
\quad \text{end} \\

\{ u : n, u = Av, \text{distr}(u) = \text{distr}(v) \} \\

Figure 9.1: Sparse matrix-vector multiplication algorithm for processor \((s, t)\)
culation superstep. Let \( h_r(s, t) \) be the number of components \( v_j \) received by processor \((s, t)\) and \( h_s(s, t) \) the number of components sent. Then define

\[
    h_r = \max\{h_r(s, t) : 0 \leq s < q_0 \land 0 \leq t < q_1\},
\]

\( (9.7) \)

\[
    h_s = \max\{h_s(s, t) : 0 \leq s < q_0 \land 0 \leq t < q_1\},
\]

\( (9.8) \)

\[
    h = \max\{h_r, h_s\}.
\]

\( (9.9) \)

The BSP cost of the first superstep is \( l + h \cdot g \).

The second superstep is the local sparse matrix-vector multiplication, which is a computation superstep. Let

\[
    r_i(t) = |\{j : 0 \leq j < n \land a_{ij} \neq 0 \land \phi_1(j) = t\}|,
\]

\( (9.10) \)

be the number of nonzeros in processor part \( t \) of matrix row \( i \), \( 0 \leq i < n \).

Then the number of floating point operations of processor \((s, t)\) is

\[
    w(s, t) = \sum_{i = 0}^{n-1} (2r_i(t) - 1).
\]

\( (9.11) \)

In this operation count, we include only non-trivial floating point operations; we exclude trivial operations involving zero operands. The maximum amount of work of a processor is

\[
    w = \max\{w(s, t) : 0 \leq s < q_0 \land 0 \leq t < q_1\}.
\]

\( (9.12) \)

The BSP cost of the second superstep is \( w + l \).

The third superstep is similar to the first, except that partial sums \( u_{ik} \) are communicated, instead of vector components \( v_j \). The fourth superstep is similar to the second; its cost is determined as follows. Let

\[
    s_i = |\{k : 0 \leq k < q_1 \land u_{ik} \neq 0\}|,
\]

\( (9.13) \)

be the number of nonzero partial sums produced by matrix row \( i \), \( 0 \leq i < n \).

Then the number of floating point operations of processor \((s, t)\) is

\[
    w(s, t) = \sum_{i = 0}^{n-1} (s_i - 1).
\]

\( (9.14) \)
The total BSP cost of the algorithm is obtained by adding the costs of the four supersteps. We denote the BSP cost for $p$ processors by $T(p)$.

The BSP cost as defined above can be used to compare the efficiency of different distributions of the same matrix. To obtain a meaningful measure for comparison of different matrices it is necessary to normalise the cost. We define the *normalised BSP cost* $C(p)$ by

$$ C(p) = \frac{pT(p)}{T_{\text{seq}}}, $$

where $T_{\text{seq}}$ is the cost of the sequential algorithm. This sequential cost is defined by

$$ T_{\text{seq}} = \sum_{i = 0}^{n-1} (2r_i - 1), $$

where

$$ r_i = |\{j : 0 \leq j < n \land a_{ij} \neq 0\}|, $$

for $0 \leq i < n$. In other words, the normalised BSP cost $C(p)$ of an algorithm is the ratio between the time $T(p)$ of that algorithm on a BSP computer and the time $T_{\text{seq}}$ of a perfectly parallelised sequential algorithm. The normalised BSP cost of an algorithm is an expression of the form $a + bg + cl$, where $a$, $b$, and $c$ are scalar values which depend on the algorithm, on the number of processors, and on the chosen data distribution. The scalars $g$ and $l$ are parameters that characterise the hardware. The normalised BSP cost of an ideal parallel algorithm is $1 + 0g + 0l$.

In summary, we have presented a simple methodology that leads to a useful measure of the efficiency of BSP algorithms and distributions. This measure, the normalised BSP cost $C(p)$, can, of course, be used to distinguish good algorithms and distributions from bad ones, but also to identify easy and hard problems for BSP computers.
9.4 Results for Structure Independent Distributions

We have implemented a program that computes the normalised BSP cost $a + bg + cl$ of the sparse matrix-vector multiplication algorithm of Fig. 9.1 for a given sparse matrix and a given data distribution. In this section, we use this program to obtain experimental results on the performance of different data distribution schemes in a wide range of problem areas. Our cost statistics can be used to predict the computing time on an actual BSP computer, provided that the $g$ and $l$ parameters of the machine are available. For our experiments, we fix the number of processors at $p = 100$. The problem size, however, may vary, so that we are still able to investigate scalability.

Table 9.2 presents the normalised computing cost $a$ for seven different data distributions and for all sparse matrices from MLIB, cf. Table 9.1. Table 9.3 presents the normalised communication cost $b$ for the different data distributions and the normalised synchronisation cost $c$ for a distribution that requires all the four supersteps of the algorithm to be present. (For a row distribution, with $q_1 = 1$, there is no need for a fan-in and a summation of partial sums, so that the number of supersteps becomes two and $c$ is halved.) The value of $c$ depends only on the number of supersteps, the number of processors, and the amount of work of the sequential algorithm, but in general not on the chosen distribution. For all distributions, the vectors $u$ and $v$ are distributed in the same manner as the diagonal of the matrix. All distributions, except “PRAM”, are Cartesian, cf. eqn. 9.4.

The “PRAM” distribution is obtained by assigning nonzero elements randomly to the processors. This distribution is non-Cartesian, since in general there do not exist mappings $\phi_0$ and $\phi_1$ that satisfy eqn. 9.4. The “PRAM” distribution is included in the table, because it simulates the use of a BSP machine in PRAM mode, with randomised allocation of data by hashing. This mode of operation may be advantageous on machines with a low value of $g$. In Tables 9.2 and 9.3, following the column of the “PRAM” distribution, there are three columns with results for random distributions. The random distribution with $q_0 = 100$ and $q_1 = 1$ assigns matrix rows $i$ randomly to processors $(\phi_0(i), 0)$, with $0 \leq \phi_0(i) < 100$. The random distribution with $q_0 = q_1 = 10$ assigns an identifier $\phi_0(i)$, with $0 \leq \phi_0(i) < 10$, randomly to each matrix row $i$, and, independently, an identifier $\phi_1(j)$, with
<table>
<thead>
<tr>
<th>row distr.</th>
<th>PRAM</th>
<th>random 100</th>
<th>random 10</th>
<th>eq. random 10</th>
<th>grid 10</th>
<th>block 10</th>
<th>diag. 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>column distr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hyp.2.10.1</td>
<td>1.44</td>
<td>1.74</td>
<td>1.58</td>
<td>1.41</td>
<td>4.26</td>
<td>1.07</td>
<td>1.26</td>
</tr>
<tr>
<td>hyp.2.10.2</td>
<td>1.34</td>
<td>1.72</td>
<td>1.39</td>
<td>1.16</td>
<td>2.43</td>
<td>1.03</td>
<td>1.15</td>
</tr>
<tr>
<td>hyp.2.10.3</td>
<td>1.20</td>
<td>1.74</td>
<td>1.33</td>
<td>1.07</td>
<td>1.74</td>
<td>1.03</td>
<td>1.12</td>
</tr>
<tr>
<td>hyp.3.10.1</td>
<td>1.04</td>
<td>1.08</td>
<td>1.05</td>
<td>1.04</td>
<td>3.21</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>hyp.3.8.1</td>
<td>1.14</td>
<td>1.24</td>
<td>1.16</td>
<td>1.13</td>
<td>3.52</td>
<td>1.02</td>
<td>1.08</td>
</tr>
<tr>
<td>hyp.20.4.1</td>
<td>1.02</td>
<td>1.03</td>
<td>1.04</td>
<td>1.03</td>
<td>8.82</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>hyp.30.3.1</td>
<td>1.08</td>
<td>1.13</td>
<td>1.17</td>
<td>1.09</td>
<td>8.46</td>
<td>1.00</td>
<td>1.05</td>
</tr>
<tr>
<td>hyp.50.3.1</td>
<td>1.03</td>
<td>1.05</td>
<td>1.05</td>
<td>1.04</td>
<td>8.46</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>hyp.50.2.1</td>
<td>1.29</td>
<td>1.43</td>
<td>1.39</td>
<td>1.33</td>
<td>7.78</td>
<td>1.00</td>
<td>1.19</td>
</tr>
<tr>
<td>hyp.100.2.1</td>
<td>1.14</td>
<td>1.20</td>
<td>1.17</td>
<td>1.16</td>
<td>7.78</td>
<td>1.00</td>
<td>1.10</td>
</tr>
<tr>
<td>hyp.200.2.1</td>
<td>1.06</td>
<td>1.10</td>
<td>1.09</td>
<td>1.08</td>
<td>7.78</td>
<td>1.00</td>
<td>1.05</td>
</tr>
<tr>
<td>dense.100</td>
<td>2.14</td>
<td>4.03</td>
<td>2.41</td>
<td>1.12</td>
<td>1.41</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>dense.500</td>
<td>1.12</td>
<td>2.12</td>
<td>1.48</td>
<td>1.01</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>random.1000.1000</td>
<td>2.10</td>
<td>2.32</td>
<td>2.30</td>
<td>2.18</td>
<td>4.21</td>
<td>1.88</td>
<td>2.13</td>
</tr>
<tr>
<td>random.1000.100</td>
<td>1.48</td>
<td>1.77</td>
<td>1.61</td>
<td>1.46</td>
<td>4.00</td>
<td>1.29</td>
<td>1.28</td>
</tr>
<tr>
<td>random.1000.10</td>
<td>1.28</td>
<td>1.73</td>
<td>1.36</td>
<td>1.11</td>
<td>1.49</td>
<td>1.09</td>
<td>1.08</td>
</tr>
<tr>
<td>hb.impcolb</td>
<td>4.14</td>
<td>5.88</td>
<td>5.11</td>
<td>4.06</td>
<td>5.66</td>
<td>4.43</td>
<td>3.84</td>
</tr>
<tr>
<td>hb.west0067</td>
<td>3.90</td>
<td>5.33</td>
<td>4.70</td>
<td>3.74</td>
<td>7.29</td>
<td>3.84</td>
<td>3.42</td>
</tr>
<tr>
<td>hb.fs5411</td>
<td>1.69</td>
<td>2.12</td>
<td>2.71</td>
<td>2.50</td>
<td>6.41</td>
<td>2.42</td>
<td>2.24</td>
</tr>
<tr>
<td>hb.steam2</td>
<td>1.55</td>
<td>2.02</td>
<td>1.68</td>
<td>1.40</td>
<td>3.11</td>
<td>1.11</td>
<td>1.22</td>
</tr>
<tr>
<td>hb.shl400</td>
<td>4.98</td>
<td>31.52</td>
<td>5.05</td>
<td>4.74</td>
<td>6.99</td>
<td>4.38</td>
<td>4.74</td>
</tr>
<tr>
<td>hb.bp1600</td>
<td>2.24</td>
<td>7.73</td>
<td>2.32</td>
<td>2.15</td>
<td>4.64</td>
<td>2.34</td>
<td>2.11</td>
</tr>
<tr>
<td>hb.jpwh991</td>
<td>1.53</td>
<td>1.86</td>
<td>1.69</td>
<td>1.58</td>
<td>5.52</td>
<td>1.48</td>
<td>1.39</td>
</tr>
<tr>
<td>hb.sherman1</td>
<td>1.61</td>
<td>1.93</td>
<td>1.80</td>
<td>1.68</td>
<td>11.29</td>
<td>1.85</td>
<td>1.52</td>
</tr>
<tr>
<td>hb.sherman2</td>
<td>1.47</td>
<td>1.84</td>
<td>1.51</td>
<td>1.34</td>
<td>3.79</td>
<td>1.27</td>
<td>1.27</td>
</tr>
<tr>
<td>hb.lns3937</td>
<td>1.25</td>
<td>1.25</td>
<td>1.31</td>
<td>1.27</td>
<td>4.61</td>
<td>1.62</td>
<td>1.21</td>
</tr>
<tr>
<td>hb.gemat11</td>
<td>1.23</td>
<td>1.33</td>
<td>1.26</td>
<td>1.24</td>
<td>4.30</td>
<td>1.28</td>
<td>1.18</td>
</tr>
<tr>
<td>lp.1000</td>
<td>1.37</td>
<td>2.08</td>
<td>1.49</td>
<td>1.34</td>
<td>1.72</td>
<td>1.93</td>
<td>1.31</td>
</tr>
<tr>
<td>lp.6000</td>
<td>1.14</td>
<td>1.43</td>
<td>1.18</td>
<td>1.15</td>
<td>1.81</td>
<td>1.42</td>
<td>1.15</td>
</tr>
<tr>
<td>md.6000.20</td>
<td>1.21</td>
<td>1.31</td>
<td>1.26</td>
<td>1.25</td>
<td>5.78</td>
<td>1.19</td>
<td>1.18</td>
</tr>
<tr>
<td>md.6000.10</td>
<td>1.14</td>
<td>1.26</td>
<td>1.15</td>
<td>1.11</td>
<td>2.83</td>
<td>1.07</td>
<td>1.07</td>
</tr>
<tr>
<td>md.6000.8</td>
<td>1.11</td>
<td>1.26</td>
<td>1.11</td>
<td>1.08</td>
<td>2.04</td>
<td>1.05</td>
<td>1.05</td>
</tr>
<tr>
<td>mdr.6000.10.2000</td>
<td>1.13</td>
<td>1.25</td>
<td>1.14</td>
<td>1.11</td>
<td>2.71</td>
<td>1.06</td>
<td>1.06</td>
</tr>
<tr>
<td>mdr.6000.8.1000</td>
<td>1.11</td>
<td>1.26</td>
<td>1.11</td>
<td>1.07</td>
<td>1.91</td>
<td>1.05</td>
<td>1.04</td>
</tr>
</tbody>
</table>

Table 9.2: Computation cost for data distributions with $p = 100$
Table 9.3: Communication and synchronisation cost for data distributions with \( p = 100 \)
\(0 \leq \phi_1(j) < 10\), to each matrix column \(j\). An equalised random distribution of rows is similar to a random distribution, but it assigns the same number of rows to each identifier, if \(n \mod q_0 = 0\). Otherwise, the number of rows will differ by at most one. This procedure is equivalent to randomly permuting the rows and then distributing them according to the block distribution \(\phi_0(i) = i \div \ell\), where \(\ell = n/q_0\) and it is assumed that \(n \mod q_0 = 0\).

The cost results for the random distributions are the averages over 100 runs of the random distribution program. The standard deviations are small, so that we consider the results to be reliable.

Tables 9.2 and 9.3 also present results for two deterministic distributions: the grid/grid distribution, which is the square grid distribution of eqn. 9.6, and the block/grid distribution, which is defined for the general case by

\[
\ell_0 = \left\lfloor \frac{n}{q_0} \right\rfloor, \quad \ell_1 = \left\lceil \frac{n}{q_0} \right\rceil, \quad r = n \mod q_0,
\]

(9.18)

\[
\phi_0(i) = \begin{cases} 
    i \div \ell_1 & \text{if } i < r\ell_1, \\
    r + (i - r\ell_1) \div \ell_0 & \text{if } i \geq r\ell_1,
\end{cases}
\]

(9.19)

\[
\phi_1(i) = i \mod q_1, \text{ for } 0 \leq i < n.
\]

(9.20)

This distribution allocates rows in consecutive blocks to processors, and columns in a cyclic fashion. It was proposed as a suitable distribution for iterative linear system solvers [9], because it distributes the matrix diagonal over all the processors so that it can easily be matched with a vector distribution. (The square grid distribution does not have this advantage, because it distributes the diagonal over only \(\sqrt{p}\) processors.) Finally, Tables 9.2 and 9.3 present a column with the results for the “diagonal” distribution. This distribution is determined by taking an equalised random distribution of the matrix diagonal over the processors. Note that in our distribution scheme, for a given choice of \(q_0\) and \(q_1\), the distribution of the matrix diagonal fully determines the distribution of the complete matrix and that of the vectors, see eqns 9.4 and 9.5.

The results of Table 9.2 show that it is relatively easy to obtain a good load balance, i.e. \(a \approx 1\), and hence a minimal computation cost, except for very small matrices such as dense.100, hb.impcolb, and hb.west0067, and for extremely sparse ones such as hb.shl400. Most distributions lead to a normalised computation cost of between one and two. The exception is the square grid distribution, which leads to excessive workloads on diagonal
processors \((s, s)\) in the summation of the partial sums, because these are the only processors that participate in this superstep. (Note that this is directly related to the heavy communication obligations of the diagonal processors in the fan-in, since these processors are the only receivers of data.) A breakdown of the total BSP cost into the contributions of the separate supersteps confirms this analysis. Furthermore, it shows that the load balance of the grid distribution in the local sparse matrix-vector multiplication is about the same as that of the other distributions, except in the case of matrices with an unfavourable nonzero structure. This may occur if there is a correlation between the row and the column nonzero structures, resulting e.g. in diagonals of nonzeros. This may lead to a bad load balance for certain numbers of processors. This phenomenon can be observed for some of the hypercube matrices and the \texttt{hb.sherman} matrices. Furthermore, Table 9.2 shows that equalised random distributions lead to a better load balance than standard random distributions. In general, distributions that impose constraints balance the workload better. For example, the “PRAM” distribution does not impose any constraints except for an identical distribution of matrix diagonal and vectors. It does not perform very well on small problems and even for larger problems there are superior distributions, such as the “diagonal” distribution, which imposes an equal division of the matrix diagonal over the processors and hence causes a good load balance in the summation of partial sums.

The results of Table 9.3 show that it is quite hard to achieve a low communication cost for general sparse matrices, i.e. if one cannot exploit any structural knowledge about the matrix. Even for the best structure independent distributions, block/grid and “diagonal”, one needs a BSP computer with \(g \leq 10\) to solve most problems efficiently. The best performance is obtained by square distributions, i.e. distributions with \(q_0 = q_1 = \sqrt{p}\). This leads to a factor of \(\sqrt{p}/2\) communication reduction for dense [9] and general sparse matrices, compared to a row distribution. This is due to a \(\sqrt{p}\)-fold increase in the reuse of communicated data, at the cost of an extra communication phase, the fan-in. (A similar analysis can be performed for the BSP model.) This effect can most clearly be seen by comparing the random distribution for \(q_0 = 100\) and \(q_1 = 1\) with the random/random distribution for \(q_0 = q_1 = 10\), in particular for relatively dense matrices such as \texttt{hyp.2.10.3}, \texttt{dense.500}, \texttt{random.1000.10}, \texttt{md.6000.8}, and \texttt{md.6000.8.1000}. On the other hand, for very sparse matrices such as \texttt{random.1000.1000} and \texttt{md.6000.2}, the introduction of the fan-in for \(q_1 > 1\) doubles the communication, without much
compensation by reuse of data. The “PRAM” distribution performs poorly, because nearly all the vector data must be fetched from non-local memories. This distribution is viable only if \( g \) is very close to one. Again, the square grid distribution is the worst distribution: the diagonal processors are the only ones that send data in the fan-out, and they are also the only ones that receive data in the fan-in; this may degrade performance by a factor of \( \sqrt{p} \).

The best distributions are the block/grid distribution and the “diagonal” distribution. They perform equally well for problems that have a random nature, such as the random, md, and mdr matrices. For problems that have some local structure that is reflected in the matrix, the block/grid distribution is able to discover part of this structure and to exploit it, to some extent. This can be observed for the hyp matrices, hb.steam2, and the hb.sherman matrices, which are all derived from multidimensional grids. Obviously, the random construction of the “diagonal” distribution prevents discovery of any structure. In a few cases, hb.fs5411 and hb.lns3937, the block/grid distribution is outperformed by the “diagonal” one; this may be caused by an unfavourable structure that does not suit the block/grid distribution.

The synchronisation cost of the sparse matrix-vector multiplication is low, because it has at most four supersteps. The normalised synchronisation cost is

\[
    c \approx \frac{4}{2 \text{nz}(A)/p} = \frac{2p}{\text{nz}(A)}.
\]

This implies that problems with more than 200,000 nonzeros can be solved efficiently on a 100-processor BSP computer with \( l \leq 1000 \).

## 9.5 Results for Structure Dependent Distributions

Table 9.4 shows the normalised communication cost for hypercube matrices of distance one and dimension \( d = 2, 3, 4 \), distributed by domain partitioning of the corresponding hypercube graph. The radix \( r \) is the number of points in each dimension, and \( P_k \), \( 0 \leq k < d \), is the number of subdomains into which dimension \( k \) is split. For example, the first line of the table gives the cost for a \( 50 \times 50 \) grid that is split into \( 50 \times 2 \) blocks, each of size \( 1 \times 25 \). In all cases, we choose \( q_0 = p \) and \( q_1 = 1 \), because we found no advantage in other choices of \( q_0 \) and \( q_1 \) for domain distribution of hypercube matrices.
of distance one. The distribution of the grid points and hence of the vector components uniquely determines the distribution of the matrix.

The results of Table 9.4 show that the lowest communication cost for separate dimension splitting is achieved if the resulting blocks are cubic. This is an immediate consequence of the surface-to-volume effect, where the communication across the block boundaries grows as the number of points near the surface, and the computation as the number of points within the volume of the block. In two dimensions, partitioning the grid into square blocks of size $r/\sqrt{p} \times r/\sqrt{p}$ reduces the communication by a factor of about $\sqrt{p}/2$, compared to splitting it into strips of size $r/p \times r$. This can be seen for example in the reduction by a factor of five for the $200 \times 200$ hypercube grid, comparing the cost for $P_0 = 100, P_1 = 1$ with that for $P_0 = 10, P_1 = 10$. The surface-to-volume ratio for cubes in dimension $d$ is $2dp^{1/d}/r$. For each grid point, $4d + 1$ floating point operations must be performed. The value $h$ of the $h$-relation to be realised equals the number of exterior boundary points, because all the values of these points must be received. By symmetry, the same argument holds for sending. Therefore, the normalised communication cost for cubic partitioning is

$$b = \frac{2dp^{1/d}}{(4d + 1)r} \approx \frac{p^{1/d}}{2r}.$$  (9.22)

This formula explains the results for $d = 2$ and $P_0 = P_1 = 10$ in Table 9.4. It implies for instance that two-dimensional grid problems with more than 45 grid points per direction can be solved efficiently on 100-processor BSP computers with $g \leq 10$. This indicates that PDE solving on such a BSP computer is feasible, already for relatively small problem sizes.

It is possible to improve the distribution further, by partitioning the domain along specific hyperplanes, not necessarily parallel to the coordinate hyperplanes. (Note that this implies that the dimensions are not split up separately.) An example is the case of the two-dimensional hypercube grid, which can be split into digital spheres of the form

$$B_R(\mathbf{a}) = \{ \mathbf{x} \in \mathbb{Z}^2 : \|\mathbf{x} - \mathbf{a}\|_1 \leq R \},$$  (9.23)

where the norm in dimension $d$ is defined by $\|\mathbf{x}\|_1 = \sum_{i=0}^{d-1} |x_i|$. In other words, all grid points with a Manhattan distance less than or equal to $R$ to the centre $\mathbf{a}$ of such a sphere are allocated to the same processor. The spheres wrap around the boundaries of the grid. Figure 9.2 illustrates this distribution.
Table 9.4: Communication cost for low-dimensional hypercube matrices with domain partitioning for $p = 100$

For an infinite grid, the centers of the spheres form a lattice, consisting of all integer linear combinations of the vectors $v_0 = (R + 1, R)$ and $v_1 = (-R, R + 1)$. Together the spheres form a tiling of the plane $\mathbb{Z}^2$. The advantage of tile partitioning over block partitioning is that there are a factor of $\sqrt{2}$ less points in the boundary layer, for sufficiently large partition sizes. Therefore, the normalised communication cost $b$ is reduced by a factor of $\sqrt{2}$. For the example of Fig. 9.2, the cost is $b \approx 0.071$, compared to $b \approx 0.088$ for the corresponding block partitioning. Fig. 9.3 shows the normalised communication cost for the two distributions as a function of the number of grid points per processor. The tile distribution is clearly superior, showing for instance a reduction by a factor of 1.34 for 221 grid points per processor. Note that problems of this size can be solved efficiently on BSP computers with $g \leq 50$, provided the tile distribution is used.

A complication that should be mentioned is that there may be a mismatch between the number of processors and the size of the grid. A perfect block distribution is possible only for very specific (square) numbers of grid points per processor, and similarly a perfect tile distribution is possible only for $2R^2 + 2R + 1$ grid points per processor, with $R$ a non-negative integer. In
the non-ideal case, a good distribution can still be obtained by splitting the plane along diagonal lines at suitable distances and assigning grid points accordingly.

Table 9.5 shows the BSP cost for various distributions of the molecular dynamics matrix \texttt{md.6000.10}. This matrix represents a three-dimensional universe of 6000 particles, contained in a box of size $1 \times 1 \times 1$ with periodic boundary conditions. Particles interact if their distance is less than $r_c = 0.1$. For convenience, the upper part of the table repeats the cost results for a few structure independent distributions from Tables 9.2 and 9.3. The lower part of the table presents the cost of structure dependent distributions; these exploit additional knowledge about the particle positions to assign particles to subdomains and hence to processors.

The results for the structure independent distributions show that they
Figure 9.3: Communication cost comparison between block distribution and tile distribution of two-dimensional hypercube matrices.
achieve a good load balance but that they suffer from large amounts of communication. Even the best distributions of this type, block/grid and "diagonal", need BSP computers with a low value of $g$, $g \leq 3$, to prevent communication dominance. One can view these distributions as being based on so-called particle parallelism. Another approach is to distribute particles by using geometric parallelism. This leads to structure dependent distributions as given in the lower part of the table. These distributions have lower communication requirements, but the price to be paid is a possible deterioration of the load balance, due to an inhomogeneous particle density.

Table 9.5 indicates that cubic subdomains are optimal among the orthogonal partitioning schemes, i.e., those schemes that split each dimension separately. Note that for non-cubic subdomains such as slabs or piles, choosing a square Cartesian distribution (with $q_0 = q_1$) improves the performance significantly. This is also done by Hendrickson and Plimpton [39] in the case of particle parallelism. For cubic subdomains, communication requirements are already reduced to such a low level, that this procedure, based on aggregation of partial sums, does not lead to further improvement. Note that the cut-off radius $r_c = 0.1$ of this matrix is quite large compared to the subdomain size. For partitioning into slabs this implies that particle information must be sent to 20 other processors (so that it pays to aggregate information); for piles it must be sent to 4–8 other processors, depending on the position; and for near-cubes to 2–6 processors. (In our discussion we ignore the symmetry of particle interactions, which may be used to reduce the computation and the communication by a factor of two.)

It is possible to further improve the distribution by allowing cuts of the domain in any direction. This can be done efficiently by taking a suitable sphere packing lattice [19] and assigning particles to the nearest centre of a sphere. (Sphere packing lattices have been used in other areas of scientific computing; for instance, it has been proposed to use them to decrease anisotropy in pseudo-spectral PDE solving on multidimensional grids.) This method splits the universe into Voronoi cells, each of which corresponds to a processor.

For the cube distribution, the universe is split into cubes of size $p^{-1/3} \times p^{-1/3} \times p^{-1/3}$. This perfect splitting is, of course, only possible if the number of processors $p$ is a cube. For the sphere packing distribution, we used a body-centred-cubic (bcc) lattice, defined by three basis vectors $v_0 = (2, 0, 0)$, $v_1 = (0, 2, 0)$, and $v_2 = (1, 1, 1)$. The lattice is scaled by a factor $\lambda = (p/2)^{-1/3}$.
<table>
<thead>
<tr>
<th>distribution</th>
<th>$q_0$</th>
<th>$q_1$</th>
<th>comp. (in g)</th>
<th>comm. (in l)</th>
<th>synch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRAM</td>
<td>1.14</td>
<td>1.087</td>
<td>0.0013</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random row</td>
<td></td>
<td></td>
<td>1.26</td>
<td>0.541</td>
<td>0.0007</td>
</tr>
<tr>
<td>eq. random row and column</td>
<td>10</td>
<td>1</td>
<td>1.11</td>
<td>0.426</td>
<td>0.0013</td>
</tr>
<tr>
<td>block/grid</td>
<td>10</td>
<td>10</td>
<td>1.07</td>
<td>0.358</td>
<td>0.0013</td>
</tr>
<tr>
<td>diagonal</td>
<td>10</td>
<td>10</td>
<td>1.07</td>
<td>0.337</td>
<td>0.0013</td>
</tr>
<tr>
<td>slabs of size $0.01 \times 1.0 \times 1.0$</td>
<td>100</td>
<td>1</td>
<td>1.34</td>
<td>0.320</td>
<td>0.0007</td>
</tr>
<tr>
<td>slabs of size $0.01 \times 1.0 \times 1.0$</td>
<td>10</td>
<td>10</td>
<td>1.28</td>
<td>0.259</td>
<td>0.0013</td>
</tr>
<tr>
<td>piles of size $0.1 \times 0.1 \times 1.0$</td>
<td>100</td>
<td>1</td>
<td>1.41</td>
<td>0.108</td>
<td>0.0007</td>
</tr>
<tr>
<td>piles of size $0.1 \times 0.1 \times 1.0$</td>
<td>10</td>
<td>10</td>
<td>1.41</td>
<td>0.081</td>
<td>0.0013</td>
</tr>
<tr>
<td>near-cubes of size $0.2 \times 0.2 \times 0.25$</td>
<td>100</td>
<td>1</td>
<td>1.54</td>
<td>0.075</td>
<td>0.0007</td>
</tr>
<tr>
<td>near-cubes of size $0.2 \times 0.2 \times 0.25$</td>
<td>10</td>
<td>10</td>
<td>1.54</td>
<td>0.087</td>
<td>0.0013</td>
</tr>
</tbody>
</table>

Table 9.5: Normalised BSP cost for distributions with $p = 100$ of the matrix $md_{6000 \times 10}$

This leads to a perfect splitting of the box if $p$ equals two times a cube. The figure shows that the bcc distribution is slightly superior. (Note that this figure is based on experiments for one randomly generated $md$ matrix and not on the average for a set of randomly generated matrices.) We chose the bcc lattice for this experiment because it has been conjectured that it solves the sphere covering problem. Our results indicate that sphere packing techniques may be useful in distributing physical domains over the processor of a parallel computer. This holds in particular for a BSP computer, because it liberates us from considerations of network locality. Therefore, there is no need for rigid partitioning schemes that produce highly regular domains. Further investigation of this issue is needed; for instance, there may exist better lattices for our purpose. Furthermore, to be useful in practice, finite-size effects must be taken care of.

### 9.6 Asymptotic Analysis

Let $C(r, d)$ denote the adjacency matrix of the directed $r$-ary, $d$-dimensional hypercube graph. The nodes of this graph form a $d$-dimensional grid of $n = r^d$ points which are numbered lexicographically. Each node has directed arcs to itself and to its immediate neighbours in each dimension. For the purposes of discussion we will consider just four $n \times n$ sparse matrices. Three
of the four are instances of \( C(r, d) \). They are: 2D-MESH = \( C(n^{1/2}, 2) \), 3D-MESH = \( C(n^{1/3}, 3) \) and HYPERCUBE = \( C(2, \log n) \). Matrices of this kind are often used to model finite-difference operators in the solution of partial differential equations [11]. The fourth matrix has a random structure in which each row and each column contains four nonzeros. We will refer to it as the EXPANDER matrix. [Note. The value four is not particularly significant. We could have chosen any small integer value greater than one.]

The above analysis shows that, compared with random data distributions, matrix-based distributions such as block-grid can offer some reductions in the BSP communication cost \( H(n, p) \) of most sparse matrix-vector multiplication problems. It also shows that in many cases, much more significant reductions in \( H(n, p) \) can be achieved by using a data distribution based on an efficient decomposition of the underlying graph. For example, the nodes of the 2D-MESH graph can be partitioned into \( p \) regions, each of which corresponds to a 2D-MESH on \( n/p \) nodes. The corresponding data distribution for matrix elements gives a BSP algorithm for \( u = M \cdot v \), where \( M \) is the 2D-MESH matrix, with total cost \( n/p + (n^{1/2}/p^{1/2}) \cdot g + l \). The same approach, applied to the 3D-MESH matrix, gives a BSP algorithm with total cost \( n/p + (n^{2/3}/p^{2/3}) \cdot g + l \) and, applied to the HYPERCUBE matrix, gives a BSP algorithm with total cost \( (n \log n)/p + ((n \log p)/p) \cdot g + l \). In each case we minimise the communication cost of the algorithm by partitioning the nodes of the graph into \( p \) equal sized subsets in a way which minimises the number of arcs between different subsets. Lower bounds on the efficiency of such partitions can be derived from known isoperimetric inequalities in graph theory, see e.g. [14].

For the EXPANDER matrix, the best upper bound which we have is the trivial one, \( n/p + (n/p) \cdot g + l \), which can be obtained by randomly distributing the matrix elements. Techniques similar to those in [60] can be used to show that for the EXPANDER matrix there is no partition of the nodes into \( p \) equal sized subsets which gives a value for \( H(n, p) \) which is less than the trivial \( n/p \). Therefore, for the EXPANDER matrix, \( u = M \cdot v \) is an inherently non-local problem.

### 9.7 Conclusions

In this chapter we theoretically and experimentally analysed the efficiency with which a wide range of important scientific computations can be per-
formed on bulk synchronous architectures. The computations considered include the iterative solution of sparse linear systems, molecular dynamics, linear programming, and the solution of partial differential equations on a multidimensional discrete grid.

The analysis shows that the exploitation of knowledge about the underlying structure of the problem is the key to achieving efficient parallel computations on a BSP computer. We have shown that grid computations and molecular dynamics simulations are feasible on BSP computers with realistic values for the machine characteristics \( g \) and \( t \). Therefore, the BSP computers that can be built in the foreseeable future will be able to solve problems from several important problem classes. Highly irregular scientific computing problems without a known structure are much harder to solve on BSP computers. We have introduced two distributions, block/grid and “diagonal”, see Section 9.4, that perform reasonably well on a variety of such problems. Our results show that structure independent parallel computations require extremely high communication performance and demand values of \( g \) that at present are difficult to achieve. This holds even more for the PRAM approach, which completely ignores the problem structure.

Providing a library of parallel algorithms to solve general sparse problems is a first step towards efficient parallel scientific computing, but to make further progress, this should be combined with developing algorithms that find structure in the problems. The BSP model facilitates developing such algorithms, because it focuses attention on the partitioning of the problem to be solved and not on the mapping to any particular hardware.

The initial techniques and results described here show clearly that the network independent approach of the BSP model gives rise to a whole range of interesting new theoretical questions concerning load balancing, communication complexity, and domain partitioning for parallel scientific computing. In contrast to the many network specific (e.g. hypercube, mesh, or butterfly) process mapping and domain decomposition methods which were developed over the last decade, the techniques and results described here have an advantage in that they are of relevance to any parallel computing system.
Chapter 10

The PRAM Model and Virtual Shared Memory

Various idealised shared memory models of parallel computation have been used in the study of parallel algorithms and their complexity. Three such models are the PRAM, the circuit, and the comparison network. In this section we describe the PRAM and circuit models, and give a number of simple examples of efficient shared memory parallel algorithms which can be implemented on them. Most of the circuits described can be translated into PRAM algorithms in a straightforward manner. We also discuss various ways in which the efficiency of shared memory parallel algorithms can be measured. The class $\mathcal{NC}$ has, over the past two decades, provided a very simple and robust framework for the classification of problems in $\mathcal{P}$, in terms of their parallel time complexity on a PRAM. A large number of important problems have been shown to lie in $\mathcal{NC}$, i.e. to be solvable on a PRAM in polylogarithmic time using a polynomial number of processors. Other problems have been shown to be $\mathcal{P}$-complete, i.e. to have no $\mathcal{NC}$ algorithm unless $\mathcal{P} = \mathcal{NC}$. The class $\mathcal{NC}$ and the notion of $\mathcal{P}$-completeness have allowed major advances to be made in our theoretical understanding of shared memory parallel algorithms and their complexity.

10.1 The PRAM Model

A parallel random access machine (PRAM) [31, 44, 76] consists of a collection of processors which compute synchronously in parallel and which communi-
cate with a common global random access memory. In one time step, each processor can do (any subset of) the following - read two values from the common memory, perform a simple two-argument operation, write a value back to the common memory. There is no explicit communication between processors. Processors can only communicate by writing to, and reading from, the common memory. The processors have no local memory other than a small fixed number of registers which they use to temporarily store the argument and result values. In a Concurrent Read Concurrent Write (CRCW) PRAM, any number of processors can read from, or write to, a given memory cell in a single time step. In a Concurrent Read Exclusive Write (CREW) PRAM, at most one processor can write to a given memory cell at any one time. In the most restricted model, the Exclusive Read Exclusive Write (EREW) PRAM, no concurrency is permitted either in reading or in writing. The CRCW PRAM model has a large number of variants which differ in the convention they adopt for the effect of concurrent writing. Three simple examples of such conventions are: two or more processors can write so long as they write the same value, one of the processors attempting to write will succeed but the choice of which one will succeed will be made nondeterministically, the lowest numbered processor will succeed (assuming some appropriate numbering.) In other CRCW models [73] one might have the possibility of concurrent writing in which the memory location is updated to the sum of the written values, or to the minimum of the written values.

As a simple example of a CREW PRAM computation, consider the problem of computing $ab + ac + bd + cd$ from inputs $a, b, c, d$. Let $p_i t_j$ denote the computation performed by processor $i$ at time step $j$. Then we have

$$
\begin{align*}
  p_{1t_1} : & b + c \Rightarrow x \\
  p_{1t_2} : & a \ast x \Rightarrow y \\
  p_{2t_2} : & x \ast d \Rightarrow z \\
  p_{1t_3} : & y + z \Rightarrow \text{result}
\end{align*}
$$

The complexity of a PRAM algorithm is given in terms of the number of time steps and the maximum number of processors required in any one of those time steps. The above example requires three time steps and two processors.

The most important characteristic of the PRAM model is that it is a 1-level memory (or shared memory) model, i.e. all of the memory locations are uniformly far away from all of the processors, the processors have no
local memory and there is no kind of memory hierarchy based on ideas of network locality. These simplifying properties of the PRAM model have made it extremely attractive as a robust model for the design and analysis of algorithms.

10.2 Circuits

A circuit \cite{27} is a directed acyclic graph with \( n \) input nodes (in-degree 0) corresponding to the \( n \) inputs to the problem, and a number of gates (in-degree 2) corresponding to two-argument functions. In a Boolean circuit, the gates are labelled with one of the binary Boolean functions \( \text{NAND}, \land, \text{NOR}, \lor, \rightarrow \), \( \oplus \) etc. In a typical arithmetic circuit, the input nodes are labelled with some value from \( \mathbb{Q} \), the set of rational numbers, and the gates are labelled with some operation from the set \( \{+,-,*,/\} \). The size of a circuit is the number of gates.

Let \( g_i \) denote the function computed by gate \( i \). Then we have the following arithmetic circuit for \( ab + ac + bd + cd \).

\[
\begin{align*}
g_1 &= b + c \\
g_2 &= a \times g_1 \\
g_3 &= g_1 \times d \\
g_4 &= g_2 + g_3 
\end{align*}
\]

An example of a Boolean circuit is the following, which computes the two binary digits \( <d_1, d_0> \) of \( x_1 + x_2 + x_3 \).

\[
\begin{align*}
g_1 &= x_1 \land x_2 \\
g_2 &= x_1 \oplus x_2 \\
g_3 &= g_2 \land x_3 \\
g_4 &= g_2 \oplus x_3 (= d_0) \\
g_5 &= g_1 \lor g_3 (= d_1) 
\end{align*}
\]

The parallel complexity of a circuit is the depth of the circuit, i.e. the maximum number of gates on any directed path. The parallel complexity of both of the above examples is three.
10.3 Shared Memory Algorithms

10.3.1 Addition

Let \( \text{ADD}_n(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i \) where \( x_i \in \mathbb{Q} \). A circuit of depth \( \lceil \log_2 n \rceil \) for \( \text{ADD}_n \) can easily be obtained by constructing a balanced binary tree of \(+\)-gates, with \( n \) leaves corresponding to the arguments \( x_1, \ldots, x_n \). This corresponds to an \( n/2 \) processor PRAM algorithm with complexity \( \lceil \log_2 n \rceil \) for \( \text{ADD}_n \). The optimality of this construction, in terms of depth, follows from the functional dependency of \( \text{ADD}_n \) on each of its \( n \) arguments.

If we now define \( \text{OR}_n(x_1, \ldots, x_n) = \bigvee_{i=1}^{n} x_i \) where \( x_i \in \{0, 1\} \), then we have a very similar problem to that of computing \( \text{ADD}_n \). We can easily obtain a PRAM algorithm of complexity \( \lceil \log_2 n \rceil \) and a Boolean circuit of depth \( \lceil \log_2 n \rceil \) for \( \text{OR}_n \). That this circuit depth is optimal follows from functional dependency. However, as Cook and Dwork [21] observed, there is rather more to the question of the PRAM complexity of \( \text{OR}_n \). If we allow concurrent write, then \( \text{OR}_n \) can be computed in one parallel step in an obvious way; processor \( i \) reads \( x_i \) from memory location \( i \) and if \( x_i = 1 \) it writes a 1 into location 0. Cook and Dwork [21] show that even on an EREW PRAM, \( \text{OR}_n \) can be computed in less than \( \lceil \log_2 n \rceil \) steps. They derive an upper bound of \((0 \cdot 72) \log_2 n\) on the number of steps required. However, they also show that a lower bound of \( \Omega(\log_2 n) \) holds, and thus only a constant factor improvement is possible.

The method of Cook and Dwork can be described as follows. Assume that the inputs \( x_1, x_2, \ldots, x_n \) are stored initially in cells \( M_1, M_2, \ldots, M_n \) of the shared memory. Let the Fibonacci numbers be defined recursively by \( F_0 = 0, F_1 = 1, F_{m+2} = F_{m+1} + F_m, m \geq 0 \).

**Theorem 10.3.1** A PRAM can compute the OR of \( F_{2t+1} \) inputs in \( t \) steps with no read or write conflicts. \( \square \)

Since \( F_{2t} \approx \phi^{2t} \) where \( \phi = (1 + \sqrt{5})/2 \) we have an upper bound of

\[
((\log_\phi 2)/2) \log_2 n \approx (0 \cdot 72) \log_2 n
\]

for the Boolean OR of \( n \) arguments.

**Proof** Assume \( M_i = 0 \) for \( i > n \). Each processor \( P_i \) has local variables \( Y_i \) (Boolean) and \( t \) (integer) which are initially 0. While \( F_{2t+1} < n \), each processor \( P_i \) executes the instructions

For all \( i, i + F_{2t} \leq n, Y_i := Y_i \lor M_{i+F_{2t}} \);
For all $i$, $i > F_{2t+1}$ and $Y_i = 1$, $M_{i-F_{2t+1}} := 1$;
$t := t + 1$;

It is straightforward to prove by induction on $t$ that before execution of step $t$,

$$Y_i = x_i \lor x_{i+1} \lor \cdots \lor x_{i+F_{2t-1}}$$

and

$$M_i = x_i \lor x_{i+1} \lor \cdots \lor x_{i+F_{2t+1}-1}$$

In particular,

$$M_1 = x_1 \lor x_2 \lor \cdots \lor x_{F_{2t+1}}$$

before step $t$.

Each time through the loop each processor reads (at most) one memory cell. No two processors read from the same cell. Then each processor writes to at most one memory cell. Again no two processors write to the same cell.

With this algorithm, both the processors and the memory cells accumulate information about more of the input on each step. When a processor has a zero result, it passes it on to the memory by not writing; thus the memory elements as well as the processors do OR’s each time through the loop.

### 10.3.2 Polynomial Evaluation

Let $P_n(a_0, a_1, \ldots, a_n, x) = \sum_{i=0}^{n} a_i x^i$ where $a_i, x \in \mathbb{Q}$. The standard sequential algorithm for polynomial evaluation is Horner’s Rule where to calculate $P_n$ we successively compute

$$p_n = a_n$$

$$p_i = (p_{i+1} \ast x) + a_i \quad \text{for} \quad i = n - 1, n - 2, \ldots, 1, 0.$$  

Then $P_n = p_0$. The sequential complexity of polynomial evaluation has been studied for many years. It is known that $2n$ arithmetic operations are required to evaluate a general polynomial of degree $n$, given by its coefficients. Thus, in terms of sequential complexity, Horner’s Rule is optimal. However, it is very unsuitable for parallel computation since at every step in the
computation the immediately preceding subresult is required. If instead, we evaluate each term $a_i x^i$ of the polynomial independently, in parallel, using a balanced binary tree of $\ast$-gates, and we then sum the values of the terms using a balanced binary tree of $+$-gates then we have a circuit of depth $2[\log_2(n+1)]$. This circuit is exponentially better, in terms of depth, than a circuit based on Horner’s Rule, although the number of gates (sequential complexity) is now $O(n^2)$ rather than $O(n)$. The $2[\log_2(n+1)]$ upper bound can be further improved to $\log_2 n + O(\sqrt{\log_2 n})$ by using a simple recursive parallel algorithm due to Munro and Paterson [68] which splits the polynomial into consecutive blocks of terms and factors out the appropriate power of $x$. Kosaraju [50] has shown that the algorithm of Munro and Paterson is optimal, in terms of circuit depth, for polynomial evaluation.

The Munro-Paterson algorithm can be described as follows. We define a recursive polynomial evaluation procedure for polynomials of degree $n \cdot 2^k$. The polynomial $P(x)$ may be expressed in the form

$$
P(x) = q_0(x) + q_1(x) \cdot x^n + q_2(x) \cdot x^{2n} + \cdots + q_{2^k-1}(x) \cdot x^{(2^k-1)n}
$$

where the $q_i(x)$ are polynomials of degree $\leq n$. So, to compute $P(x)$, we compute $\{q_i\}$, multiply by the appropriate power of $x$ at the next step, and then use $k$ further steps to combine the $2^k$ terms by binary addition.

Let $D(n)$ denote the maximum number of steps required to evaluate any $n^{th}$ degree polynomial on a shared memory parallel machine. The evaluation procedure described above yields the following recurrence relation for $D(n)$.

$$
D(n \cdot 2^k) \leq \max\{D(n), (\log n) + k\} + 1 + k \quad \text{(Divide and Conquer)}
$$

Let $d(n) = D(n) - \log n$, then

$$
d(n \cdot 2^k) + (\log n) + k \leq \max(d(n) + (\log n), (\log n) + k) + 1 + k
$$

Therefore,

$$
d(n \cdot 2^k) \leq \max(d(n), k) + 1
$$

Let $d(n) = k$ (Balancing)

$$
d(n \cdot 2^{d(n)}) \leq d(n) + 1
$$

For the basis of our induction we observe that a polynomial of degree 1 can be evaluated in two time steps and thus

$$
D(1) = d(1) = 2
$$
So we have
\[ d(1) = 2 \]
\[ d(1 \cdot 2^2) \leq 3 \]
\[ d(1 \cdot 2^2 \cdot 2^3) \leq 4 \]
\[ d(1 \cdot 2^2 \cdot 2^3 \cdot 2^4) \leq 5 \]
and an elementary induction shows that
\[ d(2^{(x-2)(x+1)/2}) \leq x \]
Let \( x = \sqrt{2 \log n} + 2 \)
\[ d(2^{(\sqrt{2 \log n})^3/2}) \leq \sqrt{2 \log n} + 2 \]
\[ d(2^{\log n+(3/2)\sqrt{2 \log n}}) \leq \sqrt{2 \log n} + 2 \]
\[ d(2^{\log n}) \leq \sqrt{2 \log n} + 2 \]
\[ d(n) \leq \sqrt{2 \log n} + 2 \]
Therefore,
\[ D(n) \leq \log n + \sqrt{2 \log n} + O(1) \]

### 10.3.3 Prefix Sums

Let \( x_1, x_2, \ldots, x_n \) be a set of values and \( \circ \) be an associative operation on that set. The *prefix sums problem* is to compute \( p_i = x_1 \circ x_2 \circ \cdots \circ x_i \) for all \( 1 \leq i \leq n \). The straightforward method yields a circuit of size \( n - 1 \) but its depth is also \( n - 1 \). By computing each \( p_i \) independently we can obtain a circuit of size \( O(n^2) \) and depth \( \lceil \log_2 n \rceil \). An important result of Ladner and Fischer [52] shows that the prefix sums problem can be computed by a circuit of size \( O(n) \) and depth \( O(\log n) \). This construction can be applied to produce small fast parallel circuits for a variety of important problems, including \( n \)-bit addition, \( n \)-bit multiplication, and Boolean sorting. It can also be used for the efficient parallel simulation of finite state automata. We will describe the application of parallel prefix computation to the problem of \( n \)-bit addition.
A binary number \(< a_{n-1}, a_{n-2}, \ldots, a_0 >\) ∈ \(\{0, 1\}\)^n represents the value \(\sum_{i=0}^{n-1} a_i \cdot 2^i\). Given two \(n\)-bit binary numbers \(X = < x_{n-1}, x_{n-2}, \ldots, x_0 >\) and \(Y = < y_{n-1}, y_{n-2}, \ldots, y_0 >\), the \(n\)-bit addition problem is to compute the \((n + 1)\)-bit representation \(Z = < z_n, z_{n-1}, \ldots, z_0 >\) of \(X + Y\).

In the normal “school method” we first compute \(z_0 = x_0 \oplus y_0\) and the initial carry bit \(c_0 = x_0 \land y_0\). We then use \(n - 1\) full adders to compute \(z_i, c_i\) from \(x_i, y_i\) and \(c_{i-1}\). Finally, we let \(z_n = c_{n-1}\). This method yields a circuit of size \(O(n)\) and depth \(O(n)\). The prefix method consists of three stages:

**Stage 1:** Compute \(u_j = x_j \land y_j, v_j = x_j \oplus y_j\) for all \(0 \leq j \leq n\).

**Stage 2:** Compute the carry bits \(c_j\) for all \(0 \leq j < n\).

**Stage 3:** Compute the outputs. \(z_0 = v_0, z_j = v_j \oplus c_{j-1}\) for all \(1 \leq j < n, z_n = c_{n-1}\).

Stages 1 and 3 can both be carried out in linear size and constant depth, therefore we need only consider Stage 2. Let \(A_{(u,v)}(c) = u \lor (v \land c)\) for \(c \in \{0, 1\}\). Then we have

\[
c_i = A_{(u_i, v_i)} \circ A_{(u_{i-1}, v_{i-1})} \circ \cdots \circ A_{(u_0, v_0)}(0)
\]

where \(\circ\) denotes function composition. Since function composition is associative, we can use a prefix circuit to compute all the carry bits \(c_i\). We need only design a circuit for the operation \(\circ\). Let \(A_{(u,v)} = A_{(u_2, v_2)} \circ A_{(u_1, v_1)}\). Then \((u, v) = (u_2 \lor (u_1 \land v_2), v_1 \land v_2)\) and therefore we can construct a subcircuit of size three and depth two for the operation \(\circ\). We have shown that \(n\)-bit addition can be realised by a Boolean circuit of size \(O(n)\) and depth \(O(\log n)\).

In functional programming, the second-order function \textit{scan} corresponds to the prefix sums computation. The above mentioned results, and the work of Blelloch [13] and others, have shown it to be a parallel primitive of extremely wide applicability.

### 10.3.4 Matrix Multiplication

Let \(A, B\) be two \(n \times n\) matrices of rational numbers. Then the product of \(A, B\) is an \(n \times n\) matrix \(C\), where \(c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}\). The exact determination of the sequential complexity of matrix multiplication is a major open problem in the field of computational complexity [20, 70, 84]. At the present time, the
best known algorithm (asymptotically, as \( n \to \infty \)) requires only \( O(n^{2.376}) \) arithmetic operations [22] as opposed to the standard \( O(n^3) \) which follows from the definition. No lower bound larger than the trivial \( \Omega(n^2) \) is known.

In contrast, determining the shared memory parallel complexity of matrix multiplication is trivial. We can evaluate each \( c_{i,j} \) term independently, in parallel, by a balanced binary tree of depth \( \lceil \log_2 n \rceil + 1 \). Functional dependency shows this bound for \( c_{i,j} \) to be optimal and so we have an optimal time bound for parallel matrix multiplication on the PRAM and circuit models.

10.3.5 Linear Recurrences

The parallel evaluation of recurrences was discussed in the mid 1960s by Karp, Miller and Winograd [46]. Let us first consider the computation of the very simple recurrence which defines Fibonacci numbers. The \( m^{th} \) Fibonacci number \( f_m \) is given by the second order linear recurrence

\[
\begin{align*}
    f_0 &= 0 \\
    f_1 &= 1 \\
    f_m &= f_{m-1} + f_{m-2} \quad \text{for} \, \, m \geq 2
\end{align*}
\]

This definition can be directly translated into an arithmetic circuit with \( m-1 \) gates (and depth \( m-1 \)) which successively computes \( f_2, f_3, \ldots, f_n \). As in the case of Horner’s Rule we have a circuit with no direct parallel speedup. If instead, we use the unconventional definition

\[
(f_{m-1} \, \ f_m) = (f_0 \, f_1) \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{m-1}
\]

then we see immediately that \( f_m \) can be calculated by an arithmetic circuit of size and depth \( O(\log_2 m) \) if we compute the matrix power efficiently by repeated squaring.

The above result for Fibonacci numbers is a special case of the following more general result by Greenberg et al. [36] on the parallel evaluation of \( k^{th} \) order linear recurrences. If we have \( \mathbf{F} = (f_0 \ f_1 \ \ldots \ f_{k-1}) \) and \( f_m = \sum_{j=1}^{k} a_{k-j} \cdot f_{m-j} \) for \( m \geq k \), then \( (f_{m-k+1} \ \ldots \ f_m) = \mathbf{F} \cdot \mathbf{M}^{m-k+1} \) where \( \mathbf{M} \)
is the $k \times k$ matrix

$$
\begin{pmatrix}
0 & \ldots & 0 & a_0 \\
& \ddots & & \\
& & a_i & \\
I & & & a_{k-1}
\end{pmatrix}
$$

and therefore the parallel complexity of computing $f_m$ is at most $O(\log_2 k \cdot \log_2 (m - k))$.

For a practical application of this result we consider the problem of solving linear systems. Let $B$ be an $n \times n$ non-singular, lower triangular matrix, and $c$ be an $n$-element vector. In solving the linear system $Bx = c$ by ‘back substitution’ we use the recurrence $x_i = (c_i - \sum_{j=1}^{i-1} b_{i,j} \cdot x_j) / b_{i,i}$ for $1 \leq i \leq n$. If we let $x_i = 0$ for $i < 1$ then we can rewrite this recurrence in the form $(x_i \ x_{i-1} \ \ldots \ x_{i-n+1} \ 1) = (x_{i-1} \ x_{i-2} \ \ldots \ x_{i-n} \ 1) \cdot M_i$ where

$$
M_i =
\begin{pmatrix}
-\frac{b_{i,i-1}}{b_{i,i}} & & & & \\
\vdots & & & & \\
-\frac{b_{i,1}}{b_{i,i}} & 0 & & & \\
\vdots & & & & \\
0 & & & & \\
0 & 0 & \ldots & & 0 \\
0 & 0 & \ldots & & 0 \\
-\frac{c_i}{b_{1,i}} & 0 & \ldots & & 1
\end{pmatrix}
$$

Therefore we can design an arithmetic circuit of depth $O(\log^2 n)$ which solves $Bx = c$ to obtain $x$. This circuit corresponds to a PRAM algorithm with parallel time $O(\log^2 n)$ and number of processors $O(n^4)$. In contrast, a direct PRAM implementation of back substitution would have parallel time $O(n)$, but would require only $O(n)$ processors.

### 10.3.6 Algebraic Path Problem

The Algebraic Path Problem, see section 7.2.3, is to compute $M^* = \bigoplus_{k=0}^{\infty} M^k$ where matrix product is defined in terms of the two operations $\oplus$ and $\otimes$. ($M^0$ is the identity matrix with diagonal elements $I_\phi$). Noting that node $i$ is connected to node $j$ by a directed path if and only if it is connected by a
directed path of length $\leq n - 1$, we have

$$M^* = \bigoplus_{k=0}^{n-1} M^k = (M^0 \oplus M)^{n-1} = (M^0 \oplus M)^{2^l} \text{ for } 2^l \geq n - 1$$

Therefore, to obtain an efficient shared memory parallel algorithm for the computation of the APP on matrix $M$ we need only set the main diagonal to $I_\otimes$ and repeatedly square the resulting matrix until we have a sufficiently large power. For an $n \times n$ matrix $M$, this method yields a circuit of depth $O(\log^2 n)$ or, equivalently, a PRAM algorithm of time complexity $O(\log^2 n)$

### 10.4 The Class $\mathcal{NC}$: Parallelisable Problems

We have described a number of shared memory parallel algorithms for the PRAM, circuit and comparison network models. The circuits and comparison networks given can be directly transformed into corresponding PRAM algorithms. These idealised models have provided a robust framework for the investigation of parallel algorithms and their complexity [31, 44, 76]. One outcome of this work has been the development of an extensive set of results concerning $\mathcal{NC}$, the class of computational problems which can be solved on a PRAM by a deterministic algorithm in polylogarithmic time using only a polynomial number of processors. A major open problem in theoretical computer science is to determine whether $\mathcal{P}$, the class of polynomial time computable problems, is contained in $\mathcal{NC}$. If this were shown to be true then it would imply that every problem which had a fast (polynomial time) sequential algorithm also had a fast (polylogarithmic time), efficient (polynomial number of processors) parallel algorithm. Over the last decade, a large number of important problems in $\mathcal{P}$ have been shown to also lie in $\mathcal{NC}$. The following list of such problems is by no means complete.

**Evaluation of expressions and programs:** Tree restructuring, tree contraction, expression evaluation, evaluation of straight-line algebraic programs of polynomial degree over a commutative semiring, evaluation of straight-line programs corresponding to
dynamic programming algorithms, context-free recognition, parallel simulation of finite state automata, circuit value problem for planar monotone circuits, evaluation of set expressions.

Logic: Term matching, term equivalence, evaluation of DATA-LOG logic programs with the polynomial-fringe property.

Sets: Sorting, selection, set operations, constructing Huffman trees.

Sequences and strings: Prefix sums, merging, sequence comparison (string edit problem), string matching, recognising shuffle of two strings, longest common substring, finding squares in a string, pattern matching for d-dimensional patterns.

Lists: List ranking.

Arithmetic: n-bit integer arithmetic (addition, multiplication, division), linear recurrences, polynomial arithmetic (evaluation, multiplication, division, GCD), evaluation of elementary functions (exp, ln, sin, etc.).

Matrices: Matrix multiplication, determinant, rank, inverse, solution of linear system, sparse Cholesky factorisation.

Graphs: Algebraic path problem (transitive closure, shortest paths, minimum cost spanning tree, topological ordering of dag), transitive reduction, connected components, biconnectivity, triconnectivity, Euler tours, ear decomposition, maximal independent set, symmetry breaking, lowest common ancestors, planarity, tree isomorphism, bipartite perfect matching, minimal elimination ordering.

Combinatorial optimisation: Fixed dimension linear programming.

Geometry: Convex hull in two and three dimensions, Voronoi diagrams and proximity problems, detecting segment intersections, triangulating a polygon, point location.

A number of interesting and important randomised \( \mathcal{NC} \) algorithms have also been produced for graph problems such as depth-first search, constructing a perfect matching, maximum cardinality matching, maximum \( s - t \) flow, planar graph isomorphism, and subtree isomorphism, and for various problems in computational geometry.
An interesting theoretical development over the last few years has been the development of a large number of new randomised parallel algorithms for important problems, which achieve nearly-constant time performance on a CRCW PRAM. The problems include hashing, dictionary (insert, delete, query operations), integer sorting, integer chain sorting, space allocation, linear approximate compaction, estimation, load balancing, leaders election, representative selection, generation of random permutations.

10.5 \( P \)-Completeness: Inherently Sequential Problems?

Using techniques analogous to those used in sequential computation to establish \( \mathcal{NP} \)-completeness, a number of problems in \( P \) have been shown to be \( P \)-complete. A computational problem \( \Pi \) is \( P \)-complete if and only if \( \Pi \in P \) and \( (\Pi \in \mathcal{NC} \Rightarrow P \subseteq \mathcal{NC}) \). The \( P \)-complete problems are, in a sense, those in \( P \) for which it is hardest to obtain a fast, efficient PRAM algorithm. Showing that any one of them was in \( \mathcal{NC} \) would imply that all problems in \( P \) had fast, efficient PRAM algorithms. The first \( P \)-complete problems were established in the early 1970s. A recent book [37] lists several hundred \( P \)-complete problems. Two very simple \( P \)-complete problems are the following.

**Subset Closure**
- **Given**: A finite set \( X \), a binary operation \( \circ \) on \( X \), a subset \( S \subseteq X \), and an element \( x \in X \).
- **To determine**: Whether \( x \) is contained in the smallest subset of \( X \) which contains \( S \) and is closed under \( \circ \).

**Monotone Circuit Value Problem**
- **Given**: A single-output Boolean circuit with \( \{\land, \lor\} \) gates, and a set of values for the inputs.
- **To determine**: The output.

Some other examples of \( P \)-complete problems are:

**Evaluation of expressions and programs**: Planar circuit value problem, arithmetic circuit value problem, type inference, deadlock detection.
Logic: Unification, propositional Horn clause satisfiability, path systems, context-free grammar membership.

Algebra: Finite algebra, generalised word problem, subgroup equality, subgroup isomorphism, group rank.

Arithmetic: Iterated mod.

Matrices: Gaussian elimination with partial pivoting.

Graphs: Maximum flow, lexicographically first maximal independent set, lexicographically first maximal path, lexicographically first depth-first search ordering, high degree subgraph, minimum degree elimination order.

Combinatorial optimisation: Linear programming, linear inequalities, first fit decreasing bin packing, nearest neighbour travelling salesman heuristic, two-player game.

Geometry: Plane sweep triangulation, visibility layers.

Some simple examples of problems in $\mathcal{P}$ which are not currently known to be in $\mathcal{NC}$ or to be $\mathcal{P}$-complete are the following:

**Integer GCD**

*Given:* Two $n$-bit positive integers $a, b$.

*To determine:* $\text{GCD}(a, b)$

**Relative Primeness**

*Given:* Two $n$-bit positive integers $a, b$.

*To determine:* Whether $a, b$ are relatively prime.

**Stable Marriage**

*Given:* $n$ men and $n$ women plus a list of marital preferences for each person.

*To determine:* $n$ marriages that will stand the test of time.

**Ray Tracing**

*Given:* A set of $n$ mirrors of lengths $l_1, l_2, \ldots, l_n$ and their placements, a source $S$ and the trajectory of a single beam emitted from $S$, a designated mirror $M$.

*To determine:* If $M$ is hit by the beam. At the mirrors the angle of incidence of the beam equals the angle of reflection.
Serna and Spirakis [79, 80] have investigated the extent to which solutions to important \( P \)-complete problems such as linear programming, maximum flow and high degree subgraph can be approximated by fast, efficient PRAM algorithms if \( P \neq NC \).

### 10.6 Work Efficient PRAM Algorithms

Matrix multiplication and the algebraic path problem are two fundamental computational problems which have fast \( NC \) algorithms. A large number of important problems in \( P \) can be shown to be in \( NC \) by a reduction to one of these two problems. (In the case of reductions to the APP, these are usually reductions to the transitive closure instance of that problem.) Unfortunately, many of the parallel algorithms so produced are extremely inefficient in terms of the number of processors required. For example, many problems on graphs with \( v \) vertices and \( e \) edges can be solved sequentially in time \( O(v + e) \) or \( O((v \log v) + e) \). For a number of these problems, one can obtain a parallel algorithm with time complexity \( O(\log^2 v) \), but the algorithm requires \( M(v) \) processors, where \( M(v) \) is the sequential complexity of \( v \times v \) matrix multiplication. As noted earlier, the best known upper bound on \( M(v) \) is \( O(v^{2.376}) \) and we know that it cannot be less than proportional to \( v^2 \). Thus we have a number of fast PRAM algorithms for which the processor-time product is much greater than the time required to solve the problem by a sequential algorithm. To produce fast practical parallel algorithms for such problems we must avoid the brute force use of matrix multiplication and transitive closure on dense matrices. This difficulty has come to be known as the *matrix multiplication / transitive closure bottleneck*. It is particularly serious in applications where one is dealing with highly sparse matrices or graphs. In such cases, by embedding the problem in one involving dense matrices one may produce a theoretically fast algorithm, but it is unlikely to be of much practical value. In the last few years, substantial progress has been made on overcoming this bottleneck. A number of important new PRAM algorithms for sparse matrix and graph problems have been developed which are very efficient in their use of processors.

As we have seen, the robustness of the PRAM model and the class \( NC \) has permitted the development of a rich theory of parallel algorithms and their complexity. However, the above considerations show that a naive preoccupation with \( NC \) may not result in efficient parallel algorithms for practical
implementation. Probing $\mathcal{NC}$ further, it is not even clear that the class captures the informal notion of “problems which are amenable to parallel solution”. Vitter and Simons [93] have shown that some $\mathcal{P}$-complete problems may be solved by parallel algorithms which are in a very reasonable sense, efficient. On the other hand, a problem such as searching an ordered list, which runs in logarithmic sequential time, is in $\mathcal{NC}$, irrespective of the existence of efficient parallel algorithms for that problem. In fact, searching does not admit efficient parallel algorithms.

Efficiency is a prime consideration in the design of parallel algorithms: one would like to solve a problem roughly $p$ times faster when using $p$ processors. This consideration is missing from the definition of $\mathcal{NC}$, instead the emphasis of $\mathcal{NC}$ theory is simply on the development of parallel algorithms which have polylogarithmic time complexity. Motivated by these considerations, Kruskal, Rudolph and Snir [51] have developed an alternative set of complexity classes for PRAM computations, and have demonstrated convincingly that they provide an equally robust framework for studying parallel algorithms and complexity, but one which is more relevant in the context of practical parallel computing. Their emphasis is much more on the performance of a parallel algorithm relative to the best known sequential algorithm for the same problem. In describing the approach of [51] we will use the following notation. For a given problem, $S(n)$ will denote the sequential running time, $T(n)$ the parallel running time, and $P(n)$ the number of processors.

One very weak requirement, in terms of parallel performance, would be that the parallel algorithm demonstrate some unbounded speedup, i.e. $\lim_{n \to \infty} T(n)/S(n) = 0$. A parallel algorithm for which $T(n) = O(n/\log \log n)$ when $S(n) = O(n)$ would satisfy this condition. That kind of small improvement is unlikely to be sufficient in many cases. We are more likely to want to claim that a significant reduction in running time can be achieved through the use of parallelism, i.e. that $T(n)$ is a fast decreasing function of $S(n)$. The two obvious choices for such a function are captured in the following definition.

**Definition 10.6.1** A parallel algorithm is polynomially fast if $T(n) = O(S(n)^\epsilon)$ for some $\epsilon < 1$, and it is polylogarithmically fast if $T(n) = O(\log^k S(n))$ for some fixed $k$.

Reduction in running time has a cost. The number of processors must increase as fast as the speedup; generally it increases faster. The inefficiency of a parallel algorithm is the ratio $T(n) \times P(n)/S(n)$, i.e. the ratio between
the time-processor product for the parallel algorithm and the number of operations performed by the sequential algorithm.

**Definition 10.6.2** A parallel algorithm has constant inefficiency if $T(n) \cdot P(n) = O(S(n))$, it has polylogarithmically bounded inefficiency if $T(n) \cdot P(n) = O(S(n) \log^k S(n))$ for some fixed $k$, and it has polynomially bounded inefficiency if $T(n) \cdot P(n) = O(S(n)^k)$ for some fixed $k$.

Six interesting classes can be obtained by combining the two requirements on speedup with these three constraints on inefficiency.

<table>
<thead>
<tr>
<th>Inefficiency</th>
<th>Polylog Fast</th>
<th>Poly Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>ENC (Efficient, NC fast)</td>
<td>EP (Efficient, Parallel)</td>
</tr>
<tr>
<td>Polylog</td>
<td>ANC (Almost efficient, NC fast)</td>
<td>AP (Almost efficient, Parallel)</td>
</tr>
<tr>
<td>Poly</td>
<td>SNC (Semi efficient, NC fast)</td>
<td>SP (Semi efficient, Parallel)</td>
</tr>
</tbody>
</table>

Kruskal, Rudolph and Snir [51] classify a large number of important problems within this framework.

**ENC**: Sorting, merging, selection, prefix sums, polynomial evaluation, expression evaluation, fast Fourier transform, connected and biconnected components of dense graphs.

**EP**: Various dense graph computations (strongly connected components, single source shortest paths, minimum cost spanning tree, directed graph reachability), monotone circuit value problem.

**ANC**: Connectivity and biconnectivity of sparse graphs with $e = O(v)$ edges.

**SP**: Depth-first search of undirected graphs, weighted bipartite matching, flows in $0-1$ networks.

### 10.7 PRAM Simulation

We have seen that an idealised model of parallel computation such as the PRAM can provide a robust framework within which to develop techniques for the design, analysis and comparison of parallel algorithms. A major issue in theoretical computer science since the late 1970s has been to determine the extent to which the PRAM and related models can be efficiently implemented on physically realistic distributed memory architectures.
The techniques and results that we have described for various types of randomised routing show convincingly that for the problem of routing h-relations at least, there are a variety of theoretically and practically efficient methods which can be used. However, in order to show that we can efficiently simulate shared memory on a distributed memory architecture we also need to show that we can deal with the problem of “hot spots”, i.e. where a large number of processors simultaneously try to access the same memory module.

10.7.1 Hashing

In theoretical terms, one very effective method of uniformly distributing memory references is to hash the single address space. The hash function has, of course, to be efficiently computable. Hash functions for this purpose have been proposed and analysed by Mehlhorn and Vishkin [65]. They suggest using an elegant class of functions with some provably desirable properties: the class of polynomials of degree $O(\log p)$ in arithmetic modulo $m$, where $p$ is the number of processors and $m$ is the total number of words in the shared address space. As in the case of randomised routing, the idea of hashing the address space in this way has been subjected to extensive scrutiny in terms of both its theoretical and its practical performance. All of the available evidence suggests that it works extremely well in both respects. In fact, even constant degree polynomial hash functions, e.g. degree two, seem to work well in practice. One additional advantage of using a hashed address space is that we do not need then to resort to randomising to avoid bottlenecks in packet routing, simple deterministic methods will suffice.

Detailed technical accounts of the role of hashing in achieving efficient general purpose parallel computing can be found in [66, 89]. We will mention only the following three results which demonstrate that distributed memory architectures can efficiently simulate PRAMs. Let $\text{EPRAM}(p, t)$, $\text{CPRAM}(p, t)$, $\text{HYPERCUBE}(p, t)$, $\text{COMPLETE}(p, t)$, $\text{BSP1}(L, p, t)$ denote the class of problems which can be solved on a $p$ processor EREW PRAM, CPRAM, hypercube, completely connected network, BSP computer with $g = O(1)$ and $l = L$ respectively in $t$ time steps.

**Theorem 10.7.1 (Valiant [89])**

$\text{EPRAM}(p \cdot \log p, t/\log p) \subseteq \text{HYPERCUBE}(p, t)$.

**Theorem 10.7.2 (Valiant [88])**

$\text{EPRAM}(p \cdot L, t/L) \subseteq \text{BSP1}(L, p, t)$.
Theorem 10.7.3 (Karp, Luby and Meyer auf der Heide [45])

\[ \text{CPRAM}(p \log \log p \log^* p, t) \subseteq \text{COMPLETE}(p, t \log \log p \log^* p). \]

Theorems 10.7.1, 10.7.2 and 10.7.3 show that PRAM algorithms with a degree of parallel slackness can be implemented on distributed memory architectures with powerful global communications in a way which is optimal in terms of the processor-time product.

Definition 10.7.4 An \( m \) processor algorithm, when implemented on an \( n \) processor machine, where \( n \leq m \), is said to have a parallel slackness factor of \( m/n \) for that machine.

Parallel slackness is an idea of fundamental importance in the area of general purpose parallel computing. If parallel algorithms and programs are designed so that they have more parallelism than is available in the machine, then the available parallel slackness can be effectively exploited to hide the kind of network latencies one finds in distributed memory architectures. The only requirement is that the processors provide efficient support for multithreading and fast context switching. As a means of efficiently realising virtual shared memory on distributed memory architectures, latency tolerance via multithreading is likely to be more effective than the use of complex caching schemes for latency reduction.

The idea of exploiting parallel slackness can even be carried over into the area of sequential computing. Much effort in recent years has been devoted to the development of complex heuristic techniques for the efficient prefetching of values from memory in sequential computations. An alternative to this approach is, instead, to design parallel algorithms for implementation on sequential machines. The parallel slackness of the algorithm can then be exploited to achieve efficient prefetching.

We have seen then that by achieving a degree of parallel slackness in program designs one can provide significant opportunities for the effective scheduling of those programs, by the programmer or by a compiler, to hide the various kinds of latencies which arise in both sequential and parallel computing. This idea of exploiting parallel slackness, or overdecomposition, is not new. It was, for example, a central idea in the early HEP parallel architecture [83], and has been used in its successors, Horizon and Tera [4]. The prospects for “autoparallelising” sequential code, which may be regarded as the extreme opposite of this approach, would, on the other hand, appear to very bleak.
10.7.2 Combining

In the previous sections we have been concerned with the problems of implementing the weakest PRAM model, the EREW PRAM, on a distributed memory architecture. In practical parallel programming it is often convenient to permit concurrent access to a memory location, as in the CRCW PRAM model. An important practical case is that of broadcasting, where all processors simultaneously require the value of a single memory location.

One approach to the implementation of concurrent memory access is to use combining networks [33], i.e. networks that can combine and replicate messages in addition to delivering them in a point-to-point manner. The Fluent machine of Ranade [73] provides an excellent example of how a CRCW PRAM can be efficiently implemented on a distributed memory architecture equipped with a combining network. The interconnection network of the Fluent machine is a butterfly. Let \( \text{CBUTTERFLY}(p, t) \) denote the class of problems which can be solved on a \( p \)-processor butterfly with a combining network in \( t \) time steps.

Theorem 10.7.5 (Ranade [73])

\[
\text{CPRAM}(p, t) \subseteq \text{CBUTTERFLY}(p, t \log p).
\]

The theorem is established by showing that a \( p \)-processor Fluent machine can emulate a \( p \)-processor CRCW PRAM with only a slowdown of \( O(\log p) \), i.e. each parallel step of the PRAM requires at most \( O(\log p) \) steps on the Fluent machine, with high probability. The size of buffers required at each node of the network is constant. The following is a very brief account of the main ideas used in the emulation. The address space of the PRAM is hashed onto the memory modules of the Fluent machine using a hashing function chosen at random from a \((\log n)\)-universal class of hash functions [18]. Suppose several processors wish to read the same memory location at the same time. Each one sends a message to the appropriate memory module along some path in the network. These paths will intersect to form a tree and there is, therefore, no need to send more than one request along any branch of the tree. A request simply waits at each node until (i) another request to the same destination arrives on the other input to the node, in which case the node combines the two and forwards the result along the tree, or (ii) the node determines that no future requests arriving on the other input will have the same destination. By always transmitting messages in sorted order of their
destinations, and using “ghost messages” where necessary, one can achieve the above emulation result.

Concurrent access to shared variables in the Fluent machine is based on the multiprefix primitive. It has the form \( MP(A, v, \oplus) \) where \( A \) is a shared variable, \( v \) is a value, and \( \oplus \) is a binary associative operator. At any time step a processor can execute a multiprefix operation, with the constraint that if processors \( P_i \) and \( P_j \) execute \( MP(A, v_i, \oplus_i) \) and \( MP(A, v_j, \oplus_j) \), then \( \oplus_i = \oplus_j \).

The semantics of the multiprefix operation is defined as follows.

**Definition 10.7.6** At time step \( T \), let \( P_A = \{p_1, p_2, \ldots, p_k\} \) be the set of processors referring to variable \( A \), such that \( p_1 < p_2 < \cdots < p_k \). Suppose that \( p_i \in P_A \) executes instruction \( MP(A, v_i, \oplus) \). Let \( a_0 \) be the value of \( A \) at the start of time step \( T \). Then, at the end of time step \( T \), processor \( p_i \) will receive the value \( a_0 \oplus v_1 \oplus \cdots \oplus v_i \) and the value of variable \( A \) will be \( a_0 \oplus v_1 \oplus \cdots \oplus v_k \).

Thus, when a set of processors perform a multiprefix operation on a common variable, the result is the same as if a single prefix operation were performed with the processors ordered by their index.

A parallel architecture which is very close in design to the Fluent machine is currently under construction at the University of Saarbrucken [1]. The efficient implementation of broadcasting and combining in software on a BSP architecture is discussed in [90].

### 10.8 BSP and the PRAM Model

The BSP model can be regarded as a generalisation of the PRAM model which permits the frequency of barrier synchronisation, and hence the demands on the routing network, to be controlled. If a BSP architecture has a very small value of \( g \), e.g. \( g = 1 \), then it can be regarded as a PRAM and we can use hashing to automatically achieve efficient memory management. The value of \( l \) will determine the degree of parallel slackness required to achieve optimal efficiency. The case \( l = g = 1 \) corresponds to the idealised PRAM, where no parallel slackness is required.
Bibliography


