The Design of Numerical Algorithms for Transputer Systems

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by Timothy Alec Oliver.

July 1993

Abstract

This thesis discusses techniques for the design and implementation of parallel numerical algorithms for distributed memory MIMD architectures. The algorithms are targeted at machines based on the INMOS transputer family of microprocessors which include the T4, T8 and T9000 processors. We investigate how features of each member of the transputer family affect the design of numerical algorithms. Run-time cost models are developed to give predictions of the total execution time of algorithms. These cost models allow us to study the run-time characteristics of algorithms and the impact of the computer architecture on algorithm run-time. `occam` implementations of algorithms are developed and their performance is compared with the predictions given by the cost models to see how much credence can be given to the models. The algorithms are designed for both efficiency and portability between a wide range of distributed memory MIMD architectures. The cost models can help to estimate the performance of an algorithm on a different distributed memory architecture and on future generations of machines.

The portability of algorithms is maximised by the use of modular programming. Algorithms use low-level library routines for communications and computation. These routines are optimised for each target architecture to achieve reasonable performance for the complete algorithm.

Algorithms from a wide range of numerical programming fields have been investigated. Each area highlights different techniques that can be employed in the design of good parallel algorithms. In linear algebra we look at the Gaussian elimination algorithm. This algorithm is very important and illustrates many techniques applicable to parallel linear algebra algorithms. Another very important field is sorting. Designing good parallel sorting algorithms is very difficult because of the low computation cost compared with data size. Finally, we look at two algorithms from non-linear numerical optimisation. These algorithms clearly illustrate the use of low-level communications and computation routines to achieve portable and efficient algorithms. They indicate the way in which larger, more complicated programs may be developed from simpler, existing routines thus reducing software development time.

Acknowledgements

This thesis covers work conducted within the Centre for Mathematical Software Research, University of Liverpool, between 1987 and 1993. The work has been funded by two European ESPRIT projects: Supernode I (P1085) and PUMA (P2701). This work presents only a small fraction of the research conducted under these projects by colleagues at Liverpool and elsewhere. However, much benefit has been gained from related work within these, and other, ESPRIT projects. I am indebted to the group at Liverpool for their close collaboration and support, and in particular to Rod Cook for driving me through the final stage of writing-up.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a short overview of the aims of the thesis. The remainder of the chapter then presents an introduction to scientific parallel computing. We start this introduction with a discussion of the development of parallel computer architectures. This is followed by an overview of current MIMD architectures and in particular the architecture of transputer systems. We then discuss the types of parallel programming models and languages that are used for MIMD architectures and the different computational techniques that are available to the scientific programmer.

1.1 Overview

This thesis discusses techniques for the design and implementation of parallel numerical algorithms for distributed memory MIMD architectures. The algorithms are targeted at machines based on the INMOS transputer family of microprocessors which include the T4, T8 and most recently the T9000. We investigate how features of each member of the transputer family affect the design of numerical algorithms. Two of the most important quantities to consider are the ratio of floating point computation rate to communication rate, and the architecture of the communication subsystem.

We develop run-time cost models for each algorithm studied to give an insight into the total execution time of the algorithm. This is an extension of the well known technique of quantifying the compexity of a sequential algorithm. These cost models allow us to study the run-time characteristics of algorithms and the impact of the computer architecture on algorithm run-time. `occam` implementations of some of the algorithms are developed and their performance is compared with the predictions given by the cost models to see how much credence can be given to the models.

The algorithms are designed for both efficiency and portability between a wide range of distributed memory MIMD architectures. With this in mind, the cost models can be used to estimate the performance of an algorithm on a different distributed memory architecture and on future generations of machines. The portability and maintainability of algorithm code is maximised by the use of software encapsulation. Algorithms use low-level library routines for communications and computation. These routines are optimised for each target architecture to achieve reasonable performance for the complete algorithm.

Algorithms from a wide range of numerical programming fields have been investigated. Each area highlights different techniques that can be employed in the design of good parallel algorithms. In linear algebra we look at the solution of systems of linear equations by Gaussian elimination. The Gaussian elimination algorithm is very important and illustrates many techniques applicable to parallel linear algebra algorithms. Another very important field is sorting. Designing good parallel sorting algorithms is very difficult because of the low computation cost compared with data size. Finally, we look at two algorithms for unconstrained non-linear numerical optimisation. These algorithms clearly illustrate the use of low-level communications and computation routines to achieve portable and efficient algorithms. They indicate the way in which larger, more complicated programs may be developed from simpler, existing routines thus reducing software development time.

The thesis is arranged as follows: The remainder of this chapter presents an introduction to scientific parallel computing. Chapter 2 discusses the calculation of cost-model parameters for transputer architectures. These parameters are then used in the following chapters to predict the performance of the algorithms examined. The next 4 chapters detail the algorithms studied and present performance results: Chapter 3 discusses Gaussian elimination and Chapter 4 covers sorting. Chapters 5 and 6 describe algorithms for the Newton and quasi-Newton optimisation methods respectively. In Chapters 7 and 8, we present a discussion and summary of the results obtained, and indicate directions for future work.

In addition there are four appendices. Appendix A describes the communication routines developed in this work and Appendix B discusses the programming techniques used. The last two appendices contain the run-time measurements for the Gaussian elimination and sorting algorithms.

The thesis describes work conducted within two European ESPRIT projects: Supernode I (P1085) which ran from 1986 to 1988, and PUMA (P2701) which ran from 1989 to 1991. The Supernode I project was based on the T8 architecture. Chapters 3 and 4 describe work conducted within that project in 1988[74, 75]. The PUMA project was based on the T9000/C104 architecture. Work undertaken during this project [77, 79, 78, 76, 80] is presented in Chapters 2, 4, 5 and 6.

Parallel computing has been developing rapidly over the past few years. This is reflected in the thesis in several ways: Better parallel algorithms have now (July 1993) been developed which supersede some of the algorithms described here, most notably the Gaussian elimination algorithm.

Architecture designs have improved significantly over the years of this thesis. The limited connectivity of the T8 architecture has been replaced by the point to point connectivity of the T9000/C104 architecture. The T8 algorithms described in Chapters 3 and 4 in fact only use a chain of processors, instead of a grid, since this was dictated by the Liverpool Library [32] design. However, the T9000/C104 algorithms make full use of that architecture's rich connectivity as these algorithms were not part of the Library.

Parallel software environments have also improved significantly. At the start of this work there were no standards to ease program development and portability. Now, we will soon have a widely accepted standard for communications operations in the MPI [44], and standard parallel extensions to Fortran in HPF [43].

With these advances in mind, we trust that the work described in the following chapters gives a reasonable picture of the state of parallel computing at the time the work was undertaken.

1.2 Computer architectures

Ever since the first general purpose electronic digital computer, the ENIAC, was completed in 1946 the scientific community has demanded greater and greater performance [8]. For this community, performance is measured in terms of the time taken to perform floating point (FP) operations needed for scientific calculations. That first computer could add two ten decimal digit numbers in 200μs [58]. Each vector-processing node of a Cray Y-MP C90 supercomputer, first delivered to customers in 1992, has a double precision FP peak performance of 1Gflop/s. The largest current installed system consists of 16 vector-processing nodes giving a total peak performance of 16Gflop/s. This is an increase in performance by a factor of 10^6 over 46 years. The ENIAC could perhaps be considered to be the first parallel computer since it had many independent computing units which cooperated in solving a problem. Thus the concept of parallel computation is not new, but has been exploited in various ways throughout the short history of digital electronic computers.

Most early computers followed the von Neumann architecture where the processor made serial accesses to words of data stored in memory. To improve the performance of this architecture many parallel processing techniques were developed. Pipelining of instruction fetch, decode and execution was first introduced in the ATLAS computer in 1963. The CDC 6600, produced in 1964, featured multiprocessing with 10 independent peripheral units which executed different instructions on different data simultaneously. In 1976 the first vector computer, the Cray 1, was produced and featured special purpose vector registers and functional units for vector operations. All the functional units were highly pipelined.

The level of integration of these old supercomputers was low and so they were very expensive to manufacture. This situation changed dramatically in the 1980s with the introduction of VLSI (Very Large Scale Integration). Improvements in VLSI technology have given unprecedented performance increases over the past 10 years. Whilst the minimum feature size has decreased from 50 microns (50μ) in 1960 to 0.5μ in 1992, chip size has increased from 2mm in 1960 to 13mm in 1990 [56]. Decreasing feature size (increasing density) leads to faster circuit speed and thus faster cycle times for microprocessors. The combined increases in density and chip size give a huge increase in the number of components available per chip. Modern microprocessors are now utilising this silicon area to incorporate all the techniques developed for supercomputers over the past 40 years. Hence microprocessors feature multiple independent functional units (called a superscalar architecture) and pipelined (or superpipelined) operation, as well as large on-chip caches and advanced memory management. Microprocessors such as the DEC Alpha can provide a floating-point performance of 150Mflop/s which is about the same as the Cray 1. This trend of rapid increases in single processor performance looks set to continue for some years to come [88].

Alongside the developments in single processor architectures during the 1970s and 1980s there was also much work on the design of parallel computer architectures. The ILLIAC IV (first working system delivered in 1975) consisted of an 8x8 array of 64bit floating point processing elements (PEs) each with its own local memory and nearest-neighbour communications on a grid topology. All the PEs were controlled by a central control unit executing a single program. Another similar system, the ICL DAP, was delivered in 1980. This had a 64x64 array of single bit PEs again with nearest-neighbour grid communications.

At this time, other types of parallel architectures were also developed. These architectures had many independent processors executing different programs with different data. We distinguish these architectures from pipelined and superscalar processor architectures since the latter do not use multiple distinct instruction streams. These architectures are also distinguished from earlier systems by having tightly coupled processors, that is, processors which can communicate with one another relatively quickly to cooperate in solving a problem. For example, the Cosmic Cube, built in 1983, consisted of 64 Intel 8086/8087 nodes connected together in a six dimensional hypercube. The processing nodes communicated with their near-neighbours by passing messages along the six edges of the hypercube. This machine was produced commercially as the Intel iPSC. Another example is the Cray X-MP which was introduced in 1982 and consisted of two Cray 1 computers connected to a shared memory.

The large number of different parallel architectures that have been designed make classification difficult. The most widely known computer classification is due to Flynn [42]. This classification provides a useful framework for discussing parallel architectures. Flynn divides computer architectures into four classes distinguished by how the architecture deals with sequences of program and data values (program and data streams):

SISD single instruction stream/single data stream. This is the classic sequential von Neumann computer architecture and includes pipelined and superscalar machines.

SIMD single instruction stream/multiple data stream. This is a class of parallel architectures including array processors such as the ILLIAC IV and ICL DAP. It should perhaps also include vector processors such as the Cray 1, since these architectures execute single vector instructions which operate on multiple data values.

MISD multiple instruction stream/single data stream. This would be a class of parallel

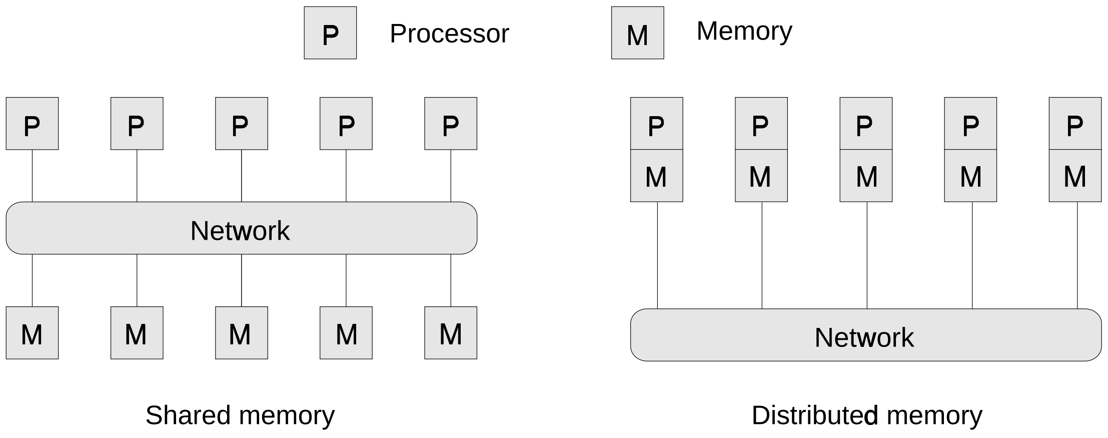


Figure 1.1: The main MIMD architectures

architectures, but no design has yet been proposed.

MIMD multiple instruction stream/multiple data stream. This is the predominant parallel architecture class, including all multiprocessor architectures such as the shared memory Cray X-MP and the distributed memory Cosmic Cube.

It should be emphasised that many architectures do not fit neatly into Flynn's classification, but may contain elements of more than one class. This is particularly true of MIMD architectures which commonly have advanced vector processors as processing nodes. Hence, for our purposes we shall consider vector processors as a special case of the conventional SISD architecture with highly advanced FP performance. The remaining SIMD architectures have been shown to be very efficient for certain application areas such as image processing. However, in recent years most research and development effort into parallel computing has been directed towards MIMD architectures. It is now widely accepted that this class of parallel computer is the best choice for high performance, general purpose computing.

1.3 MIMD architectures

The class of MIMD computers includes all computers that have multiple instruction streams processing multiple streams of data. This definition is far too broad for general use and so the class is usually further subdivided [41]. The two main MIMD architectures are distributed memory and shared memory architectures (see Figure 1.1). There are also several other architectures that do not fit easily into either of these two classes. These include dataflow architectures, reduction architectures and wavefront (systolic) architectures.

Shared memory MIMD architectures have several processors all connected to a shared memory subsystem by a network. The memory is usually partitioned into several modules to allow simultaneous access to memory addresses in different memory

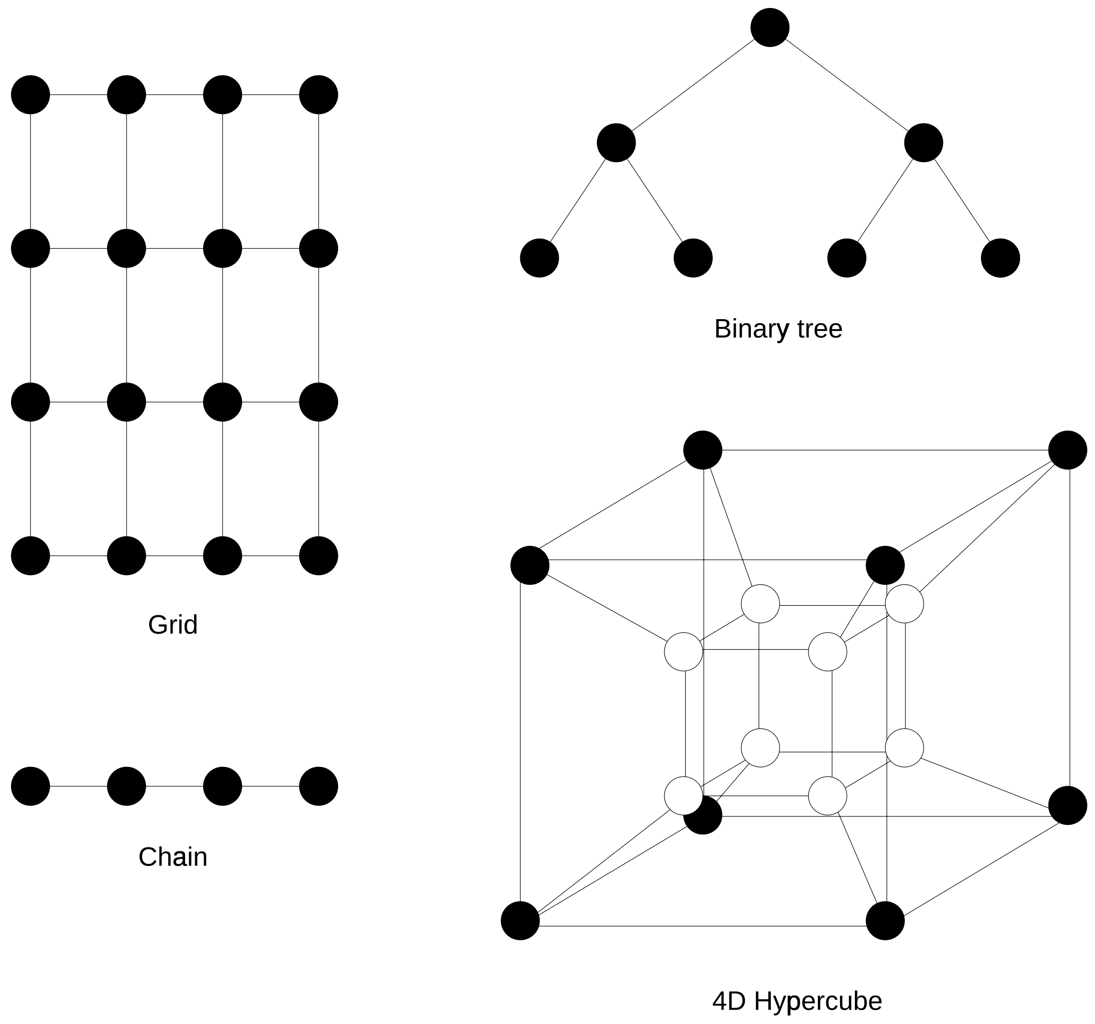


Figure 1.2: Examples of distributed memory MIMD interconnection networks

modules. To improve memory access times each processor will also have a large local cache, but these require complex hardware cache coherency protocols to function correctly. There are many different network designs including buses, cross-bar switches and multistage networks [96]. These architectures usually have only a few, high performance processors since the memory subsystem quickly becomes a bottleneck as the number of processors is increased. There are many examples of shared memory architectures including the Cray Y-MP C90 with up to 16 vector processors.

Distributed memory MIMD architectures have many processors each with its own local memory. These processors are connected together by an interconnection network. As for shared memory architectures, there are many different designs of interconnection networks (for some examples see Figure 1.2). The early Cosmic Cube and Intel iPSC both used a six dimensional hypercube network topology. Systems based around the INMOS T8 transputer usually use chain or grid topologies (see below). Most of the newest architecture designs, such as the IBM PowerParallel 9076 SP1, the Connection Machine CM-5, the Meiko CS-2, and the proposed T9000 architecture (see below), use multistage networks. One new machine, the Intel Paragon, uses a grid topology, and Intel claim the performance of their network matches those of the other manufacturers.

Although each manufacturer uses proprietary custom communications hardware, many of the processing nodes are based on standard commodity microprocessors: IBM uses its RS/6000 processor, whilst the Intel Paragon and others use the Intel i860 processor. The old Meiko Computing Surface used the i860 but the new CS-2 uses a SuperSPARC processor supported by two Fujitsu vector processing chips. The Connection Machine CM-5 also uses a SPARC processor and optional vector processors. The long awaited Cray MPP (Massively Parallel Processor) machine, expected towards the end of this year, will use the new DEC Alpha chip. The trend in distributed memory MIMD architectures is to use commodity microprocessors for individual nodes and custom interconnect probably based on a multistage network. This choice allows parallel systems to benefit immediately from developments in sequential processor architectures by simply updating the processing nodes. In the future when there is more consensus on interconnection network design, we will probably see standard interfaces to the communications network.

As well as the two distinct classes of distributed and shared memory MIMD architectures there are some systems which are a combination of both architectures. Two machines in this category are the Kendall Square Research KSR1 and the Alliant Campus/860. The Campus architecture consists of clusters of 16 processing nodes connected to a shared memory subsystem. Multiple clusters are in turn connected by an interconnection network. The KSR1 has physically distributed memory, but each processor's memory is treated as a cache and data values migrate to a processor as it requests access to them. This allows the programmer to use a shared memory programming model, but hopefully gives better scalable performance than a physically shared memory architecture.

In this work we focus our attention on the distributed memory MIMD architecture offered by the INMOS transputer. However, at all times we are interested in the applicability of principles and the portability of algorithms from this architecture to the other distributed memory MIMD architectures.

1.4 Transputer architecture

The transputer family of microprocessor products are designed and manufactured by an English company, INMOS, based in Bristol. The first generation transputer, the T4, first appeared in 1985. It has a novel architecture that combines a standard 32 bit microprocessor functionality with on-chip support for processor-to-processor communications via four bi-directional serial links. Systems can be designed with very little glue logic since the processor also has on-chip support for DRAM memory. Networks of T4 processors can easily be built using the T4's four hardware links. Early systems

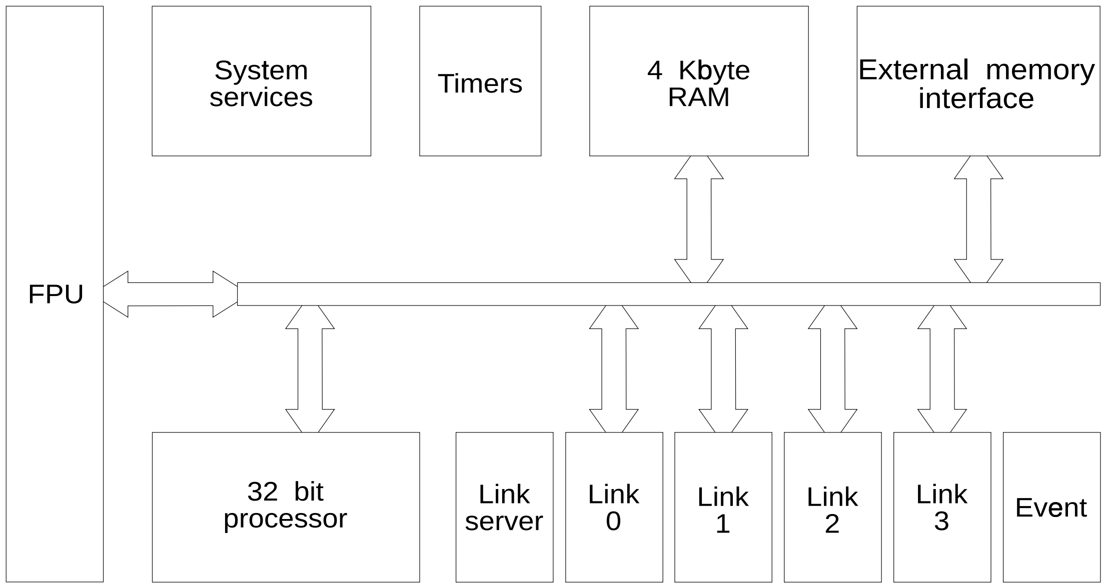


Figure 1.3: Block diagram of the INMOS T8 transputer

consisted of processor boards holding 1 or 4 T4 processors. Boards with 4 processors had their transputers hard-wired into a ring using half of the links, and the remaining links would be taken to the backplane. Rack mounted systems allowed the processors on several such boards to be connected together, by hand, using these remaining links. Reconfiguring such a system by hand was a tedious job and as a result most systems were left configured as a simple chain of processors (with perhaps a return link at the end), or as a grid of processors (see Figure 1.2). The size of networks was also very small with only around 4 to 8 processors available.

One of the main limitations of the T4 processor for use in scientific “number crunching” applications was its poor floating point performance. This quickly led to the development of the second generation of transputers, the T8, in 1988 (see Figure 1.3). The instruction set of the T8 is upwards compatible from the T4, but the main advantage of this processor is that it has floating point operations implemented in hardware. When combined with effective use of the on-chip 4K of memory, the T8 compares favourably with the INTEL i486 processor giving a floating point calculation rate of around 1Mflop/s (see Chapter 2). The T8 retained the four hardware links from the T4 providing maximum transmission rates of either 10Mbit/s or 20Mbit/s. At 10Mbit/s a link provides a usable uni-directional data bandwidth of about 1Mbyte/s (see Chapter 2) and slightly less with bi-directional communication. Larger networks of processors were also being constructed demanding a solution to the problem of hand-wiring the links. Within the ESPRIT Supernode I project (P1085) a switch chip was developed which could statically connect 16 hardware links, with only minor limitations on the connectivity permitted. Networks of these switch chips could be used to provide electronic configuration of the transputers. These developments culminated in the Parsys and Telmat Supernode machine which allows networks of up to several hundred transputers to be electronically configured (see Section 1.7). The Meiko Computing Surface is another machine with similar capabilities that was developed at this time.

All of these machines were still limited to an essentially static network configuration of a chain or grid before a program run (although the Supernode can in principle be reconfigured dynamically). Hence an entire program can use only one network configuration. This leads to compromises in the program design and performance if two sections of a program could be implemented most efficiently on two different configurations. There has been some investigation into the practicality of dynamically switching the network configuration during run-time [9] but this has high overheads and is cumbersome with current systems. Another important issue is the routing of messages within the transputer network. With the current generation of transputer machines a message can only be sent directly by a processor to one of its four immediate neighbours via the hardware links. If the message is destined for a distant processor not directly connected to the source processor then the message must be forwarded through all the intervening processors. For networks with a large diameter this operation can take far longer than communication with a neighbouring processor. Efficient algorithm design must take data locality into account to avoid as much distant communication as possible and instead perform mainly near-neighbour communication. The situation is made worse by the lack of hardware support for message forwarding in the T8 architecture which means that all forwarding must be performed by the user in software.

These considerations led to the design of the next generation of transputer, the T9000 [64], and its companion switch chip, the C104 [65]. The T9000 instruction set is again upwards compatible from the T8. However the instruction decode and execution logic has been completely redesigned. The T9000 is a superscalar, superpipelined microprocessor architecture with multiple functional units (see Figure 1.4). To keep the pipelined functional units supplied with instructions and data the instruction decode logic must process the program instruction stream extremely quickly. This arises from the decision to maintain instruction compatibility with older generations of transputer. This instruction set consists of very simple, single byte instructions which are executed in only a small number of cycles. To provide a large enough instruction throughput the T9000 instruction decode unit includes a “grouper” which groups sets of consecutive instructions in the input stream which can be executed in parallel in the different functional units. A large and carefully designed cache is also required in order to provide data at the rate required. These architectural considerations are common to all the current generation of superscalar, superpipelined RISC processors.

What distinguishes the T9000 transputer from other processors is again its communication subsystem. The T9000 provides four serial hardware links which implement a virtual link, packet-based communication protocol. To complement the T9000 links, the C104 switch chip has inputs for 32 of these hardware links. The switch chip can route messages arriving on one link to any of the other 31 links. Networks of switch

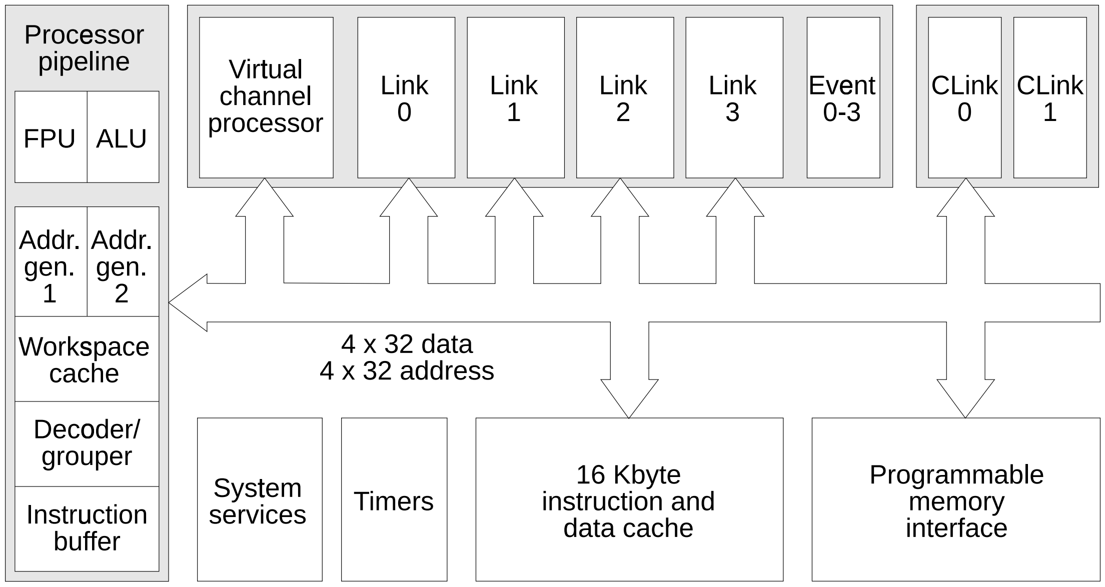


Figure 1.4: Block diagram of the INMOS T9000 transputer

chips can be constructed, with T9000 processors connected to the network on the remaining switch link inputs. This gives a system where a source processor can inject a message into the switch network and the network then routes that message to the destination processor. The source processor does not need to know the location of the destination processor nor the route through the switch network; all this is handled by the switch network itself. Details of how message routing is implemented on these systems is given in Chapter 2 and in INMOS [51].

There are many factors that can determine the choice of switch network including financial cost, network bi-sectional bandwidth, fault-tolerance, size and scalability. The size of the network is measured by the number of attached T9000 processors. This could range from 16 on small systems to several hundred on large systems. Current large T8 systems have between 100 and 400 processors: the Liverpool University Parsys Supernode has 96 T8 processors, and the Edinburgh Parallel Computing Centre Meiko Computing Surface has 400 T8 processors. It is predicted that large T9000 systems for scientific computation will have up to 512 processors, although larger systems are possible. A good measure of the scalability of the network is the degree to which the achieved IO bandwidth of a processing node remains constant as the number of processors increases. A network with good scalability allows extra processors to be added without significantly degrading the achieved IO bandwidth of the original processors. For example although a bus-based architecture allows a large number of processors to be attached, the bandwidth available to each processor decreases rapidly as more processors are added. The bi-sectional bandwidth of the network is the maximum bandwidth available between two halves of the network. This determines the capacity of the network to support heavy communication loads.

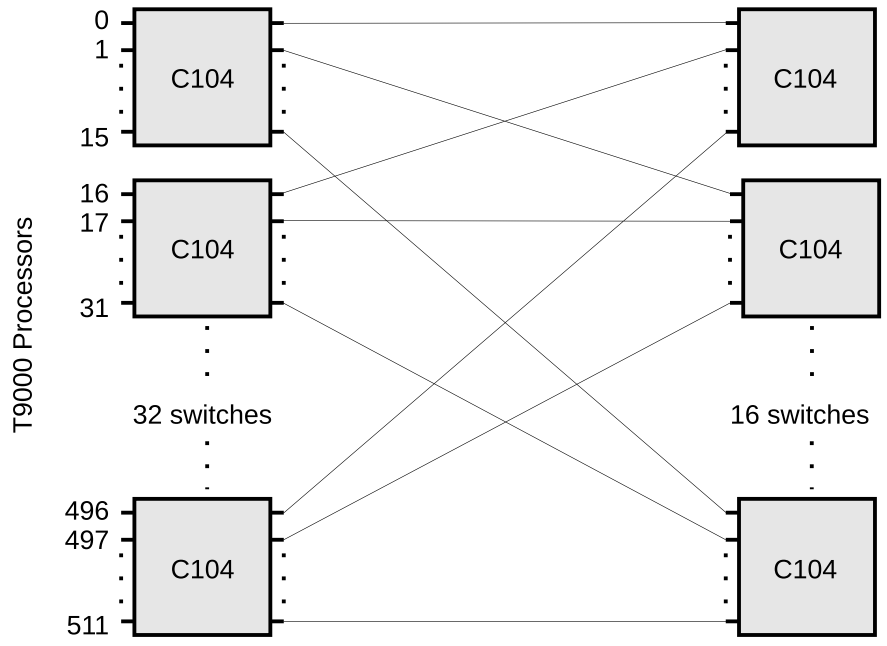


Figure 1.5: three-stage folded Clos-type multistage network

For large systems it is important that faults with one component or another do not cause undue interference to the rest of the system, but instead a graceful degradation in service occurs. In the case of the switch network this means that failure of one switch chip or inter-chip link does not bring the whole system down, but instead alternative message routes are used that avoid the failed components. Of course, improvements in all these areas must be balanced against the increase in financial cost of the system.

Considerable study of these issues has been done by a group at Siemens AG [59, 60]. They recommend using a replicated folded Clos-type multistage network. (See Wilkinson [96] for a comparison of the different types of networks.) This type of network offers good scalability and has a low cost/performance ratio. Figure 1.5 shows an example of a single three-stage folded Clos-type multistage network. This network has 512 external links for connection to up to 512 T9000 processors. The diameter of the network, i.e.. the maximum number of switch chips that a message must pass through, is only three. To increase the network bandwidth and provide good fault tolerance, Hofestädt et al. recommend replicating the network four times, connecting a link from each processor into each network.

Although development of these chips was partly funded by the ESPRIT PUMA project (P2701) which finished at the end of 1991, they are not yet being manufactured in quantity. They are expected to become widely available in 1994.

1.5 Programming models and languages

Accompanying the developments in distributed memory MIMD computer architectures there has been much research into the design of suitable programming models for these new machines. This has led to the development of a wide range of programming languages based on several different programming models [10].

The simplest model is that of a group of sequential processes executing in parallel and communicating with each other by explicit message passing. This model was first introduced by Hoare [57] and called the CSP (Communicating Sequential Process) programming model. The model maps naturally onto a distributed memory MIMD architecture and its low level specification permits efficient implementation on these systems.

Other models have a higher level abstraction from the underlying hardware design. Two of the most important features of these programming models are how the distributed system is viewed and how parallelism is expressed. Although the hardware is a physically distributed system, some programming models have a non-distributed view of the system. Thus the programmer sees a global memory space which is supported by the language implementation. For example the Linda [4] programming language views memory as a global tuple space. Another example is Parlog [22], a parallel Prolog-like logic language. As well as having a non-distributed system view, Parlog also expresses parallelism differently from CSP. In Parlog parallelism is expressed at the level of the logic clause. Parallelism may also be expressed differently in functional and object-oriented languages (see Bal [10] for details). However, the most common way to express parallelism is at the process level with multiple sequential processes executing in parallel, as exemplified by CSP. Most parallel languages like Ada and Concurrent C support this style of parallelism. The `occam` [95] language takes this process model one stage further by considering each individual program statement to be a complete process and thus allows them to be executed in parallel.

Programming models based on parallel process execution may be subdivided according to the model of interprocess communication that is used. The simplest model for distributed memory MIMD systems is message passing between processes: messages traverse the switch network between the source and destination processors. Languages that support a global memory space, such as Linda, allow interprocess communication through shared data structures. This is the model most appropriate for shared memory MIMD systems where processes executing on distinct processors all access a common memory subsystem. For this model, processes must coordinate accesses to shared data structures to ensure correct program behaviour. This synchronisation between processes is achieved by the use of such language constructs as semaphores, critical regions and monitors. On a physically distributed memory system, the use of a global memory programming model must be supported either by virtual shared memory hardware (for example the KSR1) or by software built on message passing.

Message passing models may allow only point to point communications (CSP) or they may include one-to-many messages such as broadcast. Some modern architecture designs provide hardware support for broadcast which can improve program performance significantly (for example, the CM-5). The rendezvous and remote procedure call (RPC) constructs are further examples of message passing models although the interprocess communication is not explicit.

The point-to-point message passing model is supported directly in the CSP and `occam` languages. However, many more parallel systems support this model through the use of extensions to the standard sequential languages such as FORTRAN and C. There is no standard yet for message passing primitives and so many incompatible systems exist including MPI [44], PARMACS [54, 55], Express [27], PVM [14, 90], the proposed BLACS [7, 40] and CS-Tools [71]. Communications systems may provide up to three different types of message passing synchronisation: non-blocking, blocking and synchronous communications. Non-blocking and blocking communications are both asynchronous operations. In a non-blocking communication the source process tells the communication system the area of memory to be sent and immediately continues program execution. The process is informed when the message has been sent. Similarly, the destination process can tell the communication system that it is expecting a message and provide an area of memory for the message. The process continues execution and is informed when the message has arrived. In a blocking communication the source process is blocked until the message has been transferred from its memory into the communication system. A receiving process is blocked until the message arrives in its memory area. In a synchronous communication the source process is blocked until the destination process has received the message.

All the programs described in this thesis have been developed using the point-to-point synchronous explicit message passing model provided by the `occam` language. This language allows very easy expression of parallel constructs and leads to well structured and easily understood parallel programs. The language is also well matched to the transputer architecture. There is, however, an important restriction on the point-to-point communications supported on T8 based transputer systems. The standard `occam` implementation only allows a single communication channel to be placed on each of the four transputer links, and this channel can only connect two processes on neighbouring transputers. Hence, point-to-point communications between arbitrary processes are not supported by the system. This restriction leads to the design of algorithms which use a logical process connectivity which can be mapped onto the physical processor array. Thus most `occam` programs are designed using a chain or grid of processes. The programs in this thesis which have been designed for T8 architectures use a chain of processes (see Chapters 3 and 4), except for the Newton algorithm (see Chapter 5) which uses a grid.

To overcome this restriction a software message routing system for T8 systems has been developed [30]. This system, called the VCR (for Virtual Channel Router), allows communication between processes that are not on adjacent processors. Using the VCR incurs a substantial communication performance penalty on T8 systems so it is not used where efficiency is important. However, the VCR provides an excellent environment for the development of `occam` programs for T9000/C104 machines, which will have hardware support for arbitrary point-to-point communications. Chapter 6 describes a BFGS algorithm which has been designed for the T9000/C104 architecture, but implemented on a current T8 system using the VCR.

As well as looking at parallel program models and language design from the perspective of process parallelism, very important progress has been made using structural parallelism. Structural parallelism partitions data structures across the processes and exploits parallelism by operating on the data elements in parallel in each process. This approach is similar to that taken by languages available on SIMD architectures and is often called the Single Program Multiple Data (SPMD) model. Most commercial MIMD machines have FORTRAN dialects with structural parallelism extensions and the forthcoming High Performance Forum Fortran (HPF [43]) language attempts to standardise these extensions to aid portability of FORTRAN codes. Structural parallelism has been one of the most successful programming techniques and is discussed further in the next section.

1.6 Computational models

Although program design for MIMD systems is still in its infancy, there are already a number of recognised programming techniques, or computational models. These computational models are concerned with the way in which program functionality and data are allocated to different processes. The different models are suitable for different application areas.

The simplest computational model is the process farm. In this model many independent processes are generated and fed with packets of data by a master process which coordinates the work. The processes do not have to be identical, although they usually are. Since the processes are independent there is no communication between the processes; the only communication occurs when the master farms out data to a process and when a process returns a result to the master. This technique is ideal for many image processing tasks, such as ray tracing.

Another technique is to exploit the functional parallelism in an algorithm. The algorithm is partitioned into several processes which perform different functions and data is then input to these processes. The output data of one process may be required as input to another process in which case we construct a pipeline of processes to keep the processes busy and maintain high performance. This differs from the simple farm in that processes are not necessarily independent but may communicate with one another. The degree of parallelism that can be obtained from pipelined functional parallelism is limited by the number of processing steps that the algorithm can be partitioned into and this is frequently rather a small number. For example a compiler might perhaps be partitioned into four pipelined processes: a tokeniser, parser, intermediate code generator and back-end code generator.

The third technique exploits structural, or geometric, parallelism in an algorithm. In this case the data is partitioned instead of partitioning the algorithm functionality.

Each process executes the entire algorithm but only manipulates that part of the data which it holds. The processes will usually exchange data and intermediate results with one another during the course of the algorithm. This is the type of parallelism that is most frequently used in scientific programs and has proved to be very successful. As mentioned in the previous section, this is also the model of parallel computation best supported by most commercial Fortran dialects including HPF. The design of algorithms using structural parallelism is primarily concerned with finding a decomposition of the data which minimises the amount of interprocess communication without unduly increasing the amount of computation required. For scientific applications, which frequently consist of operations on large dense arrays, the decomposition is usually achieved by partitioning the arrays into equally sized blocks of some regular shape. The algorithms examined in this thesis all make use of structural parallelism.

Other computational models include the divide-and-conquer, dataflow, and systolic models. The divide-and-conquer model is ideal for graph operations. The systolic model has been used successfully for simulating molecular dynamics and has similarities to a fine-grained structural parallelism computational model. Dataflow and systolic models of computation both have very fine-grained parallelism and so do not perform well on most general purpose distributed memory MIMD architectures. However, specialised architectures that support each of these models do exist, for example the Manchester Data-Flow Computer [53] and the iWARP [82].

1.7 Implementation and performance

The algorithms studied in this thesis have been designed and implemented for T8 and T9000 based transputer architectures. The programs have been developed on the Parsys Supernode system available at Liverpool. This is a multi-user machine and currently has a total of 96 T8 processors. Each user may request a domain of processors from the pool of free processors, and electronically configure these processors to his required topology. For this thesis, the programs have been benchmarked on networks of up to 48 T8 transputers. These processors were all running at 25MHz, and the transputer links were running at 10MHz.

Most of the programs for T8 transputer systems were designed for a simple chain of processors (see Chapters 3, 4). When this work was started, this topology was the one that was most likely to be available on transputer systems, since all machines required the links to be hand-wired. This consideration is not as important now as most new systems, like the Parsys Supernode, have electronic reconfiguration. Also, although the chain topology does not provide as much connectivity as a grid of processors, it has been found that many of the communication operations required performed almost as well on a chain as on a grid.

Chapters 4, 5 and 6 describe the design of algorithms for T9000/C104 architectures. Even though T9000 machines are not yet available, we have implemented some T9000 programs using the VCR communications system [30] on the Supernode (see

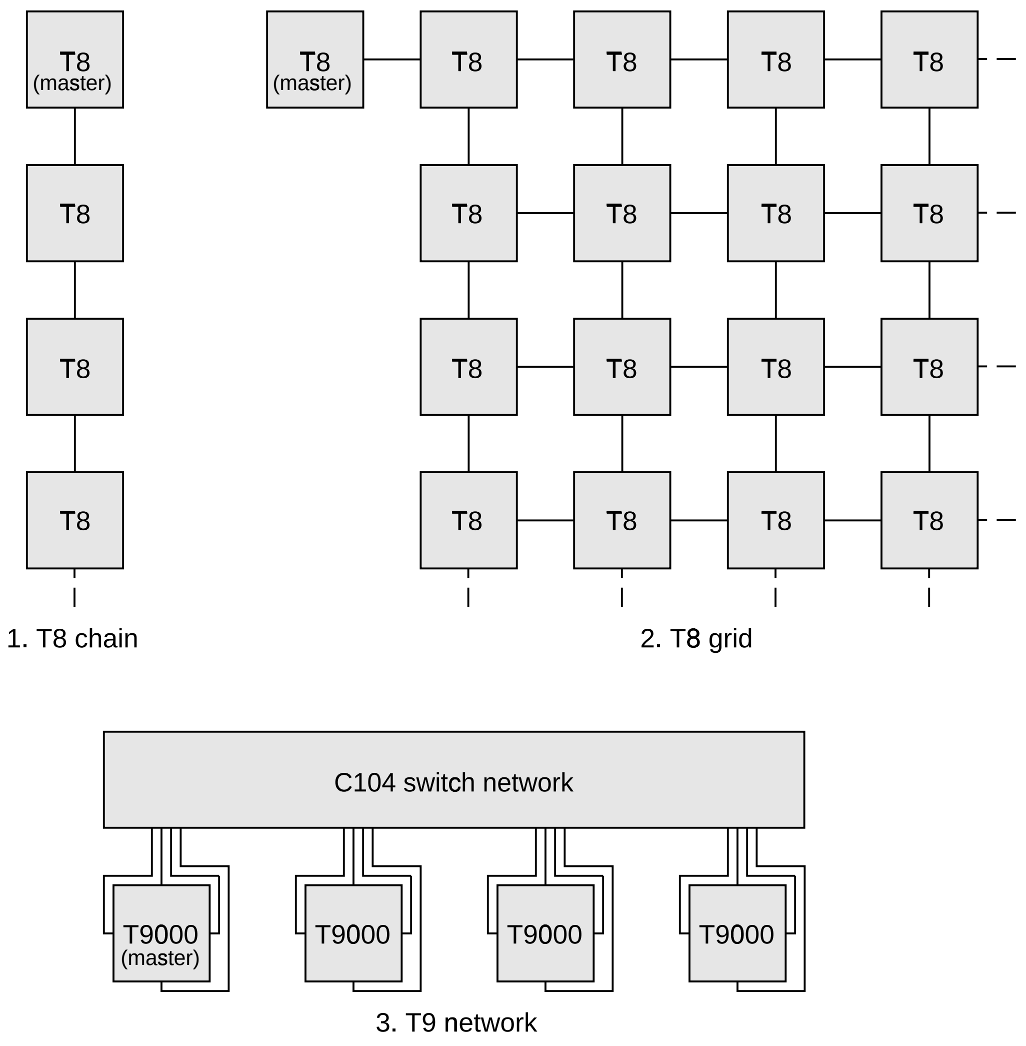


Figure 1.6: Network topologies used by algorithms

Chapter 6). This allows the development of programs which use arbitrary point-to-point communications which will be supported by the C104 switch network. Unfortunately, as mentioned previously, the VCR introduces a large performance penalty, so the performance of these programs on the T8/VCR system cannot be compared with the performance predicted by the T9000 cost models. However, when T9000 systems are available it will be possible to simply recompile the programs for the new machine and immediately benefit from the improved communications capabilities of the C104 switch network.

Figure 1.6 shows the network topologies used for programs in this work. The programs all use the same model of computation which consists of a “master” process coordinating and controlling a number of “slave” processes. The slave processes can communicate with one another and with the master process. The master process also communicates with the outside world to receive input data and return results to the "user." This structure leads to algorithms with three distinct stages:

1. master scatters input data to slaves,
2. slaves solve the problem, and
3. master gathers result data from slaves.

In most instances the "user" will in fact be a program which calls the master process passing data as parameters. Hence the calling program and master process will both be executing on the same processor.

For T8 networks the need for communication with the outside world determines where the master process is placed in the network. We cannot assume that all the master processor's links are available for connection to slave processors since some must be used for connection to the outside world. At a minimum, one link must be available for connection to the slave network, and in general most T8 machines such as the Supernode do have only a single link connection from the master processor to the slave network. With only a single link available for T8 networks the master process is placed at one end of the chain of slave processes or hangs off one corner (or edge) of a grid of slave processes (see Figure 1.6 parts 1. and 2.). This single link may be a bottleneck for communications, especially for the grid topology, and in general it is best if the algorithm makes minimal use of the master process. This leads to T8 algorithms where the slave processes perform almost all of the computation and the master process just handles communication with the user. For this reason, we specify the size, p, of a T8 network as the number of slave processors and assume the master process is running on another processor.

Communications with the outside world on a T9000/C104 machine can take place over a virtual link through the C104 switch network. Hence all the links on a processor may be connected to the switch network and available for communication with other processors. This means that the master process can be located on any processor in the array and has the same communication ability as any other slave processes (see Figure 1.6 part 3.). For this reason, in the algorithms designed for a T9000/C104 machine the master process takes a full part in the solution of the problem. It handles communication with the user and also executes the slave algorithm. This arrangement still requires two programs: one for the master process and one for the slave processes, but makes more effective use of the master processor instead of leaving it idle much of the time as is the case for the T8 algorithms. Hence, we specify the size, p, of a T9000/C104 network as the total number of processors used including the master processor. See Chapter 7 for a discussion of the merits of these algorithmic models.

We have chosen `occam` as the implementation language for these programs. `occam` allows very simple expression of parallel constructs and the CSP model. This leads to well structured and easily understood parallel programs. Unfortunately, this language is not widely available on other architectures. Thus the desire for portability of programs from one architecture to another has now led to the use of Fortran for all further development work (see Chapter 7).

Measuring and presenting the performance of computer programs, and especially parallel computer programs, has always been a contentious issue. In this thesis we have tried to present data in a manner which most clearly demonstrates the analysis given in the text. The elapsed time, or wall-clock time, for trial program runs is given in tables to allow further analysis to be conducted. All other performance information is derived from this raw data.

For each parallel program we are interested in how the program performance scales with changes in problem size, $\mathbf{n}$ ($\mathbf{n}$ is a vector of problem parameters), and the number of processors, p. We also wish to compare this performance with that of the best sequential program for the same problem. Let us define the wall-clock execution time of the best sequential program for the problem to be $T_s(\mathbf{n})$, and the wall-clock execution time for the parallel program to be $T(\mathbf{n}, p)$. We now define the speedup, $S(\mathbf{n}, p)$, for the parallel program as

$$S(\mathbf{n}, p) = \frac{T_s(\mathbf{n})}{T(\mathbf{n}, p)}.$$

The speedup, as defined here, shows the variation in performance, i.e., the rate of computation, of a parallel program as the number of processors or problem size vary. Speedup is a relative measure, with no units, and is normalised so that a parallel program with a speedup of 1 represents the same computation rate as that given by the best sequential program for the same problem size. We present graphs of speedup, $S(\mathbf{n}, p)$, against the number of processors, p, for fixed problem sizes, $\mathbf{n}$. These graphs show clearly how well the parallel program scales with machine size. Of special interest is the location of the peak of the graph which shows the number of processors which gives the maximum performance for a given problem size. Since this definition of speedup is normalised to the cost of the best sequential program for the problem, speedup can also be used to compare different parallel program which solve the same problem (see Chapter 6).

In order to be able to compare the performance of parallel programs which solve different problems we must define a measure for the absolute performance of a parallel program. The absolute performance of scientific computer programs is usually expressed in terms of millions of floating point operations per second (Mflop/s). Let us define $F(\mathbf{n})$ to be the number of floating point operations required by the best sequential program for a given problem. The absolute performance of the parallel program is then given by

$$R(\mathbf{n}, p) = \frac{F(\mathbf{n})}{T(\mathbf{n}, p)}.$$

Graphs of $R(\mathbf{n}, p)$ against p for fixed $\mathbf{n}$ are identical to the speedup graphs defined above, except for one crucial difference: The scale on the y-axis for $R(\mathbf{n}, p)$ has units of Mflop/s. The performance of one parallel program in Mflop/s may be compared with another parallel program for a different problem. The performance of programs running on different architectures may also be compared. We can get a measure of the efficiency of the parallel program by comparing the absolute performance with the maximum performance available from the number of processors used. When making these comparisons, we must be aware that this measure of absolute performance, $R(\mathbf{n}, p)$, is not constant, but varies with problem size and number of processors, and varies between different parallel programs.

For practical purposes, such as estimating the run-time for a particular problem or finding the number of solutions that can be obtained in a given time, we define the temporal performance, $R_T(\mathbf{n}, p)$, to be

$$R_T(\mathbf{n}, p) = \frac{1}{T(\mathbf{n}, p)}.$$

This gives us the number of solutions per second that a program delivers for a given problem size and number of processors.

To accompany these practical measurements of the performance of each parallel program we develop analytical cost models for each program. These cost models express the wall-clock execution time for a program in terms of a number of problem parameters, $\mathbf{n}$, and a set of hardware parameters which measure the performance of important components of the architecture such as the floating point unit and communication network. In the next chapter we discuss suitable parameters for transputer systems and estimate values for these parameters.

Chapter 2

Modelling parameters

It is common for the description of a sequential numerical algorithm to include an expression which specifies the "cost" of that algorithm in terms of the "size" of the problem to be solved. This allows different algorithms to be compared and the most efficient for the intended purpose to be selected. We can extend that idea to develop cost models for parallel local memory architectures with point to point communications such as the transputer. In this chapter we propose sets of modelling parameters for T8 and T9000 based architectures.

2.1 Introduction

We are concerned here primarily with the total execution time of an algorithm on a given architecture, and give only secondary consideration to factors such as efficient use of processor and memory resources. Hence the "cost" of an algorithm is the execution time of that algorithm and the algorithm with the best performance is that which has the smallest execution time. When investigating different algorithms for any type of computer, it is very useful to be able to predict before coding how well the algorithm will perform. This would allow us to reject costly algorithms without spending time coding and debugging them. One way of doing this is to find a cost model for an algorithm. A cost model is an expression which gives the total cost of an algorithm for different problems. It allows you to compare the performance of different algorithms, and also allows you to study the variation in cost of one algorithm for different problems. Cost models should be relatively easily to develop compared with the time required to actually code an algorithm. This requirement means that the model must be simple and only take account of the most significant features of an algorithm and architecture. The costs predicted by the model need not be accurate but should be good enough to allow realistic comparisons of performance. If more accurate costs are required then detailed models may be developed or the algorithm may be coded and run on a simulator.

In traditional complexity analysis an algebraic model is derived which gives the worst case cost of the algorithm. Each feature of the problem to be solved which affects the cost of the algorithm is modelled by a *problem parameter*. In most numerical algorithms these parameters measure the "size" of the problem. For example a general matrix vector multiply requires two parameters m and n which give the row and column dimension of the matrix. The algorithmic cost is then given in terms of the number of floating point (FP) multiplications required as a function of the problem parameters. This is mn for our matrix vector multiply example. The decision to include only the number of FP multiplications is due to the significantly higher cost of these operations compared with FP addition and subtraction on older machines. This decision requires careful reconsideration for modern architectures such as the transputer (see Section 2.2).

In more complicated numerical algorithms the algorithmic cost is not so easily predicted. The cost of iterative algorithms cannot be given exactly since the number of iterations performed is problem dependent. However, the rate of convergence of an algorithm allows some comparisons to be made between different algorithms. Also the cost of each individual iteration is important; even if more iterations are required, an algorithm with cheap iterations may perform better than one requiring fewer, more expensive iterations. The iteration cost may be modelled in terms of problem parameters, e.g. in an eigenvalue problem we use n the matrix size. For some algorithms the number of primitive FP operations in an iteration is not known since it may, for example, depend on a user-supplied function of arbitrary complexity. The Newton method for unconstrained optimisation has this feature. In this case, the cost of an iteration may be given as an estimate of the required number of evaluations of the user-supplied function.

For modern computers, using the total number of FP multiplications as the basis for the cost model is not acceptable since other FP operations such as addition take comparable time and have comparable operation counts. For example, the Level 1 BLAS dot product has only 1 less addition than multiplication. One early suggestion was thus to use the multiply-add operation as the basic FP operation (flop). However, it is now generally accepted that each individual FP operation, such as addition, subtraction, multiplication and division, should be counted separately and a combined total number of FP operations given as the cost of an algorithm. An algorithm can thus be said to have a cost of say 10Mflop if it entails the execution of a total of 10 million FP operations. The computation rate of a particular architecture for a given algorithm is then given as say 1Mflop/s if that architecture executes 1 million FP operations per second.

In the prior discussion the cost expression gives values proportional to the execution time of the algorithm. If the actual time taken is required then the expression must be multiplied by a constant of proportionality which is the time taken to perform a single FP operation.

The problem parameters for an algorithm on a parallel local memory machine will include all those used in sequential algorithms to specify the problem size. The most important additional parameter is the number of processors, p, used to solve the problem.

A cost model expression for a parallel algorithm on a local memory machine will also include *architecture constants.* Architecture constants are introduced for each feature of the machine architecture that affects the cost of the algorithm. These constants are required so that terms in the cost expression due to each architectural feature are added in correct proportions. For example, if a unit of communication takes twice as long as a unit of computation then the communication constant must be twice as large as the computation constant. We chose to give each constant a value which is an estimate of the real cost of the operation so that the cost model expression gives the actual execution time for the algorithm. By varying the values given to the architecture constants we can predict the behaviour of an algorithm on future generations of machines. For example we can study the effect of varying the communication rate relative to the computation rate to try and find the balance that gives the best performance for an algorithm.

There are many features of transputer architectures which could be incorporated into a cost model as architecture constants. However to keep the models relatively simple and quick to develop we must select only the most important features. The two fundamental operations that must be included in the model are FP computation and inter-processor communication; but even these have hidden complications.

2.2 Modelling computation

For simplicity we wish to model FP computation by a single architecture constant, T_f. As explained earlier, we model the computation cost of the algorithm by the total number of FP operations, and so our architecture constant should be an "average" of the individual FP operation costs combined in proportion to the frequency of their use. For some architectures the cost of different FP operations varies so much that using a single architecture parameter may not be sufficient to model the performance well. For example heavily vectorised architectures such as the Cray require two parameters to model computation: one parameter reflects the startup cost for a FP vector operation and the other reflects the cost of an FP operation on a single vector element. However for most modern microprocessor architectures scalar FP operations are performed in hardware and the operations have comparable cost. This is true of the T9000 (double precision additions and subtractions take 2 cycles while multiplications take 3 cycle [64]) and, to a reasonable extent, true of the T8 (addition and subtraction cost 7 cycles; multiplication 20 cycles [63]).

There are other microprocessor operations which take a comparable time to perform, for example replicated `SEQ` constructs and FP comparisons. We do not want to include these operations into our model as separate constants or the model will become too complicated. Provided an operation is only rarely used compared to normal arithmetic (for example FP comparisons to control branching in many algorithms), then we can omit it without too much loss of accuracy. To take account of frequently used program control flow code (like the replicated `SEQ` construct which is very common) we use a value for T_f which has been measured experimentally from an arithmetic expression in a program loop. Numerical algorithms usually operate on arrays so array indexing is also included in the test arithmetic expression. It is hoped that this method of estimating T_f will also take into account the advanced pipelining of the T9000 as well as the parallel execution of the floating point unit and integer unit. To measure a value for T_f on a T8 a test program was timed which performed a dot product between two vectors each of 10000 double precision elements. The test was repeated 5 times and the average time of the runs calculated. This gives a value for $T_f = 1.62\mu$s for double precision, which is a rate of 0.617Mflop/s. This experimental measurement of T_f works well for predicting algorithmic costs on T8 processors (see Chapter 3 and [74, 62]). It should be noted that hand-crafted assembler code for compute intensive loops can achieve much higher levels of performance than a high level language. For example, an assembler coded dot product routine achieves 1.14Mflop/s. If implementations make good use of assembler code routines then the latter value may be substituted. While such practical measurements are not possible for the T9000 we must estimate the FP computation rate. At a 50 MHz clock frequency the T9000 is capable of a peak rate of 25Mflop/s double precision [64]. INMOS estimates that sustained rates will be at least 10Mflop/s, and we expect that hand-coded assembler should reach 20Mflop/s. For this work we will use the conservative sustained rate of 10Mflop/s which gives a value of $T_f = 100$ns.

2.3 Modelling communication

Modelling the hardware communication cost for the T8 architecture is fairly straightforward. For numerical algorithms, we are interested in the cost of communication of vectors between neighbouring T8 transputers on a single link. Experimentation, and experience, showed that a single parameter T_c, the cost of sending one double precision value between neighbouring transputers, can model communication of large vectors quite well. To find a value for T_c a test program was timed which passed a vector of 10000 double precision elements down a hardware link. The test was repeated 5 times and the average time of the runs calculated. The resulting value was $T_c = 8.80\mu$s, which is a rate of 0.909Mbyte/s. Numerical algorithms developed for T8 networks are in general specific to a particular configuration of transputers, for example a chain, ring or grid topology. This is due to the lack of high-level communications functionality initially provided with transputer systems, and to the significant improvement in performance gained when an algorithm is tuned to exploit a particular topology. This improvement in performance comes from the ability to exploit locality of data and use near-neighbour communications instead of communication with distant processors which requires forwarding of messages through several intermediate processors before reaching the destination. Hence, to retain accuracy, cost models for algorithms running on the T8 architecture will be specific to the topology of the T8

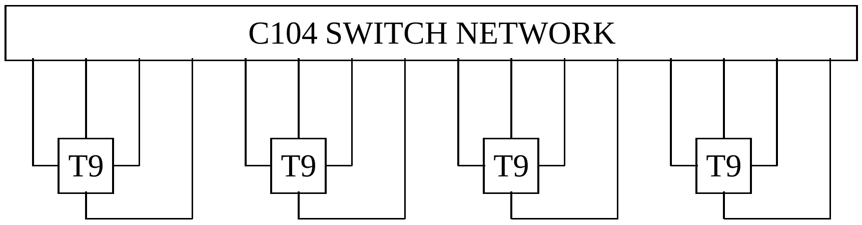


Figure 2.1: Possible T9000/C104 configuration

network for which the algorithm is designed. Models for high-level communications operations for T8 networks are thus quite complicated (see Appendix A).

Modelling communication costs for the T9000/C104 architecture is much more difficult than modelling its computation cost. Again we wish to keep the number of constants required to a minimum. For networks of T8s a single constant fits the actual cost quite well. However the more sophisticated technique used to implement channel communications on T9000/C104 networks (see [64, 65, 51]) requires a different model. For T8 networks an algorithm and accompanying cost model was specific to a particular configuration of transputers. However, the physical configuration of C104 switch chips on a T9000/C104 machine does not restrict the logical configuration of processors required by an algorithm since, as mentioned in Chapter 1, the architecture provides "virtual links" between any pair of processes. As an algorithm is now independent of the switch network configuration, we would like the cost model for the algorithm also to be independent of the configuration. In general the configuration of the switch network will be unknown; machines with the same number, p, of T9000 processors may have different numbers of C104 switch chips connected in different topologies balancing the requirements of network performance and financial cost. Hence, for both programming and modelling purposes we consider a T9000/C104 machine to be a set of p processors with all links connected to a general switch network (see Figure 2.1).

Unfortunately, this flexibility in physical configuration makes it difficult to model the cost of communication accurately. This is because each switch chip through which a packet travels introduces a significant delay of about 1μs (ten times the cost of FP multiplication) and the number of switch chips that a packet crosses is unknown. There are also many further complications such as the possibility of saturation of the network bandwidth, hot-spots and random routing of messages [60]. Within the T9000 processor itself we should consider the costs of message packetisation, utilisation of the link bandwidth, process scheduling, memory bandwidth and caching.

In the following sections we examine the T9000/C104 architecture in more detail developing cost models that include some of the architectural features explicitly. Then we see how we can develop simple cost models of communication for use in modelling numerical algorithms.

2.4 A Plethora of Parameters

Message communication is explained fully in the INMOS documents [64, 65, 51], however a brief overview of the process will be helpful in understanding the following sections. The source process initiates a communication by setting up a Virtual Link Control Block (VLCB) with information about the message size and location and the destination processor. It is then descheduled while the Virtual Channel Processor (VCP) co-ordinates the message transmission. The VCP splits the message into packets of 32 bytes in length. Each packet is injected into the switch network with a header and End Of Packet (EOP) marker attached. The header contains the identity of the virtual link which the switch network uses to route the packet through the network to its destination. The packets are sent one at a time and the VCP only transmits another packet when it has received from the destination process an acknowledge packet (ACK) for the last transmitted packet. This maintains synchronised communication and ensures that no data is lost. The last packet must be acknowledged before the source process is rescheduled.

Let us consider the following scenario: a source process on a processor wishes to send a multi-packet message to a destination process on another processor connected to the switch network. There are s switch chips on the path that the message will travel from source to destination processor. We assume that the bandwidth of the network is great enough to allow us to neglect the possibility of collisions between different messages. For simplicity we also assume that the destination process is ready to receive the message when the first packet arrives. This avoids the need to consider idle time in source and destination processor. However, the model for a complete numerical algorithm should take account of the idle time due to synchronisations between processes.

The following parameters may be useful in modelling the cost of this scenario:

n *number of bytes in message*

Let us assume that n is a multiple of b (see below).

h *number of bytes in header*

We will use a value $h = 3$ giving 2^{24} distinct channels which should be ample even for large networks and algorithms which use many channels.

b *maximum number of bytes in a packet*

Value: $b = 32$.

s *number of C104 switch chips a packet passes through*

α *transmission time*

The time taken to transmit a single byte on a link. We will use a value of $\alpha = 100$ns (i.e.10 Mbyte/s link speed).

β *packet initialise time*

The time taken for the Virtual Channel Processor (VCP) to initialise the output of a packet on a link. Value: $\beta = 200$ns.

γ *channel initialise time*

The time taken by the integer unit to set up a Virtual Link Control Block (VLCB) for output of a message on a virtual link. Value: $\gamma = 500$ns.

δ *switch delay time*

The delay introduced by a C104 switch chip as a packet passes through it. Value: $\delta = 1\mu$s.

(Values for these parameters are estimates only.)

2.5 Packet Transmission

For the transmission of a single full packet between processors there are two different expressions for the cost (as measured by the source processor) depending on the number, s, of switch chips that the packet traverses. If s is small enough so that the source receives an acknowledge packet (ACK) before it has transmitted all of the bytes in the packet then the cost is simply the time taken for the source processor's VCP to be initialised and to transmit the header, data bytes and EOP token i.e., $\beta + (h + b + 1)\alpha$. (For simplicity, we use the same cost for communicating a token as for a single byte). However, as s increases the time taken to receive the acknowledge is increased by the delay introduced by the intervening switch chips. For sufficiently large s the time to receive the ACK is larger than the cost to output the packet and so the cost is determined by the time taken to receive the ACK. The time taken for the VCP on the source processor to initialise and output the header is $\beta + h\alpha$. After a delay of $s\delta$ the header has been received by the destination VCP. The VCP processes the header to generate an ACK in time β and this is transmitted back at cost $(h + 1)\alpha$. The ACK is received by the source VCP after a further delay $s\delta$. The total cost in this case is $2\beta + (2h + 1)\alpha + 2s\delta$. So the cost of transmitting a full packet is given by:

$$T_p = \max \left\{ \begin{array}{l} \beta + (h + b + 1)\alpha \\ 2\beta + (2h + 1)\alpha + 2s\delta \end{array} \right. \tag{2.1}$$

Figure 2.2 shows the cost for transmitting a full packet across varying numbers of switch chips using estimates for the parameters as given in Section 2.4. The point at which the cost of transmitting a packet begins to depend on s, the number of switch

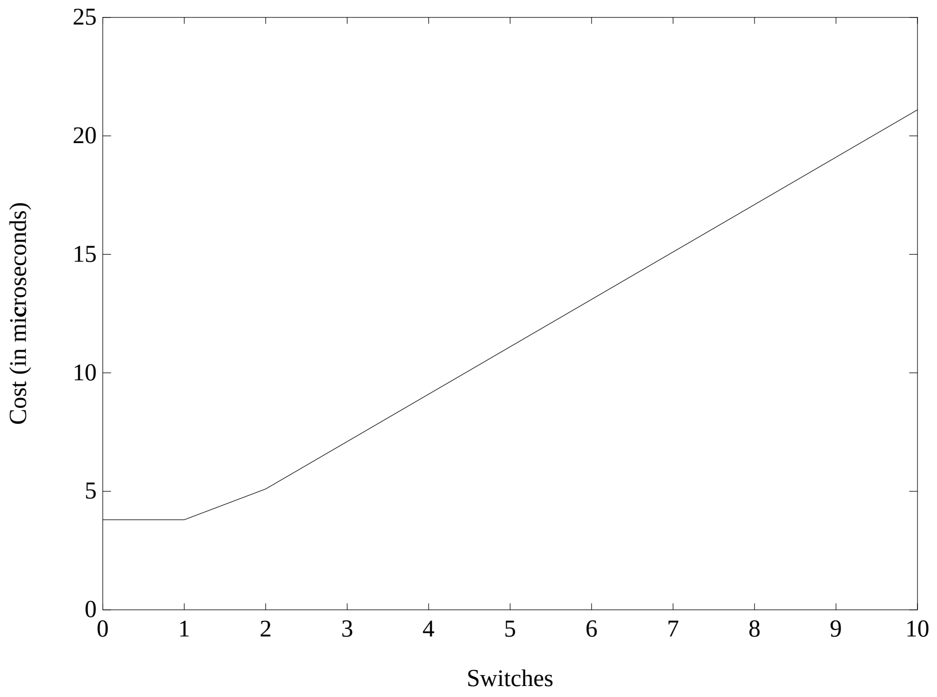


Figure 2.2: Cost of transmitting a single packet

chips traversed, is derived from Equation 2.1:

$$s \geq \frac{(b-h)\alpha - \beta}{2\delta} \tag{2.2}$$

2.6 Message Transmission

In this section we are concerned mainly with the communication of long messages ($n > 100$bytes). This is common in numerical library codes, which frequently communicate large matrices and vectors. To simplify the models we assume that n is also a multiple of b. Given the time to transmit a packet in Equation 2.1 the cost to transmit a message of length n is:

$$T_m = \gamma + \frac{n}{b} T_p$$

i.e.,

$$T_m = \max \left\{ \begin{array}{l} \gamma + \frac{n}{b}(\beta + (h+b+1)\alpha) \\ \gamma + \frac{n}{b}(2\beta + (2h+1)\alpha + 2s\delta) \end{array} \right. \tag{2.3}$$

Figure 2.3 shows the cost for transmitting messages of various sizes across varying numbers of switch chips.

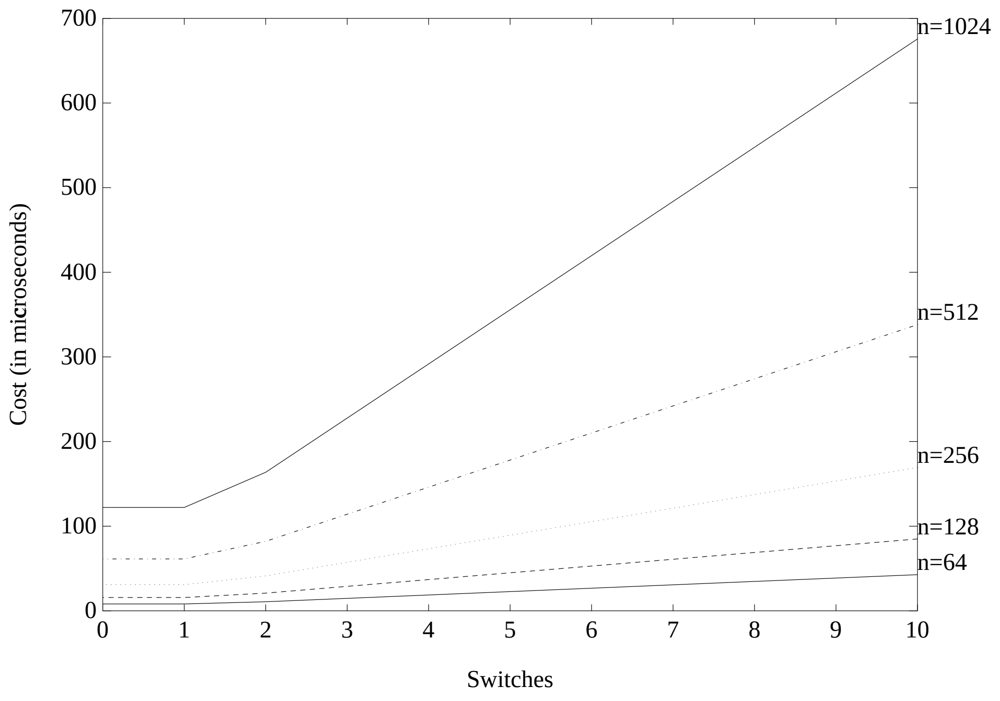


Figure 2.3: Cost of transmitting a single message on one channel

Equation 2.2 gives the point at which the message transmission cost begins to depend on s. With the estimated values for the parameters this means that the best utilisation of a link for a single channel communication is only obtained when at most a single switch chip is traversed. If there are two or more switch chips on the path then idle time of the source VCP and link engine are introduced whilst the ACK is awaited. T9000/C104 machines consisting of more than 32 T9000 processors connected by a switch network will require more than one switch chip and so the best link utilisation will not be obtained on a general purpose machine with only one channel to a link.

There are two ways of improving the utilisation of the link bandwidth. One way would be to introduce *excess parallelism* into the algorithm, placing m identical processes on each processor. In this case while a delayed acknowledge packet prevents transmission of the next packet on one channel on the link, packets on other channels mapped onto this link may be transmitted. If there are enough channels wishing to communicate at a time then the full utilisation of the link can be achieved. This method does not require complex programming by the algorithm developer, but only that the algorithm is fine-grained enough to be able to use mp processes. It should be noted that compute processes may also execute in parallel with the communicating processes since they use separate execution units within the processor.

The second method for increasing link utilisation is to split the single message that a process wants to transmit into multiple blocks and transmit these blocks on several

```
...  some code
-- Declare c channels
[4]CHAN OF []REAL32 chanvec:
PLACED PAR
  -- Source process
  [1024]REAL32 source.vec:
  INT block.size:
  SEQ
    ...  some code
    block.size := 256
    PAR i = 0 FOR 4
      chanvec[i] ! [source.vec FROM block.size*i FOR block.size]
    ...  some more code
  -- Destination process
  [1024]REAL32 dest.vec:
  INT block.size:
  SEQ
    ... some code
    block.size := 256
    PAR i = 0 FOR 4
      chanvec[i] ? [dest.vec FROM block.size*i FOR block.size]
    ...  some more code
...  some more code
```

Listing 1: Full `occam` Code to Communicate a Message using Multiple Channels

channels mapped onto the same link. This technique will require complicated communication routines to divide, transmit and reassemble the message correctly. Complete routines should be provided as part of a standard communications library so that the user does not need to consider their implementation. A collection of suitable routines that have been required for the algorithms developed in this work are described in Appendix A.

Listing 1 shows a full `occam` fragment that communicates a vector between two processes using multiple channels. Ways in which this can be implemented as a general library routine are unclear; for example, how can a user specify the source and destination processes to a library routine? This requires “named process” functionality in the compiler.

The number of channels that need to be used to saturate a link depends on the number of switch chips that the message must traverse. The number of channels, c say, should at least be large enough so that when the first channel to transmit a packet has just received an `ACK` the last channel has finished transmitting its packet, i.e.:

$$\begin{aligned}
\text{time to output on other channels} &\geq \text{time from end of output to receive } \texttt{ACK} \\
(c-1)(\beta+(h+b+1)\alpha) &\geq 2\beta+(2h+1)\alpha+2s\delta-\beta-(h+b+1)\alpha \\
c &\geq \frac{2\beta+2s\delta+(2h+1)\alpha}{\beta+(h+b+1)\alpha} \qquad (2.4)
\end{aligned}$$

Let us consider the example three-stage Clos-type network in Chapter 1. For this example machine with random routing of messages disabled the number of switch chips that a message traverses will be at most 3. Using $s = 3$ in Equation 2.4 suggests that about 2 channels will be required to communicate in parallel on a link to fully utilise the link bandwidth. It should be noted that if $s = 0$, i.e., the processors are connected together directly, one channel on a link saturates the link bandwidth.

The total cost of communication using multiple channels for a single message is derived as follows. If the number of channels used is not enough to saturate the link bandwidth then each channel transmits its n/cb packets with a wait for ACK between each one. The time between the output of the first packet on the first channel and the receipt of the last ACK on that channel is $(n/cb)(2\beta + (2h+1)\alpha + 2s\delta)$. The last channel receives its last ACK after a further time $(c-1)(\beta + (h+b+1)\alpha)$. On the other hand, if enough channels are used to saturate the link bandwidth transmission of packets is continuous with no waiting for ACK packets and the transmission cost is $(n/b)(\beta + (h+b+1)\alpha)$. In both cases there is the additional cost, $c\gamma$, incurred by the initialisation of the VLCBs by the integer unit. However, because of the parallel execution of the integer unit and the VCP, communication may take place on the first channels while the remaining channels are still being initialised. In this thesis we have included the full cost of the channel initialisations in the models. The total cost for a multi-channel communication is:

$$T_{mc} = \begin{cases} c\gamma + \frac{n}{cb}(2\beta + (2h+1)\alpha + 2s\delta) & \text{if } c < \frac{2\beta+2s\delta+(2h+1)\alpha}{\beta+(h+b+1)\alpha} \\ \qquad +(c-1)(\beta + (h+b+1)\alpha) & \\ c\gamma + \frac{n}{b}(\beta + (h+b+1)\alpha) & \text{otherwise} \end{cases}$$

Figure 2.4 shows how the cost of the multi-channel communication of a single message varies with the number of channels and switches. Figure 2.5 shows that the cost of the communication will be independent of the number of switch chips traversed provided that enough channels are used.

At worst, this method gives as good a performance as when a single channel saturates a link, except for the cost of the additional channel initialisations. However, the overhead due to channel initialisation is small compared with the full cost of the communication. For large networks where the round trip time to receive the ACK dominates the cost for the single channel communication, this method provides a significant improvement in performance. Hence this method of communication seems worthwhile in all practical situations whether c and s are known or not. If these routines are provided as part of a communications library then a fixed value of c, say $c = 2$, would probably be used which was sufficiently large to saturate the link bandwidth for communications across the largest number of switch chips that is likely to occur on a machine. For the example three-stage Close-type network in Chapter 1 Figure 2.5 shows that only 2 channels are needed to saturate the link.

If small messages (a few packets in length) are transmitted then the multi-channel method still provides a significant performance improvement over a single channel communication provided that more than one switch chip is traversed (and thus a single channel cannot saturate the link bandwidth). Even with only one packet per channel

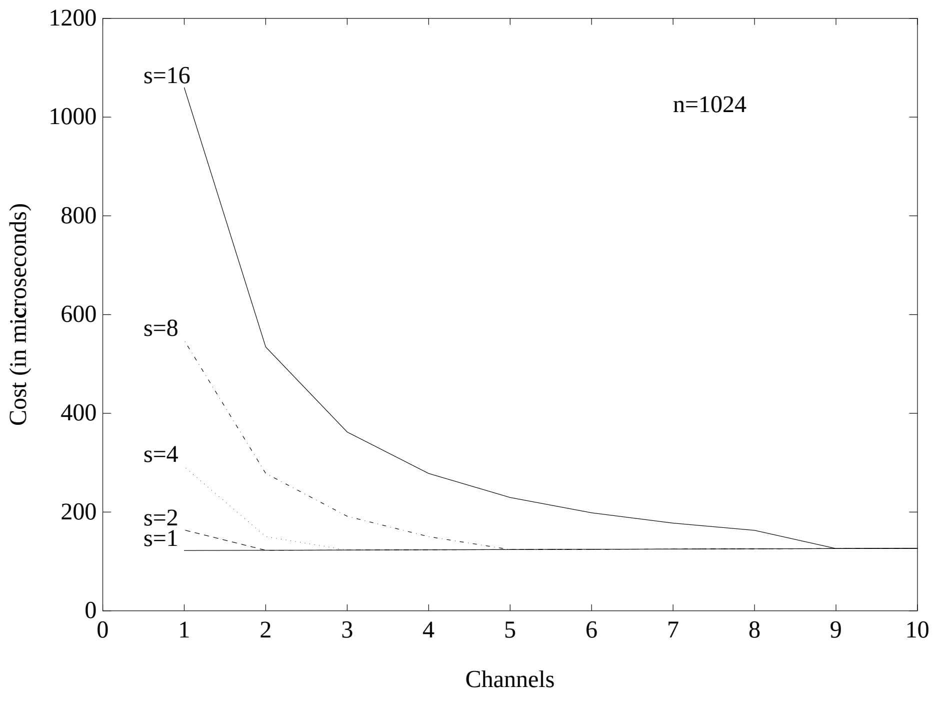


Figure 2.4: Cost of multi-channel communication against number of channels

the cost of the extra channel initialisations is only a small part of the total cost. For a very small message (up to one full packet i.e., $n \leq 32$) the simple single channel communication is better. Most numerical library programs will communicate messages of a wide range of lengths, so both multi-channel communications and single channel communications will be used. For example in Gaussian elimination multi-channel communications would be used for scattering the matrix, and a single channel might be used to broadcast pivot information at each step of the algorithm.

A further refinement to the technique for communicating a message between two processors would be to make use of several, l say, of the links on both the source and destination processor. Each link would have enough channels mapped onto it to saturate the link bandwidth. The VCP would first initialise a packet communication on a channel on one link. After that, in the previous multi-channel method, the VCP must wait until that packet had been output before it could initialise output of a packet on a different channel mapped onto the same link. To avoid this VCP idle time, the new method instead initialises the transmission of a packet on a channel mapped onto a different link. The VCP cycles round the links outputting packets on each in turn and returns to the first link hopefully before it has finished transmitting its packet. The VCP then outputs a packet on the first link again. This method again tries to saturate the bandwidth of each link but uses multiple links to increase the output rate of the source processor. Full utilisation of each link is only attained if the time taken by the

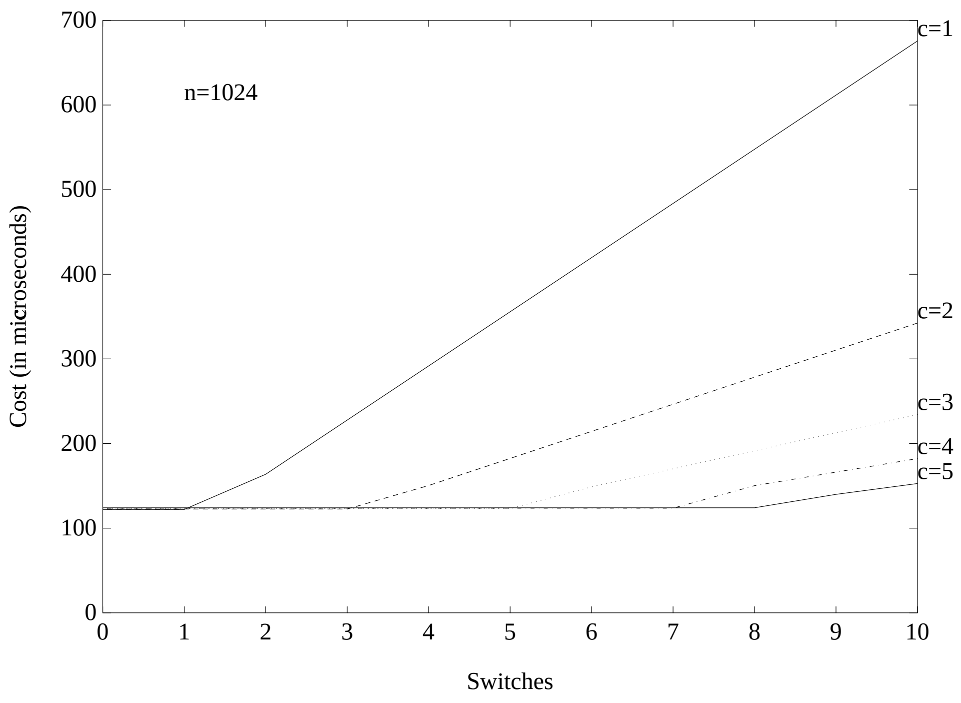


Figure 2.5: Cost of multi-channel communication against number of switches

VCP to service all four links is less than or equal to the time taken by a link engine to transmit a packet:

$$4\beta \leq (b + h + 1)\alpha$$

With the estimated values of Section 2.4 this is true and so we expect the VCP to be able to sustain communication on all the links at their full bandwidth simultaneously. This reduces the number of packets output on each link by a factor l, giving a cost of:

$$T_{ml} = \begin{cases} lc\gamma + \frac{n}{lcb}(2\beta + (2h+1)\alpha + 2s\delta) & \text{if } c < \frac{2\beta+2s\delta+(2h+1)\alpha}{\beta+(h+b+1)\alpha} \\ \qquad +(c-1)(\beta + (h+b+1)\alpha) & \\ lc\gamma + \frac{n}{lb}(\beta + (h+b+1)\alpha) & \text{otherwise} \end{cases}$$

A comparison of the predicted costs of the three methods of communicating a message is shown in Figure 2.6. This graph shows that the use of multi-channel (and even better multi-link) message communication routines significantly decreases the cost of any communication across more than one switch chip.

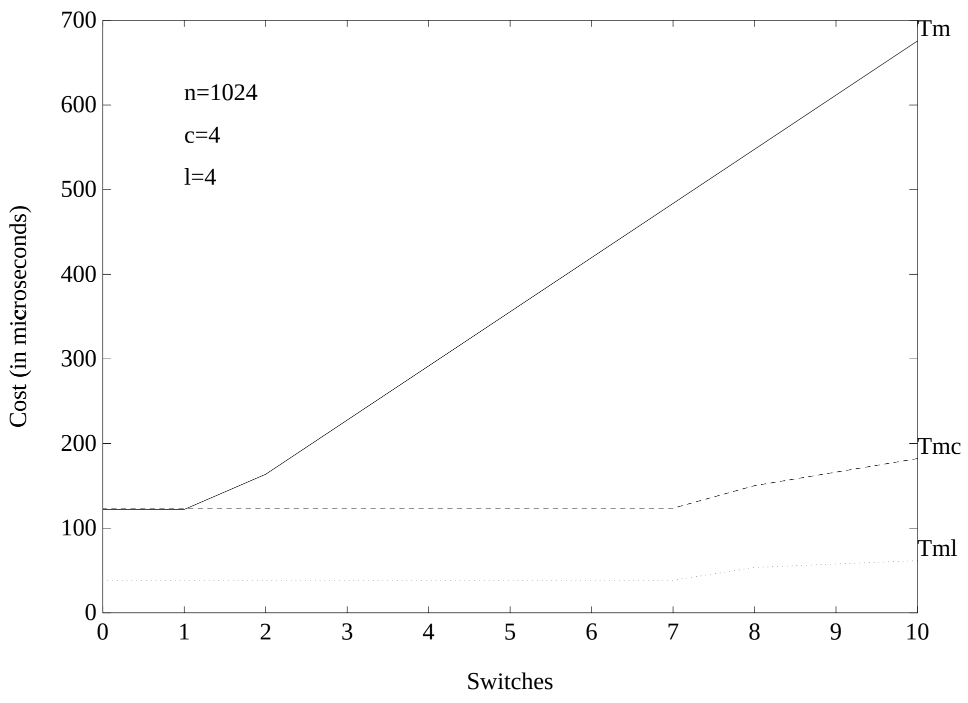


Figure 2.6: Comparison of the costs of transmitting a message

2.7 A Simple Model for Communications

The communications models presented above are not suitable for use in our algorithm models because they are far too complicated for our desired aim of ease of model development. This complexity is introduced to present an accurate model of the true costs of communication. However, it is unlikely that the degree of detail involved will give the expected accuracy in practice. There are many other factors which have not been included in the models which may be significant. These include the overheads of program code execution, inefficiency of the implementation language, transmission of partially full packets, memory bandwidth, caching, pipelining and parallel execution of the processor units. As far as the switch network is concerned two important unknowns which will affect costs are the actual number of switch chips traversed and the presence of hot-spots in the network causing collisions. On top of all of these concerns there is the problem of measuring idle time as the two communicating processes synchronise initially. This idle time may be due to the presence of other processes on the communicating processors or to imbalance in the workload of the two processes. For all of these reasons we need a much simpler model which will predict the communication cost adequately enough for us to be able to make decisions about the general performance of different algorithms on a T9000/C104 machine.

Let us consider first a simple model for a single channel communication. As mentioned in Section 2.3 the simplest model $T = kn$ does not match the true cost well enough, since it ignores the significant start up time, γ, as the channel is initialised. The model $T = k_1 + k_2 n$ includes this initialisation time but does not take account of the variation in cost with s. This variation is very significant (refer to Figure 2.3) and dominates the cost when a large number of switch chips are traversed. The extra cost incurred by switch chips scales with the message size since a delay is introduced for each packet, not just for the whole message. Hence the delay is equally significant for both small and large messages.

One way to model this delay is to introduce a further term in the cost expression involving n and s:

$$T = k_1 + k_2 n + k_3 s n \tag{2.5}$$

Unfortunately, the resulting cost expression is becoming as complicated as the models presented above. Also, we will not know the value of s to be used; even for a specific machine with a known number of switch chips in a known configuration each individual channel communication will cross different numbers of switch chips depending on the locations of the processors involved and whether random routing is enabled or disabled. It has been suggested that the expression $s = \log p$ could be used as an average, assuming the switch network was configured as a hypercube. This is not satisfactory as the dimension of the hypercube, if a hypercube configuration is indeed used, will vary depending on the number of switch chips, and this is unlikely to be the same as the number of processors in the machine for reasons of economy. Alternative configurations of switch chips would give different relationships between s and p: a grid configuration would require $s \propto \sqrt{p}$; and a linear chain: $s \propto p$. A further complication in evaluating a value for s occurs if an algorithm uses only one partition of a large multi-user machine: in this case the number of switch chips is more likely to depend upon the total number of processors in the entire machine rather than the number being used for this particular computation. A reasonable solution seems to be to use Equation 2.5 as a model and expect a suitable expression for s to be specified by the user which reflects the size and configuration of his machine. This expression for s must give an approximation to the average number of switch chips in the path of a general message, for example half the worst case number. This model does not satisfy one of our main aims, that the cost model be independent of the switch network configuration, but this seems unavoidable.

A simple cost model for the multi-channel or multi-link communication methods is much easier to specify since this dependence on s is removed provided that the link is saturated. In this situation we can use the model $T = k_1 + k_2 n$ without losing any accuracy. The relationship between the multi-channel cost model and our simple model is given by:

$$\begin{aligned} k_1 &\equiv c\gamma \\ k_2 &\equiv \frac{\beta + (h + b + 1)\alpha}{b} \end{aligned}$$

In practice values for k_1 and k_2 would be estimated by running test communications programs on a real T9000/C104 machine and fitting the model to the results obtained. This attempts to take account of the cost of program control flow and other features not included in the models.

We are now in the difficult situation where we propose two different cost models for point to point communications depending upon the implementation of that communication. The cost model for a numerical algorithm may contain terms due to both types of communication and will only be correct if each individual communication in the algorithm implementation has been performed using the method expected by the model. For numerical libraries this may not be too much of a problem; for many algorithms most communications are of vectors which may be large. In these situations multi-channel communications would be used throughout and the associated cost model would be independent of s and therefore applicable to machines with any switch network configuration. On the other hand many algorithms involve the communication of small amounts of data. These would be implemented using single channels for communication and hence the algorithmic cost would depend on the switch network.

To summarise this discussion of communication models for the T9000/C104 architecture, we proposed to model single channel communication by:

$$T = k_1 + k_2 n + k_3 s n.$$

This expression is dependent on the configuration of the switch network whilst the algorithm itself is not. It has been shown that utilising the full bandwidth of the links is not possible with only a single channel on a link when multiple switch chips are traversed by a message. To overcome this inefficiency, we suggest that a communications library is developed which includes a primitive routine to implement multi-channel communications for a single message. We model the cost of such a routine:

$$T = k_4 + k_5 n$$

This cost model is independent of the switch network. This method of communication may be used to transmit longer messages such as the vectors and matrices in numerical algorithms. A further set of library routines implementing higher level communications operations such as broadcast and scatter could use this primitive to improve their performance.

In Chapter 1 it was stated that a multistage network gave the best cost/performance ratio of any network design. We consider it very likely that production T9000/C104 systems will indeed be based on this type of network. In particular, the example of the three-stage folded Clos-type network (Figure 1.5) seems ideal since it permits network sizes up to 512 processors which is the probable upper bound on network size over the next few years. A three-stage network can also support almost 1000 processors (precisely 992) though with reduced bandwidth. Hence we think it very likely that production systems will be three-stage networks which means that in the worst case there will be only three switch chips along the path of a message. In addition, we consider it unlikely that the necessary effort will be invested in the development of multi-channel communication routines.

Taking these expectations into account, we have chosen to use the first single-channel communication cost model (Equation 2.3) with a value for s of 3. Hence, for the T9000/C104 architecture we have a communications model consisting of two constant parameters T_s and T_c. When T9000/C104 machines are available these parameters will be estimated by running test communications and fitting the timings to the model. In the absence of real machines we make estimates for the values using Equation 2.3. This equation gives estimated values of $T_s = 1\mu$s and $T_c = 888$ns for a double precision value, which is equivalent to a sustained bandwidth of about 4.5Mbyte/s.

This work complements the work presented in [51, Section 6] which describes the costs of link communication in terms of the number of cycles required by the various processing units to perform a communication. That work also shows the effect on performance of partially full packets. It is comforting to note that these two different approaches broadly agree on the costs predicted for channel communication on a T9000/C104 machine. Gee [48] presents an alternative communication cost model which he tests on the Virtual Channel Router (VCR) [31]. The VCR is a software implementation of the T9000 communications model which runs on a network of T8 processors.

2.8 Summary

The fundamental modelling parameters and expressions for computation and communication of double precision values for transputer systems are as follows:

For T8 networks:

computation model	$T = nT_f$	where	$T_f = 1.62\mu$s
communication model	$T = nT_c$	where	$T_c = 8.80\mu$s

For T9000 networks:

computation model	$T = nT_f$	where	$T_f = 100$ns
communication model	$T = T_s + nT_c$	where	$T_s = 1\mu$s and $T_c = 888$ns

In the following sections these simple expressions are used to develop cost models for complete algorithms. Examples of cost models using these expressions may also be found in several working papers [3, 23, 61, 74, 62].

Chapter 3

Gaussian elimination

This chapter describes some of our earliest work on transputer systems, namely, to design a parallel algorithm to solve a set of simultaneous linear equations using Gaussian elimination [74]. The original program was developed and tested on the RSRE Protonode in early 1988. This prototype machine consisted of sixteen T4 processors whose connectivity could be controlled electronically. In its time this machine was quite advanced and was the forerunner of the successful Supernode machines. In this chapter we describe the algorithm developed and present updated experimental results obtained using the Parsys Supernode, a member of the current generation of T8 based Supernode machines. The chapter provides an introduction to many of the design techniques that apply to local memory MIMD algorithms and algorithm cost modelling. These techniques are then used and expanded throughout the rest of this work.

3.1 Introduction

Almost every area of scientific programming involves the use of linear algebra calculations. These matrix operations frequently govern the overall performance of numerical applications. Such widespread use has led to this field of numerical computing receiving more attention than any other field. Linear algebra computations may be divided into two fields depending on the structure of the underlying data: dense linear algebra and sparse linear algebra. Dense linear algebra covers all use of matrices and vectors which are mainly filled with non-zero values. If matrices or vectors contain only a few non-zero values then sparse linear algebra algorithms are used to manipulate these sparse arrays.

One of the dominant operations in dense linear algebra is the solution of a system of linear equations. There are several methods to solve this problem but the most common method is Gaussian elimination. Gallivan et al. [47] gives an excellent overview of parallel algorithms for dense linear algebra including parallel Gaussian elimination algorithms. Much work has been done on the design of parallel linear equation solvers. Chu and George [21] describe a parallel Gaussian elimination algorithm that distributes rows of the matrix to processors. Howard [62] describes an algorithm which distributes the matrix by columns. Two other interesting papers [16, 93] discuss the performance of parallel linear equation solvers on very large systems. In this chapter we describe the implementation of a row-distributed parallel Gaussian elimination routine on a chain of processors.

3.2 Algorithm

The solution of a dense system of linear equations can be expressed as

$$A\mathbf{x} = \mathbf{b}$$

where a_{ij} is the coefficient of the jth unknown in equation i, b_i is the right hand side value for equation i, and x_i is the solution value of the ith unknown variable.

For a real, dense matrix A, the best method to solve the system of linear equations is Gaussian elimination with partial pivoting. This method has the lowest cost and partial pivoting maintains good accuracy in A by minimising the accumulation of rounding errors.

The calculation of $\mathbf{x}$ can be split into two separate operations:

1. reduction of the matrix A to upper triangular form,
2. forwards elimination and backwards substitution of $\mathbf{b}$ the right hand side (RHS) vector.

Reduction of the matrix has a cost of order n^3, where n is the dimension of A, whilst the calculation of the new vector $\mathbf{x}$ has a cost of order n^2.

The kij forward looking version of the Gaussian elimination algorithm with partial pivoting may be expressed as follows:

Algorithm 1 (Gaussian elimination)

```
1. matrix reduction
   for $i = 1, \ldots, n-1$
    1.1. find $p_i$ which minimises $|a_{p_i i}|$, where $i \leq p_i \leq n$
    1.2. if $p_i \neq i$ then for $k = i, \ldots, n$ perform $a_{p_i k} \leftrightarrow a_{ik}$
    1.3. for $j = i+1, \ldots, n$
       1.3.1. set $a_{ji} = a_{ji}/a_{ii}$
       1.3.2. for $k = i+1, \ldots, n$ set $a_{jk} = a_{jk} - a_{ji}a_{ik}$
2. RHS forwards elimination
   for $i = 1, \ldots, n-1$
```

```
    2.1. if $p_i \neq i$ then perform $b_{p_i} \leftrightarrow b_i$
    2.2. for $j = i+1, \ldots, n$ set $b_j = b_j - a_{ji} b_i$
3. RHS backwards substitution
    3.1. set $x_n = b_n / a_{nn}$
    3.2. for $i = n-1, \ldots, 1$ set $x_i = \left[ a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij} x_j \right] / a_{ii}$
```

□

More details of the sequential algorithm can be found in any linear algebra textbook, for example [52].

3.3 Parallel implementation

The parallel Gaussian elimination algorithm was developed as part of the Supernode I Liverpool Parallel Library. Compatibility with this mark of the Library dictates a number of algorithm design decisions. Firstly, the Library assumes that the processor network is configured in a chain (see Section 1.7). Secondly, parallel Library routines are called from a sequential master program executing on a single master processor. This second design decision requires that at the start of a parallel algorithm initial data must be scattered from the master to the slave processors, and at the end of the algorithm the master must gather back the result data from the slave processors. (See Chapter 7 for a discussion of the issues involved in the design of a parallel library.)

The Library specification does not place any restrictions on the data distributions used in the algorithm. The RHS vector $\mathbf{b}$ will be distributed between the processors. We wish to use a distribution for the matrix A which minimises the amount of mid-algorithm communication. First let us consider a row distribution for A. For each step i in the matrix reduction phase of the algorithm, all the processors must communicate to find the best pivot row for this iteration. Once chosen, this pivot row must be broadcast to all the processors (and swapped with row i if necessary). The processors can then update their rows of the matrix independently. The RHS forwards elimination requires the pivot value b_i to be broadcast at each step i. For the backwards substitution phase, at each iteration i the value x_i is broadcast to all the processors which then update their partial sums $\sum_{j=i+1}^{n} a_{ij} x_j$ in parallel. The communication requirements are different for a distribution of A by columns. In the matrix reduction phase, in iteration i, one processor chooses the best pivot, calculates the pivot factors $a_{ij} \quad j > i$ and broadcasts this column of pivot factors to all the other processors. This avoids the communication of single values to chose the best pivot by performing the whole calculation on one processor. The column broadcast costs the same amount as the row broadcast in the row distributed case, but does avoid any extra communication to swap the best pivot row and row i. For the RHS operations, the forwards elimination requires the column

$a_{.i}$ to be scattered at iteration i to allow the processors to perform pivoting on their elements of b. This is a greater communication cost than for the row distribution of A. In the back substitution phase the algorithm is essentially sequentialised: at each iteration i only one new value x_i is calculated and this only contributes to the summation $\sum_{j=i+1}^{n} a_{ij}x_j$ on the processor which owns column j of A unless at each iteration the column of A is re-distributed. On balance, the communication costs for the two distribution schemes do not differ by very much. As a routine to perform Gaussian elimination based upon a column distribution of A was already available [62], for comparison, this implementation distributes A by rows.

3.3.1 Matrix reduction

The algorithm starts by scattering the rows of the matrix A from the master to the slave processors. Each processor does not receive a block of consecutive rows but instead processor $p-i+1$ receives rows i, $i+p$, $i+2p$, etc.. This is called a cyclic distribution with wrapping. This method of distribution helps to keep an even workload between the processors throughout execution. To make this operation as efficient as possible, whilst one row is being output to the right by a slave, it will also be receiving the next row from the left. In this way the total time for a row to be output and the next row to be input is only slightly greater than that required simply to output the row. All the rows for the rightmost slave are sent first, so that whilst rows are still being sent to other slaves, those on the right can begin their calculations.

The parallel matrix reduction consists of a main loop $i = 1, \ldots, n-1$, within which the best pivot element for column i is chosen and pivoting performed.

Choosing the best pivot element is divided into two parts. Firstly, each processor chooses its element with largest magnitude in column i as a pivot. Then, the rightmost processor p passes its pivot choice to processor $p-1$. Processor $p-1$ compares this pivot value with its own choice for the pivot and passes the pivot with the largest magnitude on up the chain to processor $p-2$. This continues until processor 1 chooses the best pivot value, which is then broadcast back down the chain. The pivot row numbers chosen at each step are recorded by each processor for subsequent use in the formation of the result vector.

Secondly, the owner of the best pivot row broadcasts the slice of that row from element i to n. The broadcast is performed as follows: the source processor outputs the vector to left and right if there are any processors on each side. The other processors input the vector and pass it on along the chain if there are other processors beyond them. If a row swap is required, when the processor owning row i receives the pivot row it passes its row i from element i to n back to the processor with the best pivot row for this iteration. The rows are exchanged instead of being renumbered to try to maintain a good workload balance.

The pivoting then proceeds with each processor updating its rows of the matrix according to Step 1.3.2 of the sequential algorithm.

When this main loop is complete each processor in the network has a sub-set of rows of the reduced matrix and a vector of the pivot rows chosen at each pivoting step.

3.3.2 RHS computation

At the start of the second phase of Gaussian elimination the RHS vector, $\mathbf{b}$, is scattered to the processors. As for the first phase, the distribution of vector elements is such that processor $p - i + 1$ receives elements i, $i + p$, $i + 2p$,etc.. This distribution ensures that all the data required to perform pivoting on a slave's RHS elements (with the exception of the pivot element for each step) is already available on that slave.

The calculation of the result vector is split into two phases: forwards elimination (Step 2 of the sequential algorithm) and backwards substitution (Step 3).

The first phase is performed in a manner very similar to that employed for the matrix reduction.

We have a loop in i for $i = 1, \ldots, n - 1$. The owner of the ith element of $\mathbf{b}$ broadcasts b_i to all the processors. If the pivot row chosen in the matrix reduction was not also the ith row then the corresponding elements of the RHS vector, $\mathbf{b}$, are also exchanged. This may require communication between processors to accomplish. The processors then perform the pivoting of Step 2.2 of the sequential algorithm on their RHS elements, utilising the factors, a_{ji} calculated in the matrix reduction phase.

Once the pivoting is completed, the new, equivalent system of linear equations is solved by backwards substitution. The result vector $\mathbf{x}$ which is calculated is distributed in the same way as the RHS vector $\mathbf{b}$.

Initially, the last element of the result vector, $\mathbf{x}$, is calculated: $x_n = b_n/a_{nn}$. This value is then broadcast to all the other processors.

A loop in i, with $i = n - 1, \ldots 1$, is then performed to calculate the remaining elements of the result vector (Step 3.2 of the sequential algorithm). At the start of each iteration of this loop each processor holds the result vector element x_{i+1} and a partial sum:

$$\text{tot}_r = \sum_{j=i+2}^{n} a_{rj} x_j$$

for each row r of the matrix that it holds. Each of the partial sums is updated:

$$\text{tot}_r = \text{tot}_r + a_{r,i+1} x_{i+1} \quad \forall r < i + 1.$$

The processor which owns the element x_i then calculates x_i:

$$x_i = (b_i - \text{tot}_i)/a_{ii}$$

This value is then broadcast to all the other processors and the next iteration of the loop starts.

After completing the loop each slave processor passes back to the master processor its elements of the result vector $\mathbf{x}$.

3.4 Cost models

We now present run-time cost models for the matrix reduction and RHS operations. These models are expressed in terms of the matrix size, n, the number of slave processors, p, and the hardware parameters T_f and T_c:

T_f	the time taken to perform a double precision arithmetic operation, $T_f = 1.62\mu$s;
T_c	the total time taken to communicate a message of n double precision values is given by nT_c, $T_c = 8.80\mu$s.

Chapter 2 gives full details about the hardware parameters and their values for both T8 and T9000/C104 machines.

3.4.1 Matrix reduction

The total cost of the sequential matrix reduction is:

$$\left(\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6}\right) T_f.$$

This is the cost of the arithmetic operations in Step 1.3.2 of the sequential algorithm. In the parallel algorithm this workload is distributed over the processors since they all have rows of the matrix. The cyclic distribution of the rows helps to ensure that even towards the end of the parallel algorithm when fewer rows of A are being updated the workload is still reasonably distributed between the processors. With this assumption of a well balanced workload we can say that the total time taken by a processor to update its rows is approximately:

$$\left(\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6}\right) \frac{T_f}{p}.$$

The communication costs for the parallel algorithm are more difficult to estimate. Assuming that two parallel communications on different transputer links can both execute at full speed, the time taken to scatter the matrix from the master will be n^2/T_c. The main loop in i involves communication to find the best pivot element and to broadcast the pivot row and swap that row with the ith row. In each iteration the cost to find and broadcast the best pivot element is $2(p-1)T_c$. This is summed to give the total cost to the algorithm for this operation: $\sum_{i=1}^{n-1} 2(p-1)/T_c = 2(p-1)(n-1)T_c$.

The cost of the pivot row broadcast and row swap will vary depending on which processors own the pivot row (processor i) and the best pivot row (processor p_i) at each iteration. Assuming that a row swap is required on every iteration, which is most likely, the number of consecutive interprocessor communications required to broadcast the pivot row and perform the row swap is $\max(p-p_i, p_i-1, 2|i-p_i|)$, where $p-p_i$ and p_i-1 are the number of communications required to broadcast the best pivot row to either end of the chain, and $2|i-p_i|$ is the number of communications for processor i to receive the best pivot row from processor p_i and send back the ith row. Since p_i is unknown we want to find the average value of this function over all i and p_i, but we have found no analytical solution to this function. Instead we evaluated the function over the complete range of possible values for i and p_i for several values of p. These results give $\frac{9}{10}p$ as an approximate value for the average number of communications required to broadcast the best pivot row and perform the swap. The total cost to perform the row broadcasts and swaps for the whole algorithm is then given by $\sum_{i=1}^{n-1}(n-i+1)\frac{9}{10}pT_c = (\frac{n^2}{2}+\frac{n}{2}-1)\frac{9}{10}pT_c$.

The total cost for the matrix reduction is thus given by:

$$\left(\frac{2n^3}{3}-\frac{n^2}{2}-\frac{n}{6}\right)\frac{T_f}{p}+n^2T_c+2(p-1)(n-1)T_c+(\frac{n^2}{2}+\frac{n}{2}-1)\frac{9}{10}pT_c,$$

or

$$\left(\frac{2n^3}{3}-\frac{n^2}{2}-\frac{n}{6}\right)\frac{T_f}{p}+\left(\frac{9}{20}n^2p+n^2+\frac{49}{20}np-2n-\frac{29}{10}p+2\right)T_c.$$

3.4.2 RHS computation

The number of operations for the sequential forwards elimination is $n(n-1)$, and for the backwards substitution n^2. This gives a total cost for the sequential RHS computations of

$$n(2n-1)T_f.$$

The cost model for the parallel RHS forwards elimination and backwards substitution follows the same principles as the parallel matrix reduction cost model.

The cost to initially scatter the RHS vector, $\mathbf{b}$, is nT_c. The forwards elimination phase which follows uses almost the same algorithm as the matrix reduction except that only single values are communicated and updated instead of vectors. Hence the total cost to broadcast the best pivot element of $\mathbf{b}$, b_{p_i} and swap b_i and b_{p_i} in all iterations is $(n-1)\frac{9}{10}pT_c$. The cost to perform the update is given by $n(n-1)\frac{T_f}{p}$.

For the backwards substitution we need to model the cost to calculate and broadcast the elements of the result vector x_i and perform the partial sum updates. The number of consecutive communications required to broadcast an element of the result vector depends on the source processor and varies in value from $p/2$ when the source processor is at the centre of the chain, to $p-1$ when the source processor is at one end of the chain. If we again take the average number of communications, which is $\frac{3p-2}{4}$, and sum over all the iterations we have a total communication cost for broadcasting the elements of $\mathbf{x}$ of $\sum_{i=n-1}^{1}\frac{3p-2}{4}T_c = (n-1)\frac{3p-2}{4}T_c$.

Assuming the workload is well balanced, the partial sum updates cost $\sum_{i=n-1}^{1}2(n-i)\frac{T_f}{p} = n(n-1)\frac{T_f}{p}$. The calculation of each new element x_i from the partial sums costs an additional $2T_f$. This gives a total cost for calculating elements of $\mathbf{x}$ of $(2n-1)T_f$.

The result vector is then returned to the master at a cost of nT_c.

Thus, the total cost for the parallel RHS operations is given by:

$$2nT_c + (n-1)\tfrac{9}{10}pT_c + n(n-1)\tfrac{T_f}{p} + (n-1)\tfrac{3p-2}{4}T_c + n(n-1)\tfrac{T_f}{p} + (2n-1)T_f,$$

or

$$\left(\tfrac{2n^2}{p} - \tfrac{2n}{p} + 2n - 1\right)T_f + \left(\tfrac{33}{20}np + \tfrac{3}{2}n - \tfrac{33}{20}p + \tfrac{1}{2}\right)T_c.$$

3.5 Results

We implemented both the parallel algorithm and an efficient sequential algorithm. The performance of these programs was then measured for a range of problem sizes and number of processors on the Parsys Supernode (see Chapter 1). We used problem sizes of $n = 50, 100, 200, 400$ and 600. These cover the range of problem sizes that can be stored on a master processor with 4MB of memory. The parallel program was timed for these problem sizes using $p = 1, 2, 4, 8, 16, 32$ and 48 slave processors. For the sequential program we measured the time taken to reduce the matrix and the time taken to perform the RHS computations. For the parallel program we measured the time taken to scatter the matrix, the time required to reduce the distributed matrix, and the time taken to perform the RHS computations including the vector scatter from and gather to the master processor. The run-times were measured using the built-in timer on the master processor and averaged over five program runs. The variation in run-time between program runs for the same problem is very small. For example, for $n = 400$ the variation in total time was under ± 0.005s (under 0.1%), and for $n = 50$ the variation was ± 0.001s (under 1%). The timings are presented in Appendix C.

To test the accuracy of the cost models at predicting the run-time of the parallel program we evaluated the cost model functions for each problem size and number of processors used in the experimental measurements.

3.5.1 Matrix reduction

T8 architecture

Looking at the sequential matrix reduction costs, we see that the model underestimates the total cost by between around 20% for small problems and around 10% for larger problems. This error occurs because the actual algorithm tested has a greater overhead for each arithmetic operation performed than the simple test program used to evaluate T_f. This results in lower cost predictions especially for small problems. A least-squares fit of the model to the measured values suggests a value of $T_f = 1.76\mu$s instead of $T_f = 1.62\mu$s, an increase in T_f of 8%. An error of 10% for the large problems is within reasonable bounds for using the model as a predictor of program cost.

The parallel matrix reduction model exhibits errors of similar magnitude to the sequential model. For small problems ($n = 50$) the error ranges from 22% for $p = 1$ to 28% for $p = 48$, and for large problems ($n = 600$) the error ranges from 16% for $p = 1$

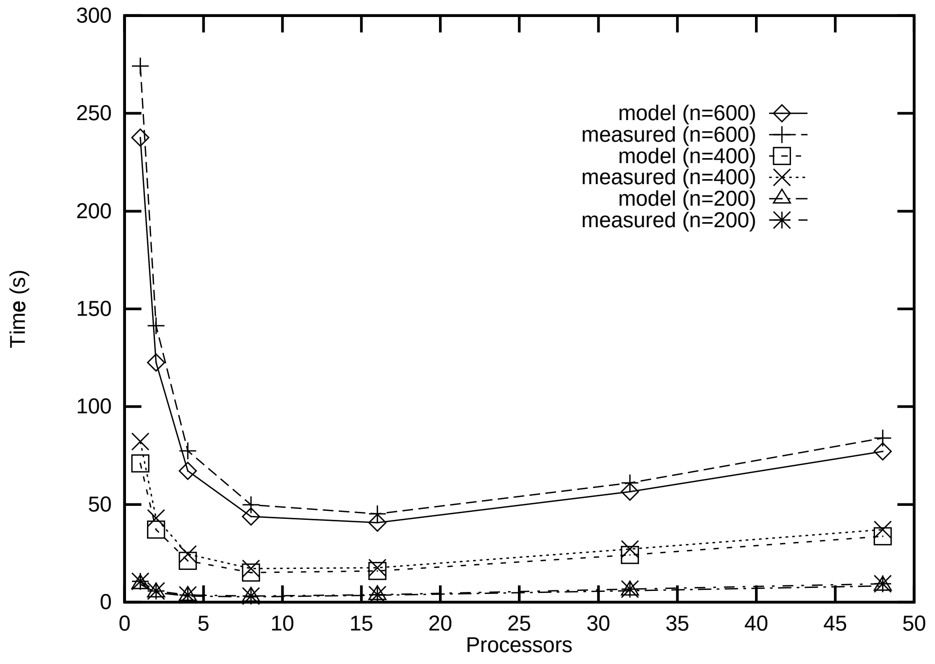


Figure 3.1: Matrix reduction model costs

to 9% for $p = 48$. As for the sequential case we would expect the error in the smaller problems to be larger due to the increased overheads compared to useful computation performed. Figure 3.1 compares the predicted model cost with the measured cost for the larger problem sizes $n = 200, 400$ and 600. This shows the model to give a good estimate of the actual run-time for the matrix reduction. The larger errors for smaller p is due to the underestimate for T_f which affects the cost for small p much more than the cost for large p. This is because for small p the term in T_f is much more significant than for large p where the communication cost adds significantly to the total run-time. If we perform a least-squares fit of the parallel model to the measured timings, using the fitted value for T_f, to find a better value for T_c we get $T_c = 9.79\mu$s instead of $T_c = 8.80\mu$s, an increase in T_c of 11%. Figure 3.2 shows the predicted and measured times for $n = 200, 400$ and 600 using these fitted hardware parameters. In general we cannot use fitted parameter values for our cost models, since we will be using the cost models to predict the performance of algorithms that have not yet been implemented and therefore we cannot have fitted parameter values for those algorithms. However, if we are using a cost model to predict the performance of an existing algorithm for larger problems on larger machines then we may use fitted parameter values obtained from measurements for small problems on small machines. The advantage of fitted parameters over general parameter estimates is that they more accurately reflect the overhead cost per arithmetic or communication operation for a particular algorithm. But by definition that overhead is specific to one algorithm and a particular implementation and will vary from one algorithm to another, so we recommend using one set of

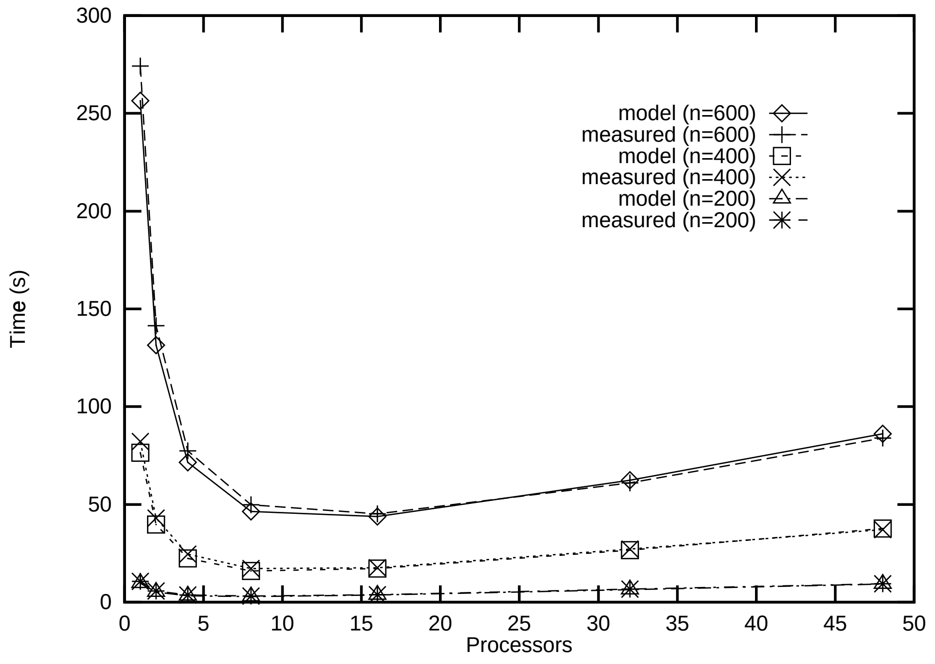


Figure 3.2: Matrix reduction costs with fitted model

general parameter values for all algorithms on a particular architecture instead of different fitted values for each algorithm. So for the T8 architecture we have two general parameters values for T_f and T_c.

So, the parallel matrix reduction model gives good timing predictions for all but the smallest problem sizes, keeping the error below 20%. Compare this with the most naïve model possible which assumes perfect parallelism, i.e., $\left(\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6}\right)\frac{T_f}{p}$. A graph showing the measured costs and costs predicted by this naïve model for a problem size of $n = 600$ is shown in Figure 3.3. This clearly shows that the small amount of effort invested in developing a cost model produces a much better cost predictor than the naïve model. We can also simplify the cost model by dropping terms in only n, p or n/p. This gives us a simplified cost model of

$$\left(\frac{2n^3}{3} - \frac{n^2}{2}\right)\frac{T_f}{p} + \left(\frac{9}{20}n^2p + n^2 + \frac{49}{20}np\right)T_c.$$

This new model predicts almost exactly the same costs as the fuller model.

Having established that the cost model gives a good prediction of the run-time of the parallel matrix reduction let us examine the performance of the algorithm using both the measured timings and timings predicted by the model. Figure 3.4 shows the speedup for the parallel matrix reduction algorithm derived from the measured timings. These values are calculated using the cost of the efficient sequential algorithm as the sequential cost. (See Section 1.7 for details of these derived quantities.) We are most interested in the point at which the parallel algorithm achieves the greatest speedup

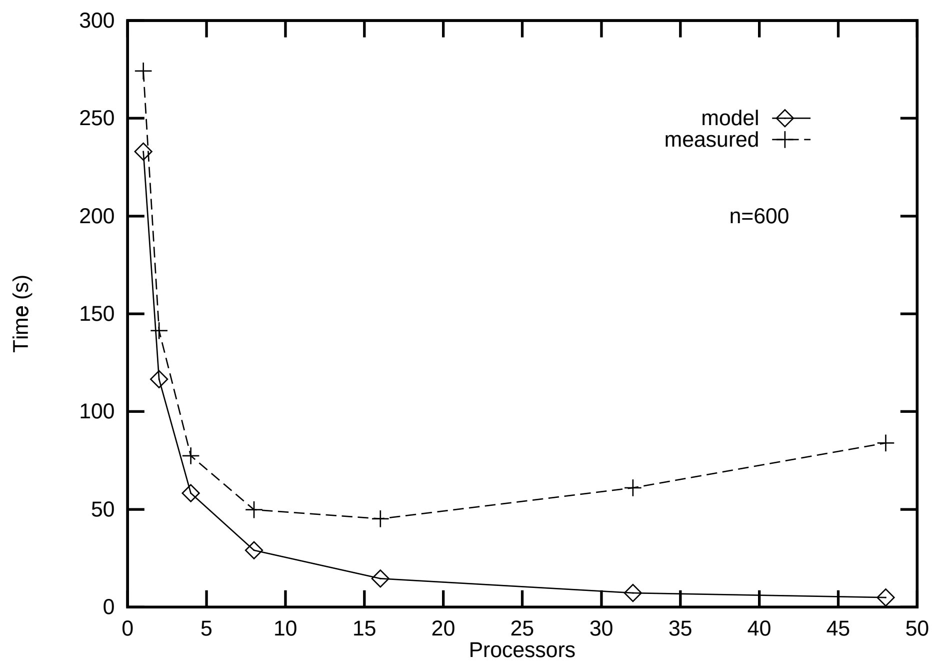


Figure 3.3: Simple model

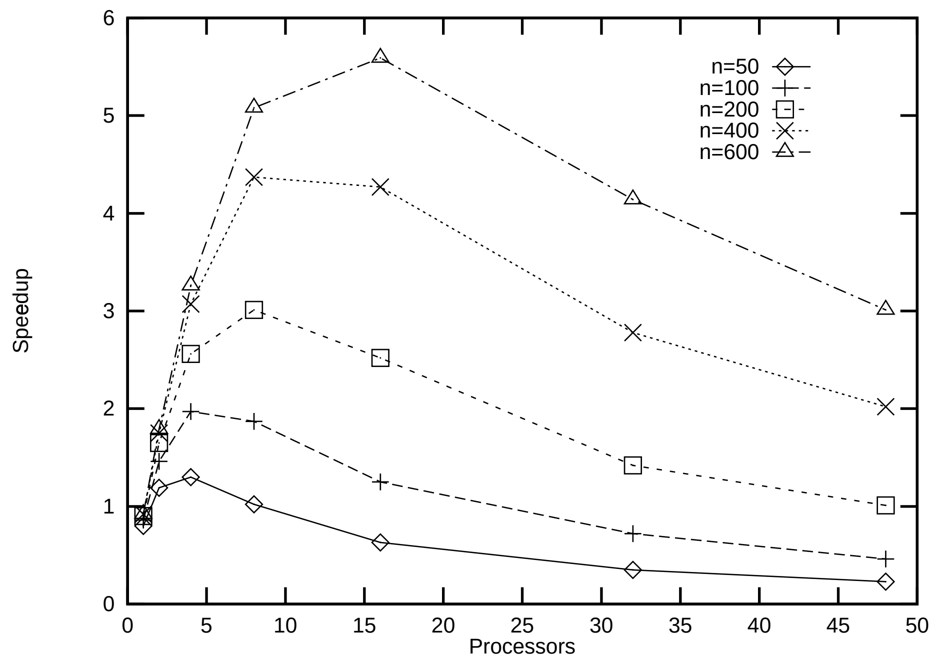


Figure 3.4: Speedup for parallel matrix reduction

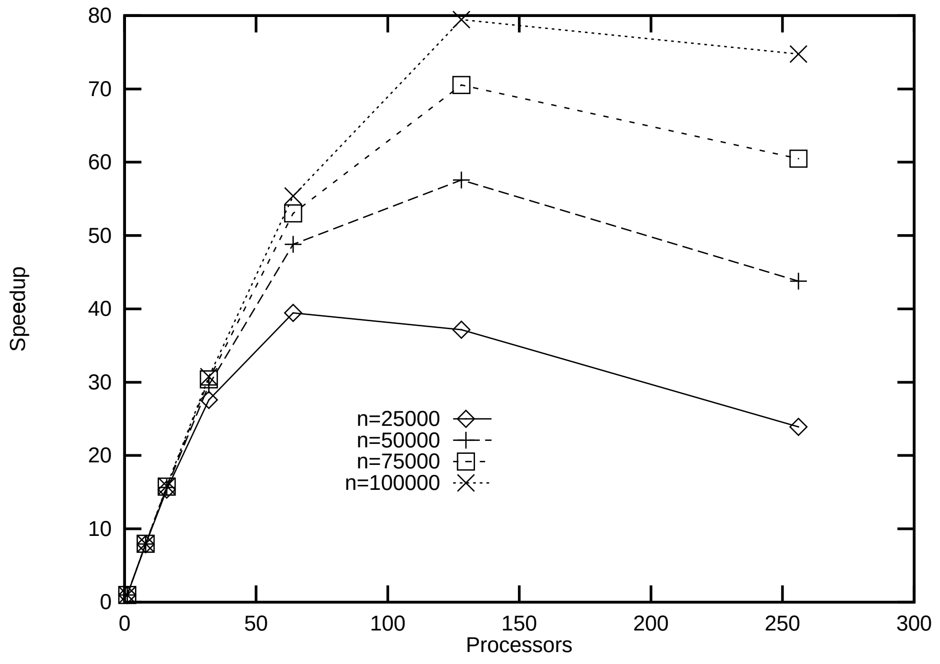


Figure 3.5: Predicted speedup for parallel matrix reduction

over the sequential algorithm. This tells us the number of processors to use to get the shortest run-time for a given problem size. The speedup at this point tells us how much quicker the parallel algorithm will be than the best sequential algorithm. For the problem sizes measured, the best performance is achieved for only small numbers of processors, e.g., between 4 and 16 processors and the speed is at best only 5.6 times quicker than the sequential algorithm. These results indicate that for problems in this size range a significant fraction of the total time is taken up with communication once the number of processors exceeds around 16. The best performance is also only a small fraction of the maximum floating-point performance available from that number of processors.

We can see how larger problems on larger machines will perform using cost predictions given by the cost model. Figure 3.5 shows the predicted speedup for the parallel matrix reduction algorithm. The graph shows that as we increase the problem size we can achieve better speedups and use more processors to get the maximum speedup. For these problem sizes the data will not fit on a single master processor, but will have to be read in from secondary disk storage and scattered over the processors or generated directly by the slave processors. Such problem sizes occur in the aerospace industry where there is a desire to solve dense systems of equations in up to 100,000 variables.

If we compare this algorithm with the algorithm by Howard [62] we see that both algorithms have a very similar performance. That algorithm also uses a chain of processors but distributes the columns of the matrix cyclically. Thus there is not much difference in the communication performance of a row- or column-distributed algorithm on the same chain topology.

However, the algorithm by Bisseling and van de Vorst [17, 16] achieves a much better performance than our algorithm and scales well to much larger network sizes. For example, Bisseling's algorithm achieves a speedup of 219 on 400 transputers for a problem size of 1000. For $n = 600$ and $p = 16$ his algorithm takes 15.2s to factorise the matrix compared with 45.2s for our algorithm. Some of this difference can be explained by the use of assembler BLAS in Bisseling's algorithm whilst our algorithm uses only `occam` code. But the main advantage of his algorithm is that the communication operations in the factorisation phase send vectors of length at most $n/\sqrt{p}$ compared with n for our algorithm. This is achieved by using a grid topology of processors and a double-cyclic distribution of the matrix, i.e., element a_{ij} is placed on processor ($i \bmod \sqrt{p}$, $j \bmod \sqrt{p}$).

The results indicate that our parallel matrix reduction algorithm running on current generation T8 transputers will only achieve small speedups for problem sizes that fit on a single master processor, and the best speedups for such problems will be achieved for small numbers of processors. However, for much larger problems, where the data is pre-distributed or generated by the slave processors, the algorithm can make good use of larger arrays of transputers to achieve higher speedups.

T9000 and T4 architectures

At this point it is interesting to speculate on the performance of this algorithm on different parallel architectures and also on future generations of transputers. We can attempt to do this by using the matrix reduction cost model and substituting in new values for the hardware parameters T_f and T_c that reflect the new architecture. This assumes that the processors in the new architecture are connected in a chain and that computation and communication can be modelled by a single parameter each. It is reasonable enough to assume that a single parameter can be used to model the computation cost. Only vector processing nodes or heavily pipelined RISC architectures may need an extra parameter to model the startup time for a vector arithmetic operation. However, the complexity of communication architectures requires that most architectures will need at least two communication parameters to model the communication cost reasonably well. These parameters would model the data throughput of the network and the message startup or latency time. An example of such a communication model has already been presented in Chapter 2 for the T9000/C104 architecture.

Bearing the preceding comments in mind let us consider a T9000 machine with the processors connected directly together in a chain. Chapter 2 gave a value for $T_f = 0.1\mu$s, and assuming the full link bandwidth is utilised by direct connections we can use $T_c = 0.8\mu$s. Now these times are both about ten times quicker than the times for the T8 architecture, but still in the same ratio to one another. This means that the cost model will give very similar run-time curves except that all timings will be about ten times quicker. But the sequential timings will also be ten times quicker so the speedup of the algorithm on such a T9000 machine will be the same as for the T8 machine.

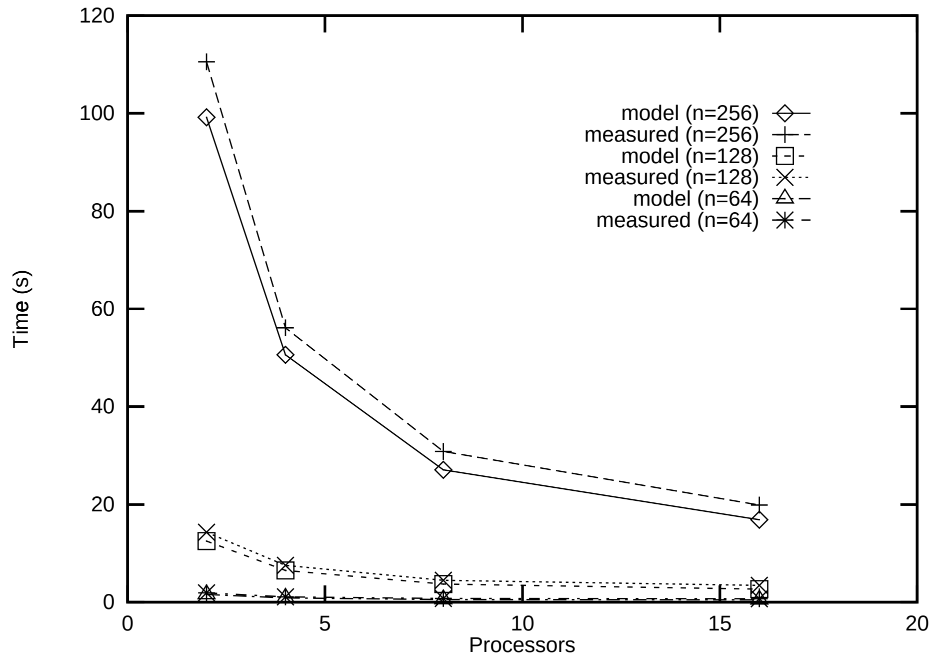


Figure 3.6: T4 matrix reduction

This also means that the algorithm will only be able to effectively use small numbers of processors in this new architecture. In order to get a better performance from this algorithm we must use an architecture with a much lower communication cost relative to the computation cost. This will decrease the relative cost of the communications operations in the algorithm giving better speedups for a given number of processors and will also increase the number of processors which gives the largest speedup for a given problem size allowing effective use to be made of larger networks of processors.

This is demonstrated by the original timings for this algorithm which were made on a T4 machine. When this algorithm was first developed and tested on a 16 processor T4 machine the algorithm worked with single precision floating-point numbers. This means that only half the number of bytes need to be sent for any given communication compared with the double precision version tested here. Also, and more importantly, floating point arithmetic was performed in software by the T4 processor and so the arithmetic cost was very large compared with communication and compared with the hardware floating-point arithmetic of the T8. In fact, the T4 processor has values of $T_f = 17.6\mu$s and $T_c = 8.4\mu$s which gives a ratio of $T_f/T_c = 2.1$. This compares with a ratio of $T_f/T_c = 0.2$ for the T8 processor. Measured and predicted run-times for the algorithm on the T4 machine are given in Appendix C. Figure 3.6 compares the measured run-times with those predicted by the model. The model exhibits the same tendency on the T4 as on the T8 to underestimate the cost, but the accuracy of the model is at least as good as for the T8 machine. The measured speedup of the algorithm is shown in Figure 3.7. This graph shows that even for the much smaller

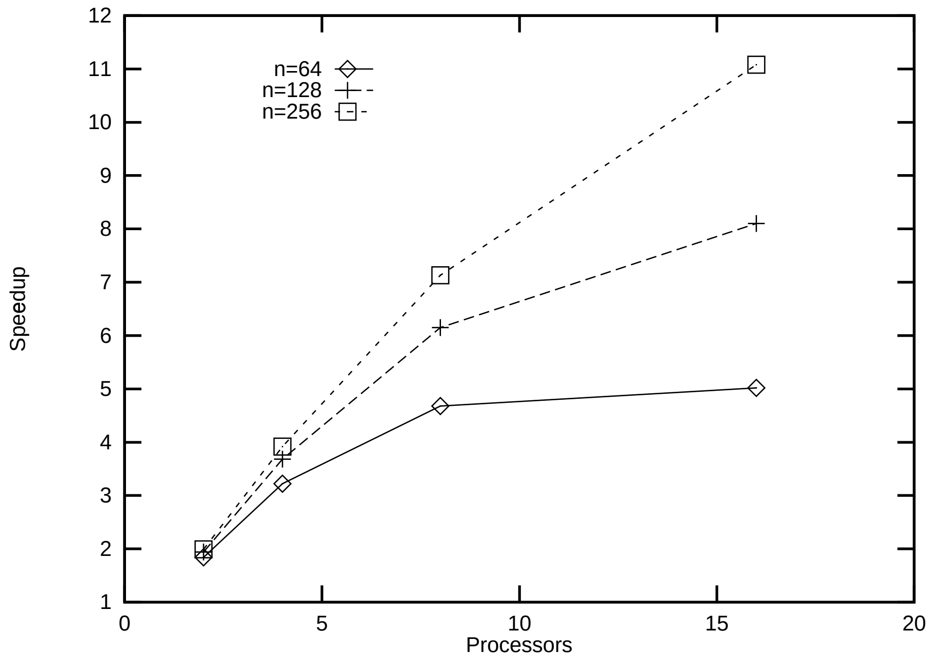


Figure 3.7: T4 matrix reduction speedup

problem sizes tested on the T4 machine better speedups were achieved than for the T8 machine. Full details of the T4 implementation of the algorithm can be found in Oliver [74].

Another way to achieve a lower communication cost for the algorithm is to use a richer communications topology. For example a T9000/C104 machine with a multi-stage switch network will provide much better performance for broadcast and scatter operations than a simple chain of processors. This will reduce the total time spent performing communications in the algorithm and result in better speedups and allow more processors to be used effectively to give the best performance. Chapter 5 includes a cost model for Gaussian elimination on a T9000/C104 machine using communications operations described in Appendix A.

3.5.2 RHS computation

We now turn our attention to the RHS computation phase. As for the sequential matrix reduction algorithm we find that the sequential RHS computation model underestimates the actual costs. In this case the overheads per arithmetic operation are even greater than for the matrix reduction and the error is about 38% for all problem sizes. A least-squares fit of the model to the measured timings gives a fitted value of $T_f = 2.23\mu$s. This is to be compared with the initial approximation of $T_f = 1.62\mu$s and the matrix reduction fitted value of $T_f = 1.76\mu$s. The fitted value does give a very good fit to the measured data indicating that the sequential cost model accurately

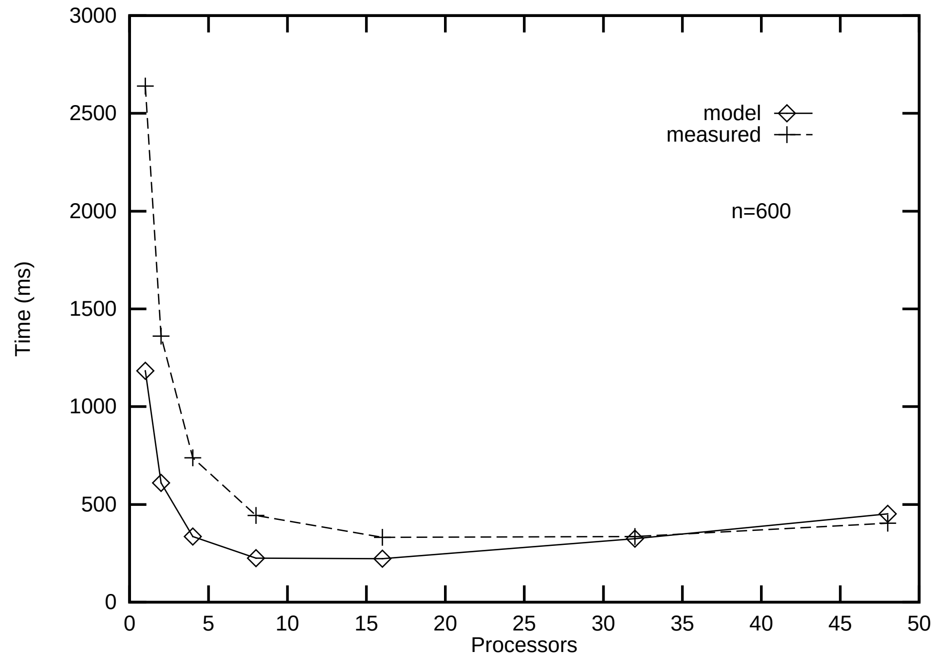


Figure 3.8: RHS computation

models the number of arithmetic operations performed even if the initial estimate of the cost of a single arithmetic operation is not accurate.

However, the parallel RHS algorithm model gives a very poor fit to the measured timings. Individual data points are in error by around 100%, but more significantly the general trend of the cost model does not agree well with the timings. This is shown clearly in Figure 3.8 which plots the modelled and measured costs for a problem size of $n = 600$. For small numbers of processors the model again underestimates the true cost of the algorithm but by an even greater margin than for the sequential RHS algorithm. This suggests that the parallel RHS algorithm has even greater overheads per arithmetic operation than the sequential algorithm. For larger numbers of processors the predicted timings are greater than the measured timings and the trend appears to increase this overestimate for larger numbers of processors giving increasingly larger errors. A least-squares fit of the model to the measured values using the fitted T_f gives $T_c = 10.07\mu$s. However these fitted values do not change the model predictions greatly: for small p the model still underestimates the cost by about 100% and for large p the model overestimates the cost following the same trend as the unfitted model.

A more detailed examination of the practical run-time costs for the parallel RHS computation shows that almost all of the total cost of the algorithm comes from the actual forwards elimination and backwards substitution and not from the scatter and gather of the vector. The model predicts the initial and final communication costs well and so the error in the model is an error in predicting the cost of the forwards elimination and backwards substitution. The modelling techniques used for the RHS computation are the same as for the matrix reduction which predicted the cost very well. The difference between the two algorithms is that the matrix reduction is a coarse-grained algorithm and the RHS computation is a fine-grained algorithm. The matrix reduction performs a lot of computation between synchronisations, although the communications are also of large amounts of data. For the RHS computation a processor only executes a few arithmetic operations before having to synchronise with other processors again for communication, although communications here are only of a few bytes of data. These frequent synchronisations and small amounts of computation are difficult to model using a *global* view of the machine where, for example, we try to model the cost of an algorithmic broadcast by the total time taken by the whole network of processors to perform the broadcast. This means the model time starts from the moment the source processor is ready to start the broadcast operation and finishes only when the last destination processor has received the broadcast data. In practice, during the broadcast operation some processors will receive the data early on and continue executing subsequent computation operations whilst other processors are still waiting for the broadcast data. The execution of the broadcast operation on the processors moves like a wave along the chain of processors away from the source processor. Due to the waveform nature of these communications operations, which are especially noticeable for fine-grained algorithms like the RHS computation, a better cost model might be obtained by taking a *local* view of each processor in the machine and considering the cost of the broadcast as the cost of the broadcast operations on just a single processor. Such a model would give a lower cost for communications operations like a broadcast, and for the RHS computation this would decrease the cost gradient for large p and hopefully give a closer fit to the measured timings for large p.

Figure 3.9 shows the measured speedup achieved by the RHS computation. As for the matrix reduction the speedups are small due to the large fraction of communication operations in the algorithm. The highest speedup is only 4.8 times faster than the sequential algorithm for a problem size of $n = 600$. Notice also that the speedups achieved are lower than the speedups for the matrix reduction algorithm. This is because the RHS computation has a larger fraction of communication operations than the matrix reduction. Another important feature is that for a given problem size n the optimum number of processors to achieve the highest speedup for the RHS operation is different from the optimum number for the matrix reduction. This will be true of most parallel algorithms since the optimum number of processors depends on the fraction of communication in an algorithm. When we solve a problem we wish to use the optimum number of processors for that problem size. But if the optimum number of processors varies between different parallel algorithms that are called in the process of solving the problem how many processors do we use? If the data is always gathered back to the master processor between each parallel algorithm then we can use the different optimum number of processors that each parallel algorithm requires. Unfortunately the overhead in scattering and gathering data to the master processor may be a significant fraction of the total run-time. If instead we leave the data distributed over the slave processors between parallel algorithms then we can either choose to re-distribute data

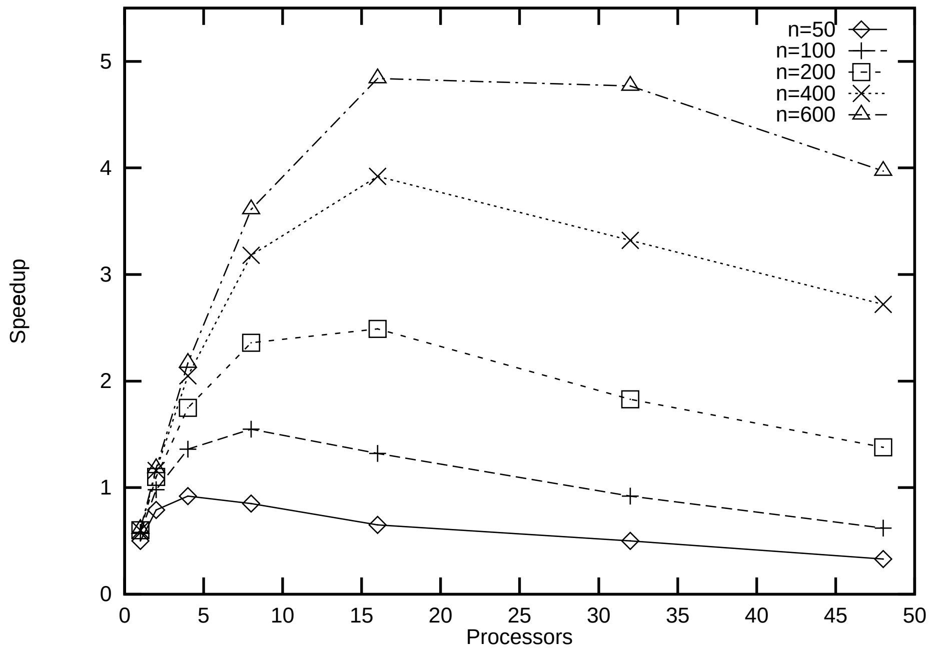


Figure 3.9: Speedup for parallel RHS computation

between algorithms to allow us to use the optimum number of processors or we can use the same number of processors for all the parallel algorithms and leave the data in place. Re-distributing data between algorithms may have a large cost and the improvement in performance from using the optimum number of processors will rarely offset the re-distribution cost. Hence the best approach is to leave the data distributed over a fixed number of processors for all the parallel algorithms. The number of processors chosen should be the number that gives the lowest overall run-time cost. For Gaussian elimination we use the optimum number of processors for the matrix reduction algorithm since the cost of this algorithm dominates the total cost of the algorithm.

If we incorporate the cost model for an algorithm into the implementation, the program itself can select the optimum number of processors to be used for a given problem size. The RHS computation model, even though it does not give a good prediction of the total cost of the algorithm, still gives a good estimate of the optimum number of processors to use.

3.6 Conclusions

This chapter has described the implementation of Gaussian elimination on current generation T8 networks. We have shown the need for complex communication operations to achieve good performance from a chain topology. The advantage of using the richest processor configuration available has been shown by the high performance of Bisseling's grid-based algorithm. Also we have considered the effect of different distribution strategies for the matrix. We have introduced the techniques for developing cost models for parallel algorithms using the parallel Gaussian elimination algorithm as the example. The difficulties in modelling have been introduced along with an indication of the accuracy that can be expected from a model. In particular, we have shown that coarse-grained algorithms (such as the matrix reduction) can be modelled very well, but other fine-grained algorithms (such as the RHS computation) cannot be modelled well using the simple techniques discussed here. We have compared the performance of this algorithm on T4 and T8 machines to show the importance of the balance between computation rate and communication rate.

In the next chapter we describe a completely different type of algorithm: parallel sorting. We will show how the same programming and modelling techniques that have been used for the linear algebra problem in this chapter can be used in this different area of computational mathematics.

Chapter 4

Sorting

In this chapter we look at the suitability of the transputer architecture for parallel sorting algorithms. We present two algorithms which implement bitonic sorting; one designed for the T8 architecture and one for the T9000/C104 architecture. Practical measurements of the performance of the T8 algorithm are given and compared with the cost model. We then use the cost models for the two algorithms to compare the performance of the algorithms for large problems.

4.1 Introduction

Sorting is an important operation in data processing and there has been a lot of activity in the design of parallel sorting algorithms. Most of the earlier work was concerned with the design of algorithms using simple comparators suitable for VLSI implementation. Akl [5] gives a good description of many of the algorithms designed for use in VLSI sorting networks. These algorithms include Batcher's classic odd-even sort and bitonic sort[12]. With the widespread availability of massively parallel computers recent work has focused on designing algorithms for these architectures. Much of this work has been theoretical in nature, with algorithms based on the PRAM shared memory parallel computation model. In practice, most machines do not exhibit the fixed cost communication assumed by the PRAM model, but instead the interprocessor communication has a high cost which affects the performance of algorithms significantly.

Designing sorting algorithms for distributed memory architectures is difficult. One approach is to partition the unsorted data between the processors, perform a sequential sort and then merge the sorted subsets. Loots and Smith[69] describe an implementation of this method for small transputer networks. One difficulty with this basic algorithm is the large cost of the final merge phase. In [97] the merge phase is avoided by initially partitioning the unsorted data into p buckets which are ordered. The sequential partitioning phase now represents a significant proportion of the total algorithm cost. Parallel partitioning algorithms have been suggested, for example [45]. Algorithms which use data partitioning have the disadvantage that the data distribution might not be very well balanced between the processors. This impacts on the memory requirements of each node and the processing time at each node.

Parallel algorithms such as odd-even sorting and bitonic sorting do not have this problem. The main problem with these sorting algorithms is that the sequential cost is much greater than the cost of good sequential algorithms such as quicksort[66]. For example, the cost of bitonic sort is $\frac{1}{4}n\log_2 n(\log_2 n + 1)$ compared with $n\log_2 n$ for quicksort. The advantage of bitonic sort is that it should scale well to large arrays of processors since it is inherently parallel. Bitonic sort algorithms for a mesh of processors are described in [26] and [5]. In this chapter we look at bitonic sort algorithms for a chain of T8 processors and a T9000/C104 machine.

In the next section we describe the generic bitonic sort algorithm for distributed memory MIMD architectures. The two sections following that present details of the design of the algorithm for the specific transputer architectures: Section 4.3 describes the T8 algorithm and Section 4.4 describes the T9000/C104 algorithm. This is followed in Section 4.5 by a discussion of the performance of the two algorithms and our conclusions in Section 4.6.

4.2 Bitonic sort

A bitonic sequence of n $(= 2^k)$ elements $\{x_0, x_1, \ldots, x_{n-1}\}$ is such that

$$x_0 \leq x_1 \leq \ldots \leq x_{j-1} \leq x_j \geq x_{j+1} \geq \ldots \geq x_{n-2} \geq x_{n-1}$$

or the sequence can be shifted cyclically to satisfy this condition.

At the heart of the bitonic sort algorithm is a routine which merges a bitonic sequence to give a sorted sequence. To merge a bitonic sequence of n elements:

Algorithm 2 (Bitonic merge)

```
1. let $i = n$
2. while $i \geq 2$
   2.1. for each element $x_j$ in the sequences of elements

        $x_0 \ldots x_{i/2-1}, \quad x_i \ldots x_{3i/2-1}, \quad \ldots, \quad x_{n-i} \ldots x_{n-i/2-1}$

        compare each element $x_j$ with the element $x_{j+i/2}$;
          for ascending order,   if $x_j > x_{j+i/2}$ then exchange ($x_j$,$x_{j+i/2}$),
          for descending order,  if $x_j < x_{j+i/2}$ then exchange ($x_j$,$x_{j+i/2}$)
```

```
    2.2. let $i = i/2$
```

□

To construct a sorted sequence from an unsorted sequence of length n the bitonic merge procedure is used to produce successively larger sequences of sorted elements:

Algorithm 3 (Bitonic sort)

```
1. let i = 2
2. while $i \leq n$
    2.1. for each bitonic sequence of elements
             $$x_0 \ldots x_{i-1}, \quad x_i \ldots x_{2i-1}, \quad \ldots, \quad x_{n-i} \ldots x_{n-1}$$
         sort the sequence using bitonic merging (Algorithm 2) with alternating sort
         order.
         For ascending sort order, sort first sequence into ascending order and sec-
         ond into descending order, etc., and vice versa for descending sort order.
    2.2. let $i = 2i$
```

□

The adaptation of this algorithm to sort n elements on a distributed memory MIMD machine with p ($= 2^m$, $m \leq k$) processors is as follows:

Initially the unsorted vector is scattered block-wise to the processes, with each process receiving n/p elements. To build up a sorted sequence on each process a sequential quicksort routine is used as it has a better performance than sequential bitonic sorting. Alternate processes perform sorting in ascending and descending order. Now each pair of adjacent processes p_k, p_{k+1} holds a bitonic sequence.

Next the merging stage is entered as followed:

Algorithm 4 (Parallel bitonic sort)

```
1. let $j = 2$
2. while $j \leq p$
    2.1. let $i = j$
    2.2. while $i > 1$
```

```
        2.2.1. for each process $p_k$ in the blocks of processes

                    $p_0 \dots p_{i/2-1}, \quad p_i \dots p_{3i/2-1}, \quad \dots, \quad p_{p-i} \dots p_{p-i/2-1}$

               processes $p_k$ and $p_{k+i/2}$ perform a compare-exchange operation on
               their vector elements:
               for $m = 1$ to $n/p$
                 ascending:    if $p_k(m) > p_{k+i/2}(m)$ then
                                   exchange($p_k(m)$,$p_{k+i/2}(m)$)
                 descending:   if $p_k(m) < p_{k+i/2}(m)$ then
                                   exchange($p_k(m)$,$p_{k+i/2}(m)$)
               (The element at index $m$ of the vector on process $p_k$ is compared with
               element $m$ of the vector on $p_{k+i/2}$.)
        2.2.2. let $i = i/2$
      2.3. perform bitonic merging (Algorithm 2) on each process
      2.4. let $j = 2j$
```

□

The complete sequence has now been sorted and each process returns its vector back to the master process.

We are also interested in a version of the bitonic sort algorithm which uses pre-distributed data. In this situation the sorted vector is left distributed and there is no need for the initial scatter or final gather operations.

Step 2.2.1 of the parallel algorithm contains all the communications for the algorithm except for the initial scatter and final gather of the vector. Hence, the detailed code for this step will be optimised to reduce or hide the communication cost as much as possible for each architecture. This results in different implementation details for this step for the T8 and T9000/C104 algorithms.

4.3 T8 algorithm

4.3.1 Algorithm

The processor configuration used for the T8 algorithm is a chain of p slave processors connected at one end to a master processor. The selection of this configuration is discussed in Chapter 1. With such a configuration step 2.2.1 of the parallel algorithm will involve many communications of long vectors between distant processors. In order to reduce the cost of these operations a packet-based point to point communications routine was developed. This routine splits the communication of a single large vector into many small packets which are transmitted one after another. This has a much smaller cost than a single communication. For sufficiently large vectors, of size n, the cost for the packet-based communication approaches nT_c for communication between slaves a distance of p apart. This compares with a cost of npT_c for the single communication. Full details of the algorithm are given in Appendix A.

The bitonic sort algorithm for the T8 architecture proceeds as follows. Initially the master process scatters the vector to the p slaves using the packet-based communication routine, sending the block for the furthest process first. Once the slaves have received their blocks they sort them using an efficient sequential quicksort routine. Next the main phase of the parallel bitonic sort starts.

Step 2.2.1 of Algorithm 4 breaks down into three operations: an initial communication, the compare-exchange, and a final communication. Within one iteration of step 2.2.1 all the communications are between processes in a block of adjacent processes, e.g., $p_0 \ldots p_{i-1}$ and $p_i \ldots p_{2i-1}$. Each block of processes performs the same operations within its own block. We will consider the case for the process block $p_0 \ldots p_{i-1}$. First, each process p_k in the first half of the process block, i.e., $p_0 \ldots p_{i/2-1}$, receives the vector from process $p_{k+i/2}$ in the second half of the process block. Vectors from the lower numbered processes are sent first, with all communications using the packet-based routine. Once each process, p_k, in the first half of the process block has received a vector it then performs the sequential compare-exchange operation on elements in its own vector and the vector received from process $p_{k+i/2}$. If ascending sort order is required then the process keeps in its own vector the smaller of each pair of elements $p_k(m)$ and $p_{k+i/2}(m)$ and places the larger in the vector received from process $p_{k+i/2}$. If descending order is required the process keeps the larger element. After the compare-exchange operation the updated vector from process $p_{k+i/2}$ is returned. The vectors are sent from higher numbered processes first. These three operations are repeated for each iteration of step 2.2.1.

Once the parallel bitonic sort has completed each slave process has a sorted vector and these vectors are in sorted order over the processes, i.e., for an ascending order sort, the largest element on process p_k is smaller than the smallest element on process p_{k+1} and all subsequent processes. To finish the algorithm the master process gathers the distributed vector. Each slave process sends its vector back to the master process starting with the slave nearest the master process.

This algorithm requires storage on each slave process for $2n/p$ data values.

4.3.2 Model

In order to predict the performance of the algorithm we develop a run-time cost model. As for the Gaussian elimination algorithm of Chapter 3 the model will be given in terms of a set of hardware and problem parameters. In that chapter we modelled the computation cost by T_f. This is the time taken to perform a single double precision floating point arithmetic operation such as multiply or add. For a sorting algorithm the “computation” operation is the comparison and exchange of pairs of data values. The cost of this comparison will vary depending on the type of the data values which may include integers, floating point values and characters. In this chapter we model the comparison-exchange cost by a new hardware parameter, T_e, which represents the time taken to perform a single compare-exchange operation on a pair of data values. The other hardware and problem parameters remain the same as in the last chapter: T_c, the cost of communicating a single data value; n, the problem size, i.e., the size of the unsorted vector; and p, the number of processors. Presented in its simplest form, the parallel bitonic sort algorithm requires both n and p to have values that are powers of two, i.e., $n = 2^k$ and $p = 2^m$, $(m \leq k)$. These constraints on n and p are assumed throughout this chapter.

We now develop a cost model for the T8 algorithm. In this model we use the simplified form of the cost of the packet-based communication, i.e., nT_c.

Scattering the vector to slave processes and gathering the sorted vector using packet-based communications costs approximately $2nT_c$. The cost of the initial sequential quicksort on each process is $(n/p)\log_2(n/p)T_e$.

In each iteration of the parallel bitonic sort algorithm we have the following. For slave k to get a vector of n/p values from slave $k + 2^{i-1}$ in step 2.2.1 involves first reading and passing on $(2^{i-1} - 1)$ vectors before receiving its own. Using the packet-based routine this costs approximately $2^{i-1}(n/p)T_c$. This is followed by the compare-exchange at a cost of $(n/p)T_e$. Sending the rejected vector back to slave $k + 2^{i-1}$ costs $2^{i-1}(n/p)T_c$.

To form sorted sequences across j processes, step 2.2.1 is repeated for $i = 1 \ldots \log_2 j$. To this cost we add the cost of the bitonic merge on each process in step 2.3. This costs $(n/2p)\log_2(n/p)T_e$.

To completely sort the unsorted vector, we need to form sorted sequences across j processes for $j = 2, 4, 8, \ldots, p$.

The total parallel bitonic sort cost for the T8 architecture is thus given by:

$$2nT_c + \frac{n}{p}\left[2(2p - \log_2 p - 2)T_c + (\log_2(n/p) + \tfrac{1}{2}(\log_2 n + 1)\log_2 p)T_e\right].$$

For large n and large p the cost may be simplified to $6nT_c$. The cost for a sequential quicksort of n items is $T_e n \log_2 n$ and hence the speedup for large n and p has a simplified form of $\frac{T_e}{6T_c}\log_2 n$.

4.4 T9000/C104 algorithm

4.4.1 Algorithm

The implementation of the parallel bitonic sort algorithm for the T9000/C104 architecture is similar to that for the T8 architecture. A significant advantage of the T9000/C104 architecture for this algorithm is that the switch network will give a great reduction in the cost of communications between non-neighbour processes which is the dominant cost of step 2.2.1 of the algorithm. In the T9000/C104 algorithm direct point to point communications are used for step 2.2.1 of the algorithm. This should allow the performance of the T9000 algorithm to scale well to large array sizes which is not true for the T8 algorithm.

A second disadvantage of the T8 algorithm, as presented above, is that step 2.2.1 only uses half of the available processes for computation, the other half remain idle waiting to receive back the rejected vector. In the T9000/C104 algorithm we remedy this situation by splitting the vector on each process into two halves. Step 2.2.1 then proceeds as follows. Each pair of processes p_k and $p_{k+i/2}$ exchange half of their vectors: process p_k sends its high half vector to process $p_{k+i/2}$ and receives the low half vector of $p_{k+i/2}$ in exchange. These two communications can be executed in parallel provided the communicated vectors are input into temporary vectors of size $n/2p$. Process p_k then performs a compare-exchange operation on elements from its low half vector and the low half vector received from $p_{k+i/2}$. Similarly, process $p_{k+i/2}$ performs a compare-exchange operation on the high half vectors it holds. The step is completed by returning the high half vector to p_k and the low half vector to $p_{k+i/2}$ in parallel. Hence this method has two communications operations per step, each operation consisting of the communication of two half vectors in parallel.

This method keeps all the processes busy, but the total amount of communication performed is greater than necessary: the half vectors are returned to their source at the end of the iteration and then at the beginning of the next iteration output again if the source process has the same state as in the previous iteration (either expecting high or low half vectors). The communication cost can be reduced by fetching the half vectors at the beginning of the step directly from their location in the previous iteration. This removes the need to return half vectors at the end of the step, but may instead require two half vectors of a process to be input at the start of the step: both the half vector it would input under the previous method and its own half vector for this iteration if this was stored on a different process in the previous iteration. The total amount of communication in the worst case is as much as for the previous method, i.e., 4 half vectors per iteration, but all 4 communications can be executed in parallel in one operation at no additional cost in storage space, compared with 2 operations for the previous method.

Simple expressions specify the location of the 2 half vectors that process p_k requires in iteration i of Algorithm 4. We define the state of process p_k, $oddk$, as

$$oddk = (k \operatorname{div} i/2) \bmod 2.$$

If $oddk$, i.e., $oddk = 1$, then process p_k requires high half vectors, otherwise it requires low half vectors. Process p_k is paired with process p_{k1} where

$$k1 = k + i/2 - (i * oddk).$$

In the previous iteration the state of process p_{k1}, $oddk1p$, was

$$oddk1p = (k1 \operatorname{div} i) \bmod 2,$$

		$oddkp$	
		0	1
$oddk$	0	k_{low}	$k3_{high}$
	1	$k3_{low}$	k_{high}

own half vector

		$oddkp$	
		0	1
$oddk$	0	k_{low}	$k3_{high}$
	1	$k3_{low}$	k_{high}

other half vector

Figure 4.1: Locations of half vectors

and it was paired with process p_{k2}

$$k2 = k1 + i - (2i * oddk1p).$$

If $oddk1p$ then p_{k1} holds high half vectors from the previous iteration and p_{k2} holds low half vectors. Hence, from these expressions each process can calculate whether it needs to input vectors from p_{k1} or p_{k2} and whether these vectors are stored in low or high half vectors on those processes.

To find its own half vector, similar expressions are used. In the previous iteration p_k had state

$$oddkp = (k \operatorname{div} i) \bmod 2,$$

and was paired with process p_{k3} where

$$k3 = k + i - (2i * oddkp).$$

If $oddkp$ then p_k holds high half vectors from the previous iteration and p_{k3} holds low half vectors. Hence, from these expressions each process can calculate whether it needs to input its own high or low half vector from process p_{k3}.

These expressions are more clearly shown in Figure 4.1 as two tables which give the location of the 2 half vectors that process p_k requires at iteration i for the various values of $oddk$, $oddkp$ and $oddk1p$.

Similar expressions can be found for the destination processes for the half vectors held by a process at the start of an iteration.

This operation is repeated for the loop in Step 2.2 of the algorithm. On exit from this loop the half vectors must be returned to their correct processes before the next step.

This second method decreases the communication cost of the algorithm by performing communications in parallel, however, even better performance can be achieved by overlapping communications with comparisons. Step 2.2 may be rearranged to use parallel threads which operate on quarter vectors, i.e., halves of the half vectors. Whilst one thread performs the compare-exchange on a pair of quarter vectors, another thread inputs the next pair of quarter vectors. Step 2.2 becomes:

Algorithm 5 (Multi-threaded T9000/C104 main loop)

```
2.2.1 process $p_k$ gets first $1/4$ vector from $p_{k+i/2}$

2.2.2 while $i > 1$

    2.2.3 par
              perform compare-exchange on first $1/4$ vector
              get second $1/4$ vector

    2.2.4 par
              perform compare-exchange on second $1/4$ vector
              get first $1/4$ vector for next iteration (if $i > 2$)

    2.2.5 let $i = i/2$

2.2.6 return $1/4$ vectors to owner processes
```

□

This third method requires the same amount of storage as the previous method, but hides most of the communications behind compare-exchange operations.

The other phases of the parallel algorithm are the scatter, gather and quicksort. The T9000/C104 algorithm differs from the T8 algorithm in that the master process is also a slave process (see Chapter 1). Hence in the initial scatter of the unsorted vector the master process keeps n/p elements of the vector for itself, distributing the remainder over the $p - 1$ slave processes. We also use communications operations that have been optimised for the T9000/C104 architecture. The scatter and gather operations are described in Appendix A. The quicksort on each process is the same as for the T8 algorithm.

4.4.2 Model

The cost model for this algorithm on a T9000/C104 machine uses the same hardware and problem parameters as the T8 algorithm with the addition of an extra hardware parameter, T_s, for communications. The hardware parameters are: T_e, the cost of the compare-exchange operation for a single pair of data values; T_s, the startup cost for a communication; T_c, the cost of communicating a single data value. (See Chapter 2 for details of the hardware parameters.) The problem parameters are: n, the problem size, i.e., the size of the unsorted vector; and p, the number of processors.

If the vector to be sorted is stored on one master process then we must include the cost of the initial scatter of the unsorted vector, and the gather of the sorted vector back to the master at the end of the algorithm. The algorithm and cost model for these standard operations are given in Appendix A. They have a combined cost of

$$\frac{p-1}{2}\left(T_s + \frac{n}{p}T_c\right).$$

If the vector is already distributed before the algorithm starts we leave the sorted vector distributed also.

The main algorithm begins with each process performing an initial quicksort on its vector costing $(n/p)\log_2(n/p)T_e$.

The parallel threads in Step 2.2 cost $\max(T_s + (n/4p)T_c, (n/4p)T_e)$. The loop in i executes $\log_2 j$ times, and on the last iteration the second thread only performs a compare-exchange operation. Before the first iteration a $1/4$ vector is input at cost $T_s + (n/4p)T_c$. At the end of the loop two $1/4$ vectors are returned to their owner process in parallel at cost $T_s + (n/4p)T_c$. This gives a total cost for the i loop of:

$$(2\log_2 j - 1)\max\left(T_s + \frac{n}{4p}T_c, \frac{n}{4p}T_e\right) + 2(T_s + \frac{n}{4p}T_c) + \frac{n}{4p}T_e.$$

After this loop ends each process performs a bitonic merge (Algorithm 2) on its own vector at cost $(n/2p)\log_2(n/p)T_e$. All of this work in the loop of Step 2 is repeated over j, with $j = 2^k$ where $k = 1\ldots\log_2 p$. The total cost of this step is thus

$$\max\left(T_s + \frac{n}{4p}T_c, \frac{n}{4p}T_e\right)(\log_2 p)^2 + 2(T_s + \frac{n}{4p}T_c)\log_2 p + \left(1 + 2\log_2\frac{n}{p}\right)\frac{n}{4p}\log_2 pT_e.$$

This gives the total cost for a pre-distributed bitonic sort as

$$\begin{gathered}\tfrac{n}{p}\log_2\tfrac{n}{p}T_e + \max\left(T_s + \tfrac{n}{4p}T_c, \tfrac{n}{4p}T_e\right)(\log_2 p)^2 + \\ 2\left(T_s + \tfrac{n}{4p}T_c\right)\log_2 p + \left(1 + 2\log_2\tfrac{n}{p}\right)\tfrac{n}{4p}\log_2 pT_e.\end{gathered}$$

The total cost for a parallel bitonic sort of a vector initially on a single process is:

$$\begin{gathered}\tfrac{p-1}{2}\left(T_s + \tfrac{m}{p}T_c\right) + \tfrac{n}{p}\log_2\tfrac{n}{p}T_e + \max\left(T_s + \tfrac{n}{4p}T_c, \tfrac{n}{4p}T_e\right)(\log_2 p)^2 + \\ 2\left(T_s + \tfrac{n}{4p}T_c\right)\log_2 p + \left(1 + 2\log_2\tfrac{n}{p}\right)\tfrac{n}{4p}\log_2 pT_e.\end{gathered}$$

4.5 Performance

4.5.1 T8 algorithm

We have implemented the parallel T8 algorithm and an efficient sequential algorithm on the Parsys Supernode (see Chapter 1). These programs sort vectors of 32 bit integer values. Programs to sort the other data types are easily generated by source level global replacements of the data type name (e.g., replace INT by REAL32). The performance of the programs was measured for a range of problem sizes and number of processors. The problem size n was limited by the number of values that can be stored on a master processor with 4MB of memory and the requirement for n to be a power of two. This allowed problem sizes up to 524288. Since the slave processes require space for twice

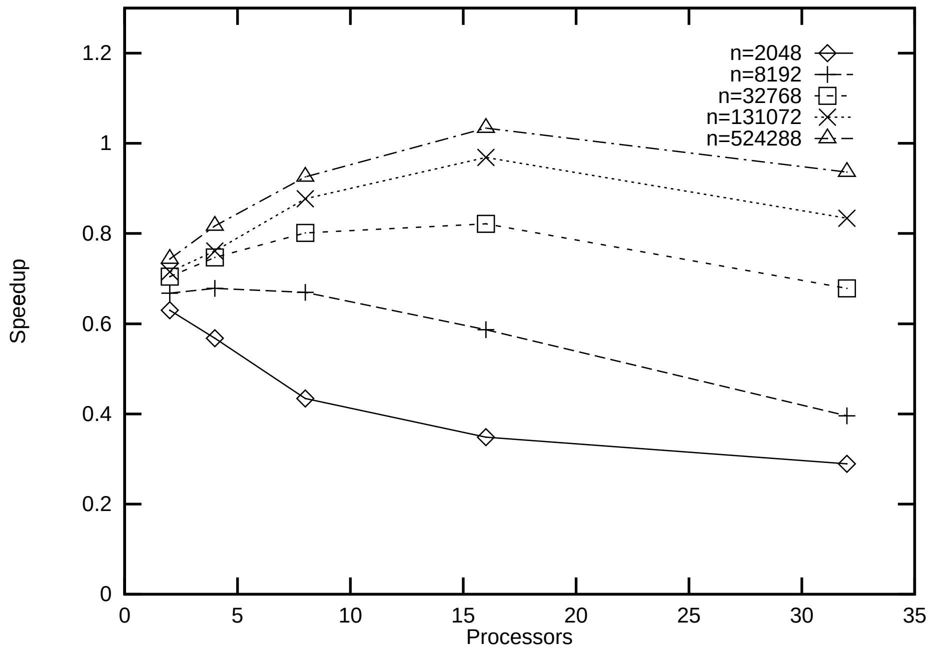


Figure 4.2: Measured speedup of algorithm for T8 architecture

the amount of data they initially receive, this largest problem size cannot be used on only a single slave processor. The parallel program was timed using $p = 2, 4, 8, 16$ and 32 slave processors.

We recorded the total time taken by the algorithm as measured by the master processor. In addition we measured the time taken by four parts of the parallel algorithm to illustrate the proportion of the total time that they consumed. These four parts are the initial scatter of the unsorted data, the sequential quick sort on each processor, the parallel bitonic sort, and the final gather of the sorted data. The time given for these phases of the algorithm is the time measured by the first slave processor. The times measured on other processors in the array vary as follows: As the processor number increases, i.e., the processors get further away from the master processor, the costs vary as follows: the scatter cost decreases, the sequential quick sort cost remains constant, and the bitonic sort and gather costs increase. Hence these figures are presented only to illustrate approximately the proportion of the total time spent in each phase of the algorithm. All these timings are given in Appendix D.

Figure 4.2 shows the speedup of the algorithm over an efficient sequential quicksort algorithm. These results are very disappointing showing that the parallel algorithm does not give any speedup at all for most problem sizes tested. Instead the sequential quicksort algorithm is shown to be quicker. Looking at the times for the four phases of the algorithm given in Table D.2 shows that, for all but the smallest arrays, the cost of the bitonic sort phase dominates the total cost. The cost of the initial scatter and final gather also make up a significant proportion of the total cost. This poor performance is due to several factors. The most important factor is the high cost of communications in the bitonic sort phase compared with the amount of computation performed. For each comparison operation the communication of two data values is required.

The cost to communicate a single 32 bit integer is $T_c = 4.40\mu$s. This value was obtained by measuring the time taken to send a large vector across a link (see Chapter 2). It is not so easy to obtain a reasonable estimate for the cost of a single comparison operation. In this case, we decided to estimate a value for T_e by fitting the sequential timings to the sequential cost model using a least squares fit. This gives a value of $T_e = 1.9\mu$s. This estimate for T_e takes into account the cost of the IF language construct and the exchange of data values as well as the comparison operation. Notice that this value is higher than the average cost of an integer arithmetic operation ($T_f = 1.2\mu$s).

From these values for T_e and T_c we see that the communication cost in the central bitonic sort phase is about 5 times greater than the computation cost. So we can say that the run-time cost of the parallel algorithm is dominated by the cost of the communications. Hence as more processors are used, small reductions in computation cost are hidden by the much larger, and almost constant, communication costs. At best the costs of the packet-based communications will remain constant as network size increases, for constant problem size n, but for small problems (such as those measured) communication costs increase with network size. Thus the speedup, if one can call it such, reaches a maximum and then falls as further processors are added. So for problems of similar size to those tested here, the parallel algorithm will provide almost no benefit and will probably cost more than a good sequential quicksort.

Another factor which affects the speedup of the algorithm is the use of only half the processors in the bitonic computation steps. But this only has a small effect on the total cost of the algorithm since we have already determined that communication costs dominate total costs. To reduce the communication costs we could implement some of the techniques used in the T9000/C104 algorithm. Splitting the vectors into half vectors and sending half vectors in both directions down links might reduce communication costs slightly, but for the T8 architecture there is not much spare link bandwidth for this extra communication on each link. Using quarter vectors and overlapping communication and computation would also improve the performance of the algorithm slightly. With the given values for T_e and T_c all the computation could be hidden behind communication. However, since it is the communication cost that dominates this again gives only a slight improvement. The only other way to improve the performance of this algorithm would be to use a richer topology such as a grid of processors. This would increase the available bandwidth and decrease the diameter of the network. Unfortunately, the current implementation was limited to a chain topology for compatibility with the Liverpool Parallel Library (see Chapter 1).

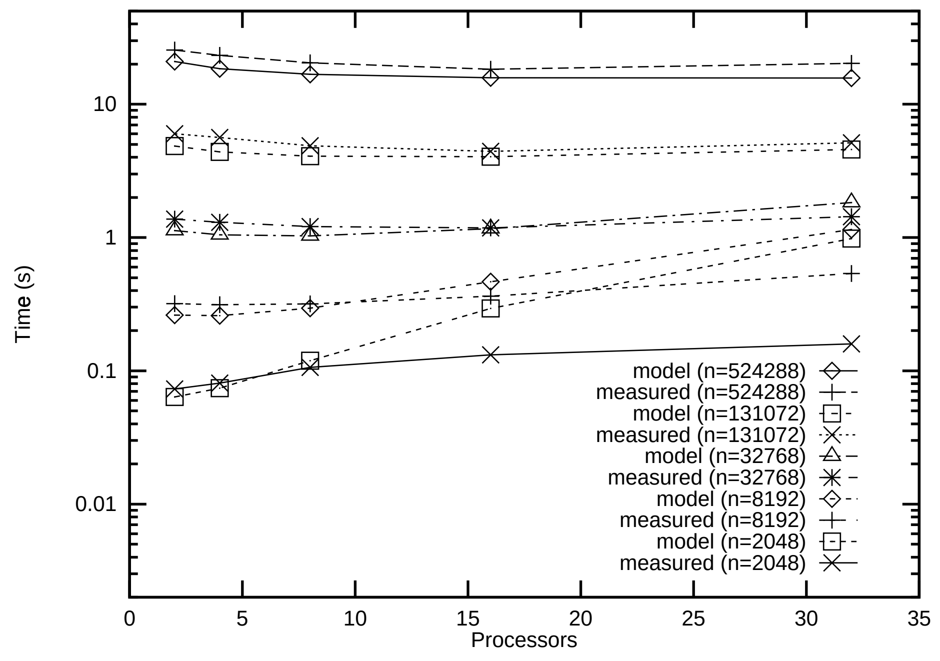


Figure 4.3: Comparison of measured time and model time for T8 algorithm

4.5.2 Comparison with T8 model

We have shown that for problem sizes that can be stored on a single master processor a sequential quicksort algorithm will give comparable if not better performance than the parallel algorithm. Now, we compare the measured costs with the costs predicted by the model and use the model to indicate the performance that might be obtained for very large pre-distributed problems.

Figure 4.3 shows a graph of the measured run-times and predicted run-times for the parallel algorithm. For the larger problem sizes and for small networks for smaller problem sizes the model predicts lower run-times than were actually measured. The large divergence between predicted times and measured times for smaller problems and larger network sizes is difficult to explain. One factor is that the communications model becomes invalid for such small problems. However this should lead to predicted timings which are less than the measured timings, but in fact the predicted timings are much greater than the measured ones. The error in the cost model for larger problem sizes is about 25%. These observations lead us to conclude that the cost model can only be used to give a rough guide to the actual run-time of the algorithm, but it does not model the true costs of the algorithm accurately enough to place any greater confidence in it.

Bearing in mind these considerations, Figures 4.4 and 4.5 show the speedups that the model predicts for very large problem sizes and large arrays of processors. Figure 4.4 shows the predicted speedup of the algorithm if the unsorted data were initially

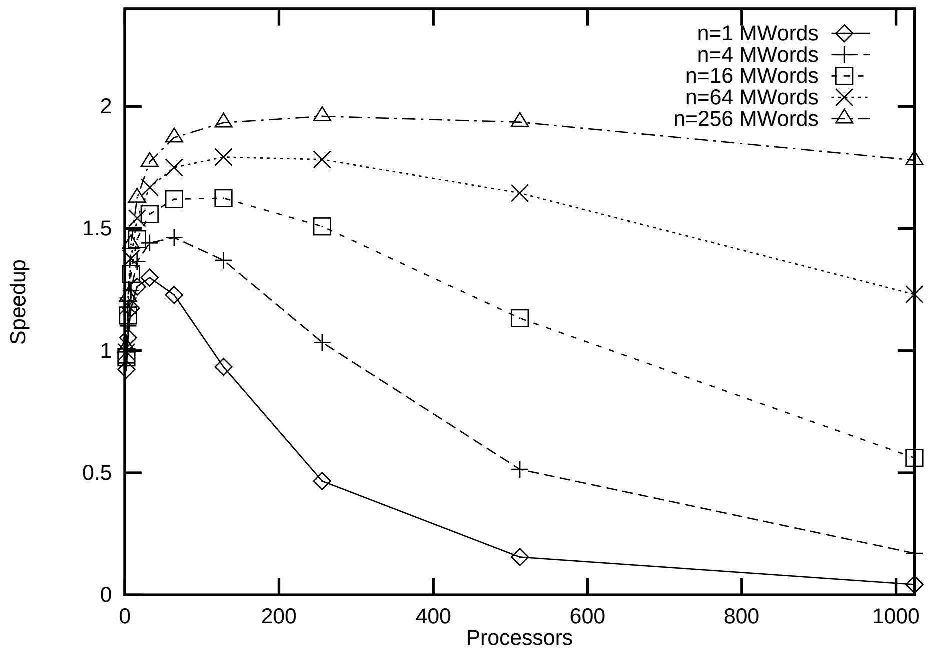


Figure 4.4: Speedup for single source algorithm on T8 architecture

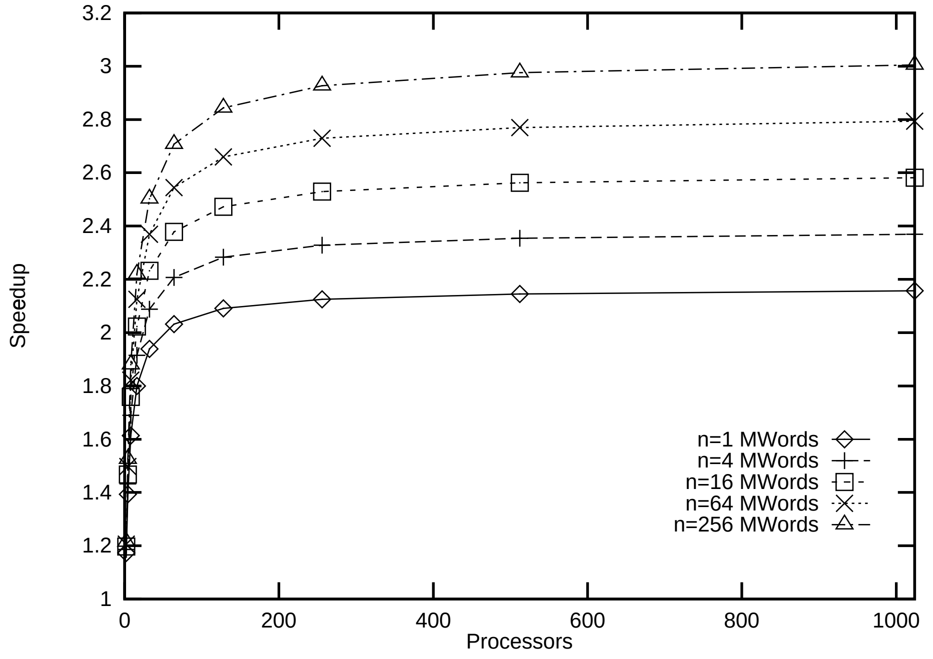


Figure 4.5: Speedup for pre-distributed algorithm on T8 architecture

stored on a single master processor. Figure 4.5 shows the speedup of the algorithm when the unsorted data is pre-distributed over the slave processors before the algorithm begins, and the sorted data is left distributed over the array. In practice, the volume of data that the largest of these problem sizes represent could not be stored on a single processor but can only be processed in a distributed manner. The only difference in cost between the two algorithms is the cost of scattering and gathering the data, but it is clear that this extra cost reduces the performance of the single source algorithm considerably. The model for the pre-distributed algorithm predicts that the performance reaches a plateau very quickly as the number of processors is increased and thereafter no improvement is achieved by adding more processors. This supports the observation made in the previous subsection, that the run-time cost of the parallel algorithm is dominated by the cost of the communications. The communication cost is constant for constant problem size (assuming the simpler packet-based communication model is valid) and the small decreases in the computation cost obtained by using additional processors are hidden by the much higher communication cost. Notice also that even for these very large problem sizes the run-time is only reduced by a small factor. Even the algorithmic improvements suggested in the previous subsection would only give a small increase in performance.

These results lead to the conclusion that for a chain of T8 processors the parallel bitonic sort algorithm is not worth using. Instead, a good sequential quicksort algorithm should be used. If the problem is too large to be held on a single processor, then the parallel algorithm can be used to allow the data to be sorted but the performance will not be good. The poor performance of the algorithm is fundamentally due to the large ratio of communication cost to computation cost for the bitonic sort phase of the algorithm. This ratio is determined by the hardware parameters and the ratio of communication operations to computation operations in this phase of the algorithm. Thus a better T8 algorithm must decrease the ratio of communication operations to computation operations. This cannot be done with the bitonic sort algorithm so other sorting algorithms must be considered. The other way to achieve better performance is to change the ratio of the hardware parameter values. The next subsection looks at the performance of the algorithm on the T9000/C104 architecture which does indeed have a different value for this ratio.

4.5.3 T9000/C104 algorithm

In the absence of the T9000 processor we must make an estimate for the cost of the compare-exchange operation. Here we have made the assumption that this cost will be the same as the cost for double precision floating point operations, i.e., $T_e = T_f = 100$ns (see Chapter 2). The true cost is probably higher than this if the example of the T8 architecture is relevant to the T9000 as well. The cost of communications on the T9000/C104 architecture is detailed in Chapter 2. For 32 bit data values we use $T_s = 1\mu$s and $T_c = 444$ns.

Neglecting the communication startup time the ratio of communication cost to

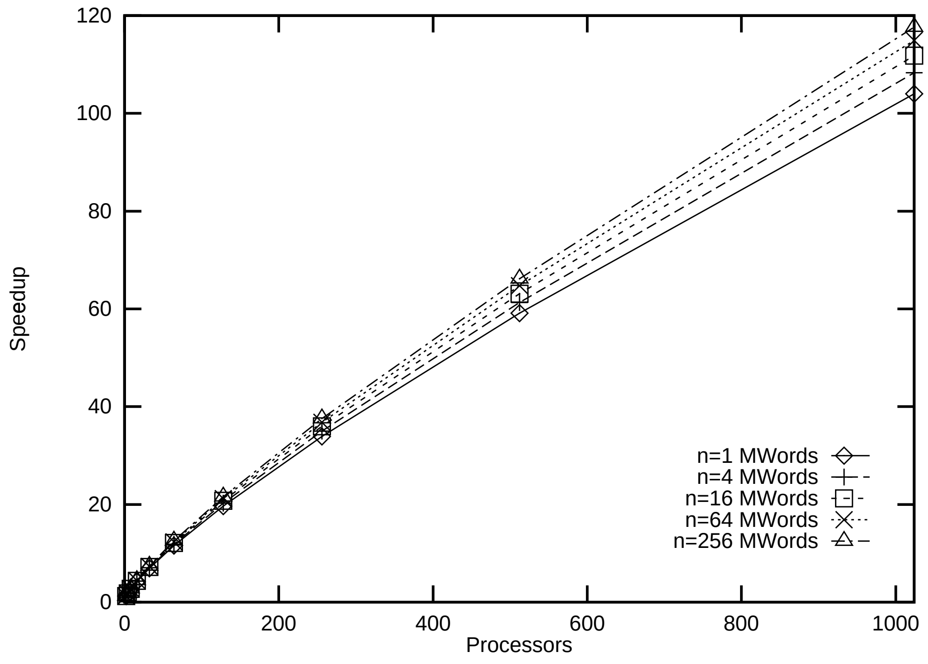


Figure 4.6: Speedup for pre-distributed algorithm on T9000 architecture

computation cost is $T_c/T_e = 4.4$ for the T9000/C104 architecture compared with $T_c/T_e = 2.3$ for the T8 architecture. The T9000/C104 ratio is worse than the T8 ratio, suggesting that the performance of the T9000/C104 algorithm will also be worse than that of the T8. However, this is not the case. In the bitonic sort phase of the T9000/C104 algorithm the exchanges of quarter vectors take place in parallel instead of sequentially within each block of processors as for the T8 algorithm. This is a considerable advantage especially for large networks where the number of vector exchanges becomes large. This factor allows the T9000/C104 algorithm to scale well to large networks. Figure 4.6 shows the speedup achieved by the pre-distributed algorithm for very large problem sizes. This shows that the algorithm does scale well to large networks. The algorithm also benefits from the overlap of communications and computation, essentially hiding the computation entirely behind communications.

Even though the pre-distributed T9000/C104 algorithm scales well for large problems and large networks its efficiency is very small. This is because of the much higher cost of the sequential bitonic sort algorithm compared with sequential quicksort and also because of the dominant communication costs in the parallel algorithm. If the hardware parameter ratio was less than 1, the communications would be hidden behind computation and the efficiency of the algorithm would be much higher. For the fundamental data types such a ratio is very unlikely for any architecture since communication between processors will always take longer than local memory accesses and the comparison itself has a very low cost. On the other hand, sorting string elements in a database would have a better balance between communication and comparison cost.

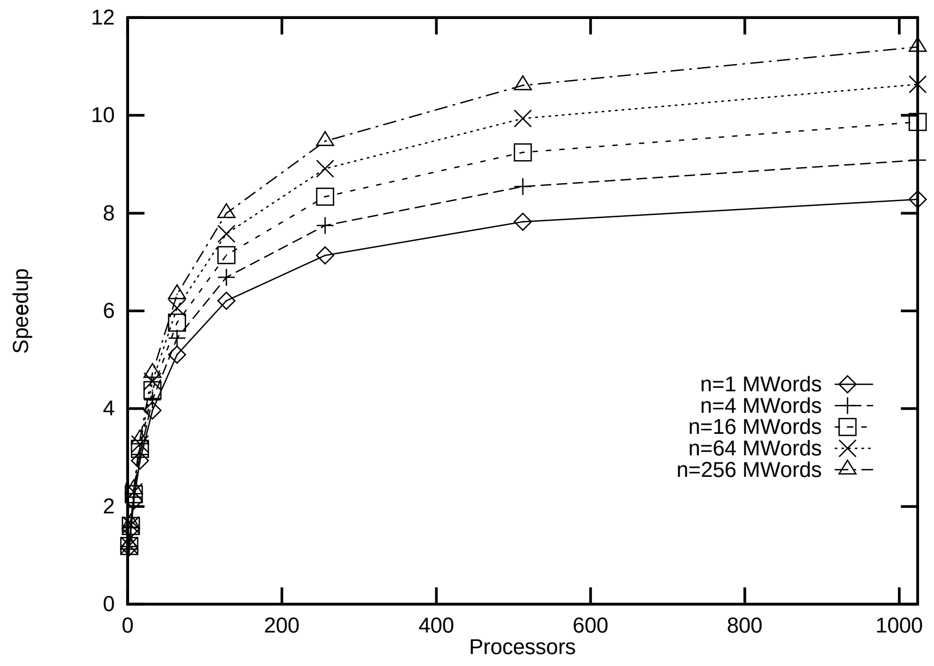


Figure 4.7: Speedup for single source algorithm on T9000 architecture

The low efficiency of the T9 parallel algorithm can be improved somewhat by the use of a variation in the bitonic sort algorithm. Instead of performing compare-exchange operations on the vector pairs, this improved algorithm merges and splits the sorted vectors. The communication pattern remains the same as for the original algorithm. This change doubles the arithmetic cost in the parallel threads and removes the need for the bitonic sort on each processor at the end of each loop. Hence, given the current ratio between communication and comparison the extra work performed in the threads is still hidden behind communications and the overall cost of the algorithm is reduced.

Figure 4.7 shows the speedup for the algorithm when data is initially held on a single source processor. This clearly shows that the scatter and gather of the vector does not scale well for large networks and since this is a large proportion of the total costs, the overall speedup is greatly reduced from that of the pre-distributed algorithm. The scatter operation cannot scale well because there are only 4 links out of the single source processor. Pre-distributed parallel algorithms in other fields of numerical methods often have a much higher cost compared with problem size which means that the scatter cost is much less significant and both forms of the algorithm will give a similar performance. However, the total cost for all pre-distributed parallel sort algorithms is low compared with the problem size and so this scatter bottleneck will affect all the sort algorithms and result in poor speedups.

4.6 Conclusions

The T9 pre-distributed parallel algorithm gives reasonable performance and scales very well onto large networks. This is due to the ability to communicate directly between all T9 processors in the network. Unfortunately, the relatively high cost of initial scatter means that the single source algorithm has a very poor performance. In conclusion, the parallel T9 bitonic sort algorithm can be recommended for applications where the data is pre-distributed, especially for extremely large vectors which cannot be held on a single processor. For smaller vectors which are initially stored on a single processor, a sequential quicksort is easier to use and is only a few times slower than the single source parallel algorithm. Other parallel sort algorithms may yield a greater efficiency than the bitonic sort algorithm, but all will give only a poor performance on the T9 when initial scatter is required.

For the T8 architecture a good sequential quicksort algorithm is preferable for small problem sizes. Very large problems which can only be stored distributed will have to use a parallel sort algorithm, but the parallel bitonic sort algorithm on a chain of processors cannot be recommended.

Chapter 5

Newton

In the last two chapters we have examined parallel algorithms for two important areas of numerical methods: linear algebra (Gaussian elimination) and sorting (bitonic sorting). We now turn our attention to parallel numerical optimisation. In previous work [78] we presented an overview of parallel optimisation algorithms which highlighted two algorithms for detailed study on the transputer architecture. The two algorithms are the Newton method and quasi-Newton method. These two algorithms illustrate the problems that arise when designing parallel optimisation routines. The Newton method highlights the use of parallel evaluations of the objective function to utilise arrays of processors. The quasi-Newton method emphasizes the need for efficient parallel linear algebra routines for the Hessian updates. This chapter discusses the Newton method in detail and the quasi-Newton method is covered in Chapter 6.

In this chapter we describe parallel implementations of the Newton algorithm for both T8 machines and T9000/C104 machines. The chapter highlights two main issues: parallel function evaluations and communication network performance. Parallel evaluations of the user's objective function, gradient vector and Hessian matrix are essential to achieve good performance from a parallel machine since in practice these evaluations are often very expensive. The algorithm also includes typical communication primitives in the linear algebra phase. We examine the improvement in communications performance that the C104 switch network achieves over the best T8 configuration—a grid of processors. We develop run-time cost models for the algorithms and compare their performance using an example function.

We use the following notation throughout the next two chapters:

- f, a continuous, twice differentiable function in n variables, with value $f(\mathbf{x})$ at the point $\mathbf{x}$,
- $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$, the gradient vector at the point $\mathbf{x}$,
- $G(\mathbf{x}) = \nabla^2 f(\mathbf{x})$, the Hessian matrix at the point $\mathbf{x}$,
- $\mathbf{p}$, a suitable search direction vector,

- $B(\mathbf{x})$, an approximation to the Hessian matrix $G(\mathbf{x})$, and
- $H(\mathbf{x})$, an approximation to the inverse Hessian matrix $G(\mathbf{x})^{-1}$.

Usually these notations are abbreviated by dropping the reference to $\mathbf{x}$, and a subscript is used to indicate the value at a particular iteration k, e.g., B_k.

5.1 Newton and quasi-Newton methods

The unconstrained optimisation problem is of the form:

$$\min_{\mathbf{x}\in\Re^n} f(\mathbf{x}) : \Re^n \to \Re$$

where $f(\mathbf{x})$ is at least twice continuously differentiable. Sequential unconstrained optimisation algorithms are usually based on the following algorithm:

Algorithm 6 (Model algorithm)

Let $\mathbf{x}_k$ be the current estimate of $\mathbf{x}^*$ the minimum of the function $f(\mathbf{x})$.

1. *Test for convergence.*
   Terminate the algorithm if the convergence conditions are satisfied returning $\mathbf{x}_k$ as the solution.

2. *Compute a search direction.*
   Compute a vector $\mathbf{p}_k$ to use as a search direction.

3. *Compute a step length.*
   Compute a scalar α_k such that $f(\mathbf{x}_k + \alpha_k\mathbf{p}_k)$ satisfies the condition
   $f(\mathbf{x}_k + \alpha_k\mathbf{p}_k) < f(\mathbf{x}_k)$.

4. *Update estimate for the minimum.*
   Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$, $k = k + 1$ and go to step 1.

□

The Newton and quasi-Newton family of methods computes the search direction, $\mathbf{p}_k$, in step 2 of the model algorithm directly as the solution to:

$$G_k\mathbf{p}_k = -\mathbf{g}_k$$

In Newton's method G_k is computed explicitly either analytically or using finite differences. Quasi-Newton methods do not compute G_k directly but instead an approximation, B_k, to the Hessian is maintained and updated in step 4 of the model algorithm.

The most widely used update is the rank 2 Broyden-Fletcher-Goldfarb-Shanno (BFGS) update:

$$B_{k+1} = B_k - \frac{B_k \mathbf{s}_k \mathbf{s}_k^T B_k}{\mathbf{s}_k^T B_k \mathbf{s}_k} + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{s}_k^T \mathbf{y}_k}$$

where $\mathbf{s}_k = \alpha \mathbf{p}_k$ and $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$. If an analytic Hessian is available, the Newton method is generally used for its superior convergence properties. When the analytic Hessian is not available the quasi-Newton method is used if function evaluation is expensive and the problem size n is not too large, otherwise if the function evaluation is cheap a finite difference Newton method is used.

Several papers have looked at methods for utilising parallel function and gradient evaluation. van Laarhoven [94] has examined Straeter's parallel variable metric algorithm [89]. During step 4 this method computes in parallel the gradient at n points displaced by small steps from $\mathbf{x}_k$ along n linearly independent directions. These gradient values are then used to sequentially update the approximate inverse Hessian H_k by the symmetric rank 1 update. The resulting algorithm exhibits quadratic termination. van Laarhoven also develops a parallel generalisation of Broyden's rank 1 formula. In addition, he uses parallel function and gradient evaluation during the line search. His algorithm computes $f(\mathbf{x})$ at multiple points along the search direction and chooses three points which bracket the minimum. The line search also incorporates a parallel interpolation procedure.

Freeman [46] points out that these algorithms are not guaranteed to produce positive definite matrices H_k. He then proceeds to describe a similar algorithm which utilises a parallel symmetric rank 2 updating formula based on Davidon's updating formula [28]. This algorithm has quadratic termination and guarantees that the approximate matrices H_k exist and are positive definite.

Although the preceding algorithms utilise parallel computation for gradient evaluation they do not allow the n matrix updates to be performed in parallel. This operation could then become the major cost within each iteration. These algorithms find the minimum of a quadratic function in one iteration, compared with n iterations for a sequential quasi-Newton update algorithm. This is because each iteration of the parallel algorithms performs n quasi-Newton updates. Hence these algorithms show good performance through a much reduced number of iterations. However, each individual iteration has a greatly increased function evaluation cost, but this is mainly hidden by the use of parallel gradient evaluations. Nevertheless there is still scope for improved performance. Only a low level of parallelism is achieved in the line search step—most processors remaining idle, and as already mentioned the updates must be performed sequentially—all extra processors being idle.

The normal description and cost analysis of these methods assumes that at least $n + 1$ processors are available to compute analytic gradients or $(n + 1)^2$ processors if gradients are approximated by finite differences. If only $n/2$, say, processors are available then the algorithms in their proposed forms cannot be executed, also any processors available in addition to those required cannot be utilised. This highlights a general problem for any parallel optimisation routine, namely that the grain size of the parallel algorithm will be determined directly by n.

Many of the optimisation test functions provide good examples of this difficulty. Rosenbrock's [72] function in 2 variables ($n = 2$) with analytic first derivatives can use at most 3 processors to calculate the gradient values used to update the inverse Hessian approximation. If gradients are approximated by finite differences then 9 processors could be utilised. Hence, even if the parallel array contained many more processors the maximum speedup possible would be less then 9 since only that number of processors could be used by the algorithm. A further point follows from this. The expense of function evaluation, although dependent on n, can be considerable even for small n since the function itself may be, for example, a sub-optimisation problem or a simulation. Thus a problem function with large cost but only small dimension n which requires an impracticably large run time to solve on conventional computer architectures can only have the solution time reduced by a small factor of order n or perhaps n^2 but not p. Of course, many small dimension problem functions, including the Rosenbrock function, are very cheap to solve and in these cases the parallel methods would probably have poorer performance than their sequential counterparts due to the high communication to computation ratio.

A well known extension to Rosenbrock's function gives a general function of dimension n (n even). For a larger dimension problem of, say, $n = 20$ either 21 or 441 processors could be utilised. However, it is likely that processor arrays would have some intermediate number of processors. If an analytic gradient algorithm is executed only 21 processors are used and the maximum speedup attainable is only 21. The remaining processors not used by the algorithm could be made available to other tasks running on the array, requiring multi-user operating system support on the machine. To make use of all the available processors on the array executing a finite difference gradient algorithm could be achieved in two ways. Firstly, the 441 processes required by the algorithm could be mapped onto the smaller number of available processors. This use of excess parallelism would allow the maximum possible speedup given the limited resources. A second approach would be to develop algorithms which compute only as many of the gradient vectors as the number of available processors allows.

We now describe a family of parallel methods due to Byrd, Schnabel and Shultz [19] which follow this approach and may use a variable number of processors. These algorithms are interpolations between Newton's method and a quasi-Newton method. During the line search, step 3 of the model algorithm, these methods utilise idle processors to calculate gradient information which is then used in step 4 to update the Hessian approximation B_k. As in the previous algorithms, the use of gradients to improve the Hessian approximation is expected to decrease the number of iterations required. In their papers Byrd et al. consider the case where gradient vectors are computed by finite differences, but the same ideas may be applied when gradients are computed analytically with only the number of processors utilised changing.

The algorithms due to Straeter and Freeman incur an extra cost in each iteration in the computation of the gradient information in step 4 of the model algorithm. In contrast, the algorithms due to Byrd, Schnabel and Shultz do not increase the cost of each iteration since the gradient information is computed in parallel with the line search of step 3 and only as many function evaluations as the number of processors permits are computed. Hence full use is made of the available processors during step 3 without increasing the run time. Schnabel calls this process speculative gradient evaluation.

There are alternative uses of the processors during the line search step. As already mentioned, van Laarhoven uses multiple processors to bracket the minimum and compute the interpolation. Lootsma [70] and others have also suggested evaluating $f(\mathbf{x})$ at multiple points in the search direction and even in other directions in parallel with the computation of $f(\mathbf{x})$ at the trial point. However, it does not seem likely that points off the search direction will be better choices for the next iterate than those on that line. Also the large body of results from sequential quasi-Newton methods clearly show that on average only between 1 and 2 trial points are selected before an acceptable point is found and at each new accepted point a complete gradient evaluation is performed. This ratio of function evaluations to gradient evaluations suggests that processors would be much better utilised in computing a speculative gradient than in computing additional trial points when the current trial point is likely to be accepted anyway.

Schnabel [87] makes some suggestions about the use of the gradient information computed at a rejected trial point. One simple approach would be to use the search direction gradient to help compute the step length to the next trial point along the current search direction. Some current sequential line search algorithms use gradients at trial points but these do not improve the algorithm's performance significantly. Another suggestion is to use gradient information to solve a tensor model of the function rather than the standard quadratic model to find the next iterate [86]. Finally he proposes updating the Hessian G_k immediately using the gradient information at the rejected trial point and computing a new search direction.

If the number of processors available p is less than or equal to $n + 1$ then Byrd et al. recommend using the quasi-Newton algorithm with $p - 1$ elements of the gradient vector at the trial point speculatively computed in parallel with the trial point function evaluation. When a trial point is accepted the remaining elements of $\mathbf{g}$ are computed in parallel, otherwise a new trial point is chosen and the process repeated.

When $p \geq (n^2 + 3n + 2)/2$ then the entire Hessian matrix can be approximated by finite differences along with the trial point function and gradient in one concurrent function evaluation step. This method is a parallel discrete Newton method. More than $(n^2 + 3n + 2)/2$ processors cannot be utilised.

The most common situation will be $n + 1 \leq p \leq (n^2 + 3n + 2)/2$. In this case $m = \lfloor (p - (n+1))/(n+1) \rfloor$ gradient vectors from around the trial point are computed in step 3 of the model algorithm. In [19] they describe and compare 11 such algorithms including the quasi-Newton and discrete Newton algorithms mentioned above. These algorithms all include at step 4 a multiple update of the Hessian approximation G_k using the m gradient vectors along the finite difference directions. The methods fall into 3 categories based on the use made of the standard quasi-Newton update in the search direction:

- Only the gradients along the finite difference directions are used to update G_k.
- The search direction update of G_k is performed after the finite difference updates.
- A temporary update of G_k in the search direction is made after the finite difference updates, and the resulting matrix is used to compute the search direction for the next iteration.

Two different methods are used to select the finite difference directions. One method simply cycles through the unit vectors and the other computes a set of m directions that are orthogonal to the previous $m-1$ directions. Also, the BFGS and other update formulae are used. The first and third categories of methods have m-step quadratic convergence and 1-step Q-superlinear convergence rates. Neither of these results has been shown for the second category in general.

Computational results are presented for the case $m = 1$. These show that the first category of methods all perform more poorly than the parallel quasi-Newton method using BFGS updates even though they use finite difference Hessian information. Temporary search direction update methods gave better performances but were still not better than the BFGS method. The third category of methods using both finite difference and search direction updates has the best performance being significantly better than the BFGS method. Of the three methods in this category the two which utilised the BFGS update were superior and Byrd et al. recommend a unit vector finite difference, BFGS method since it does not have the complication of calculating the conjugate directions which are required by the other BFGS method.

In [18] simulated results are presented for the BFGS unit vector finite difference algorithm and the parallel BFGS quasi-Newton and Newton methods when $n = 20$. Over the range of problems tested the average speedup over the sequential BFGS method of the parallel BFGS quasi-Newton method was 17.5, and for the parallel Newton method the average speedup was 82.3. With m ranging from 1 to n, the BFGS unit vector finite difference algorithm gave average speedups between 32.6 and 69.5. These simulated speedups assume that each algorithm had sufficient processors to compute all the speculative gradient evaluations in a single concurrent evaluation step. That paper also shows that the BFGS unit vector finite differences algorithm has superlinear convergence.

If function evaluation is not expensive the linear algebra involved in steps 2 and 4 becomes significant. It is thus essential to parallelise these steps if possible to maintain good performance. In [18] four different implementations of the linear algebra are considered. This shows that an efficient algorithm can be developed by storing and updating G_k^{-1}, the inverse Hessian approximation, using only matrix-vector and rank 1 updates which parallelise well. They demonstrate this by implementing a parallel algorithm based on an unfactored G_k^{-1}, distributed by rows.

In the next chapter we discuss the design and implementation of a parallel BFGS algorithm for transputer systems. In the rest of this chapter we look at the design of a parallel Newton method for transputers.

5.2 Newton algorithm

Iteration k of Newton's method consists of the following steps:

Algorithm 7 (Newton method)

1. Evaluate gradient vector $\mathbf{g}_k$ at $\mathbf{x}_k$.
2. Evaluate Hessian matrix G_k at $\mathbf{x}_k$.
3. Solve linear system for search direction vector $\mathbf{p}_k$:

$$G_k \mathbf{p}_k = -\mathbf{g}_k$$

4. Perform line search along $\mathbf{p}_k$ to find new estimate $\mathbf{x}_{k+1}$ of the minimum point of $f(\mathbf{x})$ in the direction $\mathbf{p}_k$:

$$\mathbf{x}_{k+1} : \qquad f(\mathbf{x}_{k+1}) = \min_{\alpha} f(\mathbf{x}_k + \alpha \mathbf{p}_k)$$

□

The first three steps within each iteration are suitable for parallelisation and the last step can also be parallelised to a lesser extent. For optimum efficiency the distributed data structures used by these parallel steps must be compatible. Without this provision there would be a requirement to re-distribute the data between steps. For Newton's method the data structure is determined by the algorithm which solves the linear system to find the search direction. The parallel algorithm we have chosen is row-oriented Gaussian elimination with partial pivoting. This parallel algorithm is known to perform well and has already been implemented on the current transputer architecture (see [74] and Chapter 3 for details). The distributed data structure therefore consists of scattered rows of G on each processor along with the associated elements of $\mathbf{g}$. When the system of equations is solved the search direction $\mathbf{p}$ is also distributed, with each element of $\mathbf{p}$ located on the processor that holds the associated element of $\mathbf{g}$. The other parallel steps, namely the evaluation of the Hessian matrix, G, and gradient vector, $\mathbf{g}$, must therefore produce a distributed data structure of scattered rows of G and the associated elements of $\mathbf{g}$.

Methods to parallelise the line search of step 4 have been discussed in the preceding section. In this section we use a sequential control thread to determine a new minimum point with parallel function evaluations to calculate required gradient vectors. This step must be preceded by gathering the distributed elements of $\mathbf{p}_k$ at one processor. Similarly, after the new minimum point $\mathbf{x}_{k+1}$ is found it must be broadcast to all the other processors before the next iteration can proceed.

We assume that a network of p slave processors is available, connected to a single master processor. The T9000/C104 network is configured as a three stage folded Clos network as described in Chapter 1. In order to make a fair comparison between the performance of the T9000/C104 communication architecture and the current generation T8 architecture we assume that the T8 machine is configured as a grid of processors with no wrap round of edge links. For simplicity of the run-time cost model we assume further that the T8 processors are configured in a square grid.

5.3 The user interface

The user interface to the algorithm is through the user-provided functions to evaluate the objective function, f, the gradient vector, $\mathbf{g}$, and the Hessian matrix, G. Sequential Newton algorithms expect the user to provide sequential functions that will calculate all of the Hessian and gradient in a single call. If this same user interface were retained for the parallel algorithm then it would not be possible to exploit parallel evaluation of the Hessian or gradient. However with only small changes to the user interface parallelism can be achieved.

We modify the Hessian evaluation function parameter list to include a vector of n elements which specify to the user function which rows of the $n \times n$ Hessian matrix it should evaluate. This allows the Newton algorithm to utilise all the available processors in parallel to calculate different rows of the matrix. The parameter list could be further extended to allow specification of whether the function should calculate rows or columns or even blocks of the matrix. The gradient evaluation function parameter list is similarly extended to specify to the user function which elements of the gradient vector it should calculate.

The user's functions remain largely unaltered except that they should perform checks to ensure that they only evaluate those elements specified. The user-provided function to evaluate f is identical to that required by current sequential algorithms.

5.4 Modelling the algorithm

To develop run-time cost models for the algorithm we need to identify the parameters that characterise the performance of the architectures. For distributed memory MIMD machines these parameters represent the computation and communication costs. We use T_f to model the computation cost for both architectures. This parameter is the time taken to perform one double precision floating point operation. Each architecture actually has a different value for the computation cost but the use of a single parameter T_f

makes the cost models clearer. It should be clear from the context which architecture, and hence which value for T_f, is being referred to in this chapter.

We model the communications using a similar group of parameters. For the T8 architecture we use a single parameter T_c to model the cost of sending a double precision value across a link. It has been shown [74, 62] that this single parameter models communication costs well without the addition of a communication startup parameter. The cost of communication over the switch network of the T9000/C104 architecture can be modelled by two parameters: a message latency or startup time, T_s, and a cost for each value transmitted through the network, T_c. Again we use the same parameter T_c for both the T8 and T9000/C104 architecture but the context should make clear which value is intended. Full details of the communications networks and modelling parameters can be found in Chapter 2.

5.5 Gradient and Hessian evaluation

To model the cost of the user-provided functions to evaluate f, $\mathbf{g}$ and G we introduce three parameters T_u, T_g and T_G. T_u is the cost of a single function evaluation. T_g and T_G are the costs for evaluating the most expensive single element of $\mathbf{g}$ and G respectively. These parameters all specify a cost in terms of the number of floating point operations required, i.e., a cost given as kT_G means a real time cost of kT_GT_f.

The gradient vector and Hessian matrix calculations require no communication and so will have the same cost expression for both architectures. Each of the p processors evaluates n/p rows of G and n/p elements of $\mathbf{g}$. This has a cost of:

$$\left(\frac{n^2}{p}T_G + \frac{n}{p}T_g\right)T_f.$$

5.6 Solving the linear system

Step 3 of the Newton method requires the solution of a system of linear equations to find a new search direction $\mathbf{g}_k$. This is achieved using a parallel Gaussian elimination algorithm with partial pivoting. The algorithm consists of two phases: forwards elimination of the full matrix G and right-hand-side (RHS) vector $-\mathbf{g}$, followed by backwards substitution of the RHS vector. Full details of the algorithm are given in Chapter 3, but a summary of the parallel algorithm is given below:

Algorithm 8 (Forwards elimination)

For the forwards elimination phase the following steps are performed:
For $i = 1, \ldots, n-1$

1. find best pivot element on each process,

2. global max operation performed on individual processors' best pivots with ith row holder as destination,
3. broadcast of best pivot holder from ith row holder,
4. broadcast $(n-i+1)$ elements of best pivot row and associated RHS element by its holder,
5. send $(n-i+1)$ elements of ith row and associated RHS element to pivot row holder if row swap required,
6. pivot $(n-i)$ rows and RHS elements.

□

Algorithm 9 (Backwards substitution)

For the backwards substitution phase the following steps are performed:
For $i = n, \ldots, 1$

1. holder of row i evaluates element p_i,
2. broadcast element p_i,
3. Update subtotals on processors.

□

Essentially the same parallel algorithm is used for both the T8 architecture and the T9000/C104 architecture. The only difference between the programs concerns the implementation of communication operations. We have been careful to specify the algorithm in terms of reasonably high-level communication primitives such as broadcast and global operations. This ensures that the program code for both architectures is almost the same. Slave codes in both programs execute communications operations by calling communications subroutines. The T8 program will link in versions of the communication subroutines specifically for the T8 architecture and the T9000/C104 version will link in a T9000/C104 library of communication subroutines. Appendix A describes these common communication primitives and gives cost models for both T8 and T9000/C104 architectures.

The following two subsections briefly describe the steps in the forwards elimination and backwards substitution algorithms and present models for the run-time cost for the T8 and T9000/C104 architectures.

5.6.1 Forwards elimination

The problem of finding the best pivot value for each iteration is divided into two steps (steps 1 and 2). First, each processor finds the best element it holds in step 1. Let us say for simplicity that the average number of element pairs compared in each iteration is $n/2p$ then the cost of this step for both architectures is $\frac{n}{2p}T_f$. The second step involves a global communication operation in which the processors communicate their local best pivot values to find the global best pivot value. This operation is coded as a subroutine so that each architecture can use a routine which is optimised for its communication network. Each interprocess communication consists of two data values: a pivot value and the pivot owner process number. The communication algorithms are given in Appendix A. The destination for the global operation is the processor owning row i where i is the iteration number. The cost for the T8 architecture is $\frac{3(\sqrt{p}-1)}{2}(2T_c + 3T_f)$, and for the T9000/C104 architecture: $\lceil \log_b((b-1)p+1) - 1 \rceil (T_s + 2T_c + bT_f)$. In the latter cost expression b is the branching factor of the communications tree used for the broadcast algorithm. We use a value of $b = 4$. See Appendix A for details.

The best pivot value owner is then broadcast to all slaves from the processor owning row i in step 3. Again this step uses a communications subroutine described in the appendix. For the T8 architecture the cost of this step is $\frac{3(\sqrt{p}-1)}{2}T_c$ and for the T9000/C104: $\lceil \log_b((b-1)p+1) - 1 \rceil (T_s + T_c)$.

In step 4 the owner of the best pivot row broadcasts elements of that row and the RHS element to all the slave processors. T8 cost: $\frac{3(n-i+2)(\sqrt{p}-1)}{2}T_c$, T9000/C104 cost: $\lceil \log_b((b-1)p+1) - 1 \rceil (T_s + (n-i+2)T_c)$. In iteration i the row containing the best pivot value may not be row i. Hence a row swap may be required between row i and the best pivot row. In this situation step 5 needs to be performed to send the elements of row i to the processor owning the best pivot row. In theory, the Hessian matrix should be positive definite and no swapping should be necessary. However, let us assume that a row swap is required on every iteration. For the T8 architecture the maximum distance between the source and destination processors is $2(\sqrt{p}-1)$ when the source and destination are in opposite corners of the square grid. The minimum distance is 0 when the source and destination processors are the same. Let us use an average distance between source and destination of $(\sqrt{p}-1)$. This gives a cost for step 5 for the T8 architecture of $(\sqrt{p}-1)(n-i+2)T_c$. The cost of step 5 for the T9000/C104 architecture is $T_s + (n-i+2)T_c$.

Now that each processor has a copy of the pivot row, they update their rows of the matrix and RHS elements in step 6. This pivoting operation has the same cost for both architectures, i.e., $\frac{(n-i)(2n-2i+3)}{p}T_f$.

The total cost for the forwards elimination on each architecture is found by summing the component costs over i. For the T8 architecture the total cost for the forwards elimination is:

$$\left[\frac{n(2n+5)}{3p} + \frac{9(\sqrt{p}-1)}{2}\right](n-1)T_f +$$

$$\left[\frac{5n+38}{4}\right](\sqrt{p}-1)(n-1)T_c.$$

For the T9000/C104 architecture the total cost for the forwards elimination is:

$$\left[\frac{n(2n+5)}{3p}+(\lceil\log_b((b-1)p+1)-1\rceil-1)b\right](n-1)T_f+$$
$$(3\lceil\log_b((b-1)p+1)-1\rceil-2)(n-1)T_s+$$
$$\left[\frac{(n+10)}{2}\lceil\log_b((b-1)p+1)-1\rceil-3\right](n-1)T_c.$$

5.6.2 Backwards substitution

The backwards substitution is much simpler than the forwards elimination. In step 1 the next unknown element of $\mathbf{s}_k$ is calculated by its owning processor at a cost of $2T_f$. Step 2 involves the broadcast of this value to all the slave processors so that they can update their partial sums. For the T8 architecture this costs $\frac{3(\sqrt{p}-1)}{2}T_c$, and for the T9000/C104 $\lceil\log_b((b-1)p+1)-1\rceil(T_s+T_c)$. In step 3 each processor uses the new element of $\mathbf{s}_k$ to update its partial sums. This costs $\frac{2(i-1)}{p}T_f$.

The total cost for the backwards substitution on each architecture is found by summing the component costs over i. For the T8 architecture the total cost for the backwards substitution is:

$$\left[\frac{n(n-1)}{p}+2n\right]T_f+\frac{3n}{2}(\sqrt{p}-1)T_c.$$

For the T9000/C104 architecture the total cost for the backwards substitution is:

$$\left[\frac{n(n-1)}{p}+2n\right]T_f+n(\lceil\log_b((b-1)p+1)-1\rceil-1)(T_s+T_c).$$

5.7 Line search

After solving the system of linear equations to find a new search direction the last step in an iteration of Newton's method is to search along this new search direction for an updated estimate of the minimum point of the function. This is implemented by a sequential control thread running on the master processor which determines a new minimum point, along with parallel function evaluations to calculate required gradient vectors performed by the slave processors.

Before the line search can be performed, the distributed search direction vector, $\mathbf{p}_k$, must be gathered from the slave processors to the master processor. On the T8 architecture this has a cost of nT_c and for the T9000/C104 architecture the cost is $\left\lceil\frac{p-1}{4}\right\rceil(T_s+(n/p)T_c)$.

We will fit a third-order polynomial to the function given the function value and gradient at two distinct points along the search direction. The number of steps required in such a line search is problem dependent, but in practice reasonable performance is attained with, on average, 1.5 steps. In each step the gradient evaluations are performed in parallel on the slaves with each slave calculating n/p elements of the gradient vector. To calculate the gradient at these two points, the position vectors of these points must be broadcast, and after each slave has calculated its elements of the gradient vectors, the gradient vectors must be gathered back to the master processor. These communications cost $4\sqrt{p}nT_c$ on the T8 architecture, and $2(\lceil \log_b((b-1)p+1)-1\rceil (T_s+nT_c)+\left\lceil \frac{p-1}{4}\right\rceil (T_s+(n/p)T_c))$ on the T9 architecture per step. The function evaluations are done by the master processor. The cost of the function and gradient evaluations is approximately $2(T_u+(n/p)T_g)$ per step.

After the line search the new minimum point $\mathbf{x}_{k+1}$ must be broadcast to all the slave processors for the start of the next iteration. This costs $(2\sqrt{p}-1)nT_c$ for a T8 architecture and $\lceil \log_b((b-1)p+1)-1\rceil (T_s+nT_c)$ for the T9000/C104 architecture.

5.8 Performance

To assess the overall performance of a parallel algorithm we want to examine the run-time cost of the algorithm for problems from a range of problem sizes. In the preceding chapters the problem size has been specified in terms of a single parameter n which uniquely determines the complexity of the problem. For optimisation algorithms, and especially the Newton method, the problem complexity cannot be specified by a single parameter since it depends on the complexity of the user provided functions. In the case of the Newton method we need four problem parameters to specify the complexity of the algorithm. These parameters are n, the dimension of the problem space and hence the size of vectors and arrays; T_u, the cost of performing a single objective function evaluation; T_g, the cost of evaluating a single element of the gradient vector; and T_G, the cost of evaluating a single element of the Hessian matrix.

For a given problem, n can be specified precisely but the costs of the user provided functions, T_u, T_g and T_G, will usually vary depending on run-time conditions, such as the current trial point. In addition, for many problems that are solved in practice the user functions are very complicated and the user cannot provide an analytic run-time cost for them. In order to make use of our cost model in spite of these difficulties we proceed as follows. If the user functions are relatively simple and we can obtain analytic cost models we use the models to give values for T_u, T_g and T_G which are the maximum cost for evaluating the function. If the functions are more complicated then we use values for these parameters obtained by timing sample runs of the user's sequential functions on one processor of the parallel machine.

To vary the problem complexity we can vary the values for n, T_u, T_g and T_G and then find the run-time costs of the parallel algorithm on machines with a varying number of processors. For this chapter we have chosen as a starting function Rosenbrock's extended function [72]. This is a simple function of size n (n even) given by

$$f(x) = \sum_{i=1}^{n} f_i^2(x) \quad \text{where} \quad f_i(x) = \begin{cases} 10(x_{i+1} - x_i^2) & i \text{ odd} \\ 1 - x_{i-1} & i \text{ even.} \end{cases}$$

This function has a Jacobian gradient vector g given by:

$$g_i(x) = \begin{cases} -400x_i(x_{i+1} - x_i^2) - (1 - x_i) & i \text{ odd} \\ 200(x_i - x_{i-1}^2) & i \text{ even,} \end{cases}$$

and a Hessian matrix G:

$$G_{ij}(x) = \begin{cases} 1200x_i^2 - 400x_{i+1} + 1 & i \text{ odd, } j = i \\ 200 & i \text{ even, } j = i \\ -400x_i & i \text{ odd, } j = i + 1;\ i \text{ even, } j = i - 1 \\ 0 & \text{otherwise.} \end{cases}$$

For this function the cost to evaluate the function f is $T_u = (4n - 1)T_f$. The most expensive elements of g to calculate are the odd elements, with a cost $T_g = 6T_f$, and the most expensive elements of G are the diagonal elements on odd rows which have a cost $T_G = 5T_f$.

We use the values for the hardware parameters for the T8 and T9000/C104 architecture taken from Chapter 2.

Figures 5.1 and 5.2 show graphs of the modelled speedup of the T8 and T9 parallel algorithms for Rosenbrock's test function. We calculate the speedup using an efficient sequential Newton algorithm to give the sequential cost. These graphs show how the speedup varies when we vary the problem complexity by changing n. The other problem parameters have the values given above. Rosenbrock's function and derivatives are very cheap to evaluate and so the main contribution to the total cost of the algorithms comes from solving the system of linear equations in step 3 of the algorithm. The cost of this step is independent of the user function evaluation costs, depending only on n. With these small values for the function costs, and increasingly large values for n, the cost of the Gaussian elimination algorithm dominates the overall cost and the graphs tend to indicate the performance of the parallel Gaussian elimination algorithm on the architecture. The graphs show that the T9000/C104 architecture achieves significantly better speedups than the T8 architecture for this operation. This is in spite of the fact that the T9000/C104 architecture has a slightly worse ratio between computation and communication, $T_f/T_c = 0.11$, than the T8 architecture with $T_f/T_c = 0.18$. The better performance of the T9000/C104 architecture is due to its superior network connectivity which keeps message transmission cost down even for large network sizes and allows a higher node bandwidth to be achieved by making full use of all four links on each processor. In addition, the T9000/C104 architecture offers about a ten-fold improvement in absolute performance over the T8.

The problem parameter n determines the maximum degree of parallelism that can

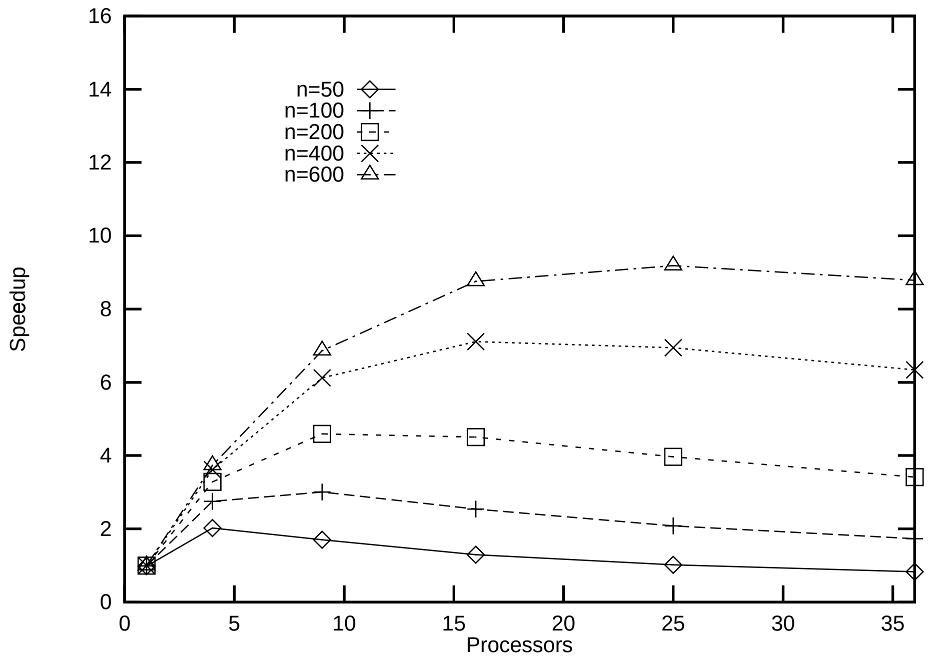


Figure 5.1: Speedup of algorithm as n varies for T8 architecture

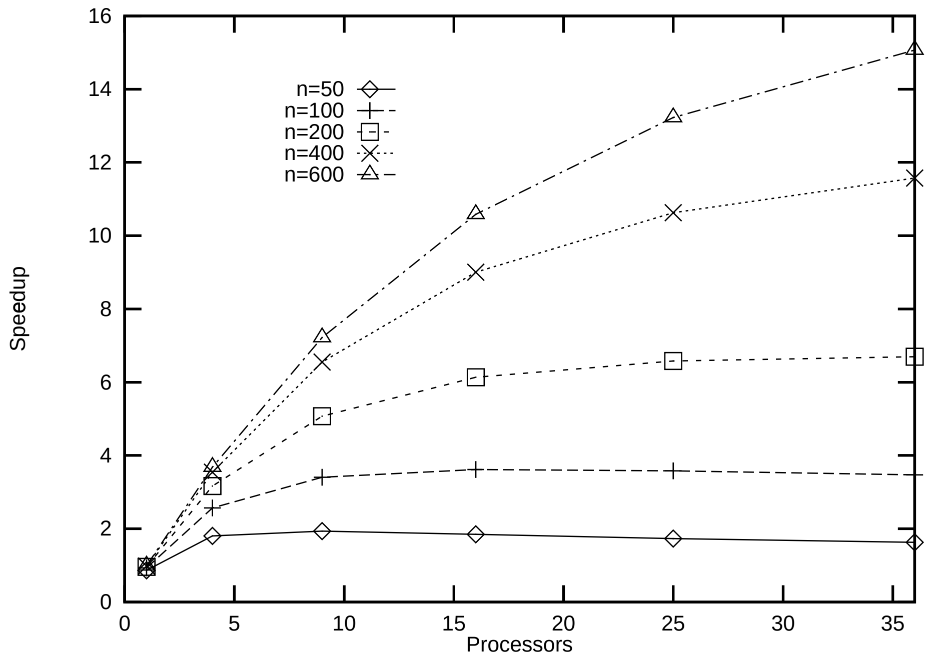


Figure 5.2: Speedup of algorithm as n varies for T9000/C104 architecture

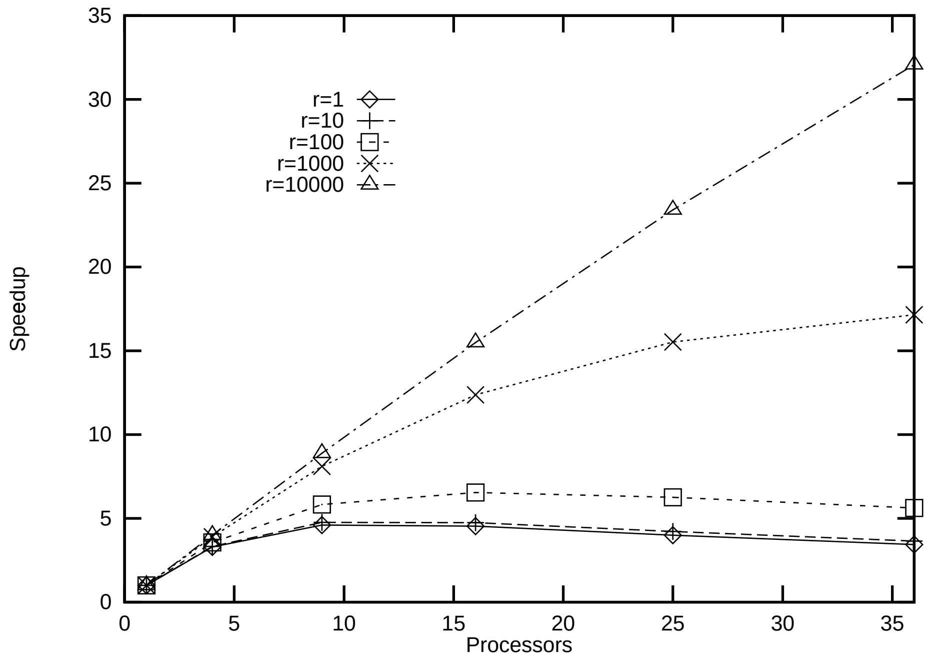


Figure 5.3: Speedup of algorithm as r varies for T8 architecture

be achieved with these algorithms. Firstly, this is because n determines the total size of the matrix for the Gaussian elimination step which puts an upper limit on the number of processors that can be used for that step of the algorithm when each processor has a single row of the matrix, i.e., $p = n$. However, the graphs show clearly that the best performance is obtained for the Gaussian elimination step for a much smaller number of processors which is indicated by the maximum point of the curves. The parameter n also determines the number of parallel gradient and Hessian evaluations that can be performed since the total number of these operations is n and n^2 respectively. If the total cost of function evaluations is small compared with the Gaussian elimination cost as in the case of this function then the algorithm should use the number of processors which optimises the performance of the Gaussian elimination step. However, we must rethink this strategy if function evaluation costs dominate the total cost of the algorithm.

To view the effects of function evaluation cost on the performance of the algorithms we again use Rosenbrock's extended function as the objective function, but instead of using the analytical gradient and Hessian functions we estimate elements of the gradient vector and Hessian matrix using finite difference approximations. This increases the cost of these calculations considerably. In addition, we artificially increase the cost of all three functions by repeating each evaluation r times. These changes give us an example problem with variable function evaluation costs as well as variable n.

Figures 5.3 and 5.4 show graphs of the modelled speedup of the T8 and T9 parallel algorithms for this new function. In these graphs we have kept n fixed at $n = 200$

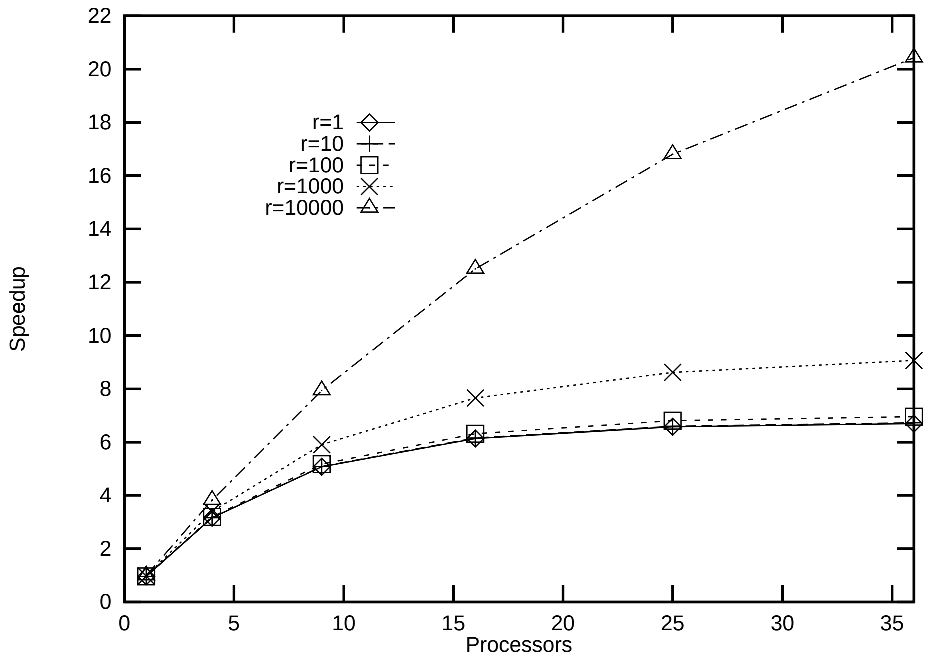


Figure 5.4: Speedup of algorithm as r varies for T9000/C104 architecture

and varied the function repeat count r. For small values of r the graphs closely follow the previous graphs since the total cost of the algorithm is still dominated by the cost of the Gaussian elimination. The Gaussian elimination cost is fixed as it depends only on n, and so as r increases the cost of function evaluations become more and more significant. Hence, these graphs show how well the algorithms parallelise the function evaluations. For both the T8 and T9000/C104 architecture the speedup increases as the repeat count increases. This is to be expected since the parallel function evaluations of steps 1 and 2 do not involve any communication overheads. It is worth noticing that the rate of improvement in performance is greater for the T8 architecture than the T9000/C104, with the former reaching a speedup around 32 for the largest problem size shown whilst the latter only achieves a speedup of about 20 for the same problem. This is because for a given increase in the total computation cost at fixed communication cost the architecture with the highest ratio of computation cost to communication cost will gain the greatest improvement in performance. As stated above the T8 architecture has the larger ratio of these values. The better speedups achieved by the T8 architecture need to be balanced against the fact that the T8 architecture is about ten times slower than the T9000/C104 architecture. This feature of these algorithms illustrates nicely the principle that one way to design an architecture which gives good speedups is to give the architecture a very poor computation rate compared with its communication rate. This is also why it is important when comparing different architectures to look at the absolute performance of algorithms as well as the speedup.

As the cost of the parallel function evaluations comes to dominate the cost of the total algorithm it will become worthwhile to determine the number of processors used not by the optimum number required for the Gaussian elimination step, which will be significantly less than n, but by the maximum number of processors that can best be used for parallel function evaluations, which is n. This balances the decreased performance of the Gaussian elimination step with increases in the speed of parallel function evaluations. If the number of processors available exceeds n then the Hessian element calculations can be redesigned so that a processor can calculate a single element of the Hessian matrix instead of a whole row as is currently the case. This change would also introduce additional communications to re-distribute the elements of the Hessian matrix to the locations expected by the Gaussian elimination algorithm.

Limiting the number of processors that can be used to n may have important consequences for many practical applications. Many real world objective functions only have a small value of n in the tens. This limits the number of processors that can be used and hence limits the maximum speedup that can be obtained however many processors are available to solve the problem. This is particularly important since many of these objective functions have very high run-time costs. If only a small number of processors can be used to solve the problem then even the parallel run-time of the algorithm may be too large to be practical.

5.9 Conclusions

This chapter has shown that the communications system of the T9000/C104 architecture gives a significant boost to the performance of algorithms compared with the best T8 configurations. For the Newton method presented here we have seen that both algorithms give reasonable performance. The performance is particularly good when the function evaluations are expensive compared with the dimension of the problem space n.

Further work should exploit the symmetry of the Hessian matrix in both the evaluation of the matrix and in the solution of the linear system. The latter will involve replacing the Gaussian elimination algorithm by a Choleski factorisation of the matrix. This factorisation will need to handle indefinite matrices. When T9000 machines become available it will be worthwhile implementing the parallel algorithm to establish the validity of the cost model.

Chapter 6

BFGS

In the previous chapter we studied the way in which parallel function evaluations can be used in the implementation of a parallel Newton method for unconstrained optimisation. In this chapter we examine quasi-Newton methods, and in particular we look at ways to parallelise the BFGS update. Alongside the parallel linear algebra algorithms developed in this chapter, a full quasi-Newton method can also exploit the parallel function evaluation technique of the previous chapter. We describe and compare three parallel algorithms for the BFGS update designed for a T9000/C104 based machine. These algorithms are distinguished by the way in which Hessian information is stored. We present run-time cost models for each algorithm and use these models to choose the best parallel algorithm for the target T9000/C104 architecture. Finally we discuss an initial implementation of this algorithm running under the VCR [30].

6.1 Introduction

An iteration, k say, of the quasi-Newton method for unconstrained optimisation may be stated as follows:

Algorithm 10 (Quasi-Newton method)

Let $\mathbf{x}_k$ be the current estimate of the minimum of the function $f(\mathbf{x})$. Also, let $\mathbf{p}_k$ be the current search direction.

1. *Test for convergence.*
   Terminate the algorithm if the convergence conditions are satisfied returning $\mathbf{x}_k$ as the solution.

2. *Compute a step length, λ_k, along the search direction $\mathbf{p}_k$.*

3. *Update estimate for the minimum.*
   Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{p}_k$.

4. *Update Hessian approximation and calculate next search direction* $\mathbf{p}_{k+1}$.
   Set $B_{k+1} = B_k + U_k$, and $\mathbf{p}_{k+1} = -B_{k+1}^{-1}\mathbf{g}_{k+1}$.

5. *Set* $k = k + 1$ *and go to step 1.*

□

Steps 2 and 4 account for most of the cost of the algorithm. Both of these steps can take advantage of parallel function evaluations discussed in the previous chapter. In this chapter we examine ways to parallelise the two linear algebra calculations of step 4: the update of the Hessian approximation, B_k, and the computation of the next search direction, $\mathbf{p}_{k+1}$.

Once a new point, $\mathbf{x}_{k+1}$, is found the Hessian approximation, B_k, is updated to reflect the new curvature information obtained:

$$B_{k+1} = B_k + U_k.$$

The update matrix, U_k, is chosen so that B_{k+1} satisfies the quasi-Newton condition:

$$B_{k+1}\mathbf{s}_k = \mathbf{y}_k,$$

where $\mathbf{s}_k$ is the change in x ($\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$), and $\mathbf{y}_k$ is the change in gradient ($\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$) in iteration k. In addition, updates B_{k+1} must possess the property of hereditary symmetry. This requires that if B_k is symmetric then B_{k+1} must be symmetric also. In most quasi-Newton methods B is positive definite (such methods are often called variable metric methods).

The simplest update matrix which satisfies these conditions is the rank one update matrix. From this we obtain the symmetric rank one update:

$$B_{k+1} = B_k + \frac{(\mathbf{y}_k - B_k\mathbf{s}_k)(\mathbf{y}_k - B_k\mathbf{s}_k)^T}{(\mathbf{y}_k - B_k\mathbf{s}_k)^T\mathbf{s}_k}.$$

This update has some drawbacks so in general rank two updates are preferred. The most popular rank two updates are the Davidon-Fletcher-Powell (DFP) update:

$$B_{k+1} = B_k - \frac{B_k\mathbf{s}_k\mathbf{s}_k^T B_k}{\mathbf{s}_k^T B_k\mathbf{s}_k} + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k} + (\mathbf{s}_k^T B_k\mathbf{s}_k)\mathbf{w}_k\mathbf{w}_k^T,$$

where

$$\mathbf{w}_k = \frac{\mathbf{y}_k}{\mathbf{y}_k^T\mathbf{s}_k} - \frac{B_k\mathbf{s}_k}{\mathbf{s}_k^T B_k\mathbf{s}_k},$$

and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update:

$$B_{k+1} = B_k - \frac{B_k\mathbf{s}_k\mathbf{s}_k^T B_k}{\mathbf{s}_k^T B_k\mathbf{s}_k} + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}.$$

	matrix unfactored	matrix factored
B	store full B $B_{k+1} = B_k +$ rank two form Choleski factors 2 triangular solves for $\mathbf{p}_{k+1}$	store L where: $B = LL^T$ $J_{k+1} = L_k +$ rank one $Q_{k+1}L_{k+1}^T = J_{k+1}$ by Jacobi rotations 2 triangular solves for $\mathbf{p}_{k+1}$
H	store full H $H_{k+1} = H_k +$ rank two m-v multiply for $\mathbf{p}_{k+1}$	store J where: $H = JJ^T$ $J_{k+1} = J_k +$ rank one 2 m-v multiplies for $\mathbf{p}_{k+1}$

Table 6.1: Four methods for implementing the BFGS update

It is generally accepted that the most effective of these is the BFGS update, and so this chapter considers the performance of parallel BFGS updates on the T9000/C104 architecture. Further details of this and other updates can be found in Fletcher[85, Chapter 3] and Gill, Murray & Wright[49, Section 4.5.2].

The Hessian approximation is used to calculate the search direction for the next iteration. For this reason the combined cost of the update and calculation of the next search direction should be considered when comparing methods.

The next search direction can be computed in two ways, either using the Hessian approximation itself:

$$B_{k+1}\mathbf{p}_{k+1} = -\mathbf{g}_{k+1}, \tag{6.1}$$

or by using the inverse Hessian approximation, H_{k+1}:

$$\mathbf{p}_{k+1} = -H_{k+1}\mathbf{g}_{k+1}. \tag{6.2}$$

Equation 6.1 involves the solution of a system of linear equations at a cost of $O(n^3)$, where n is the size of the square Hessian matrix, whilst using the inverse Hessian in Equation 6.2 only requires a matrix vector multiply costing $O(n^2)$. However, if the Hessian matrix is stored as Choleski factors, LL^T, then Equation 6.1 can also be solved in $O(n^2)$. Such considerations lead to four methods for performing the BFGS update which vary depending on how Hessian information is held: storing the Hessian matrix or inverse Hessian matrix, each one factored or unfactored. See Table 6.1 for a summary of these four methods.

Updating an unfactored inverse Hessian (we shall call this Method I) was the most popular method in early implementations since it avoided the cost of solving a linear system of equations. But, in addition to the costs of the various update methods an important consideration is maintaining positive definiteness of the matrix. It is claimed that it is easier to recognise and correct an indefinite matrix when the Choleski factors are stored and this has lead to many recent implementations updating a factored Hessian (Method II). The factored inverse Hessian update (Method III) has not received much attention due to doubts about its numerical stability. However Byrd, Schnabel & Schultz[18] and others report no significant numerical differences between any of the methods. We have chosen to examine parallel algorithms for these three methods, neglecting the unfactored Hessian update because of its significantly higher sequential cost.

To predict the performance of these algorithms on a T9000/C104 machine we have developed run-time cost models. These cost models use the same hardware constants and problem parameters as the other T9000/C104 cost models developed in this work, i.e., the three hardware constants:

T_f	the time taken to perform a double precision arithmetic operation, $T_f = 100$ns;
T_s,T_c	the total time taken to communicate a message of n double precision values is given by $T_s + nT_c$, $T_s = 1\mu$s, $T_c = 888$ns;

and the two problem parameters: n, the size of the Hessian matrix, and p the number of processors. (See Chapter 2 for details about the hardware parameters.)

In the preceding chapters, the algorithms designed for T8 networks had two distinct classes of process: a “master” process and “slave” processes. The master process was so named because it provided data and accepted results from the slave processes and controlled execution of the parallel algorithm, but did not usually take part in the computation itself. This process structure was well matched to T8 transputer systems which generally consisted of an array of processors connected to a separate control processor by a single link. In this chapter we experiment with a different process structure in which all p processes are expected to take part in the computation, and one process is additionally the source of data and destination for the result. This structure assumes that the user’s master processor is part of the T9000/C104 network and not a separate T9000. In this case the C104 switch network makes no distinction between the user’s master processor and slave processors and can provide the master processor with the same connectivity as the slave processors.

All three of these parallel algorithms use fundamental communication primitives, namely the broadcast, scatter and gather of data. A broadcast of a vector of n values from one process to $p - 1$ others is implemented as follows: the source communicates the vector to four other processes in parallel on its four links. Then each of these five processes (including the source) communicate the data to four more processes in parallel. This continues for $\lceil \log_5 p \rceil$ steps when all p processes have a copy of the data. The cost for this operation is $\lceil \log_5 p \rceil (T_s + nT_c)$. The scatter of n values between p processes (including the source process) consists of $(p - 1)/4$ steps in each of which the source process sends n/p different elements to four different processes on the four links. This has a cost of $\frac{p-1}{4}\left(T_s + \frac{n}{p}T_c\right)$. Gathering data uses the reverse of the scatter algorithm and has the same cost. Full details of these and other communication algorithms are given in Appendix A.

In the following three sections we describe parallel algorithms for each of the three update methods. The chapter then finishes with a comparison of the methods and discussion of the implementation of the best method in Section 6.5.

6.2 Method I: unfactored inverse Hessian update

The unfactored inverse Hessian update may be expressed as:

$$H_{k+1} = H_k + \frac{(\mathbf{s}_k - H_k\mathbf{y}_k)\mathbf{s}_k^T + \mathbf{s}_k(\mathbf{s}_k - H_k\mathbf{y}_k)^T}{\mathbf{y}_k^T\mathbf{s}_k} - \frac{(\mathbf{s}_k - H_k\mathbf{y}_k)^T\mathbf{y}_k\mathbf{s}_k\mathbf{s}_k^T}{(\mathbf{y}_k^T\mathbf{s}_k)^2}.$$

This is followed by a matrix vector multiplication to find the new search direction:

$$\mathbf{p}_{k+1} = H_{k+1}\mathbf{g}_{k+1}.$$

Byrd, Schnabel & Schultz[18] present a sequence of operations to perform these two steps which is cheaper than a direct implementation of these equations. We base our algorithm on their method:

Algorithm 11 (Method I)

1. $\mathbf{t} = H_k\mathbf{g}_{k+1}$
2. $\mathbf{z} = \mathbf{s}_k - \mathbf{t} + \mathbf{p}_k$
3. $\gamma = \mathbf{s}_k^T\mathbf{y}_k$
4. $\delta = \mathbf{z}^T\mathbf{y}_k$
5. $\mathbf{z}' = (\mathbf{z} - (\delta/2\gamma)\mathbf{s}_k)/\gamma$
6. $H_{k+1} = H_k + \mathbf{z}'\mathbf{s}_k^T + \mathbf{s}_k\mathbf{z}'^T$
7. $\gamma' = \mathbf{s}_k^T\mathbf{g}_{k+1}$
8. $\delta' = \mathbf{z}'^T\mathbf{g}_{k+1}$
9. $\mathbf{p}_{k+1} = \mathbf{t} + \gamma'\mathbf{z}' + \delta'\mathbf{s}_k$

□

Since H is symmetric only its lower or upper triangle need be stored. In this case the total sequential cost of the algorithm is $(4n^2+17n-1)T_f$. If the full matrix is stored the cost of the rank two update is increased and the total cost becomes $(6n^2+15n-1)T_f$. The cost of the method is dominated by the matrix vector multiply of step 1 and rank two update of step 6. Both of these steps involve the inverse Hessian approximation H and so in our parallel algorithm we need to consider the storage of H carefully since the choice of storage will affect the costs of steps 1 and 6.

For a parallel algorithm we will consider storage of both the full matrix H and triangular matrix. The update of step 6 should be cheaper if only a triangle is stored due to the smaller number of elements each processor must update. However, for triangular storage in step 1 a lot of extra communication will be required to give complete rows of H to the processors.

The storage schemes for these two algorithms are thus as follows: the inverse Hessian approximation H is stored either in full or just the upper or lower triangle with rows distributed to the processors. If a triangle only is stored then pairs of rows equidistant from the central row are given to each processor to try and balance the number of elements held by each process. Both algorithms would probably in practice require enough storage for a full matrix since the triangular matrix algorithm needs temporary storage in step 1 for complete rows of the matrix. Distributing the matrix by columns would increase the cost of step 1 by introducing more communications and so is not considered further. Block storage instead of row or column storage would increase the maximum number of processes that could be used to solve a problem of a given size. However the resulting small granularity would lead to poorer performance as the amount of communication compared with computation on each process is increased.

The vectors $\mathbf{p}_k$, $\mathbf{g}_{k+1}$, $\mathbf{s}_k$, $\mathbf{y}_k$, $\mathbf{t}$ and $\mathbf{z}$ are distributed across the processes with each process storing those elements of the vectors for which it also holds the row of the inverse Hessian H. The vectors $\mathbf{z}'$ and $\mathbf{p}_{k+1}$ overwrite $\mathbf{z}$ and $\mathbf{p}_k$. Four temporary distributed vectors are also required for communication of vector slices of $\mathbf{g}_{k+1}$, $\mathbf{s}_k$ and $\mathbf{z}$ during steps 1 and 6. We assume that at the start of an iteration of the algorithm $\mathbf{y}_k$ holds the old value of $\mathbf{g}_{k+1}$ and $\mathbf{p}_k$ holds the old value of $\mathbf{p}_{k+1}$, but all other vectors are undefined.

Parallel Method I starts with the process which holds the minimum point, $\mathbf{x}_{k+1}$, scattering the vectors $\mathbf{g}_{k+1}$ and $\mathbf{s}_k$. This has a cost of $(p-1)(T_s + (n/p)T_c)/2$. The distributed vector $\mathbf{y}_k$ is then calculated from: $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{y}_k$, costing $(n/p)T_f$.

The next step is the calculation of the distributed vector $\mathbf{t}$. For the full matrix this proceeds in p steps as follows: In each step (except the last) the processes pass segments of $\mathbf{g}_{k+1}$ to a neighbour in a ring of processes. These segments are held in a temporary vector. In parallel with this each process updates its segment of $\mathbf{t}$ using the segment of $\mathbf{g}_{k+1}$ which it input into another temporary vector in the preceding step. With current estimates of the hardware parameters this hides the communication behind computation except in the last iteration. The total cost for the p steps is:

$$\max\left(T_s + \frac{n}{p}T_c, \frac{2n^2}{p^2}T_f\right)(p-1) + \frac{2n^2}{p^2}T_f.$$

If instead a triangular matrix is stored then complete rows of the inverse Hessian must be gathered to the processes. Each process must output all elements it holds except elements in columns for which it also holds the row. Assuming an even distribution of elements between the processes then the elements a process outputs are distributed equally between the other processes. Each process must also input the same number of elements from the other processes. If each process has n/p rows of the triangular matrix distributed as described previously we approximate the number of elements held by each process as $n(n+1)/2p$. Of these $(n/p+1)n/2p$ do not need to be output. Hence the number of elements that a process must send to each other process is

$$\frac{1}{p-1}\left(\frac{n(n+1)}{2p} - \frac{n(n/p+1)}{2p}\right) = \frac{n(n-p)}{2p^2}.$$

Each process inputs $(p-1)$ messages and outputs $(p-1)$ messages using all four links giving a communication cost of

$$\frac{p-1}{2}\left(T_s + \frac{n(n-p)}{2p^2}T_c\right).$$

The computation of $\mathbf{t}$ then takes place as for the full matrix algorithm above. It may be possible to arrange for columns of the matrix to be gathered to the processes at each step of the algorithm thus reducing the storage requirement and perhaps hiding that communication behind the computation, however for simplicity this is not considered here.

The calculation of $\mathbf{z}$ which follows does not require any communication since each process can calculate those elements of $\mathbf{z}$ for which it holds corresponding elements of $\mathbf{s}_k$, $\mathbf{t}$ and $\mathbf{p}_k$ at a cost of $(2n/p)T_f$.

The two inner product calculations for γ and δ in steps 2 and 3 are executed in parallel, but for simplicity we assume for the cost model that they are executed sequentially. The first stage in the inner product calculation is for all the processes to form the partial inner product for the elements of the vectors which they hold. This costs $(2n/p-1)T_f$. This is followed by $\lceil \log_5 p \rceil$ steps in each of which groups of 5 processes communicate their partial inner products. One of the processes inputs the partial products from the other 4 processes on its 4 links and sums them with its own partial product. In the next step this process then outputs the new partial product to another process. The communications form a tree structure with 4 branches at each node. Each step i, $i = \lceil \log_5 p \rceil \ldots 1$, of the operation involves processes on level i of the tree outputting their partial products to their parent process at level $i-1$. In the last step one process forms the complete inner product of the distributed vectors. This communication phase has a cost of $(4T_f + T_s + T_c)\lceil \log_5 p \rceil$ giving a total cost for a single distributed inner product of

$$(2n/p-1)T_f + (4T_f + T_s + T_c)\lceil \log_5 p \rceil .$$

We execute the two inner product calculations so that the results end up at the same process, which is the master process. This process then calculates $\delta/2\gamma$ and broadcasts it, together with γ, to all the processes at a cost of $2T_f + 2(T_s + T_c)\lceil \log_5 p \rceil$. The calculation of $\mathbf{z}'$ follows costing $(3n/p)T_f$.

The inverse Hessian update of step 6 uses the same technique of communication hiding as step 1 does, proceeding in p steps as follows: In each step (except the last) the processes pass segments of $\mathbf{z}'$ and $\mathbf{s}_k$ to a neighbour in a ring of processes. These segments are held in two temporary vectors. In parallel with this each process updates its elements of the inverse Hessian using the segments of $\mathbf{z}'$ and $\mathbf{s}_k$ which it input into another pair of temporary vectors in the preceding step, along with its own segments of $\mathbf{z}'$ and $\mathbf{s}_k$. For full matrix storage each process updates n^2/p^2 elements of H giving a total cost for the step of

$$\max\left(T_s + \frac{2n}{p}T_c, \frac{4n^2}{p^2}T_f\right)(p-1) + \frac{4n^2}{p^2}T_f.$$

If a lower triangle is stored only approximately half the number of elements are updated at each stage giving a cost of

$$\max\left(T_s + \frac{2n}{p}T_c, \frac{2n^2}{p^2}T_f\right)(p-1) + \frac{2n^2}{p^2}T_f.$$

The next two operations which calculate γ' and δ' have the same cost as the earlier distributed inner products. These operations are followed by the broadcast of γ' and δ' costing $(T_s + 2T_c)\lceil\log_5 p\rceil$ ready for the calculation of the next search direction $\mathbf{p}_{k+1}$. The distributed vector $\mathbf{p}_{k+1}$ is calculated at a cost of $(4n/p)T_f$. The distributed partitions of $\mathbf{p}_{k+1}$ are then gathered together at the master process in a similar manner to the scatter algorithm costing $(p-1)(T_s + (n/p)T_c)/4$.

6.3 Method II: factored Hessian update

The factored Hessian update is the most widespread sequential BFGS update method. It avoids the large cost of solving a linear system of equations to find the next search direction by storing the Choleski factors LL^T of the Hessian approximation B. Methods have been developed to update the factors directly to the factors of the updated Hessian approximation. The papers by Brodlie et al. [67] and Gill et al. [81] give a good introduction to the technique whilst Goldfarb [50] presents details of the most efficient factored update algorithms.

If we store the factors LL^T of B the Hessian update can be expressed as

$$B_{k+1} = J_{k+1}{J_{k+1}}^T, \quad \text{where} \quad J_{k+1} = (L_k + \mathbf{v}_k\mathbf{u}_k^T),$$

with the vectors $\mathbf{u}_k$ and $\mathbf{v}_k$ for the BFGS update given by:

$$\begin{aligned} \mathbf{u}_k &= \mathbf{y}_k - \alpha L_k L_k^T \mathbf{s}_k, \quad \text{where} \quad \alpha = \left(\frac{\mathbf{y}_k^T\mathbf{s}_k}{\mathbf{s}_k^T L_k L_k^T \mathbf{s}_k}\right)^{1/2}, \\ \mathbf{v}_k &= \frac{L_k^T\mathbf{s}_k}{\sqrt{\mathbf{y}_k^T\mathbf{s}_k.\mathbf{s}_k^T L_k L_k^T \mathbf{s}_k}}. \end{aligned}$$

If we then calculate the factorisation $Q_{k+1}R_{k+1}$ of ${J_{k+1}}^T = (R_k + \mathbf{u}_k\mathbf{v}_k^T)$ the updated

Choleski factor L_{k+1} is given by R^T_{k+1}. The QR factorisation is performed by two sequences of Jacobi rotations. The first $n-1$ Jacobi rotations transform $\mathbf{u}_k\mathbf{v}^T_k$ to the matrix $\|\mathbf{u}_k\|_2\mathbf{e}_1\mathbf{v}^T_k$ and R_k to an upper Hessenberg matrix R^h_k. The following $n-1$ rotations then transform the matrix $R^h_k + \|\mathbf{u}_k\|_2\mathbf{e}_1\mathbf{v}^T_k$ to the upper triangular matrix R_{k+1} which then gives the updated Choleski factor L_{k+1}. An implementation of the algorithm does not need to transpose the matrices L_k and R_{k+1}. Instead, every reference to an element R_{ij} is replaced by a reference to the element L_{ji}. Hence the factored Hessian update method consists of calculating $\mathbf{u}_k$ and $\mathbf{v}_k$ followed by a QR factorisation performed by a sequence of Jacobi rotations.

Once the updated Choleski factor L_{k+1} is known, the next search direction $\mathbf{p}_{k+1}$ can be calculated:

$$B_{k+1}\mathbf{p}_{k+1} = -\mathbf{g}_{k+1}.$$

Since the factors L_{k+1} of B_{k+1} are stored and not the approximate Hessian itself this equation can be solved by performing two Choleski solves:

$$\begin{aligned} L_{k+1}\mathbf{t} &= -\mathbf{g}_{k+1}, \\ L^T_{k+1}\mathbf{p}_{k+1} &= \mathbf{t}. \end{aligned}$$

The sequential factored Hessian update algorithm which follows is based on Alg A9.4.2. given in [35].

Algorithm 12 (Method II)

Calculate $\mathbf{u}_k$ and $\mathbf{v}_k$

```
1. $\beta = \mathbf{y}^T_k\mathbf{s}_k$
2. $\mathbf{v}_k = L^T_k\mathbf{s}_k$
3. $\gamma = \mathbf{v}^T_k\mathbf{v}_k$
4. $\alpha = +\sqrt{\frac{\beta}{\gamma}}$
5. $\mathbf{u}_k = \mathbf{y}_k - \alpha L_k\mathbf{v}_k$
6. $\delta = \frac{1}{\sqrt{\beta\gamma}}$
7. $\mathbf{v}_k = \delta\mathbf{v}_k$
```

QR factorisation (dropping k subscript)

```
8. For $i = n-1 \ldots 1$
       $J(n, i, u_i, -u_{i+1})$
       $u_i = +\sqrt{u_i^2 + u_{i+1}^2}$
```

```
9. For $i = 1 \ldots n$
       $L_{i,1} = L_{i,1} + u_1 v_i$

10. For $i = 1 \ldots n-1$
       $J(n, i, L_{i,i}, -L_{i,i+1})$
```

Where the Jacobi rotation $J(n,i,a,b)$ is given by:

```
J1. $\alpha = +\sqrt{a^2+b^2}$

J2. $c = a/\alpha$, $s = b/\alpha$

J3. For $j = i \ldots n$
       $\beta = L_{j,i}$
       $\gamma = L_{j,i+1}$
       $L_{j,i} = c\beta - s\gamma$
       $L_{j,i+1} = s\beta + c\gamma$
```

Calculate next search direction: solve $L_k L_k^T \mathbf{p}_{k+1} = \mathbf{g}_{k+1}$ (dropping k subscript)

```
11. $t_1 = {g_{k+1}}_1 / L_{1,1}$

12. For $i = 2 \ldots n$
       $t_i = \frac{{g_{k+1}}_i - \sum_{j=1}^{i-1} L_{i,j} t_j}{L_{i,i}}$

13. ${p_{k+1}}_n = t_n / L_{n,n}$

14. For $i = n-1 \ldots 1$
       ${p_{k+1}}_i = \frac{t_i - \sum_{j=i+1}^{n} L_{j,i} {p_{k+1}}_j}{L_{i,i}}$
```

15. $\mathbf{p}_{k+1} = -\mathbf{p}_{k+1}$

□

When calculating the cost of this algorithm we must take careful note of the cost of the square root operations. In the past all floating point operations were implemented in software. Because of this multiplication and division cost significantly more than addition and subtraction, and the square root function itself had a huge cost compared with multiplication. For this reason much effort was expended to design algorithms which required no square root evaluations or at least kept these to a minimum. More recent microprocessor designs incorporate hardware floating point arithmetic units to improve the performance of numerical calculations. With these units the cost of simple arithmetic operations such as add and multiply are comparable. However, even when the square root function has hardware assistance it still has a cost which is larger than the simple operations. In this paper we model the cost of the square root function, T_{sq}, as a multiple of the simple arithmetic operation cost, i.e. $T_{sq} = \alpha T_f$. Using cycle times given in [64] a value of 4 is given to α.

For a sequential implementation of the algorithm, the initial calculation of $\mathbf{u}_k$ and $\mathbf{v}_k$ costs $(2n^2 + 7n + 1)T_f + 2T_{sq}$. The QR factorisation, which updates L instead of R, has a cost of $(6n^2 + 18n - 22)T_f + 2(n-1)T_{sq}$. If each negation operation of step 15 costs $1T_f$ then the calculation of the next search direction costs $n(2n+1)T_f$. This gives a total cost for the algorithm of $(10n^2 + 26n - 21)T_f + 2nT_{sq}$.

The first consideration in developing the parallel algorithm is the distributed storage of the Choleski factor L. We have again restricted our consideration to storage by rows or columns, excluding block storage. The expensive steps in the algorithm involving L are steps 2, 5, 8, 10, 12 and 14. Of the two matrix-vector multiplications of steps 2 and 5, one involves L and the other L^T so the combined cost is not affected by the choice of row or column storage. However, it may be worthwhile to store L by *both* rows and columns, which is identical to using the same storage method for both L and L^T. This would allow the more time-efficient matrix-vector algorithm to be used for both steps. The disadvantages of this are the almost doubling of the storage requirement to n^2 and the additional time cost of either communicating the transpose of the matrix or updating both L and L^T. Similarly, the Choleski solves which include steps 12 and 14 use both L and L^T so the same argument applies here too.

In the QR factorisation each Jacobi rotation operates on 2 rows of R, that is 2 columns of L. The obvious storage method is thus to store L by rows allowing all processes in parallel to update their elements of the 2 columns within a single Jacobi rotation. This requires only the broadcast of c and s at each rotation. Alternatively, if column storage of L is used then each rotation would require the communication of all elements to be updated from one process to another and then at most two processes performing the update. This very poor parallelism could be improved by beginning the next Jacobi rotation as soon as the essential element from the previous rotation had been calculated, however the large number of communications required for each rotation suggest that this method would not perform as well as using row storage.

For this work we look at the cost of a row storage algorithm for Method II. The data distribution scheme is as follows: The matrix L has its rows distributed in the same manner as used for the triangular matrix in Method I. Only space for a lower Hessenberg matrix is required giving this algorithm a great storage advantage over the other algorithms presented in this paper which require space for a full matrix. The vectors $\mathbf{s}_k$, $\mathbf{y}_k$, $\mathbf{u}_k$, $\mathbf{t}$, $\mathbf{g}_{k+1}$ and $\mathbf{p}_{k+1}$ are distributed with a process holding those elements of the vectors for which it also holds the row of L. Eight temporary vectors of size n/p are required on each process for the summation of vectors in step 2. Finally, an n-vector is stored in full on each process to hold partial results of $\mathbf{v}_k$ in step 2 and to store the partial sums from the Choleski solves in steps 12 and 14. We assume that at the start of an iteration of the algorithm $\mathbf{y}_k$ holds the old value of $\mathbf{g}_{k+1}$, but all other vectors are undefined.

The parallel algorithm starts by scattering $\mathbf{g}_{k+1}$ and $\mathbf{s}_k$ from the master process at a cost of $(p-1)(T_s + (n/p)T_c)/2$. Then $\mathbf{y}_k$ is calculated from: $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{y}_k$, costing $(n/p)T_f$. The distributed inner product of step 1 stores the result β on the master process and costs

$$(2n/p - 1)T_f + (4T_f + T_s + T_c)\left\lceil \log_5 p \right\rceil .$$

To calculate $\mathbf{v}_k$ each process first forms the partial sums (dropping k subscript) $v_i = \sum_{i=1}^{j} L_{j,i} s_j$ for each row, j, of L that it holds. This costs $(n^2/p + n/p)T_f$. The p vectors $\mathbf{v}_k$ are then summed to one process and the result vector scattered to the processes. The summation of vectors proceeds in m steps as follows: In each step, i, a process inputs a segment of the summation vector $\mathbf{v}_k$ beginning at index $(n/m)i$ and of size $d = n/m$ from its 4 children in a tree structure into 4 temporary vectors. In parallel with this the process adds the 4 vector segments input in the last step to the process's own segment at index $(n/m)(i-1)$ and the result segment is output to the parent process. This arrangement allows communications of segments to be executed in parallel with the summation. The cost to get the first segment summed at the root process is $(T_s + dT_c + 4dT_f)\left\lceil \log_5 p \right\rceil$. Reading in the remaining segments and adding them costs $\max(T_s + dT_c, 4dT_f)(m-1)$. The value of m should be chosen which give the least cost. Until the best value for m can be measured experimentally on a T9000/C104 system we shall use $m = p$. The result vector is then scattered back to the processes at a cost of $(p-1)(T_s + (n/p)T_c)/4$.

Step 3 is another distributed inner product leaving the result on the master process. The master process next calculates and broadcasts α for step 4 costing $T_f + T_{sq} + (T_s + T_c)\left\lceil \log_5 p \right\rceil$.

The matrix vector multiply of step 5 is similar to that in step 1 of Method I. The segments of $\mathbf{v}_k$ are cycled round a ring of processes, and at each step a process updates its elements of a temporary result vector. Assuming that at each step half of the full matrix elements involved in the update are zero then the total cost of the matrix vector multiply is

$$\max\left(T_s + \frac{n}{p}T_c, \frac{n^2}{p^2}T_f\right)(p-1) + \frac{n^2}{p^2}T_f.$$

The rest of the calculation of $\mathbf{u}_k$ in step 5 costs $(2n/p)T_f$.

The calculation and broadcast of δ costs $2T_f + T_{sq} + (T_s + T_c)\lceil\log_5 p\rceil$. In the final step in the initialization phase each process updates those elements of $\mathbf{v}_k$ for which it also holds the row of L at a cost of $(n/p)T_f$.

The second phase of the algorithm is the QR factorisation which is dominated by the Jacobi rotations $J(n, i, a, b)$. We parallelise rotation i as follows: For step 8 the process holding u_{i+1} communicates it to the process holding u_i. Then the process holding u_i calculates c and s and broadcasts these to the other processes. For step 10, the initial communication is not required since one process holds both $L_{i,i}$ and $L_{i,i+1}$. Next each process updates columns i and $i+1$ for those rows which it holds. Assuming the updates to L (step J3) are evenly distributed across processes then the time taken to perform all the updates in step 8 or 10 is $(3n^2 + 3n - 6)T_f/p$. To this we add the cost to calculate and broadcast c and s for $n-1$ Jacobi rotations which is $(n-1)(6T_f + T_{sq} + (T_s + 2T_c)\lceil\log_5 p\rceil)$ for step 10 and an additional $(n-1)(T_s + T_c)$ for step 8 to communicate u_{i+1}. There is no additional cost to update u_i since its new value was calculated within the Jacobi rotation. Step 9 requires u_1 to be broadcast giving a total cost for that step of $(2n/p)T_f + (T_s + T_c)\lceil\log_5 p\rceil$.

The calculation of the next search direction $\mathbf{p}_{k+1}$ in the final phase of the algorithm involves two parallel Choleski solves; one using the matrix L and the next using L^T. These require two different algorithms to solve because L is distributed. For solving $L_k\mathbf{t} = \mathbf{g}_{k+1}$ in steps 11 and 12 each process maintains a partial sum $L_{i,j}t_j$ for each row i of L that it holds and this is updated at each iteration as a new element of $\mathbf{t}$ is calculated. At iteration i in step 12 we proceed as follows: the process holding row i calculates t_i from its partial sum. This new element is then broadcast and all processes update their partial sums. Assuming that the partial sum updates take similar times on each process the total cost for steps 11 and 12 is

$$((n-1)^2/p + 2n - 1)T_f + (n-1)(T_s + T_c)\lceil\log_5 p\rceil\ .$$

In practice we would expect the broadcast of t_i to be overlapped with updating the partial sums which will give some improvement in performance.

The second Choleski solve involves L^T and hence is similar to a Choleski solve with L where L is distributed by columns instead of by rows. For this situation each process maintains a partial sum for every row of L^T. These partial sums hold the values $L_{j,i}\mathbf{p}_{k+1_j}$ for every row i of L^T and for those j for which a process holds row j of L. At the start of iteration i all processes take part in a summation of their partial sums for row i of L^T with the result finishing on the process holding row i of L. This process then calculates p_{k+1_i} and updates its partial sums before the next iteration begins. Since only one process in an iteration updates its partial sums the cost of this parallel algorithm as it stands is the cost of the sequential algorithm *plus* the cost of the distributed summations. In practice the algorithm should be coded such that as soon as the process has updated its sum $i-1$ the next iteration begins with the summation of partial sums $i-1$. In this way many processes may be updating their partial sums at a time as well as communicating for the summation of distributed partial sums. Unfortunately, this appears to be difficult to code since the time taken to perform the partial updates varies as the iterations proceed making efficient synchronisation between the communicating and updating processes difficult to achieve.

It is also very difficult to propose a cost model for this algorithm other than the worst case sequential one. A workable suggestion that will give a rough idea of the cost of this algorithm can be obtained by comparing the cost of performing an update in an iteration with the cost of the summing the partial sums and calculating a new element of $\mathbf{p}_{k+1}$. The worst case for an update involves a process updating $n-1$ partial summations at a cost of $2(n-1)T_f$. In the following $p-1$ iterations, since the rows of L are distributed cyclically, this process does not need to perform any other updates. However it does need to take part in the summation of the partial sums for these iterations. It seems reasonable therefore to compare the worst update cost with the cost of the following $p-1$ summations and calculations of $p_{k+1_j}, j = i-1 \ldots i-p$ which cost $(p-1)((T_s+T_c+4T_f)\lceil \log_5 p \rceil + 4T_f)$. With the current values for the hardware parameters the cost of the update is hidden by the cost of communications on medium to large networks ($p \geq 32$). So we assume that the partial sum updates execute in parallel with the communications and finish sooner allowing us to neglect the update cost in the cost model. This gives an estimated cost for steps 13 and 14 of

$$(2n-1)T_f + (n-1)(T_s+T_c+4T_f)\lceil \log_5 p \rceil .$$

Finally we negate the vector $\mathbf{p}_{k+1}$ and gather the full vector to the master process at a cost $(n/p)T_f + (p-1)(T_s + (n/p)T_c)/4$.

6.4 Method III: factored inverse Hessian update

This method has not received as much attention as other methods, but has been investigated by Davidon[28] and Powell[84]. As in Method II we express the inverse Hessian update in product form

$$H_{k+1} = (I + \mathbf{u}_k \mathbf{v}_k^T) H_k (I + \mathbf{u}_k \mathbf{v}_k^T)^T.$$

If we then store the factors JJ^T of H the update of the factors is given by

$$J_{k+1} = (I + \mathbf{u}_k \mathbf{v}_k^T) J_k, \quad \text{where,} \quad \begin{array}{l} \mathbf{u}_k = \frac{\mathbf{s}_k}{\mathbf{s}_k^T \mathbf{y}_k}, \\ \mathbf{v}_k = +\sqrt{\frac{\mathbf{s}_k^T \mathbf{y}_k}{\mathbf{s}_k^T H_k^{-1} \mathbf{s}_k}} H_k^{-1} \mathbf{s}_k - \mathbf{y}_k. \end{array}$$

This can be simplified since $H_k^{-1}\mathbf{s}_k = -\lambda\mathbf{g}_k$ and $\mathbf{s}_k H_k^{-1}\mathbf{s}_k = -\lambda\mathbf{s}_k\mathbf{g}_k$ where λ is the step length used in this iteration of the quasi-Newton algorithm. Once the factors have been updated a new search direction is calculated by performing two matrix vector multiplications:

$$\mathbf{p}_{k+1} = -J_{k+1}{J_{k+1}}^T\mathbf{g}_{k+1}.$$

The sequential algorithm is thus given by:

Algorithm 13 (Method III)

Calculate $\mathbf{u}_k$ and $\mathbf{v}_k$

1. $\alpha = \mathbf{s}_k^T\mathbf{y}_k$
2. $\beta = \mathbf{s}_k^T\mathbf{g}_k$
3. $\mathbf{u}_k = \mathbf{s}_k/\alpha$
4. $\gamma = \sqrt{\frac{-\lambda\alpha}{\beta}}$
5. $\mathbf{v}_k = -\gamma\mathbf{g}_k - \mathbf{y}_k$

Update J

6. $\mathbf{t}^T = \mathbf{v}_k^T J_k$
7. $J_{k+1} = J_k + \mathbf{u}_k\mathbf{t}^T$

Calculate next search direction

8. $\mathbf{t} = {J_{k+1}}^T\mathbf{g}_{k+1}$
9. $\mathbf{p}_{k+1} = -J_{k+1}\mathbf{t}$

□

The total cost for the sequential algorithm is $(8n^2 + 5n + 2)T_f + T_{sq}$.

The cost of a parallel implementation of this algorithm again depends on the storage scheme selected for the matrix J. The costly steps in this algorithm are steps 6, 7, 8 and 9. For steps 6 and 8 column storage is preferable allowing the same method to be used as is used in step 1 of Method I. Step 9, however, would be cheaper to perform if the matrix were stored by rows. The cost of the rank one update in step 7 is not affected by the selection of row or column storage. For the least cost, we choose to distribute columns of J to the processes, either cyclically or in block columns. The storage for the vectors is as follows: $\mathbf{s}_k$, $\mathbf{y}_k$, $\mathbf{u}_k$, $\mathbf{v}_k$, $\mathbf{t}$ and $\mathbf{g}_{k+1}$ are distributed to the processes whilst $\mathbf{p}_{k+1}$ is stored in full on each process. Eight temporary vectors of size n/p are required on each process for the summation of vectors in step 9. We assume that at the start of an iteration of the algorithm $\mathbf{y}_k$ holds the value of $\mathbf{g}_{k+1}$ from the previous iteration.

The first phase of the algorithm, starts with the scattering of $\mathbf{s}_k$ and $\mathbf{g}_{k+1}$ costing $(p-1)(T_s + (n/p)T_c)/2$. Then $\mathbf{y}_k$ can be calculated at cost $(n/p)T_f$ from $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{y}_k$. The two inner products follow each costing $(2n/p-1)T_f + (4T_f + T_s + T_c)\lceil \log_5 p \rceil$. Next, $-\gamma$ is calculated and broadcast along with α costing $4T_f + T_{sq} + (T_s + 2T_c)\lceil \log_5 p \rceil$. This allows the calculations of $\mathbf{u}_k$ and $\mathbf{v}_k$ to be performed at cost $(3n/p)T_f$.

Steps 6 and 8 both use the algorithm in step 1 of Method I. An additional cost of $(n/p)T_f$ is incurred in negating $\mathbf{t}$ at the end of step 8. The rank one update of step 7 is similar to the rank two update in step 6 of Method I and costs

$$\max\left(T_s + \frac{n}{p}T_c, \frac{2n^2}{p^2}T_f\right)(p-1) + \frac{2n^2}{p^2}T_f.$$

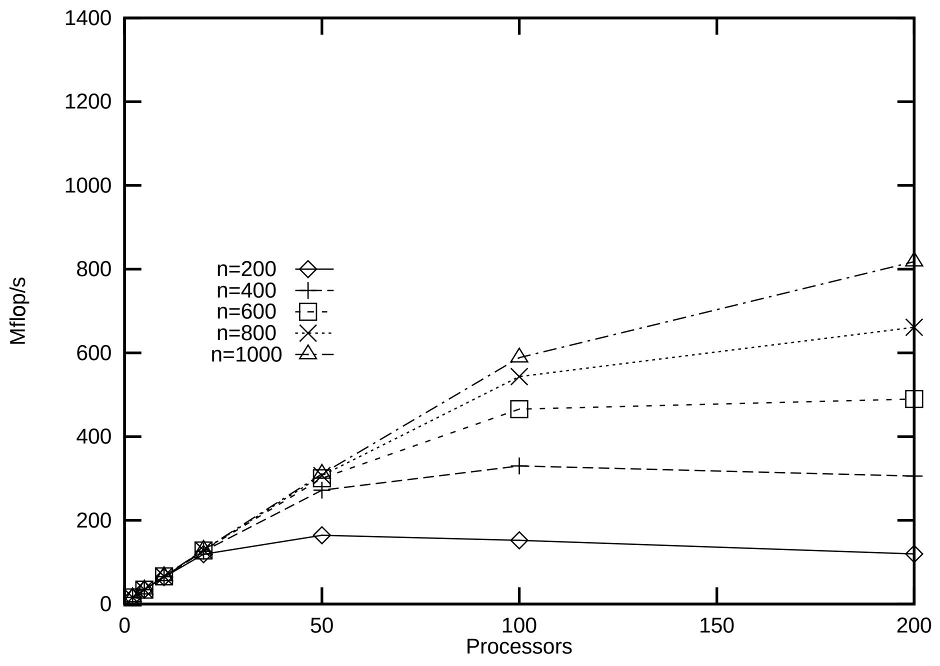
Step 9 is similar to step 2 of Method II except that in this case all elements of the full matrix are non-zero. This increases the cost of the initial partial summation on each process to $(2n^2/p)T_f$. Finally, $\mathbf{p}_{k+1}$ is gathered to the master process at cost $(p-1)(T_s + (n/p)T_c)/4$.

6.5 Comparison of methods

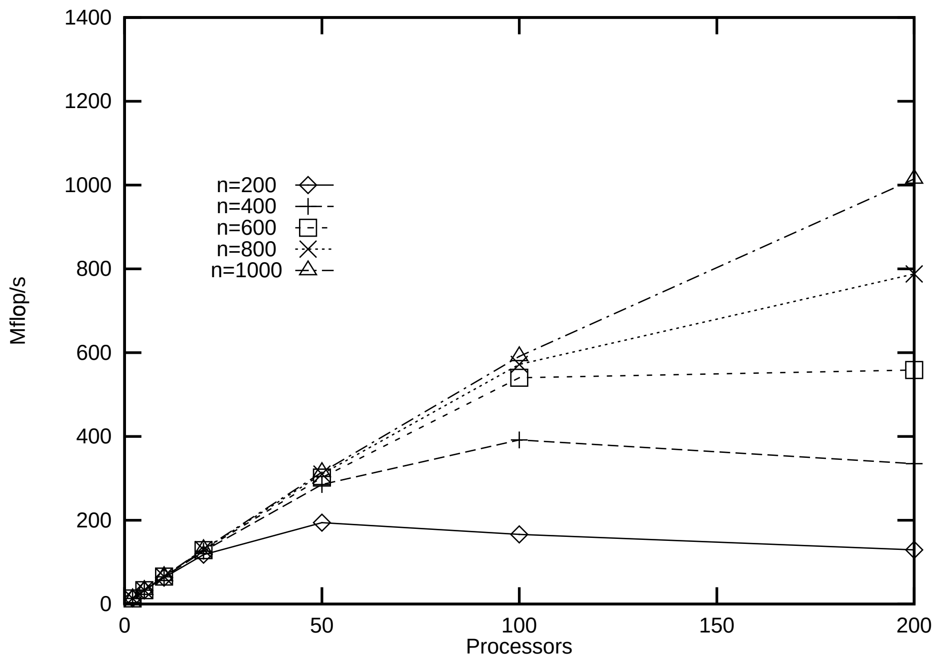
Figures 6.1 to 6.3 show graphs of the Mflop rate achieved by the algorithms for varying numbers of processors and problem sizes. The graphs show for each method the predicted Mflop rate for a given n and p. These rates should be compared with the Mflop rate of a single T9000 processor to see the speedup achieved. In this thesis we use a value of 10 Mflop/s for a single T9000. The efficiency of the parallel algorithm on p processors can be judged by comparing its Mflop rate with the maximum possible Mflop rate delivered by that number of processors. These graphs give us an indication of how efficient each parallel algorithm is and let us see how well the algorithms scale with the number of processors and problem size.

For large problem sizes ($n = 1000$) Methods I and III have good efficiencies around 60–70% for the numbers of processors shown. This indicates that both of these algorithms would scale well for big problems on big machines with Method III being preferred. The graph for Method II reveals a very poor performance. Even for large problems the algorithm achieves much lower efficiencies than the other two algorithms. Particularly important is the fact that the algorithm does not scale well as the number of processors is increased. The Mflop rate peaks at only 100 processors for the biggest problem size shown. Also for small networks the performance is not as good as for the other algorithms. Hence Methods I or III are to be preferred for all problem and network sizes.

There seem be two reasons for this contrast in performance between the algorithms. Firstly, the greater proportion of the sequential costs in Methods I and III is in level 2 BLAS operations. The computations required by these BLAS parallelise very well and



Triangular matrix storage



Full matrix storage

Figure 6.1: Performance of Method I

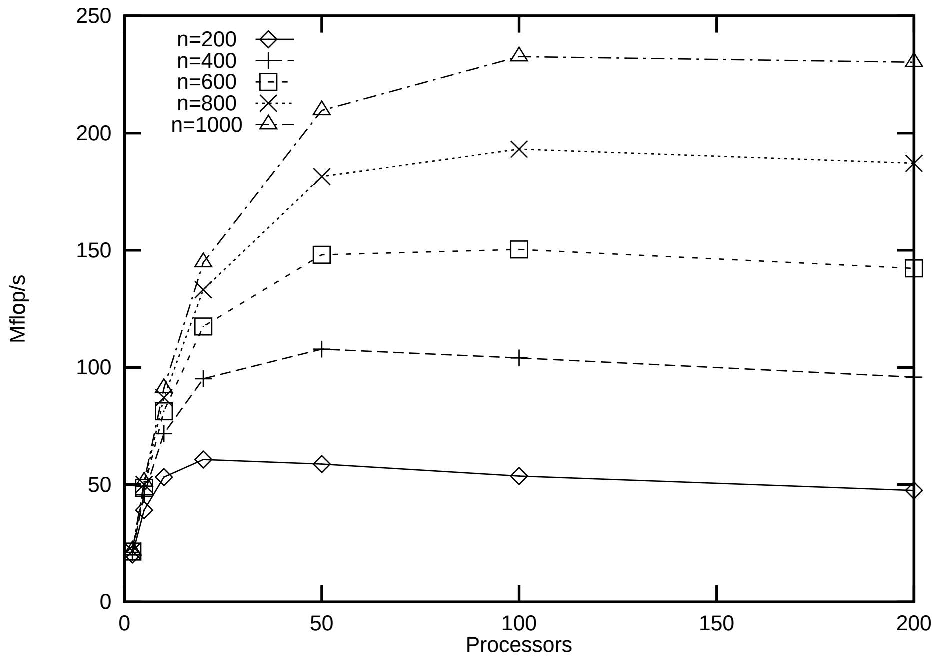


Figure 6.2: Performance of Method II

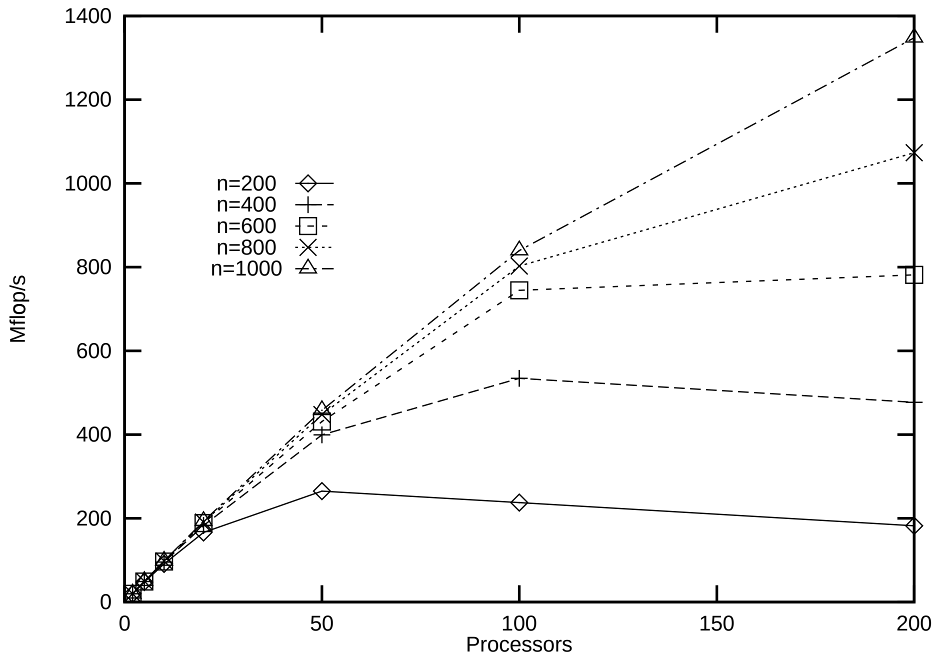


Figure 6.3: Performance of Method III

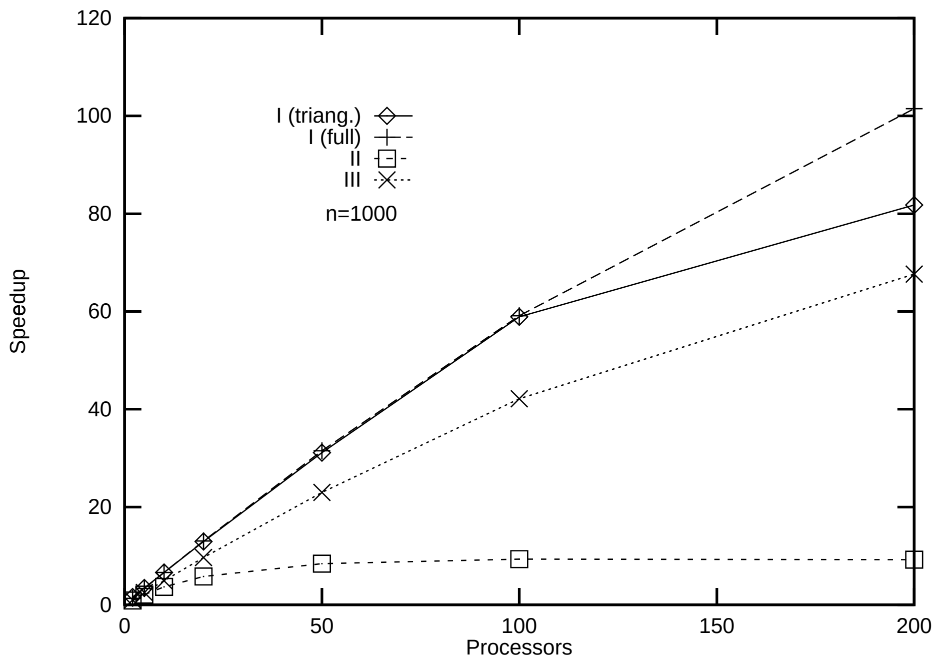
lead to efficient algorithms with few synchronizations for communication. Method II, however, contains costly Jacobi rotations and backward solves which are difficult to parallelise and require frequent communications between processes. The second factor giving Methods I and III much better performances than Method II is the way in which most of the more expensive communications in Methods I and III have been hidden behind arithmetic operations and therefore do not contribute much to the total cost of the algorithms. This is not achieved by Method II since it requires frequent small communications with only few arithmetic operations being performed in between each communication.

Figure 6.4 shows a comparison of the methods for two problem sizes. The algorithms are compared by plotting the speedup of each parallel algorithm when compared with the best sequential method, which is Method I. This gives a comparison of the "elapsed times" of the different algorithms. For large problems the dominant factor in determining the best parallel method is the cost of the method's sequential algorithm. This factor has more effect than the suitability of the sequential algorithm for parallelisation, or the details of the particular implementation of the parallel algorithm, such as whether full or triangular storage is used. Hence the top graph shows a clear distinction between the three methods with performance increasing from Method II (sequential cost $\approx 10n^2$) to Method III ($8n^2$) and then Method I ($4n^2$). There is relatively little difference in the performance of the two implementations of Method I compared with the other methods emphasising the importance of basing the parallel algorithm on the best sequential algorithm unless this is clearly unsuitable for parallelisation. The rapid fall in speedup when $n = 200$ and more than 50 processors are used indicates that the granularity of all these algorithms is quite coarse and each process should hold several (perhaps at least 4) rows of the matrix for good efficiency.

Another important factor to consider when comparing these algorithms is the amount of information which each algorithm provides. All of the algorithms give an updated Hessian or inverse Hessian matrix and the next search direction. In addition Method II, since it stores the factors LL^T of the Hessian, can easily provide information about the positive-definiteness of the matrix. This is essential for practical problems to ensure that the algorithm is robust. A further factor in favour of Method II is its lower memory requirement; Methods I and III require distributed storage for about n^2 words whilst Method II only needs about $n^2/2$. The main disadvantage of Method II is its large sequential cost. A more efficient sequential algorithm which stores the Choleski factors is given in [50]. This algorithm still has a higher cost than Method I and consists of many level 1 operations involving a lot of synchronisations but may well be worth future investigation due to the advantages of the method outlined above.

6.6 Implementation

We have implemented the parallel full matrix Method I BFGS update. This implementation runs under the VCR [30] on the Parsys Supernode. A listing of the slave process



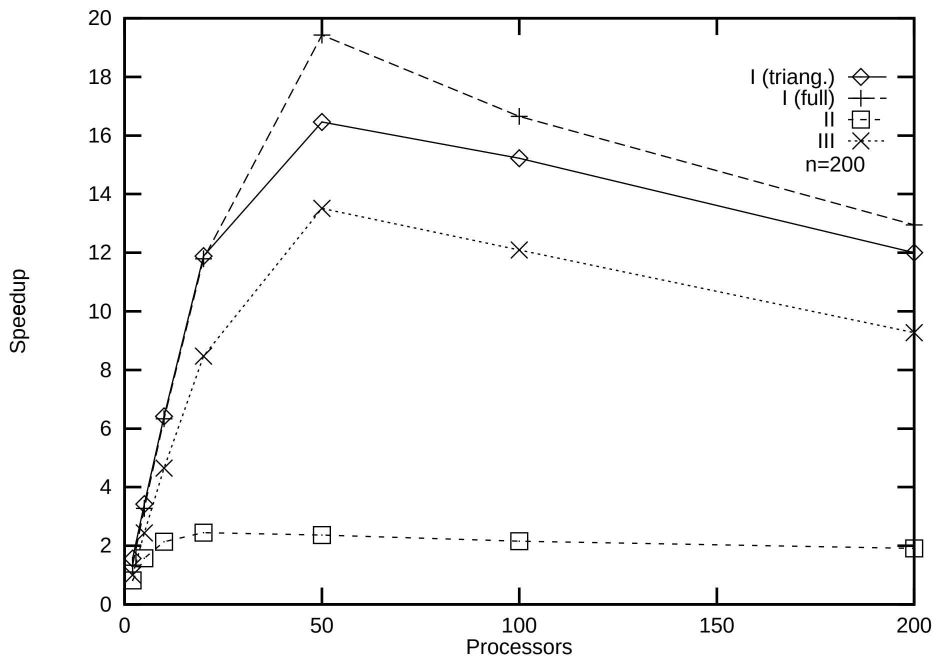


Figure 6.4: Comparison of methods

code and discussion of the programming techniques used in the program are given in Appendix B.

Tables 6.2 and 6.3 show the wall-clock execution time for the single precision algorithm on an 8×4 grid and a chain of processors respectively. Times were measured using the built-in timer on the master transputer and averaged over five program runs. All times are in seconds and are given to up to three significant figures. The variation in run-time between program runs is around 1%.

	n			
p	100	200	400	600
1	0.078	0.285	1.08	2.43
2	0.053	0.160	0.574	1.26
4	0.051	0.109	0.330	0.686
8	0.074	0.114	0.247	0.461
16	0.109	0.145	0.231	0.358
32	0.177	0.209	0.293	0.377

Table 6.2: Timings for parallel BFGS Method I on an 8×4 grid

	n			
p	100	200	400	600
1	0.078	0.285	1.08	2.43
2	0.061	0.169	0.583	1.27
4	0.063	0.121	0.339	0.695
8	0.103	0.142	0.278	0.497
16	0.182	0.224	0.332	0.472
32	0.446	0.511	0.656	0.811

Table 6.3: Timings for parallel BFGS Method I on a chain

Notice that the timings for the two processor configurations are different; this is especially noticeable when running the program on 32 processors. This variation in timings is due to the larger average number of hops that each message must make for the 1×32 grid compared with the 8×4 grid and the smaller bandwidth of the 1×32 grid.

Due to the different characteristics of the underlying communications of T8/VCR and T9000/C104 programs we do not expect our T9000/C104 run-time cost models to give a very good fit to the measured VCR timings. In fact, if we fit these results to our T9000/C104 run-time cost model we get: $T_f = 1.13\mu$s, $T_s = 373\mu$s and $T_c = 33.4\mu$s for the 8×4 grid and $T_f = 1.12\mu$s, $T_s = 1340\mu$s and $T_c = 54.8\mu$s for the 1×32 grid. These values are quite reasonable. The average number of interprocessor hops for communications on the 8×4 grid is about 5 hops. Interpolating performance information in the VCR user guide [30] suggests that we could very roughly expect the average message startup time to be around $T_s = 400\mu$s, and average transmission time to be about $T_c = 40\mu$s for single precision values with communications involving 5 hops. These values match quite well with the fitted values for the 8×4 grid (see Figure 6.5).

In general T9000/C104 run-time cost models will not predict the cost of T8/VCR programs well using fixed values for T_f, T_s and T_c. This is because the communication parameters will vary as the processor configuration is changed or larger networks are used. Thus parameter values for different topologies and network sizes would be needed to predict the run-time of T8/VCR programs. The T9000/C104 models use a single set of parameter values for a wide range of numbers of processors.

6.7 Conclusions

The results in this chapter suggest that Method I is the best parallel algorithm to use for the linear algebra sections when solving non-linear unconstrained optimisation problems using a quasi-Newton method. The algorithm gives good performance and scales very well as the number of processors increases especially for larger problems.

It is important to view these results in the context of a complete optimisation algorithm. As mentioned in Section 6.1, the calculation of the step length along the search direction is also costly. This operation normally involves function and gradient evaluations and when the objective function is complicated these calculations can dominate over the linear algebra costs. Hence, as well as using parallel linear algebra in a quasi-Newton algorithm one must use parallel function evaluations in the line search of Step 2.

In the next chapter we draw together all the work described so far and discuss the lessons learned and the direction of present and future work.

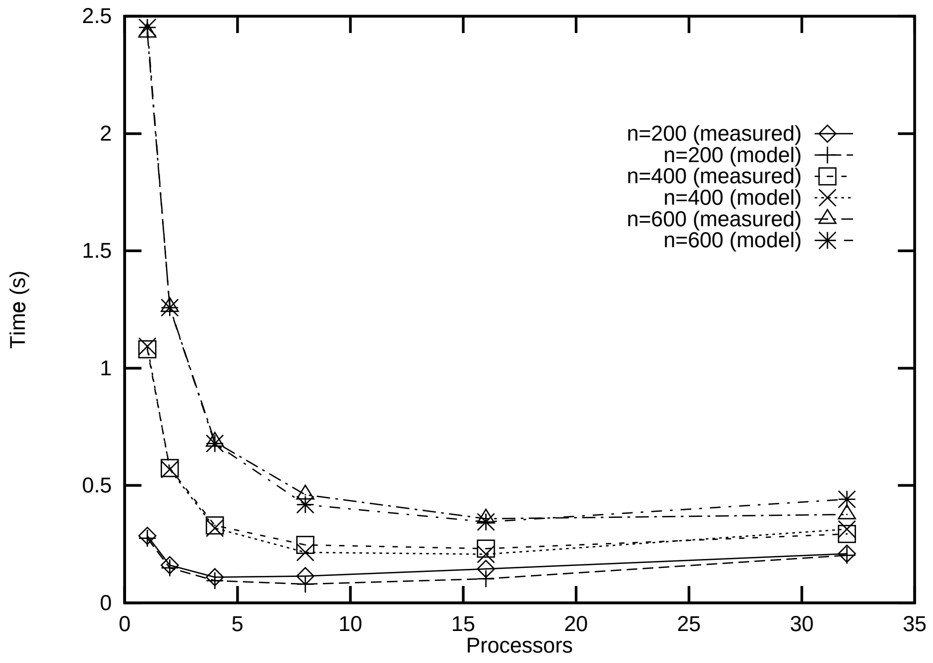


Figure 6.5: VCR implementation of Method I

Chapter 7

Discussion

In the preceding chapters we have studied many aspects of the design of parallel scientific algorithms for the distributed memory MIMD transputer architecture. In this chapter we present a discussion of some of the issues raised and suggest areas for future work.

7.1 Parallel algorithms

We have looked at several different areas of numerical computation in this thesis. These have shown a number of important principles in the design of parallel numerical algorithms.

The algorithms we have implemented have been parallel versions of well known sequential algorithms. When choosing between sequential algorithms that perform the same operation two issues must be considered. Firstly, how well can the sequential algorithm be parallelised? This is a function of the inherent parallelism in the algorithm, the ratio of computation to communication that is required by a parallel implementation of the algorithm. These factors determine the efficiency of the parallel algorithm: the proportion of time each processor performs useful work. A parallel algorithm with a higher efficiency makes better use of the hardware resources. However, alongside this consideration we must compare the relative costs of the initial sequential algorithms. If one sequential algorithm has a significantly higher sequential cost than another, then even if the parallel version of that algorithm has a much higher efficiency than the other algorithm, the lower cost sequential algorithm may still give a lower-cost parallel algorithm (see Chapter 6).

Another important issue is the amount of memory required by an algorithm on each processor. The bitonic sort algorithm of Chapter 4 is an example of a relatively memory-greedy algorithm since it requires enough storage on each processor for two copies of the processor's vector. This limits the maximum problem size that can be solved to half that of a memory efficient algorithm.

As machines become larger the scalability of the parallel algorithm to large numbers of processors will become very important. If the algorithm scales poorly and gives its best performance for only small numbers of processors then the maximum speedup achievable will only be low and the resources of the machine will be poorly utilised. The T8 sorting algorithm (Chapter 4) is an example of an algorithm that scales badly. In this case the poor scaling is due to the poor communications scalability of a chain of T8 processors. The T9000/C104 algorithm gives good scalability since it has a scalable communications system.

The scalability of an algorithm is linked to the granularity of the algorithm. The granularity of an algorithm is the amount of computation performed between synchronising communications. If the granularity is high the algorithm is said to be coarse grained and will give good efficiency. In the extreme, embarrassingly parallel applications, such as ray-tracing and other image processing tasks, require almost no communication and scale to large numbers of processors with very high efficiency. Most algorithms, like those studied here, have a more modest degree of granularity. They scale well over a range of machine sizes, but for fixed problem size give poor performance on very large systems. One way to utilise larger machines with such algorithms is to increase the problem size. If we increase the problem size as we increase the number of processors then we retain coarse granularity and can achieve very good performance from large machines. For example, in finite element applications we can use finer meshes to give more accurate solutions.

7.2 Modelling

For each of the algorithms studied we have given run-time cost models. We have shown that when the algorithm is reasonably coarse grained, such as for the forwards elimination phase of the Gaussian elimination algorithm in Chapter 3, then the cost models give a good prediction of the performance of the algorithm on T8 systems. The models can be incorporated into program code so that at run time the program can decide on the optimum number of processors to use for a given problem size.

Perhaps more importantly, the models allow us to predict the performance of an algorithm on a particular architecture before we implement the algorithm. This allows us to compare prospective algorithms and choose the most suitable one for implementation instead of implementing all the algorithms and finding the best by experiment. This approach can save a lot of development time. We applied this idea in Chapter 6 when selecting a BFGS update method for the T9000/C104 architecture.

The models also allow us to compare the performance of an algorithm on different architectures (see Chapters 3, 4 and 5). Cost models can be used to show us how much better an algorithm would perform on a new machine compared to the current machine. They can be used in the design of new architectures; for example, the models can reveal the benefits of doubling the communication rate for important algorithms.

The cost models try to add mathematical precision to the parallel programmer's intuition.

The models are not the same as a complexity analysis of an algorithm. A complexity analysis gives the order of the computation cost and communication cost separately. It does not take into account the possible overlap of communications and computation nor does it give an estimate of the absolute cost of the algorithm nor the relative costs of computation and communication. Complexity analysis gives a growth function for the cost of computation and communication of an algorithm. We try to do all of these things in the cost models presented in this thesis.

The success of this approach cannot be judged until more experience has been gained with modelling algorithms and comparing the model with actual run-times. Although the model for the forwards elimination phase of the Gaussian elimination algorithm predicts the cost well, the model for the RHS forwards elimination and backwards substitution phase gave rather poor predictions. The latter phase is very fine grained whilst the former is coarse grained. This suggests that it is difficult to model algorithms with frequent small communications accurately. Such algorithms have frequent synchronisations between processes which introduces unknown amounts of idle time, and the idle time is a significant cost compared to the communication and computation cost. In the case of the bitonic sort algorithm of Chapter 4 the error in the cost model is probably due to the difficulty of modelling the special packet-based communication routine.

During this work we have developed several techniques for modelling the cost of algorithms. In the earlier T8 work, we took an algorithm and modelled the cost of each individual computation and communication operation in turn. This followed the style of development of the algorithms which involved writing code to perform the local operations on each process. For the later T9000/C104 algorithms, which are designed as sequences of standard parallel operations, such as broadcast and BLAS operations, a different view is taken for model development. In this case the cost of a parallel algorithm is viewed as the sum of the costs of the component parallel operations. Thus cost models are developed for each standard communication and computation operation and these costs are combined to give the complete algorithm cost. This technique allows quick development of an algorithm cost model.

This latter technique generally gives worst-case costs since it assumes that each processor is busy for the entire cost of the standard operation, and so does not allow for some processors starting a subsequent operation before the other processors have completed the last operation. For example, with a global operation such as max, where the result is left on a single root processor, the time between the first leaf processors starting the operation and the last root processor completing the operation will be much greater than the time any single processor spends executing the function. Wavefront operations, where a pattern of execution operations ripples across a network of processors, also have this feature. For these types of operation, the run-time cost might be better modelled by the first T8 technique which uses a local view of the operation to determine the cost, instead of a global view of the operation across the whole network as used in the T9000/C104 models. In general, the programmer's intuition of the execution of a parallel operation can guide the use of local or global cost models.

It is important to bear in mind the expected accuracy of the cost model. We have shown that the models can give predictions within 20–30% of the measured run-time costs for T8 algorithms, with better accuracy for larger problem sizes (see Chapter 3). With this level of accuracy, less significant terms in the cost expression can be ignored without increasing the error substantially. This allows simpler cost models to be developed by neglecting certain low-cost operations in an algorithm. Chapter 3 also shows that there can be considerable variation in the estimates for the hardware parameters T_f and T_c. Variations of 10% in hardware parameter values alone will give variations of 10% in the total run-time predictions. So cost modelling cannot, in general, be expected to achieve a better accuracy than this.

We believe that the cost modelling techniques developed in this work are a useful aid to parallel algorithm design and development. There is much further work that needs to be done. When T9000/C104 systems become available we need to compare the predicted performance for the algorithms with that achieved in practice. We also wish to develop the modelling techniques to allow the same cost model to be used to predict the run-time cost of an algorithm on different distributed memory MIMD architectures. For architectures that support the same communication facilities this may be done by substituting hardware parameter values for the new architecture. For architectures with different communication subsystems, such as the grid topology of the Intel Paragon, it may be necessary to develop new cost models for the communications operations in the same way that porting the algorithm requires a new set of communication routines for the new architecture.

7.3 Implementation

When implementing the parallel algorithms a number of issues arose that are shared by all the algorithms. The two main issues are data distribution and modular program development.

One of the main considerations for any parallel algorithm is the distribution to be used for the data structures. The choice of distribution affects the volume and pattern of communications and hence can have a large impact on the performance of the algorithm. In this thesis we have used scalars, and dense vectors and matrices. The main distributions of elements of a vector are by blocks or cyclically or, in general, block cyclically where blocks of elements are distributed cyclically to the processes. Matrices may be distributed in similar manner with whole rows or columns distributed in blocks, cyclically, or block cyclically. In general, a matrix may be distributed by rectangular blocks of any size. All these distributions require the same total storage space.

The distributions of data for a particular algorithm should be compatible with one another for the operations to be performed. For example with a dot product, both of the vectors should use the same distribution. A complete algorithm will usually involve several sub-algorithms such as matrix-vector multiplication. In this case the optimum data distributions for the sub-algorithms may not all be the same. The programmer can then choose whether to re-distribute the data between sub-algorithms to utilise the optimum distribution for each part or whether to use a compromise distribution. The choice of a compromise distribution should be the optimum distribution for the most costly part of the algorithm. Interesting examples of data distribution considerations can be found in Chapter 6.

The use of modular programming techniques may be second nature to professional sequential program developers, but in the infancy of parallel programming, software engineering methods and standardisation were displaced by experimentation and independent software development. There were no standards for parallel scientific programming and some of the most important results of recent work have been the development of standard programming and computation models, parallel language extensions and communication routines. Chapter 1 discusses programming and computation models and parallel languages; Appendix A discusses standard communication routines. The use of standard communication and computation routines has reduced the difficulty of program development considerably and leads to more portable code. This is discussed in more detail in Appendix B, in Cook [25], and in the following section.

Another important consideration is the use of explicit parallelism to hide communication latency. In Appendix B we discuss several techniques to achieve this. The simplest method, which is very effective and has been used for algorithms in this thesis (for example, see Chapter 4), is to make use of two threads during communication: one thread performs the communication operation while the other thread continues with computation. This technique, like the others, requires data and control flow independence in the algorithm. Further work needs to be done to investigate the practical suitability of the excess parallelism method which at first sight appears to be a very elegant solution to the problem of hiding communication costs. This method does not require explicit parallelism in the code and so could make program development simpler.

7.4 Parallel numerical libraries

The computation model used for the algorithms in this thesis was specified in the original design of the Liverpool Parallel Library [32, 73] developed in the ESPRIT Supernode I project (P1085). That design assumed that a user would write a sequential program which made calls to routines in the Parallel Library. The parallel routines gave the user the power of the Supernode parallel machine without the user having to write parallel programs himself. This design paradigm also allows old dusty deck Fortran programs to migrate to the parallel architecture by replacing calls to sequential numerical library routines, such as those contained in the NAG Fortran Library, by an equivalent parallel routine in the Parallel Library. This strategy also retains portability of the sequential code to other architectures, such as Cray vector processors, by replacing the parallel routine call with a call to a routine optimised for the new architecture.

This high level design aim of efficiency and portability also features in the LAPACK [34, 33] and ScaLAPACK [20] numerical libraries. The LAPACK design seeks to achieve these aims by developing high level algorithms which use lower level block operations. The high level blocked algorithms are portable between platforms and efficiency is achieved by tuning the block operations to each underlying architecture. This blocked algorithm design has been shown to work extremely well for scalar processors with memory hierarchies[39, 15] and vector processors[29] as well as distributed memory MIMD machines[38, 93]. However, note that only scalar and vector processor architectures use portable blocked code; current distributed memory MIMD blocked algorithms still use explicit parallel programming. ScaLAPACK is a library design which tries to standardise the techniques used to implement distributed memory MIMD blocked algorithms. It uses three types of routines: distributed Level 3 BLAS, the BLACS[7, 40], and assembler sequential Level 3 BLAS. All communication is performed within the distributed BLAS so that the high level blocked routines, such as LU factorization, are identical with the LAPACK codes. The design of ScaLAPACK is hence very similar to the design of the Liverpool Parallel Library. It also parallels the work on communication routines and distributed BLAS described in Appendices A and B.

One important feature of the ScaLAPACK design is that it provides long-lived or persistent distributed data structures: distributed data structures are declared and initialised before being manipulated by a numerical library routine and continue to exist until explicitly deleted by the user's program. Thus a sequence of distributed operations can be invoked on a distributed data structure by the sequential user program. This is a crucial advance over the original Liverpool Parallel Library design described above and used for the algorithms in this thesis. It removes the requirement for the initial data to be distributed from the master and the final result data to be gathered back to the master. A similar system, called the DDS system[13, 1, 2], has been developed for the Liverpool Parallel Library under the ESPRIT Supernode II project (P2528). With persistent distributed data structures (DDSs) the performance of the algorithms described in this thesis improves significantly. We have illustrated this by showing the cost model performance for the parallel sort algorithm using pre-distributed data (see Chapters 4). This algorithm shows a great improvement in performance with pre-distributed data because the scatter and gather costs make up a significant proportion of the total algorithm cost.

The design of the DDS system is still in its infancy although we already have a working implementation with a set of communication routines, distributed BLAS routines and some demonstration high level algorithms.

The programming techniques described in Appendix B work well with the DDS design. An algorithm is designed using a master and slave process. The short master process checks the user parameters to ensure that the data distributions are compatible with one other for the operation to be performed, then passes the identities of the distributed data structures (DDSs) to the slaves. The design of the slave code is very similar to the slave code shown in Appendix B. The slave manipulates the distributed data through a sequence of calls to library routines. These routines may be local data assembler BLAS routines, or communication routines such as those described in Appendix A, or distributed BLAS routines described in Appendix B. There is almost no difference between the slave code presented in the appendix and slave code for the DDS system.

The user level Library will be augmented with routines that perform exactly the same functions as the slave level communication and distributed BLAS routines. These routines, in fact, will only be a short master process which checks parameters and then invokes the slave routines described above on each slave.

The portability of the Library can be enhanced by keeping to current and proposed "standards" as much as possible. Slave level communications should be implemented on top of one of the existing communication libraries such as PARMACS[54] using wrapper procedures to deal with the different requirements of different libraries. The interface to the slave level communications should be as similar as possible to the proposed MPI[44] standard. This will allow the library to be moved between platforms easily: if a platform supports the MPI then the Library can be ported without developing new communication routines; if the platform only supports PARMACS or PVM then the Library can be ported using this lower level interface, and at a later stage the communication routines could be optimised to the new architecture if required. In addition the design of the Library should ensure as much as possible that the code is compatible with the important High Performance Fortran[43] proposal. For example the Library routines to declare and create distributed data structures should map closely to the facilities of the HPF decomposition and alignment operations.

7.5 Architecture

All the work in this thesis has been based on the transputer architecture. T8 transputer networks have been very popular over the past 5 years, but the architecture is now rather dated compared with the current state-of-the-art high performance parallel computers. When this work started it was expected that T9000 systems would replace T8 systems as the next generation of high performance parallel computers. However, even the respectable 25Mflop/s peak performance of the T9000 has not kept pace with modern microprocessors. The Intel i860XP used in the Paragon has a peak rate of 50Mflop/s double precision and the DEC Alpha processor to be used in Cray's forthcoming MPP has a peak of 150Mflop/s. The Meiko CS-2 can have either SuperSPARC scalar nodes giving 50Mflop/s performance or Fujitsu vector processor nodes giving 200Mflop/s peak double precision performance.

The communication capabilities of the new T9000/C104 system have also been exceeded by other systems. The T9000 links have a bi-directional bandwidth of 10Mbyte/s giving a total node IO bandwidth of 80Mbyte/s. This compares with the 2 links per node of the Meiko CS-2 which have a bi-directional bandwidth of 50Mbyte/s giving a total node IO bandwidth of 200Mbyte/s. The CS-2 also uses a multistage switch network like the T9000/C104, whereas the Paragon and Cray MPP use a grid and 3D grid topology, respectively.

Even though the transputer architecture can no longer compete in the parallel supercomputer arena, the software techniques and algorithm design methodologies developed in this thesis can be applied equally well to the other distributed memory MIMD architectures used in today's supercomputers. In particular, the T9000/C104 based work will map very well onto the Meiko CS-2 with its similar multistage communications network. Also, the T8 work which dealt with chain and grid topologies is applicable to the Intel Paragon and Cray MPP architectures with their grid and 3D grid topologies.

7.6 Further work

There are many avenues for further work. One of the most important tasks is to demonstrate that the programming techniques, algorithm design methodologies and cost modelling techniques can be applied to distributed memory MIMD architectures other than the transputer. This will initially involve re-implementing the algorithms in Fortran on another architecture.

Another important task is the continued development of the Parallel Library design and in particular the use of standard subroutine libraries in the development of DDS codes. Further work is also needed to study different data distributions and select the most important ones for inclusion in the Library.

In addition we need to continue work on cost model development, initially by implementing the algorithms on T9000/C104 machines when these are available to confirm the accuracy of the cost models.

Chapter 8

Summary

In this thesis we have studied the design of scientific parallel algorithms for transputer systems. Using the T8 and T9000/C104 as examples, we have shown how different parallel architectures affect the design and performance of algorithms. These comparisons have highlighted the importance of a scalable interconnection network, such as that provided by the T9000/C104 system. We have also shown the importance of maintaining a balance between computation rate and communication rate.

We have placed considerable emphasis on the development of run-time cost models for parallel algorithms. We have shown that these cost models give a good prediction of the performance of algorithms for the T8 architecture, and we have then used cost models to predict the performance of algorithms on the T9000/C104 architecture. These cost models allow us to compare the performance of parallel algorithms without implemented them. Thus we can select and implement the best algorithm without spending a great deal of time implementing all the other algorithms as well. We can use the cost models to predict the performance of algorithms for large problem sizes on very large machines and to investigate the change in algorithmic performance when the hardware performance is changed, for example when the communication rate is doubled. These cost models have been shown to be a valuable aid to the algorithm designer.

We have implemented several of the algorithms in `occam`. This has revealed the need for standard communication and computation routines. We have designed a suite of communication routines for the T8 and T9000/C104 architectures. These have greatly simplified the development of programs. For computation operations, we have used local memory assembler BLAS. We have also shown the need for distributed versions of the BLAS routines. The desire for portable and efficient algorithms has highlighted the importance of the use of optimised communication and computation routines on each target architecture. In addition, the lack of `occam` implementations on any other platform coupled with the portability requirement has lead to the adoption of Fortran as the preferred language for future development.

This work has shown that the computation model assumed by the initial Liverpool Parallel Library was severely limited by its need to distribute initial data from the master process and gather results back to the master process. We have shown the significant performance improvements that can be gained from the use of pre-distributed data.

We have shown the need for careful selection of a sequential algorithm for parallelisation. The importance of the ratio between communication and computation cost for an algorithm has been emphasised along with the affect of algorithm granularity on the scalability of the algorithm to large systems. We have shown how the performance of an algorithm can be improved by hiding communication latency behind computation.

Appendix A

Communication libraries

This appendix describes a set of high level communication routines developed for use in `occam` programs for T8 and T9000/C104 machines.

A.1 Introduction

Programs developed using an explicit message passing model, such as that provided by `occam`, tend to consist of code blocks which perform one of two different functions: computation or communication. For a long time, scientific software development with sequential languages has been refining the art of code design for computation. The need for stock code fragments to perform often required computation operations has been widely recognised. Such codes would provide reliable building blocks and so reduce the cost of the development of new software. This has lead to the specification and development of libraries of reasonably low level computation operations such as the BLAS [68, 37, 36].

The use of parallel languages and the message passing model has revealed the need for similar libraries of communication operations for parallel computing. At the lowest level we need a standard for point to point communications so that explicit message passing programs can be ported from one parallel computer to another without rewriting all the communications operations. At present, most of the parallel computer manufacturers and other groups provide proprietary and incompatible low level communication routines. Currently, some of the most widely used communications libraries are PARMACS [54, 55], PVM [14, 90], NX/2 [83] and CS-Tools [71]. Attempts are now being made to define a standard set of low level communications operations [44].

These libraries help ensure portability of code from one machine to another, but the routines are generally too low level for direct use in application codes. Instead applications should be developed using an intermediate level of communications routines which provide higher level functionality. These routines would in turn use the lower level communications libraries to ensure portability to the widest range of machines.

Good communication performance can be satisfied while still maintaining application portability by tuning each level of library routines to the underlying hardware as is standard practice for sequential numerical libraries such as the BLAS. Tuning a communication routine to a particular architecture can give great improvements in performance. For example, it is often quite difficult to achieve good communication performance from the T8 architecture for the transfer of data between two distant processors connected by intermediate processors. However, we show below that a carefully written communication routine can improve upon the performance of a simple solution considerably. This improvement can be achieved whether or not the underlying hardware communication network is "difficult" to exploit effectively such as with the T8 architecture. Even the enhanced capabilities of the C104 communications network do not remove the need for tuned communications routines. The C104 switch network provides fixed distances to all processors so there is no need to consider the relative positions of source and destination processors during communication. However, this architecture provides a whole set of new opportunities for optimising communication performance. Even a single point to point communication can be implemented in a number of ways. Below we present our suggestions for gaining improvements from the T8 and T9000/C104 architectures. In this thesis we have made use of some of these techniques to improve the performance of the applications under investigation. There is more discussion of the possibilities for improving communication performance for the T9000/C104 architecture in Chapter 2.

At present there is no agreement on a standard for these high level communication operations. However there is a growing consensus concerning the type of operations that should be provided. Operations provided should include broadcast; scatter (and its complement, gather); all-to-all broadcast and scatter (also called complete exchanges); global operations such as maximum, minimum and sum; and data re-distribution operations such as shift. Examples of this type of high level communication library are the BLACS [7, 40], the Message-Passing Interface [44], and the communications facilities of Fortnet [6].

Whilst work for this thesis was being undertaken there were no high level `occam` libraries available for the T8 transputer or VCR/T9000. So communication routines required for the algorithms investigated here have been designed and implemented for both chains of T8 processors and T9000/C104 networks. Four main communication operations are required: broadcast, scatter (and its complement, gather) and the global max operation. The following sections define each communication operation, present the algorithms for the operation on both T8 networks and on T9000/C104 networks, and give run-time cost models for each operation. In the case of a square grid of T8 processors as is used for the Newton algorithm in Chapter 5 we present two algorithms for each communication operation: one algorithm is used when the master processor is not involved in the communication operation whilst a slightly different one is used otherwise. This is necessary because of the single link connection between the master and the slave network. Figure A.1 shows the process topologies that have been used in the design of the algorithms for this thesis. More information concerning the selection of these topologies can be found in Chapter 1.

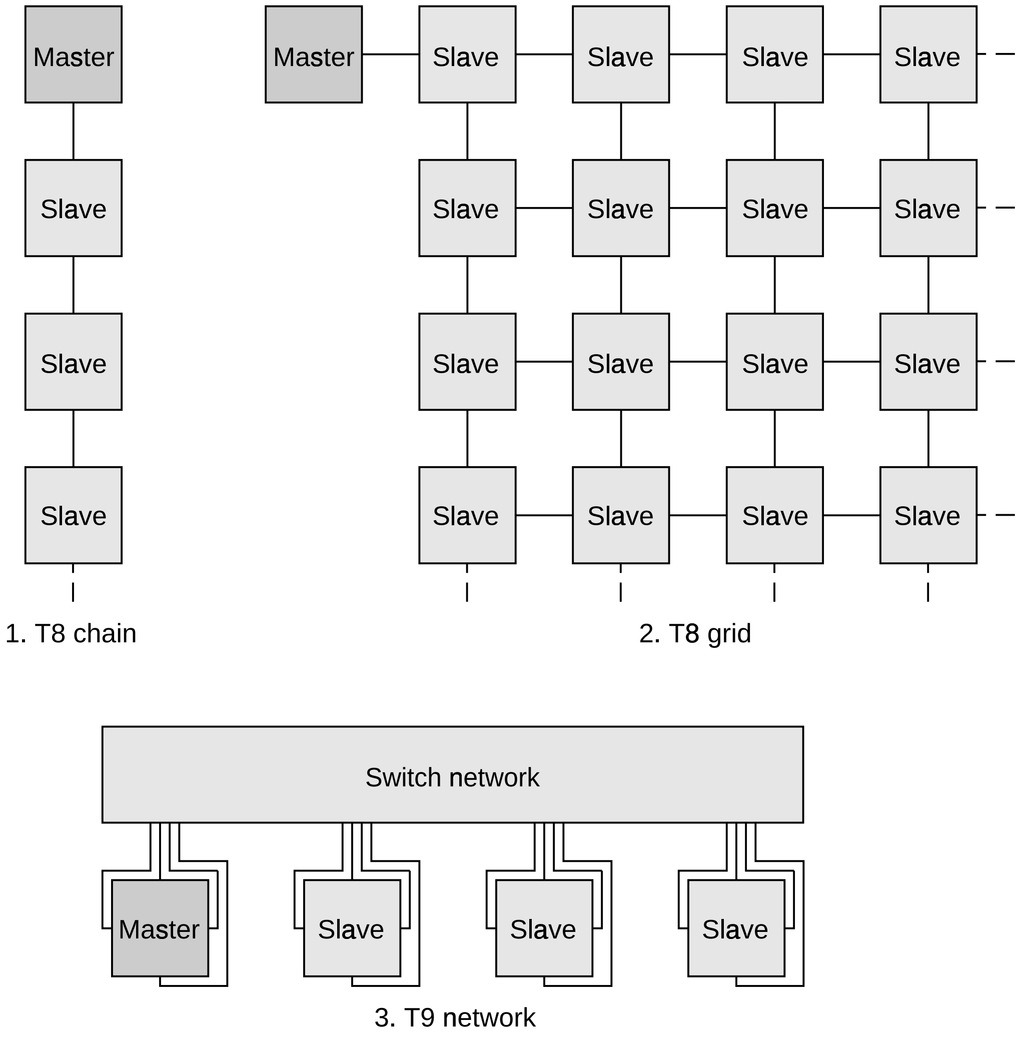


Figure A.1: Process topologies used by algorithms

In addition to these routines, for the T9000/C104 architecture only, we have developed all-to-all broadcast and scatter routines which are used in the parallel BFGS algorithms of Chapter 6. We describe the algorithms used for these operations below.

The run-time cost models use the hardware communications parameters introduced in Chapter 2, i.e., the time taken to communicate n values is given by the following:
For T8 networks: $T = nT_c$.
For T9000 networks: $T = T_s + nT_c$.
The comparison operation required for the global max function is modelled by the usual computation parameter T_f. The context makes it clear which architecture a run-time cost model is for, and hence which values are appropriate for the hardware parameters. The parameter p is the number of processors. For T8 networks this excludes the master processor. For T9000/C104 networks the value of p includes the master processor. (For a discussion of this choice see Chapter 1.)

The communication cost models make a number of assumptions about the performance of the hardware which are detailed more fully in Chapter 2. In summary, all the models in this thesis assume that the different execution units of the T8 and T9000 processors can execute concurrently at full speed, i.e., there is no decrease in the performance of one link if another link starts communicating as well, or if the integer or floating point units are executing. In the case of the C104 switch network, we also assume that we can neglect contention for network resources and the effects of hot-spots.

A.2 Point to point communication

Underlying all of these communications algorithms is a point to point communication operation. The performance of the high level communication operation can be affected considerably by the implementation of this point to point communication. This is especially noticeable for large vectors being sent between distant processors through many intermediate processors. This type of communication pattern is quite common for T8 architectures during the initial scattering of data and gathering of result data.

Let us consider first the case of sending a vector of n elements between processors p links apart in a chain of T8 processors. The simple approach of passing the vector in its entirety as one block between processors gives poor performance when the vector must pass through many intermediate processors before reaching its destination Instead a more efficient method has been implemented which splits the vector into smaller packets and sends each packet separately. In this case many packets can be communicated between intermediate processors in parallel and so greater performance is achieved. Figure A.2 illustrates these different methods. Part 1 shows the case of communicating a vector in one block. Sending the vector between adjacent processors takes three time-steps and so it takes nine steps for the vector to cross three links. Part 2 shows how the communication would be executed if the vector was split into three packets. At each of the first three steps the source would output a packet of the vector

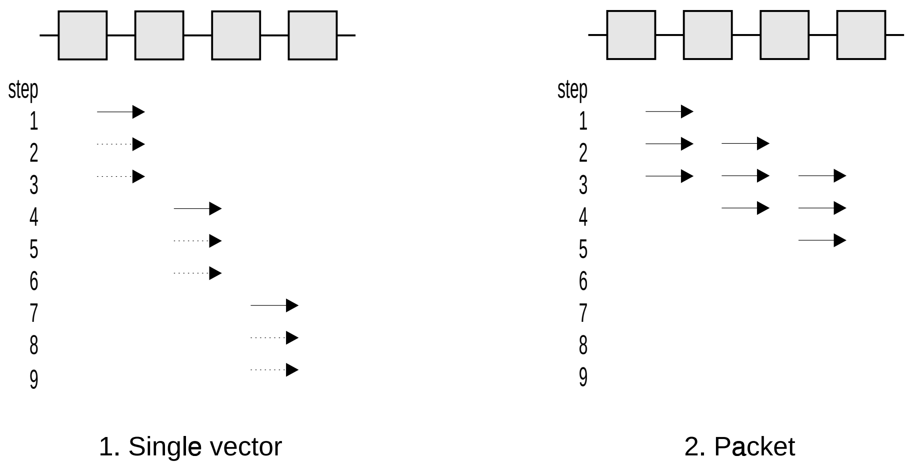


Figure A.2: Comparison of single vector and packet-based communication

to the second processor. Once the second processor has received a packet, on the next step it sends that packet to the third processor and in parallel receives the second packet from the source processor. Further processors execute in the same fashion, receiving and sending packets in parallel. For this example the whole operation takes only five steps instead of nine steps for the single block communication.

There will be an optimum packet size for this type of communication. At one extreme using one packet only gives poor performance, and at the other extreme, each packet containing only one item, management overheads and the cost of executing the code itself decreases performance.

The cost to communicate a vector of n elements between slave i and slave $i+p$ ($p \geq 1$) for each case is:

1. For a single packet

$$npT_c.$$

2. Using packets of s elements and assuming that input and output on the links of a single processor operate in parallel gives:

$$nT_c + s(p-1)T_c.$$

If $s \ll n/p$ this may be approximated by

$$nT_c.$$

Some experimentation has shown that the minimum point as s varies is in a wide trough with a range of values for s between 50 and 400 giving good performance. We have used a value of $s = 200$ in the implementation of this routine for a chain of T8 processors.

This packet-based communication works especially well for sending large vectors across many processors which is quite common for T8 algorithms during the initial scattering of data and the gathering of result data. This method of point to point communication has been used successfully for the vector exchange operations in the T8 parallel bitonic sort algorithm which account for a large fraction of the total algorithm cost (see Chapter 4). The other T8 algorithms currently use only a single block communication operation.

The T9000/C104 architecture with a multi-stage switch network has a small network diameter even for large arrays of processors, e.g., for the 3 stage folded Clos network messages need to be routed through at most 3 switch chips. And in any case, an individual channel communication is wormhole routed through the switch network by the hardware in just the manner described above for the T8 architecture. However, there are two ways we can improve the performance of a single point to point communication. The first technique available is to split the message into several parts and send each part on a different virtual channel to the destination processor. This gives better performance because it makes better use of the link bandwidth than a single channel. The number of channels that can be placed on a link before the link bandwidth is saturated depends on the number of switch chips the message must be routed through. For the three-stage folded Clos network 2 channels will saturate a link. The second technique is an extension to the first one. Instead of using a single physical link we can distribute the virtual channels over all four output links and use up to 8 channels in the case of our three-stage network. More details can be found in Chapter 2.

Until T9000/C104 machines become available, we will not know what network topologies will be available or the actual benefits that can be gained from implementing the preceding ideas. Hence for this work we have assumed that only a single channel point to point communication operation is used, and the higher level communications operations have been written to use the fundamental `occam` channel input and output functions only.

A.3 Broadcast

Description: One source process sends the same block of data of size n to all the other processes. This operation may be used by the master process to send data to all the slaves, or by one slave process to send some data to all the other slaves.

A.3.1 T9000/C104 architecture

An obvious process configuration for a broadcast is an b-ary tree of p processes rooted at the source process (b is the branching factor of the tree). For run-time cost models in this thesis we assume that a single channel is placed on each of the four links which gives $b = 4$. However, Chapter 2 suggests that for the three-stage folded Clos network one virtual channel on a link does not saturate the link bandwidth. Better performance might thus be obtained by placing up to two channels on each link giving a value of $b = 8$. Since the exact performance of the T9000/C104 links are not known the implementation of this algorithm for the T9000/C104 has been written to use a tree with a variable branching factor. The tree configuration provides the shortest link path between source and destination processes. Levels in the b-ary tree are numbered starting with the root as level 0. The algorithm proceeds as follows: at step k ($k \geq 0$) in the communication algorithm processes at level k in the b-ary tree would output the data on all their branches to processes at level $k+1$. The total number of steps required to achieve the broadcast is thus the depth of the b-ary tree, which is given by $\lceil \log_b((b-1)p+1)-1 \rceil$. The cost of the broadcast for a vector of size n is:

$$\lceil \log_b((b-1)p+1)-1 \rceil \, (T_s + nT_c).$$

This algorithm is used by the Newton method and the bitonic sort.

For the BFGS algorithm an improved algorithm was devised. This new algorithm proceeds as follows: the source process communicates the vector to four other processes in parallel on its four links. Then each of these five processes (including the source) communicate the data to four more processes in parallel. Hence at each step in the algorithm all processes having a copy of the data send it to four more processes. This continues for $\lceil \log_5 p \rceil$ steps when all p processes have a copy of the data. The cost for this operation is

$$\lceil \log_5 p \rceil \, (T_s + nT_c).$$

A.3.2 T8 architecture

Chain

For a chain of p slave processes, the algorithm proceeds as follows: the source process outputs the data in both directions (unless the process is at one end of the chain). At each subsequent step, a process to the right and left of the source process output the data to their right and left respectively until each end of the chain is reached. The number of steps required is the greater of the distances from the source process to each end of the chain which we will call d. The total cost is thus

$$dnT_c.$$

This algorithm is used in the Gaussian elimination algorithm of Chapter 3.

Square grid

In the first step in the broadcast algorithm for a square grid, the source process outputs the data concurrently on all 4 links. In the second and subsequent steps the processes that have just received the data on a link output it on their other links. The passage of data across the grid is in the form of a tree-like structure routed at the source process with each node having a link to its parent and links to up to three child nodes. There will be overheads associated with preventing multiple processes sending the data to one process, but we will not take account of this as the cost would depend upon the implementation.

For a square grid of p processes the maximum number of steps required (occurring when the source process is at a corner) is $2(\sqrt{p}-1)$. The minimum number of steps possible (occurring when the source process is centrally located) is $\sqrt{p}-1$. If the location of the source process is unknown until run-time we will assume an average of $3(\sqrt{p}-1)/2$ steps for the algorithm. With this assumption the cost to broadcast a block of n elements would be:

$$\frac{3n(\sqrt{p}-1)}{2}T_c.$$

If the master process is the source of the broadcast then the total cost is given by:

$$(2\sqrt{p}-1)nT_c.$$

A.4 Scatter

Description: One process sends a different block of data to each other process. This operation is usually used to scatter a vector of size n from one process to a number of processes such that each process gets a different block of the original vector. If the source is a slave process then the slave usually keeps one block of the vector for itself and scatters the remaining blocks over the $p-1$ other slave processes. In this case each block is of size n/p. If the master process is the source then usually the entire vector is scattered across the p slaves without the master retaining a block. On the T9000/C104 architecture the master process is also a slave process and so always retains a block of the data.

A.4.1 T9000/C104 architecture

The algorithm proceeds as follows: in each step the source process sends blocks of data directly to 4 different destination processes in parallel. The total number of steps to send blocks to $p-1$ other processes is $\lceil (p-1)/4 \rceil$. For a source vector of size n each block will be of size n/p and the total cost is:

$$\left\lceil \frac{p-1}{4} \right\rceil (T_s + (n/p)T_c).$$

The complementary operation, gather, where one destination process receives a different block of data from every other process, proceeds in a similar manner except that the communication direction is reversed. The cost is also the same.

A.4.2 T8 architecture

Chain

For a chain of processes the algorithm is similar to the broadcast algorithm described previously. In the first step the source process sends a data block out to the left and another to the right. These blocks are destined for the end processes. At each subsequent step the source process outputs blocks to left and right which are destined for the next furthest processes in the chain. The processes to the left of the source process input blocks from the right and output these blocks to their left until they receive their own block. These input and output operations execute in parallel. Processes on the right side of the source operate similarly. The number of steps required depends on the position of the source process in the chain and is the greater of the distances from the source process to each end of the chain which we will call d. For a source vector of size n the blocks will be of size n/p and the total cost is given by:

$$(n/p)dT_c.$$

This operation is usually used to scatter a vector from the master process onto the slave processes. In this case the master process does not keep a block for itself but splits the vector equally between all the slaves. In this case, for a vector of size n the blocks will be of size n/p and the total cost is given by:

$$nT_c.$$

This routine is used in the Gaussian elimination algorithm of Chapter 3. The complementary gather operation has a similar algorithm and the same cost.

Square grid

Scattering a vector on a grid of processors uses the same communications paths as broadcasting a vector on a grid from the same source process. The passage of data across the grid is in the form of a tree-like structure routed at the source process with each node having a link to its parent and links to up to three child nodes. The algorithm proceeds as follows. At each step of the algorithm the source process outputs a different block of data on each link, beginning with blocks for the processes at the greatest depth in the communication tree, i.e., the ones "furthest away" in the grid. The destination processes first input blocks destined for processes further down the tree. These blocks are routed down the output link which is the branch of the tree that includes their destination process. After passing on all the blocks for processes further down the tree, the destination process inputs its own block. As for the broadcast on a grid, this operation will require quite complex coding to determine the correct tree structure to use. The cost to scatter a vector of size n across p slave processes is

$$\frac{3n(\sqrt{p}-1)}{2p}T_c.$$

If the master process is the source of the scatter operation then the single link between the master process and corner slave process will become a bottleneck. All communications on this link will be sequentialised giving a total cost for the operation of approximately

$$nT_c.$$

The complementary gather operation has a similar algorithm and the same cost.

A.5 Global operations: max

Description: In a global operation a given function is performed on a set of data values, which are distributed across the processes. For this thesis we leave the result of the operation on a user specified process, but a useful additional routine would broadcast the result value to all the processes. The function might be max, min, sum, avg or any function that returns a single result value. We have used the max function in Chapter 5 and so we use it here to illustrate global operations. Initially, each process performs the operation locally on its data values. In the following communication phase of the global operation the objects manipulated consist of two items: a data value (calculated in the first phase) and a process label. The process label identifies the process which provided the data value.

A.5.1 T9000/C104 architecture

The algorithm uses a b-ary tree as for the broadcast operation, with the root of the tree at the process which is to receive the result. The algorithm proceeds as follows. Initially, each process performs the operation locally on its data values. In the next step all processes in the last level of the tree send their data value/label object pairs to their parents. Each of these parents receives these values and performs the specified function on the input values and its own value. The result of this operation is then used as the value for the next step of the algorithm. In the next step these processes send their result data values to their parent processes which perform the function on them and their own value. This process continues until the root process has calculated the global result.

In deriving a cost model for a global operation we have chosen to leave out the local computation cost. It is assumed that this extra cost will be incorporated separately in the complete algorithm cost model.

The cost of this operation is the same as for the broadcast operation except for the additional function evaluations at each step. We assume that the cost of a single function evaluation is $1T_f$ which is correct for max and most of the common global operations. The additional cost incurred at each step is then bT_f. Let n be the number of elements transferred in each communication. For the max operation $n = 2$, i.e., one data value and one process label. The total cost for the global operation is thus

$$\lceil \log_b((b-1)p+1)-1\rceil \,(T_s + nT_c + bT_f).$$

A.5.2 T8 architecture

Chain

On a chain of processes the destination for the result can be any of the processes. The operation proceeds as follows. After the initial local computation, the processes at the ends of the chain send their data values to the next processes in the chain. In the second and subsequent steps the processes which have just received data values from outer processes perform the required function on the received value and their own value and send the function result to the next process in the chain. Finally, the destination process receives up to two data values, one from each side of the chain, and performs the function on these values and its own to get the result of the global operation. The number of steps required is the greater of the distances from the destination process to each end of the chain which we will call d. If the destination process is not at the end of the chain the total cost, ignoring the initial local computation cost, is

$$d(nT_c + T_f) + 1T_f.$$

If the master process is the destination then the cost is

$$p(nT_c + T_f).$$

Square grid

The algorithm for a square grid uses the same tree-like structure as the broadcast and scatter operations. The algorithm proceeds just as for the T9000/C104 architecture. At each step in the algorithm processes in one level of the tree, which have just received data values from their children, perform the function on these input values and their own value and output the result to their parent. The number of input values will be between one and three. Assuming the worst case for the number of function evaluations at each step, 3, the total cost, again ignoring the initial local computation, is

$$\frac{3(\sqrt{p}-1)}{2}(nT_c + 3T_f).$$

If the master process is the destination for the global operation then the algorithm sets the corner slave process as the destination and then sends the result to the master process. This costs:

$$(2\sqrt{p}-1)(nT_c+3T_f)+nT_c.$$

A.6 All-to-all broadcast

Description: Each process broadcasts a block of data to all the other processes. This routine is used, for example, to give each process a complete copy of a distributed object. Each process broadcasts its block of the distributed object and receives the blocks from all the other processes.

A.6.1 T9000/C104 architecture

The routine is based on the broadcast operation described above. Each process starts up $p-1$ threads in parallel to receive a block from each other process using the broadcast routine. In addition another parallel thread broadcasts a block of data to each other process.

This routine is used in the implementation of the full matrix storage version of BFGS update Method I (Algorithm 11) in Chapter 6. In step 6 of that algorithm we need to perform a rank two update. Chapter 6 presents one method for performing the update using distributed data. In our initial implementation of the algorithm we have used a slightly different method. The vector $\mathbf{s}$ is stored in full on each process and $\mathbf{z}$ is distributed. Step 6 has been implemented by performing an all-to-all broadcast (exchange) of the blocks of $\mathbf{z}$ to give each process a complete copy of the vector, followed by two calls to an assembler BLAS routine, `sger`, to perform the update. See Appendix B for the program listing.

A.7 All-to-all scatter

Description: Each process sends a different block of data to every other process.

A.7.1 T9000/C104 architecture

At present, under the VCR, this algorithm has been implemented using the scatter routine. Each process starts up $p-1$ threads in parallel to receive a block from each other process using the scatter routine. In addition another parallel thread sends blocks of data to each other process.

This routine is used in the triangular matrix storage version of BFGS update Method I (Algorithm 11) in Chapter 6. In that algorithm we need to perform a matrix-vector multiplication where initially only the lower triangular part of the symmetric matrix is stored distributed by rows across the processes. The vector is distributed also with a process holding those elements of the vector for which it also holds rows of the matrix.

To perform the matrix-vector multiply we first use the all-to-all scatter routine to “fill in” the upper triangular elements of each row of the matrix. We then use a distributed full matrix-vector multiply routine.

Appendix B

Programming techniques

In this appendix we discuss the programming techniques learned during this work. Few of these techniques have been used in every program, but rather they have been developed over six years of practical experience in programming using the message passing programming model for distributed memory architectures. We discuss programming techniques for both T8 and T9000/C104 architectures and methods for enabling portability of source code between these two architectures and other distributed memory architectures. We emphasize methods which provide good performance yet retain portability between different message passing environments.

B.1 Language

All the programs described in this thesis have been developed in `occam`. This language was chosen for the ESPRIT Supernode I project (P1085) for two main reasons. Firstly, the `occam` compiler from INMOS produced much more efficient code than the early Fortran compilers and secondly, and more importantly, `occam` allows very elegant expression of the parallel constructs in the CSP model. Although it has such excellent support for parallel programming, the language is weak in other respects. It has no user defined record types, no dynamic memory allocation and hence no recursion, and only poor support for large-scale modular programming using libraries. These limitations are addressed to some extent in the `occam` 3 proposal[11]. However, these same disadvantages have not prevented Fortran from becoming the dominant sequential scientific programming language. The main disadvantage of `occam` is simply that it is non-standard—it is not Fortran nor C. The lack of `occam` compilers on other architectures makes `occam` code non-portable and prevents the language reaching a wider programming community.

In addition, the traditional sequential languages are being extended with parallel features provided mainly through libraries. Although these extensions are far from elegant they allow parallel programs to be developed while still retaining the use of the huge base of pre-existing source code. Also, the language is familiar to the programmer which speeds up the learning process. The development of High Performance Fortran (HPF) [43] with its SPMD programming model is an especially important example of the way the sequential languages are being extended. For the message passing programming model we will soon have standards for communication through the MPI initiative[44]. We already have standards for basic linear algebra computations in the BLAS. These advances in the traditional Fortran and C languages offer us the prospect of portable parallel source code.

For these reasons we would recommend using Fortran (and perhaps C) for parallel scientific programming. The exact choice of Fortran dialect and extensions is more difficult to make since we do not know what future standards will support.

However, as already stated, all the programs in this thesis have been written in `occam`. This makes the programs strictly non-portable. In the rest of this appendix when we talk about the portability of the programs we mean the portability of the program structure, understanding that individual statements will all need to be translated into another language, probably Fortran. For example, the use of communication subroutines aids portability since it hides the details of the underlying implementation of communication.

B.2 Computation model

The computation model used for the algorithms described in this work is a master/slave model where one master process controls the operation of a group of slave processes. For T8 algorithms the processes are connected in a chain with the master process at one end. For T9000/C104 algorithms the processes have all-to-all connectivity provided by the C104 switch network. For more information about the choice of topology see Section 1.7. Two programs are written for each algorithm: a master program and a slave program. All the slave processes execute the same slave program using conditional branches on the process ID to control program flow. The use of only one slave program makes debugging a program much simpler than using several slave programs and makes maintenance easier. In the T8 algorithms, the master program is responsible for distributing the initial data and gathering the final results and for controlling the progress of the algorithm. However, it takes no part in the main computation phase of the algorithm. This wastes much of the processing power of the master processor. For the T9000/C104 algorithm the master process takes a full part in the computation phase of the algorithm.

This model leads to algorithms with three distinct stages:

1. master scatters input data to slaves,
2. slaves solve the problem, and
3. master gathers result data from slaves.

In practice this model has worked well for most of the algorithms studied. However the time taken to scatter the data initially and gather the results can be a significant proportion of the total run-time. This can lead to poor overall performance of the algorithm even if the parallel computation phase of the algorithm has a good performance. See, for example the sorting algorithms of Chapter 4. In Chapter 7 we discuss ways to avoid these communication overheads.

B.3 Communications

In `occam` communication between processes is performed on channels; a process communicates on a channel assuming that that channel is connected to the correct destination process. This is different from the approach taken in Fortran where communication is between “named” processes; a process specifies the “name” or handle of a process with which it wishes to communicate. To allow complete communication between p processes in `occam` requires $2p^2$ channels to be declared although each process only needs storage for the $2p$ channels that connect to it. Using named processes only the handles of the p processes are required by each process for complete communication. Communication on a channel is strictly ordered and separated from communication on other channels. The use of named processes requires extra tag information in each message to distinguish messages sent by different threads in a single process. With `occam` separate channels would have to be declared for the use of each thread.

To hide these differences between `occam` and Fortran all communications are performed through the use of subroutines. We use simple send and receive subroutines and higher level communication subroutines described in Appendix A. On networks of T8 transputers, direct communication between all the processors is not possible and so the `occam` communications subroutines hide the complexity of the message through-routing. In order to produce efficient code, the programmer must still be aware of the underlying topology since this affects the cost of each communication operation significantly. However, he no longer needs to be concerned with the details of writing correct and efficient communication routines. The T9000/C104 architecture does allow direct communication between processors. This immediately avoids all the complexity of the T8 routines which dealt with message through-routing. However, the T9000/C104 architecture still provides plenty of scope for improving on the obvious communication algorithms. More efficient algorithms for the most common communication operations are given in Appendix A. The current version of the T9000/C104 routines has been implemented under the VCR system.

For both T8 and T9000/C104 the lowest level send and receive routines are called by the pair of processes which wish to communicate. On the T8 only communication between directly connected processors is supported. To provide point-to-point communication between distant processors would require all the other processes to call the communication routine even though they were neither the source nor destination for the message. Allowing only neighbour communication means the source code is portable to the T9000/C104 architecture although T9000/C104 code is not portable back to the T8. However the resulting code will probably be inefficient since it will use several unnecessary point-to-point communications to reach its destination. The alternative of having every process synchronise on send and receive is again portable between T8 and T9000/C104 architectures but is a horrible kludge for T9000/C104 code. The T9000/C104 send and receive routines do provide point-to-point communication without requiring synchronisation with every process.

For the high level communication routines on both the T8 and T9000/C104 all processes must synchronise by calling the routine. No facility for declaring subgroups of processes has been implemented. This is a reasonable restriction for the programs studied in this work since every process is involved in the algorithm and will want to take part in the communication to send or receive data.

Currently we have only implemented high level routines for the T9000/C104. In this implementation we use two arrays of p^2 virtual channels to provide all-to-all connectivity. At the configuration level of the `occam` program each process is passed a vector of input channels, `in`, and a vector of output channels, `out`, connecting to every other process. The process does not use these vectors in any way but merely passes them to the communication subroutines. The communication routines use parallel threads to minimise the delays due to communication latency and idle time during synchronisation. For example, here is the code for the gather operation:

```
-- gather vector from all processes to root
PROC gather.out([]CHAN OF ANY in, out, VAL INT root, num.procs, id,
                VAL []REAL32 vec, VAL INT dim)
  SEQ
    out[root-1] ! dim
    out[root-1] ! [vec FROM 0 FOR dim]
:
PROC gather.in([]CHAN OF ANY in, out, VAL INT root, num.procs, id,
               VAL []REAL32 vec, VAL INT dim, []REAL32 dest, VAL INT dest.dim)
  INT min.size, extra:
  SEQ
    min.size := dest.dim/num.procs
    extra := dest.dim\num.procs
    PAR i = 1 FOR max.procs
      INT start, size:
      SEQ
        IF
          i <= num.procs
            SEQ
              IF
                i <= extra
                  SEQ
                    size := min.size+1
                    start := (min.size+1)*(i-1)
                TRUE
                  SEQ
                    size := min.size
                    start := ((min.size+1)*extra)+(min.size*((i-extra)-1))
              IF
                i = id
                  SEQ -- just copy my slice
```

```
                    [dest FROM start FOR size] := [vec FROM 0 FOR size]
                TRUE
                  SEQ -- input slice
                    in[i-1] ? size
                    in[i-1] ? [dest FROM start FOR size]
          TRUE
            SKIP -- cope with static PAR range
:
```

Notice that the operation has actually been implemented using two routines: `gather.in` is called by the destination process and receives blocks of data from every process, including copying a block of its own; `gather.out` is called by all the other processes and outputs a block to the destination process. These two routines should be combined into a single routine called by every process, but they were initially implemented separately to simplify the parameter list of the outputting processes.

One limitation of the current implementation is that there is only a single channel between any pair of processes. For example, the all-to-all broadcast algorithm described in Appendix A uses p parallel broadcast operations. However, with our current communications implementation we cannot execute parallel broadcast operations since they will conflict over use of the channels. Instead, we have implemented the all-to-all broadcast (or exchange) routine using the gather operation given above. This routine is given below:

```
PROC exchange([]CHAN OF ANY in, out, VAL INT num.procs, id,
              VAL []REAL32 vec, VAL INT dim, []REAL32 dest, VAL INT dest.dim)
  PAR
    gather.in(in, out, id, num.procs, id, vec, dim, dest, dest.dim)
    PAR i = 1 FOR max.procs
      IF
        i = id
          SKIP
        i <= num.procs
          gather.out(in, out, i, num.procs, id, vec, dim)
        TRUE
          SKIP -- cope with static PAR range
:
```

B.4 Slave code

With this suite of communications routines, programs become much easier to develop. The following listing is the slave code for the full matrix Method I BFGS update of Chapter 6.

```
WHILE continue
  SEQ
    -- receive new g, s from master
    broadcast.in(in, out, 1, num.procs, id, g, n)
    broadcast.in(in, out, 1, num.procs, id, s, n)
    -- distribute p, y to slaves as p.s, y.s
    scatter.in(in, out, 1, num.procs, id, p.s, num.rows)
    scatter.in(in, out, 1, num.procs, id, y.s, num.rows)
```

```
-- step 1: calculate distributed t in t.s
sgemv("N", num.rows, n, 1.0(REAL32), hessin, SIZE hessin,
      g, 1, 0.0(REAL32), t.s, 1)
-- step 2: calc. z (distributed elements)
-- z[] := s[]-t[]+p[]
scopy(num.rows, [s FROM start.row FOR num.rows], 1,
      [z FROM start.row FOR num.rows], 1)
saxpy(num.rows, -1.0(REAL32), t.s, 1,
      [z FROM start.row FOR num.rows], 1)
saxpy(num.rows, 1.0(REAL32), p.s, 1,
      [z FROM start.row FOR num.rows], 1)
-- step 3:
-- ga := s[]y[]
global.sum(in, out, 1, num.procs, id,
           [s FROM start.row FOR num.rows],
           y.s, num.rows, FALSE, ga)
-- step 4:
-- de := z[]y[]
global.sum(in, out, 1, num.procs, id,
           [z FROM start.row FOR num.rows],
           y.s, num.rows, FALSE, de)
-- step 5:
-- receive de, ga
[1]REAL32 de.vec RETYPES de:
broadcast.in(in, out, 1, num.procs, id, de.vec, dummy)
[1]REAL32 ga.vec RETYPES ga:
broadcast.in(in, out, 1, num.procs, id, ga.vec, dummy)
-- z[] := (z[]-de*s[])/ga
saxpy(num.rows, -de, [s FROM start.row FOR num.rows], 1,
      [z FROM start.row FOR num.rows], 1)
sscal(num.rows, 1.0(REAL32)/ga,
      [z FROM start.row FOR num.rows], 1)
-- get complete vector z to all slaves
exchange(in, out, num.procs, id,
         [z FROM start.row FOR num.rows], num.rows,
         z, n)
-- step 6:
-- hessin[][] := hessin[][]+z[]s[]+s[]z[]
-- for full matrix distributed by block rows
sger(num.rows, n, 1.0(REAL32),
     [z FROM start.row FOR num.rows], 1,
     s, 1, hessin, SIZE hessin)
sger(num.rows, n, 1.0(REAL32),
     [s FROM start.row FOR num.rows], 1,
     z, 1, hessin, SIZE hessin)
-- step 7:
-- ga := s[]g[] distributed
global.sum(in, out, 1, num.procs, id,
           [s FROM start.row FOR num.rows],
           [g FROM start.row FOR num.rows],
           num.rows, TRUE, ga)
-- step 8:
-- de := z[]g[] distributed
global.sum(in, out, 1, num.procs, id,
           [z FROM start.row FOR num.rows],
           [g FROM start.row FOR num.rows],
           num.rows, TRUE, de)
-- step 9:
-- p[] := t[]+ga*z[]+de*s[]
-- p.s is distributed as is t.s; s and z are complete
-- negate to give same sign as m1seq
scopy(num.rows, t.s, 1, p.s, 1)
saxpy(num.rows, ga, [z FROM start.row FOR num.rows], 1, p.s, 1)
saxpy(num.rows, de, [s FROM start.row FOR num.rows], 1, p.s, 1)
-- collect p back to master
```

```
    gather.out(in, out, 1, num.procs, id,
                    p.s, num.rows)
    -- ask master whether to do an iteration
    [1]REAL32 continue.vec RETYPES continue.bytes:
    broadcast.in(in, out, 1, num.procs, id, continue.vec, dummy)
```

The program has been reduced to a sequence of calls to the communication routines and assembler coded BLAS routines that operate on local data. No detail of the underlying architecture is visible to the program, so the program is consequently portable to other parallel architectures including parallel Fortran environments. The efficiency of the program is not lost in porting the code to another architecture since we expect the new architecture to have efficient implementations of the BLAS routines and the communication routines.

This algorithm makes use of high level communication routines and local BLAS routines. Another group of routines that will be used by other algorithms are distributed BLAS routines. These routines would have the same specification as the sequential local data BLAS, except that the data manipulated would no longer be local to a single process, but would be distributed across the processes. To use one of these routines all the slave processes call the routine passing their local blocks of the distributed data objects to be manipulated. Some routines might invoke communication routines to exchange data, in which case the processes will be synchronised, whilst others may not require communication.

For example, Step 5 of the full matrix Method I BFGS algorithm involves an AXPY operation. In the T9000/C104 slave code this operation has been implemented as a call to the local data AXPY routine. Instead, a distributed AXPY routine could be invoked using the same parameter list but with the addition of communication parameters. The programmer's code would be the same except that a different BLAS routine is called. The conceptual difference to the programmer is that whereas for the first method, the programmer has to be fully aware of all the data dependencies and communication requirements for the operation, but in the second method these issues are taken care of by the implementation of the distributed BLAS routine. In the case of the AXPY operation, the first method requires the programmer to realise that the algorithmic AXPY operation can be implemented by entirely local data AXPY routines with the given data distribution for this algorithm. If the programmer called the distributed AXPY routine instead then the routine itself would ensure that any necessary communications are performed to obtain the same result as the algorithmic AXPY operation.

This distinction becomes more important for BLAS routines that do require communications. For example the rank 2 update in Step 6 has been implemented by two local data GER rank 1 update routines preceded by an all-to-all broadcast of the distributed vector z. If the programmer instead used a distributed rank 1 update routine then the need for the communication of vector z could be taken care of by the routine itself and the programmer need not be concerned with it. By extension, the programmer could scatter the vector s using the same distribution as for z and let the distributed GER routine take care of the communication required to broadcast both s and z. Using these distributed BLAS routines simplifies the programmers task considerably since any communication is performed by the routine itself instead of being the responsibility of the programmer. This shortens slave code length and decreases development time. However, the programmer should still be aware of the underlying communication operations in order to design efficient programs.

B.5 Parallelism

An important feature of the style of programming illustrated by the BFGS code just given is that it is sequential. The programmer takes the sequential algorithm and writes sequential code which operates on distributed data. The communications routines which the programmer calls may exploit parallelism for improved performance but the programmer's own code is almost a direct implementation of the sequential algorithm. Parallel performance is achieved since the code is executed on every processor. This approach makes implementation much easier than using explicit parallelism within the code for each process.

But we must ask ourselves how much performance may be improved by using explicit parallelism. Each communication subroutine synchronises all the processes, which may introduce idle time. Is there a way we can introduce asynchronous operation into an algorithm or otherwise avoid this idle time?

One technique which can be used effectively is to overlap communications operations with local computation. This technique is used in the T9000/C104 sorting algorithm of Chapter 4. The performance improvements obtained by using explicit parallelism to overlap communication and computation will depend on the ratio of the amount of communication work to computation work. If the computation cost dominates then the complete cost of the communication operation can be hidden behind the computation. This would give significant improvements in performance. On the other hand, if the communication cost is greater then the computation cost, as is the case for the sorting example, then the computation is hidden behind communication. This also gives better performance than executing the two operations sequentially, however the speedup over a sequential algorithm is less than in the other case since for most architectures the cost of an individual communication is significantly larger than the cost of an individual computation operation. To use this technique, the data operated on in the two threads of the process must be independent. In the case of the sorting algorithm data independence was obtained by splitting a single communicate-compute-communicate cycle into two similar independent threads each working on one half of the data. Then whilst one thread was in its communicate phase, the other thread was in its compute phase.

Given enough independent data, this idea could be extended to use many more parallel threads. Cook [23, 24] describes the improvement in performance obtained for two algorithms by reformulating the algorithms to introduce data and program flow independence and then using several threads to perform the independent algorithmic tasks. The number of parallel threads that can be introduced by reformulating an algorithm is usually low and a large amount of extra coding may be required to co-ordinate the execution of the threads.

Another way to achieve the same aim of hiding communication operations behind computation is suggested by Valiant [91, 92]. He shows that provided an algorithm has enough parallel slackness (or excess parallelism) it can be implemented on a distributed memory MIMD machine with optimal efficiency. For a T9000/C104 machine this is implemented by creating many more slave processes than there are processors and then mapping several processes onto each processor. Each process need only be sequential. This technique attempts to hide any communication latency when one process starts a communication by using the processor cycles to perform computation in another process. The programmer's task is simplified considerably since he does not need to use any explicit parallelism. However, increasing the number of processes decreases the amount of data held on each process and hence decreases the amount of computation performed by a process. This may lead to the communication cost for a process being greater than the computation cost. Such fine grained algorithms do not perform well on current distributed memory architectures since the ratio of communication cost to computation cost is generally too high. For coarse grained algorithms, which have a large ratio of computation to communication, this technique may work well. But such algorithms will also perform well by overlapping communication with computation. We have not tried this approach yet, but it will be very interesting to see a comparison of this technique and overlapping communications and computation on a T9000/C104 system.

Appendix C

Gaussian elimination results

This appendix contains the measured run-times for the Gaussian elimination method presented in Chapter 3.

C.1 T8 measurements

The following tables give the wall-clock execution time for the matrix factorisation and RHS forwards elimination and backwards substitution. In the tables for parallel cost measurements, the scatter column is the time taken to scatter the matrix, the factorise column is the time taken to factorise the matrix, and the RHS column includes the time taken to scatter the RHS vector and gather the result vector as well as the time for forwards elimination and backwards substitution. Times were measured using the built-in timer on the master transputer and averaged over five program runs. All times are in seconds and are given to up to three significant figures. The variation in run-time is very small. For example, for $n = 400$ the variation in total time was under ± 0.005s (under 0.1%), and for $n = 50$ the variation was ± 0.001s (under 1%).

n	factorise (s)	rhs (s)	total (s)
50	0.162	0.011	0.173
100	1.23	0.045	1.27
200	9.54	0.179	9.72
400	75.2	0.713	76.0
600	253	1.60	254

Table C.1: Timings for sequential Gaussian elimination on T8 machine

n	p	scatter (s)	factorise (s)	rhs (s)	total (s)
50	1	0.023	0.179	0.022	0.223
	2	0.024	0.112	0.014	0.150
	4	0.026	0.099	0.012	0.137
	8	0.030	0.129	0.013	0.172
	16	0.038	0.218	0.017	0.273
	32	0.063	0.400	0.022	0.484
	48	0.078	0.628	0.033	0.739
100	1	0.091	1.33	0.077	1.50
	2	0.092	0.745	0.046	0.883
	4	0.096	0.526	0.033	0.653
	8	0.103	0.553	0.029	0.686
	16	0.118	0.864	0.034	1.02
	32	0.179	1.52	0.049	1.75
	48	0.219	2.44	0.073	2.74
200	1	0.357	10.3	0.300	10.9
	2	0.360	5.44	0.163	5.96
	4	0.367	3.35	0.102	3.82
	8	0.382	2.79	0.076	3.24
	16	0.411	3.38	0.072	3.86
	32	0.593	6.12	0.098	6.81
	48	0.690	8.73	0.130	9.55
400	1	1.42	80.8	1.18	83.4
	2	1.42	41.6	0.617	43.6
	4	1.44	23.1	0.348	24.9
	8	1.47	15.8	0.224	17.5
	16	1.52	16.1	0.182	17.8
	32	2.14	25.0	0.215	27.3
	48	2.41	34.8	0.262	37.4
600	1	3.18	271	2.64	277
	2	3.19	138	1.36	143
	4	3.21	74.2	0.74	78.2
	8	3.26	46.5	0.44	50.2
	16	3.34	41.9	0.33	45.4
	32	4.61	56.4	0.34	61.3
	48	5.15	78.8	0.40	84.3

Table C.2: Timings for parallel Gaussian elimination on T8 machine

C.2 T4 measurements

n	factorise (s)	rhs (s)
16	0.06	0.01
32	0.45	0.04
64	3.51	0.17
128	27.7	0.69
256	220	2.78

Table C.3: Timings for sequential Gaussian elimination on T4 machine

n	p	factorise (s)	rhs (s)
16	2	0.05	0.009
	4	0.04	0.007
	8	0.04	0.007
	16	0.05	0.009
32	2	0.27	0.032
	4	0.18	0.021
	8	0.15	0.017
	16	0.17	0.018
64	2	1.91	0.118
	4	1.09	0.067
	8	0.75	0.046
	16	0.70	0.041
128	2	14.3	0.455
	4	7.53	0.239
	8	4.50	0.142
	16	3.42	0.105
256	2	111	1.79
	4	56.1	0.903
	8	30.8	0.492
	16	19.9	0.316

Table C.4: Timings for parallel Gaussian elimination on T4 machine

Appendix D

Bitonic sorting results

This appendix contains the measured run-times for the T8 bitonic sort algorithm presented in Chapter 4.

The parallel bitonic sort algorithm for the T8 architecture described in Chapter 4 was tested on the Parsys Supernode described in Chapter 1. Test data was derived from a random number generator. Timings were made for a range of problem sizes, n, and machine sizes, p. These times were compared with an efficient sequential quicksort algorithm. The times were measured using the built-in timer on the master transputer and averaged over four program runs. All times are in seconds and are given to up to three significant figures. The variation in run-time is small; the total execution time for the sequential and parallel algorithms varied by less than 2% over the four program runs for each particular problem. This indicates that the random number generator was producing balanced sets of data and not extreme cases which would be expected to give markedly diffferent run-times.

Table D.1 shows the measured run-times for the sequential quick sort program. In Table D.2 we show the run-time of the parallel program measured by the master processor for the different problem sizes and machine sizes. In addition we illustrate the proportion of the total time that is taken by four parts of the parallel algorithm. These four parts are the initial scatter of the unsorted data, the sequential quick sort on each processor, the parallel bitonic sort, and the final gather of the sorted data. The times given for these phases of the algorithm is the time measured by the first slave processor. The times for other processors in the array vary. As the processor number increases, i.e., the processors get further away from the master processor, the costs vary as follows: the scatter cost decreases, the sequential quick sort cost remains constant, and the bitonic sort and gather costs increase. Hence these figures are presented only to illustrate approximately the proportion of the total time spent in each phase of the algorithm.

n	time (s)
2048	0.046
8192	0.213
32768	0.970
131072	4.29
524288	19.0

Table D.1: Timings for sequential quicksort on T8 machine

n	p	scatter (s)	QS (s)	bitonic (s)	gather (s)	total (s)
2048	1	0.009	0.044	0	0.009	0.063
	2	0.010	0.020	0.033	0.010	0.073
	4	0.014	0.008	0.045	0.014	0.081
	8	0.021	0.004	0.060	0.021	0.106
	16	0.026	0.002	0.077	0.026	0.132
	32	0.031	0.001	0.095	0.031	0.159
8192	1	0.037	0.215	0	0.037	0.285
	2	0.038	0.099	0.146	0.038	0.319
	4	0.042	0.044	0.187	0.042	0.314
	8	0.049	0.020	0.201	0.049	0.318
	16	0.064	0.008	0.228	0.064	0.363
	32	0.104	0.003	0.326	0.104	0.538
32768	1	0.146	0.961	0	0.146	1.25
	2	0.148	0.435	0.639	0.148	1.38
	4	0.152	0.198	0.795	0.152	1.30
	8	0.160	0.090	0.800	0.160	1.21
	16	0.175	0.042	0.788	0.175	1.18
	32	0.250	0.019	0.912	0.250	1.43
131072	1	0.585	4.27	0	0.585	5.44
	2	0.590	2.07	2.78	0.590	6.00
	4	0.596	0.983	3.46	0.596	5.64
	8	0.604	0.456	3.23	0.604	4.89
	16	0.619	0.209	2.98	0.619	4.43
	32	0.830	0.095	3.40	0.830	5.14
524288	1	2.34	18.6	0	2.34	23.3
	2	2.36	9.20	11.8	2.36	25.5
	4	2.37	4.27	14.2	2.37	23.2
	8	2.38	1.99	13.8	2.38	20.5
	16	2.40	0.92	12.6	2.40	18.3
	32	3.15	0.46	13.5	3.16	20.3

Table D.2: Timings for parallel bitonic sort on T8 machine

Bibliography

[1] Cliff Addison, Bridget Beattie, Norman Brown, Rod Cook, Gabriel Howard, Tim Oliver, Bruce Stevens, and David Watson. Distributed objects with parallel performance. In *Sixth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 22–24 March 1993.

[2] Cliff Addison, Bridget Beattie, Norman Brown, Rod Cook, Gabriel Howard, Tim Oliver, Bruce Stevens, and David Watson. Prototype distributed data system. In *P.L.U.G Proceedings*. Parsys, October 1992.

[3] Cliff Addison, Gabriel Howard, Tim Oliver, and Chris Phillips. Interim report on algorithm design. PUMA Deliverable 5.2.1, University of Liverpool, August 1990.

[4] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[5] S. G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[6] R. J. Allan, L. Heck, and S. Zurek. Parallel Fortran in scientific computing: a new occam harness called Fortnet. *Computer Physics Comms.*, 59:325–344, 1990.

[7] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic linear algebra communication subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*. IEEE Computer Society Press, 1991.

[8] Stan Augarten. *BIT by BIT: An Illustrated History of Computers*. George Allen & Unwin, 1985.

[9] Sally Baker. Dynamic reconfiguration on the supernode. PUMA Working paper 3, RSRE, January 1990.

[10] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(2):261–322, September 1989.

[11] Geoff Barrett. *draft occam 3 reference manual*, March 1992.

[12] K. E. Batcher. Sorting networks and their applications. In *AFIPS Conference Proceedings, 1968 Spring Joint Computing Conference*, pages 307–314, 1968.

[13] Bridget J. H. Beattie, Norman G. Brown, and Bruce R. Stephens. Distributed data. Technical report, Centre for Mathematical Software Research, University of Liverpool, October 1991.

[14] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. *A users' guide to PVM: parallel virtual machine*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, July 1991.

[15] Ron A. Bell. IBM RISC System/6000 NIC tuning guide for FORTRAN and C. Technical Report GG24-3611-01, IBM United Kingdom, Ltd., July 1991.

[16] R. H. Bisseling and L. D. J. C. Loyens. Towards peak parallel LINPACK performance on 400 transputers. *Supercomputer*, 8(5):20–27, 1991.

[17] Rob H. Bisseling and Johannes G. G. van de Vorst. Parallel LU decomposition on a transputer network. In *Parallel Computing 1988*, number 384 in Lecture Notes in Computer Science, pages 61–77. Springer-Verlag, 1988.

[18] Richard H. Byrd, Robert B. Schnabel, and Gerald A. Shultz. Parallel quasi-newton methods for unconstrained optimization. Technical report, Department of Computer Science, University of Colorado, April 1988.

[19] Richard H. Byrd, Robert B. Schnabel, and Gerald A. Shultz. Using parallel function evaluations to improve hessian approximation for unconstrained optimization. In Peter L. Hammer, Robert R. Meyer, and Stavros A. Zenios, editors, *Parallel Optimization on Novel Computer Architectures*, volume 14 of *Annals of Operations Research*, pages 167–193. J. C. Baltzer AG Scientific Publishing Company, 1988.

[20] Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers.

[21] E. Chu and A. George. Gaussian elimination on a multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[22] K. L. Clark and S. Gregory. PARLOG:parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[23] Rod Cook. Timing models for preconditioned conjugate directions. PUMA Working paper 13, University of Liverpool, August 1990.

[24] Rod Cook. Preconditioned conjugate gradients squared. PUMA Working paper 31, University of Liverpool, September 1991.

[25] Rod Cook, Gabriel Howard, and Tim Oliver. Writing applications for the T9000 transputer. In *Applications of Transputers 3*. IOS Press, 1991.

[26] Peter F. Corbett and Isaac D. Scherson. Sorting in mesh connected multiprocessors. *IEEE Transactions on parallel and distributed systems*, 3(5):626–632, September 1992.

[27] Parasoft Corporation. *Express 3.0 User's Guide and Reference*, 1990.

[28] W. C. Davidon. Optimally conditioned optimization algorithms without line searches. *Mathematical Programming*, 9:1–30, 1975.

[29] M. J. Daydé and I. S. Duff. Use of parallel level 3 BLAS in LU factorization on three vector multiprocessors: the ALLIANT FX/80, the CRAY-2 and the IBM 3090 VF. In *International Conference on Supercomputing*, pages 82–95, 1990.

[30] Mark Debbage, Mark Hill, and Denis Nicole. Virtual channel router version 2.0 user guide. PUMA Working paper 25, University of Southampton, June 1991.

[31] Mark Debbage, Mark Hill, Denis Nicole, and E. J. Zaluska. Universal message router for fixed deadlock-free networks. PUMA deliverable, University of Southampton.

[32] L. M. Delves and N. G. Brown. The design of the Supernode numerical library. Technical report, Centre for Mathematical Software Research, University of Liverpool, March 1989.

[33] James Demmel, Jack Dongarra, and W. Kahan. On designing portable high performance numerical libraries. 24 March 1992.

[34] James Demmel, Jack J. Dongarra, et al. Prospectus for the development of a linear algebra library for high-performance computers. Technical report, Argonne National Laboratory, September 1987.

[35] J.E. Dennis, Jr and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, 1983.

[36] J. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[37] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[38] Jack Dongarra, Robert van de Geijn, and David Walker. A look at scalable dense linear algebra libraries. 5 May 1992.

[39] Jack J. Dongarra, Peter Mayes, and Giuseppe Radicati di Brozolo. The IBM RISC System/6000 and linear algebra operations. Technical Report CS-90-122, Computer Science Department, University of Tennessee, December 1990.

[40] Jack J. Dongarra and Robert A. van de Geijn. Two dimensional basic linear algebra communication subprograms. LAPACK Working Note 37, 28 October 1991.

[41] Ralph Duncan. A survey of parallel computer architectures. *IEEE Computer*, pages 5–16, February 1990.

[42] M. J. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[43] High Performance Fortran Forum. Draft high performance Fortran language specification. Version 1.0, 3rd May 1993.

[44] Message Passing Interface Forum. Draft document for a standard message-passing interface. 10 May 1993.

[45] R. S. Francis and L. J. H. Pannan. A parallel partition for enhanced parallel quicksort. *Parallel Computing*, 18(5):543–550, 1992.

[46] T. L. Freeman. Parallel projected variable metric algorithms for unconstrained optimisation. Technical report, Centre for Mathematical Software Research, University of Liverpool, September 25 1989.

[47] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Rev.*, 32(1):54–135, March 1990.

[48] David Gee. Report on modelling and test suite evaluation. PUMA Deliverable 5.3.1, University of Southampton, July 1991.

[49] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.

[50] Donald Goldfarb. Factorized variable metric methods for unconstrained optimization. *Mathematics of Computation*, 30(136):796–811, October 1976.

[51] M. Goldsmith, C. Liddiard, D. May, C. O'Neill, M. Poole, B. Roscoe, A Sturges, and P. Thompson. Architecture consolidated report. PUMA Deliverable 1.1.1, INMOS, August 1990.

[52] Gene H. Golub and Charles F. van Loan. *Matrix computations*. Johns Hopkins Series in the Mathematical Sciences. Johns Hopkins University Press, 2nd edition, 1989.

[53] J. Gurd, C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Comm. ACM*, 28:34–52, 1985.

[54] R. Hempel. *The ANL/GMD macros (PARMACS) in Fortran for portable parallel programming using the message passing programming model—user's guide and reference manual*. GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, November 1991.

[55] R. Hempel, H.-C. Hoppe, and A. Supalov. Parmacs-6.0 library interface specification. Technical report, Postfach 1316, D-5205 Sankt Augustin 1, Germany, December 1992.

[56] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, pages 18–29, September 1991.

[57] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, August 1978.

[58] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: architecture, programming and algorithms*. Adam Hilger, second edition, 1988.

[59] Holm Hofestädt, Axel Klein, and Erwin Reyzl. Investigation of dynamically switched, scalable network structures. PUMA Deliverable 2.1.1, Siemens AG, October 1990.

[60] Holm Hofestädt, Axel Klein, and Erwin Reyzl. Investigation of dynamically switched, scalable network structures. PUMA Deliverable 2.1.2, Siemens AG, September 1991.

[61] Gabriel Howard. Timing models for Gauss elimination on the H1 transputer. PUMA Working paper 11, University of Liverpool, May 1990.

[62] Gabriel N Howard. Solving simultaneous linear equations on transputer arrays. Working paper, Centre for Mathematical Software Research, University of Liverpool, 1988.

[63] INMOS. The transputer instruction set—a compiler writer's guide. Technical report.

[64] INMOS. H1 transputer: Advance information. Technical report, June 1990.

[65] INMOS. IMS C104 packet routing switch: Advance information. Technical report, September 1990.

[66] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[67] K.W.Brodlie, A.R.Goulay, and J.Greenstadt. Rank-one and rank-two corrections to positive definite matrices expressed in product form. *Journal of the Institute of Mathematics and its Applications*, 11:73–82, 1973.

[68] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[69] W. Loots and T. H. C. Smith. A parallel three phase sorting procedure for a k-dimensional hypercube and a transputer implementation. *Parallel Computing*, 18(3):335–344, 1992.

[70] F. A. Lootsma. Parallel non-linear optimization. Technical Report 89-45, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1989.

[71] Meiko Ltd. *SunOS CSTools*, 1990.

[72] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, March 1981.

[73] N.A. Software Ltd. *Library Manual, Liverpool Parallel Transputer Library, Occam Implementation Ver. 2.0.* N.A. Software Ltd., Meseyside Innovation Centre, 131 Mount Pleasant, Liverpool L3 5TF.

[74] Tim Oliver. Gaussian elimination on transputer arrays. Working paper, Centre for Mathematical Software Research, University of Liverpool, February 1988.

[75] Tim Oliver. Sorting algorithms for transputer arrays. Working paper, Centre for Mathematical Software Research, University of Liverpool, October 1988.

[76] Tim Oliver. A parallel Newton method for unconstrained optimisation. PUMA working paper, University of Liverpool, 28 September 1989.

[77] Tim Oliver. A communications model for a PUMA machine. PUMA Working paper 17, University of Liverpool, October 1990.

[78] Tim Oliver. An overview of parallel methods for unconstrained optimisation. PUMA working paper, University of Liverpool, 30 January 1990.

[79] Tim Oliver. The bitonic sort on transputer architectures. PUMA Working paper 33, University of Liverpool, September 1991.

[80] Tim Oliver. Parallel algorithms for the BFGS update on a PUMA machine. PUMA Working paper 32, University of Liverpool, September 1991.

[81] P.E.Gill, G.H.Golub, W.Murray, and M.A.Saunders. Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535, April 1974.

[82] Craig Peterson, James Sutton, and Paul Wiley. iWarp: A 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro*, pages 26–29,81–87, June 1991.

[83] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[84] M. J. D. Powell. Updating conjugate directions by the BFGS formula. *Mathematical Programming*, 38:29–46, 1987.

[85] R.Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 2nd edition, 1987.

[86] R. B. Schnabel and P.Frank. Tensor methods for nonlinear equations. *SIAM Journal of Numerical Analysis*, 21:815–843, 1984.

[87] Robert B. Schnabel. Concurrent function evaluations in local and global optimization. *Computer Methods in Applied Mechanics and Engineering*, 64:537–552, 1987.

[88] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30–38, September 1991.

[89] T. A. Straeter. A parallel variable metric optimization algorithm. NASA Technical Note L-8986, Langley Research Center, 1973.

[90] V. S. Sunderam. PVM: A framework for parallel distributed computing.

[91] L. G. Valiant. Optimally universal parallel computers. *Philosophical Transactions of the Royal Society*, pages 373–376, 1988.

[92] L. G. Valiant. General purpose parallel architectures. Technical report, Aiken Computation Laboratory, Harvard University, 14 April 1989.

[93] R. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. Technical Report CS-91-28, University of Texas at Austin, August 1991.

[94] P. J. M. van Laarhoven. Parallel variable metric algorithms for unconstrained optimization. *Mathematical Programming*, 33:68–81, 1985.

[95] John Wexler. *Concurrent programming in OCCAM 2*. Series in computers and their applications. Ellis Horwood, 1989.

[96] Barry Wilkinson. *Computer architecture: design and peformance*. Prentice Hall, 1991.

[97] Mark C. K. Yang, Jun S. Huang, and Yuan-Chieh Chow. Optimal parallel sorting scheme by order statistics. *SIAM Journal of Computing*, 16(6):990–1003, December 1987.