A THESIS
entitled

MAPPING NUMERICAL SOFTWARE ONTO DISTRIBUTED MEMORY PARALLEL SYSTEMS

Submitted in partial fulfilment of the
requirements for the award of the

DEGREE OF DOCTOR OF PHILOSOPHY

of the

COUNCIL FOR NATIONAL ACADEMIC AWARDS

by

STEPHEN PHILIP JOHNSON, Bsc.

Centre For Numerical Modelling And Process Analysis
School Of Mathematics, Statistics And Computing
Faculty Of Technology
Thames Polytechnic
LONDON

FEBRUARY 1992

Mapping Numerical Software Onto
Distributed Memory Parallel Systems

The aim of this thesis is to further the use of parallel computers, in particular distributed memory systems, by proving strategies for parallelisation and developing the core component of tools to aid scalar software porting. The ported code must not only efficiently exploit available parallel processing speed and distributed memory, but also enable existing users of the scalar code to use the parallel version with identical inputs and allow maintenance to be performed by the scalar code author in conjunction with the parallel code.

The data partition strategy has been used to parallelise an in-house solidification modelling code where all requirements for the parallel software were successfully met. To confirm the success of this parallelisation strategy, a much sterner test was used, parallelising the HARWELL-FLOW3D fluid flow package. The performance results of the parallel version clearly vindicate the conclusions of the first example. Speedup efficiencies of around 80 percent have been achieved on fifty processors for sizable models. In both these tests, the alterations to the code were fairly minor, maintaining the structure and style of the original scalar code which can easily be recognised by its original author.

The alterations made to these codes indicated the potential for parallelising tools since the alterations were fairly minor and usually mechanical in nature. The current generation of parallelising compilers rely heavily on heuristic guidance in parallel code generation and other decisions that may be better made by a human. As a result, the code they produce will almost certainly be inferior to manually produced code. Also, in order not to sacrifice parallel code quality when using tools, the scalar code analysis to identify inherent parallelism in a application code, as used in parallelising compilers, has been extended to eliminate dependencies conservatively assumed, since these dependencies can greatly inhibit parallelisation.

Extra information has been extracted both from control flow and from processing symbolic information. The tests devised to utilise this information enable the non-existence of a significant number of previously assumed dependencies to be proved. In some cases, the number of true dependencies has been more than halved.

The dependence graph produced is of sufficient quality to greatly aid the parallelisation, with user interaction and interpretation, parallelism detection and code transformation validity being less inhibited by assumed dependencies. The use of tools rather than the black box approach removes the handicaps associated with using heuristic methods, if any relevant heuristic methods exist.

Acknowledgements

I would like to thank Professor Mark Cross and Professor Martin Everett for their guidance and encouragement over the duration of this work and for their patience during the slow final stages of this thesis.

The useful input from, and discussions with, colleagues in the Centre For Numerical Modelling And Process Analysis is also gratefully acknowledged, particularly that from Cos Ierotheou, Peter Leggett and Steve Little.

The contribution of my parents has also been vitally important throughout the work and particularly in the writing of this thesis where their reading and correction of my English has been invaluable, despite little understanding of the content.

Finally, I would like to acknowledge the funding of the Science And Engineering Research Council without whom this work would not have been possible.

Contents

CHAPTER ONE

1. Introduction.

Computers are now used in almost all aspects of life. For many of the more mundane uses in the financial and industrial communities, the current generation of affordable computer systems are adequate. For the more significant computational tasks these machines prove totally insufficient. In particular, computers are often used in the industrial engineering community to perform computer simulations of complex physical processes providing information not available or not feasible by any other route. These simulations have proved invaluable in these industries, however, the use and quality of these simulations is greatly restricted by the limitations of today's computers.

A major field of interest for computer simulation is the use of numerical modelling software such as control volume codes and finite element codes, where the flow patterns and/or stress/strain fields of physical situations can be evaluated. The results of such simulations have many applications in industry and potentially many more if computer limitations were reduced.

The success of computer models is dependent on how accurately the physical processes are implemented in the software and on the definition of the model for the geometry and resolution of variables throughout the model space. Both these factors are influenced by two key constraints, computational time and computer memory capacity.

These constraints cause many approximations to be made in the implementation of the physical processes acting in a simulation. Micro effects are very often expressed as macro

effects and many influences are merely ignored in the hope that they have only a small effect on the overall results. The time and memory cost of adding extra detail to physical process simulation is very often considered to far outweigh any improvement in the results achieved.

The resolution of the actual model built (for example, the number of cells in a control volume model or the number and type of elements in a finite element simulation) is restricted with approximate results being accepted rather than exorbitant execution times and memory requirements. When model builders are presented with a more powerful machine than the one on which the model was originally used, they invariably increase the resolution of their models at the cost of losing some of the reduction in run time.

Processor technology is continuously improving with faster performance ratings being achieved by each new generation of processors. Even with these often impressive improvements, the desired speed for processors from the computer user is limitless with expectation always being a multiple of that available. A large demand clearly exists for a means of improving computational speed over and above the increases in processor technology.

An obvious way to achieve program execution speed of a multiple of the single processor speed is to facilitate many such processors to co-operate on the overall computation. The two questions that must have positive answers for such a system to be practical are firstly, can the required hardware be constructed, and secondly, do application codes have the required nature to allow many processors to efficiently share the overall computation ?.

The first question has clearly been answered by the large number of machines which use multiple processors in some way. Machines such as the CRAY series , AMT DAP and the Transputer processor and many others all heavily exploit the use of many processing elements in parallel.

The second question has a guarded positive response since many application codes can clearly exploit parallel execution, whilst others require alteration (for example to an inherently implicit algorithm) to allow even reasonably efficient utilisation of parallel hardware.

The creation of software to exploit parallel computer architectures raises a series of questions to the programmer : Should parallel code be written in completely new languages or should old languages be adapted ?, Should many paradigms for sequential programming be replaced by new conventions ?, Is existing sequential software obsolete ?, Will code produced be totally machine dependent (or at least architecture dependent) ?.

A range of answers have been provided to these questions. New languages for parallel programming have been developed such as SISAL (McGraw et al. 1985), BLAZE (Mehrotra and Van Resendale 1985) and the transputer specific language OCCAM (Inmos 1988). Alternatively, many dialects of existing languages have been developed with language extensions to facilitate parallel features such as IBM Parallel Fortran (IBM 1988) and 3L Parallel Fortran (3L 1988) (twelve different parallel Fortran dialects are compared in Karp and Babb 1988), as well as many versions of parallel Pascal, C amongst others. This choice of existing or new language implies answers to many of the above questions since the existing rules for sequential programming are obviously kept if the parallel language is an adapted

sequential language and existing sequential codes can more easily be adapted for parallel execution. The final question on machine or architecture dependence has a less clear answer since the rules required for parallel code and the physical differences of the computer architectures have a major influence on the final code (see chapters 2 and 5).

The remainder of this chapter concentrates on the use of an adapted serial language, specifically Fortran, discussing the relevant issues and choices available. The adaptation of correct serial code to parallel form using transformation software is a very attractive and desirable facility. This, indeed, is expected by many potential parallel machine users and the use of such machines has been lessened as they await the appearance of such 'parallelising compilers'. The reasons for this expectation and subsequently the drive for much research, can be found historically in the development of compilation software.

The development of sequential languages and their compilers/interpreters allowed understandable, English type code to be automatically converted into machine code without the user being involved. It became apparent that the machine code generated could be greatly improved (in terms of both execution speed and storage requirements) using some simple rules. These rules, known as optimisations, such as those in Allen (1972), were then included in a phase of the compilation process. These optimisations produced significant improvements in the machine code which then encourage further research leading to more sophisticated optimisations.

With the advent of vector computers (see chapter 2), a new challenge was presented to compiler designers. Vector computers enable operations on vectors of data to be performed

in parallel, however the determination of semantic validity of a particular operation for parallel execution and addition of the required code (often calls to library routines) were complex tasks, thus encouraging a great deal of research into automated techniques for producing such code.

Simple rules for legal parallel execution were given by Bernstein (1966), relating parallelism to the definition and usage of variables. This work was then greatly extended throughout the 1970's by David Kuck and his group at Illinois. The development of formal rules for legal parallelism with tests for dependence relations between subscripted variables made automatic conversion of serial code to parallel form feasible. This work eventually led to the development of Paraphrase (Kuck et al. 1980), an ever evolving vectorising/parallelising compiler, and inspired much other research such as that of Ken Kennedy and his group at Rice University (Allen 1982). It also led to a host of commercial vectorising compilers (for example Higbee 1979,Arnold 1983) which achieved various levels of success, depending on the nature of the sequential code with which they were presented.

The vectorisation phase of a compiler was very much considered as an optimisation since it could be achieved by examining only small sections of the code (i.e. inner loops as described in chapter 5), without much need to investigate external influences. As with the standard optimisations, vectorisation could be fairly successful (with a few specialised optimisations) as a fully automated process (although user directives can generally be added to the sequential code prior to vectorisation).

The availability of multi processor parallel machines thus inevitably led to the expectation of

a parallelising compiler with the same level of success as for vectorising compilers. Although some parallelising compilers have been developed for shared memory machines (see chapter 2), they have been fairly disappointing with far more user adaption of the serial code often required to enable even moderately efficient parallel execution. This lack of success can largely be attributed to the nature of parallelism best exploited by such machines, that is coarse grain parallelism between large sections of code. The 'peephole' style analysis of only a small section of code which was often sufficient for vectorisation therefore fails for parallel machines since many more influences external to this section of code, such as variable values used, need to be considered (due to the amount of code being analyzed).

Even worse, no compilers exist for distributed memory machines (see chapter 5) and even those that claim to be semi-automatic, such as the Superb compiler developed in the Suprenum project (Zima 1988), where much information and the parallelisation strategy itself being supplied by the user, has only very limited success.

Two paradigms have been used as the impetus to the work on parallelising tools :-

Firstly, no sacrifice of parallel efficiency is to be allowed for a parallel code developed using the analysis system (and all subsequent facilities) as opposed to a code parallelised fully manually (i.e. as far as possible, conservative assumptions and heuristic decisions will be avoided and the user encouraged to aid the system).

Secondly, the time and effort required to produce efficient parallel code is far less important than the parallel code efficiency, since the parallelisation may well be performed only once

whilst the resultant parallel code will be used many times.

These paradigms place the importance of parallelisation purely on the resultant code and those who use it. They are very different from those used by many others where the development of automated (often heuristic) code generation related algorithms, to satisfy the desire of the compiler user, dominate.

This thesis thus concentrates on software for distributed memory parallel machines. The following chapters are organised as follows :-

Chapter 2 describes the various types of parallel machine. The parallel Fortran dialect used in much of this work is introduced and various other issues such as techniques and utilities for parallel processing are discussed. Finally, the measures of success of parallel codes are introduced and implications of the measures discussed.

A code for modelling solidification in the casting industry is written and parallelised onto a distributed memory parallel system in chapter 3 using some of the strategies introduced in chapter 2. The technique for exploiting parallelism in the code is developed and the success of the parallel implementation assessed, the factors influencing this success are also discussed.

A commercial computational fluid dynamics package, HARWELL-FLOW3D (Jones et al 1985) is parallelised in chapter 4 using the technique developed in chapter 3. The success of this parallelisation and its implications to the value of this technique for a wide range of commercial software are discussed.

Chapter 5 introduces and discusses the theory and techniques for automated parallelisation, developed largely at Illinois. The rules for parallel code generation (i.e. for semantically valid parallel code) for the various machine types are given along with some parallelism extracting optimisations.

In chapter 6, the improvements and extensions to the analysis introduced in chapter 5 are detailed. The basic algorithms for constructing the required graphs representing the scalar program are shown. The limitations of certain stages are then indicated and cures introduced and implemented, whether these involved changes to algorithms, extensions to algorithms or additional algorithms.

Chapter 7 shows some examples of the effectiveness of the extensions to the analysis system detailed in chapter 6. The remainder of this chapter discusses various issues in extending the analysis tool into a set of semi automatic tools or components of a parallelisation system.

CHAPTER TWO

2. Parallel Architectures.

In this chapter, the different types of computer architecture currently available are introduced and broadly categorised. Emphasis is given to true parallel machines and in particular will focus on the architecture and use of the Transputer.

2.1 Inherent Parallelism.

The original serial Von Neumann computer architecture only processed instructions one at a time through a single processing unit. The codes used on such machines however, often exhibited a large amount of potential parallelism (i.e. operations that could be performed simultaneously) and it was realised that great speed improvements could be gained if computer architectures could exploit this parallelism.

2.2 Architectures To Exploit Parallelism.

Several authors have presented different classification schemes for computer architectures with those of Flynn (1966), Shore (1973) and Hockney & Jesshope (1981) proving popular. The following machine classifications are those given by Flynn which are widely used, but are by no means comprehensive. This categorisation considers computer architectures from a user perception (i.e. a code author), rather than the strict operation of the architectures concerned.

2.2.1 SISD - Single Instruction Stream, Single Data Stream.

This categorises the original serial Von-Neumann computers where only one operational instruction is processed at a time on a single item of data. The internal operation of such machines may include some parallelism such as parallel loading and storing of data items along with actual arithmetic operations. This distinction is made since the use of such architectures requires no user knowledge of any internal parallel features.

2.2.2 MISD - Multiple Instruction Stream, Single Data Stream.

In this category, several instructions are concurrently performed on a single stream of data. In a strict sense, the operation of internally parallel SISD architectures and pipeline processors (as described in the next section) could be in this category, however, since we are concerned with the user perception of computer architectures, neither is included. As a result, no commercial computer architectures are in this category.

2.2.3 SIMD - Single Instruction Stream, Multiple Data Stream.

Computer architectures in this category are commonly known as vector or pipeline computer architectures, where a single control unit issues instructions to a number of processing elements.

In a true vector processor, every processing element performs the same operation at the same moment but acts upon different sets of data. This type of parallelism is often exhibited in

operations involving vectors of data.

A pipeline processor issues different instructions to each processing element where each of these instructions is a different stage in an overall operation. The data enters the pipeline at the processing element performing the first stage of the operation, passing through others until finally passing through the processing element performing the final stage. The parallelism occurs when several data items pass through such a pipeline, each item passing through different stages at the same moment. To achieve significant speedup, every processing element in the pipeline must be kept busy. To do this, many data items that require the same overall operation to be performed on them are passed through the same pipeline. This requirement is often held in vector operations where the data passing through the pipeline consists of each consecutive element of the vector(s) concerned.

Both types of machine are classified here since they both exploit the same fine grain parallelism (i.e.parallelism of multiple instances of individual statements allowing vector operations) and appear the same from a user perspective. SIMD architectures are shown in figures 2.1 and 2.2.

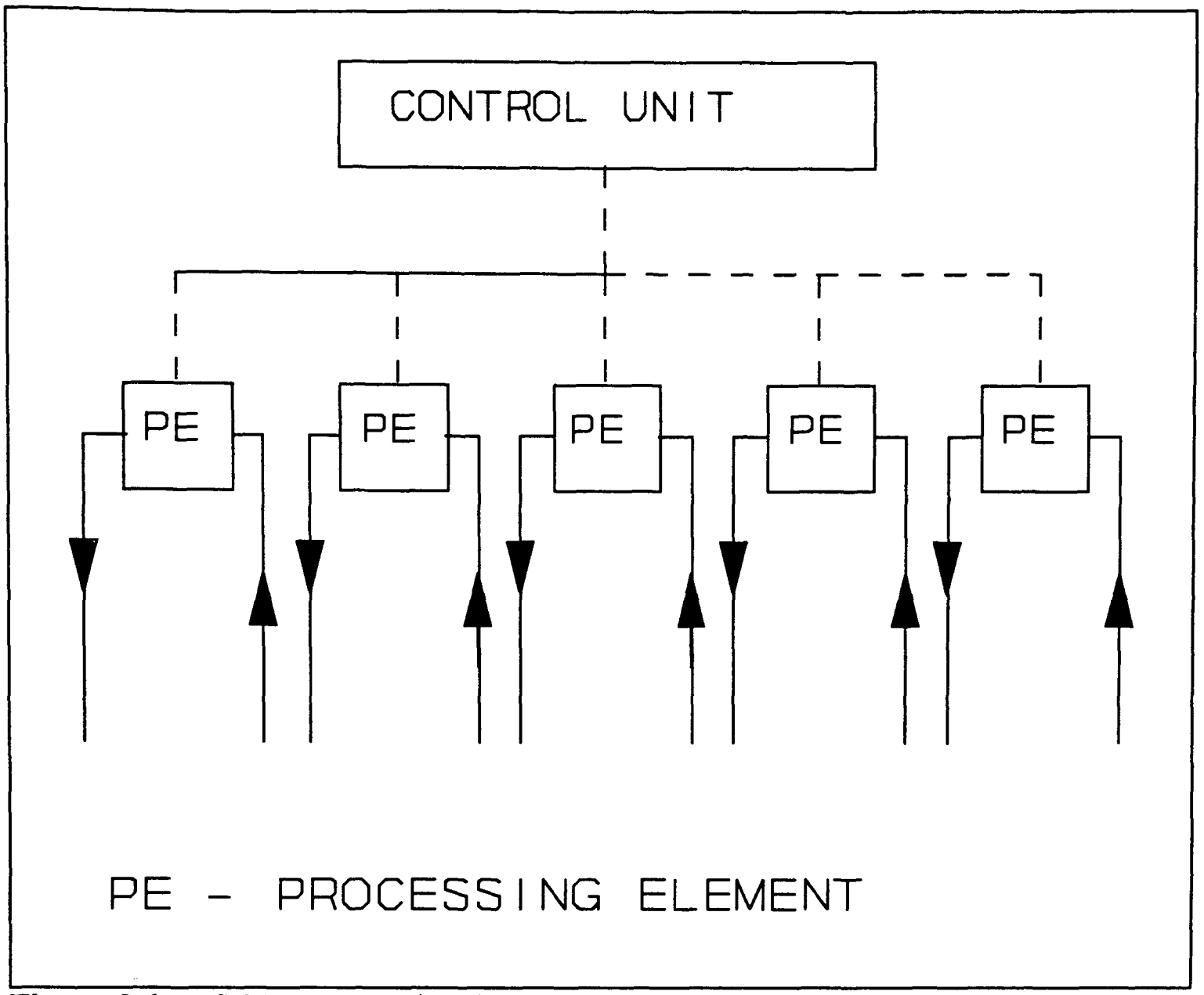


Figure 2-1 SIMD Vector Architecture.

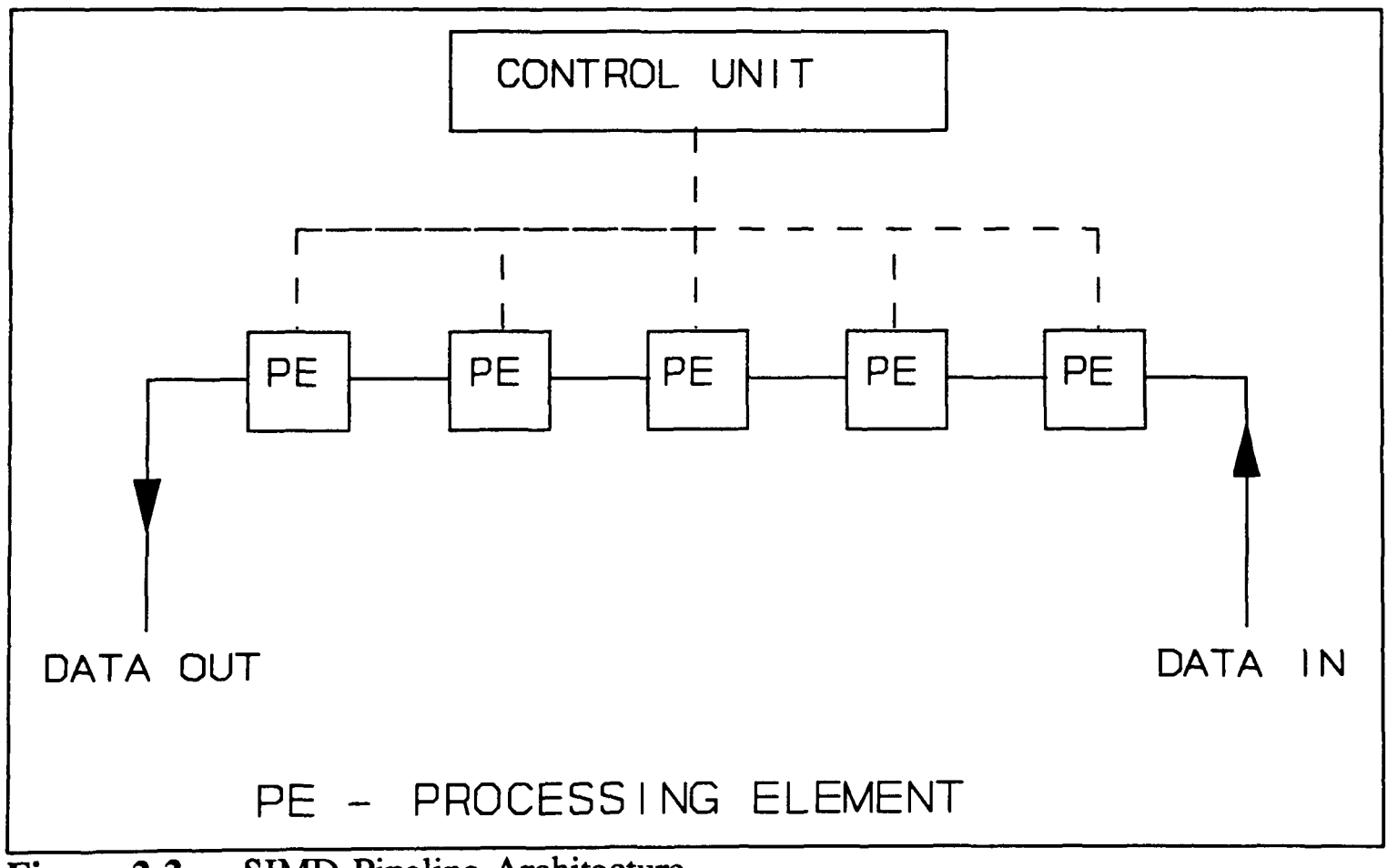


Figure 2-2 SIMD Pipeline Architecture.

For parallelism that can be exploited by SIMD architectures to exist in software, simple operations involving vectors of data must exist. In many cases this significantly restricts the exploitation of this type of machine.

2.2.4 MIMD - Multiple Instruction Stream, Multiple Data Stream.

This class contains machines where a number of separate processing units, each with its own control unit accepting its own stream of instructions, operate on separate sets of data. These processing units are then interconnected to enable them to co-operate in performing an overall task. To use such a computer architecture, the overall task must be broken down into a set of many sub tasks which can be performed in parallel.

In addition to these categories, hybrid systems of the MIMD type where each processing unit is a SIMD processor also exist.

2.3 MIMD Machine Types And Software Differences.

To illustrate the types of MIMD machine currently available, the two extreme classes of machine are discussed.

Shared memory systems have a global memory accessible directly from every processing unit via some form of communications bus. Each processor can thus be considered identical (providing each processing unit is of the same type) and a task in the computation can be

allocated arbitrarily to any free processor. The problem with such systems arises when large numbers of processing units are used, the communications bus hardware becomes a bottleneck with many requests for global memory accessing from every processor. Global memory accessing is also slowed by access conflicts between different processors which must be handled by some form of semaphore to ensure only one processor accesses a data item at a given moment, all others waiting for that data must be forced to idle. Figure 2.3 shows an example of a shared memory system.

Distributed memory systems have processing units consisting of a processing element and a local memory only accessible from that processor unit. An interconnection network joining many such processors allows an overall task to be performed on many processors by message passing i.e. data can be sent from one processor to another, possibly via other processors. This type of machine overcomes the bottleneck of a communications bus and many access conflicts encountered in shared memory systems, but can produce problems of data traffic bottlenecks in a large processor network. Also, unlike the shared memory systems, processor allocation is not arbitrary, for maximum efficiency a computation should be placed on a processor that either holds the data accessed in the computation or can access the data through as short as possible communications route. To efficiently use all the distributed memory of such a system, the program data should, if possible, be divided over all the local memories with a minimum of duplication. Figure 2.4 shows an example of a distributed memory system.

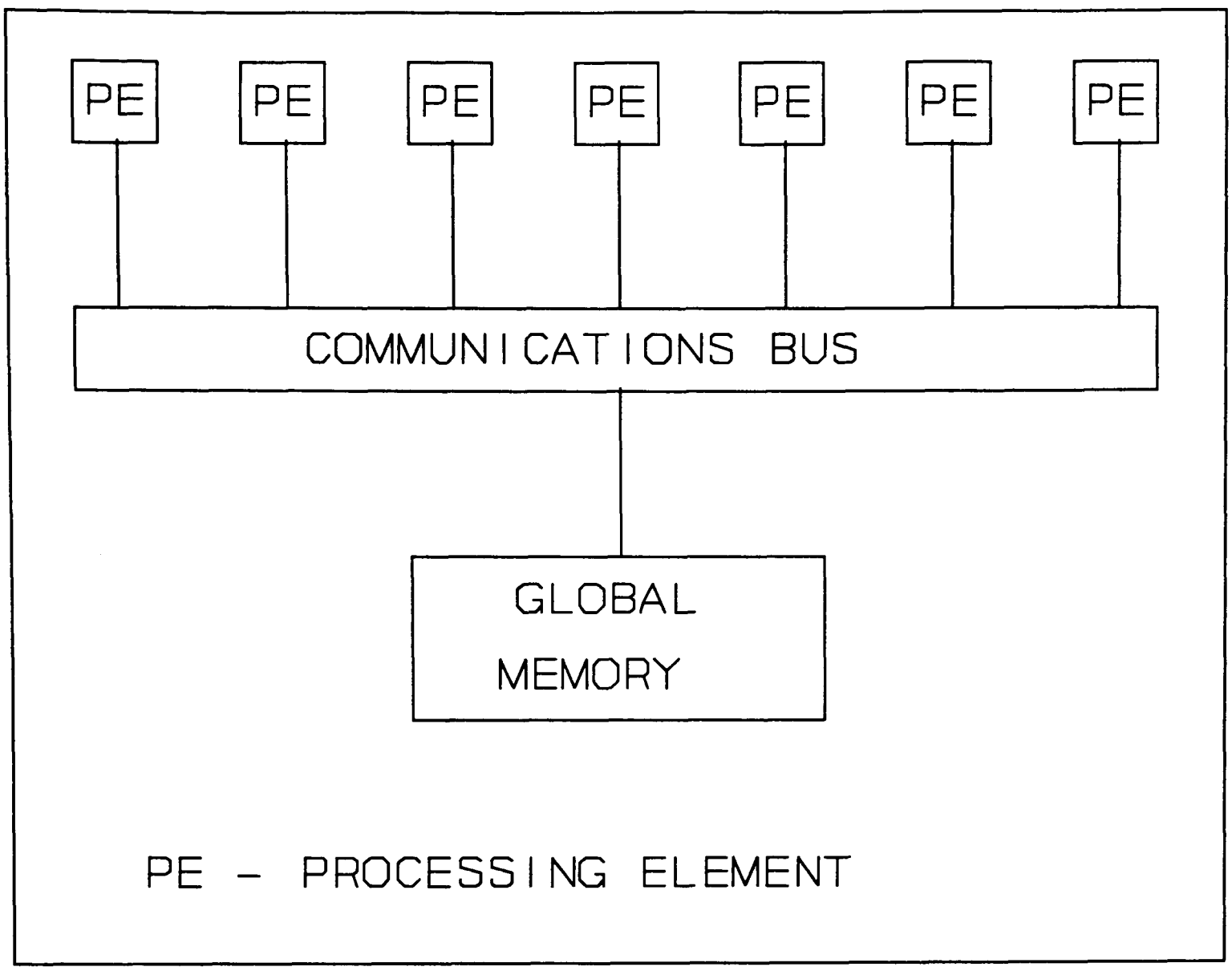


Figure 2-3 Shared Memory System.

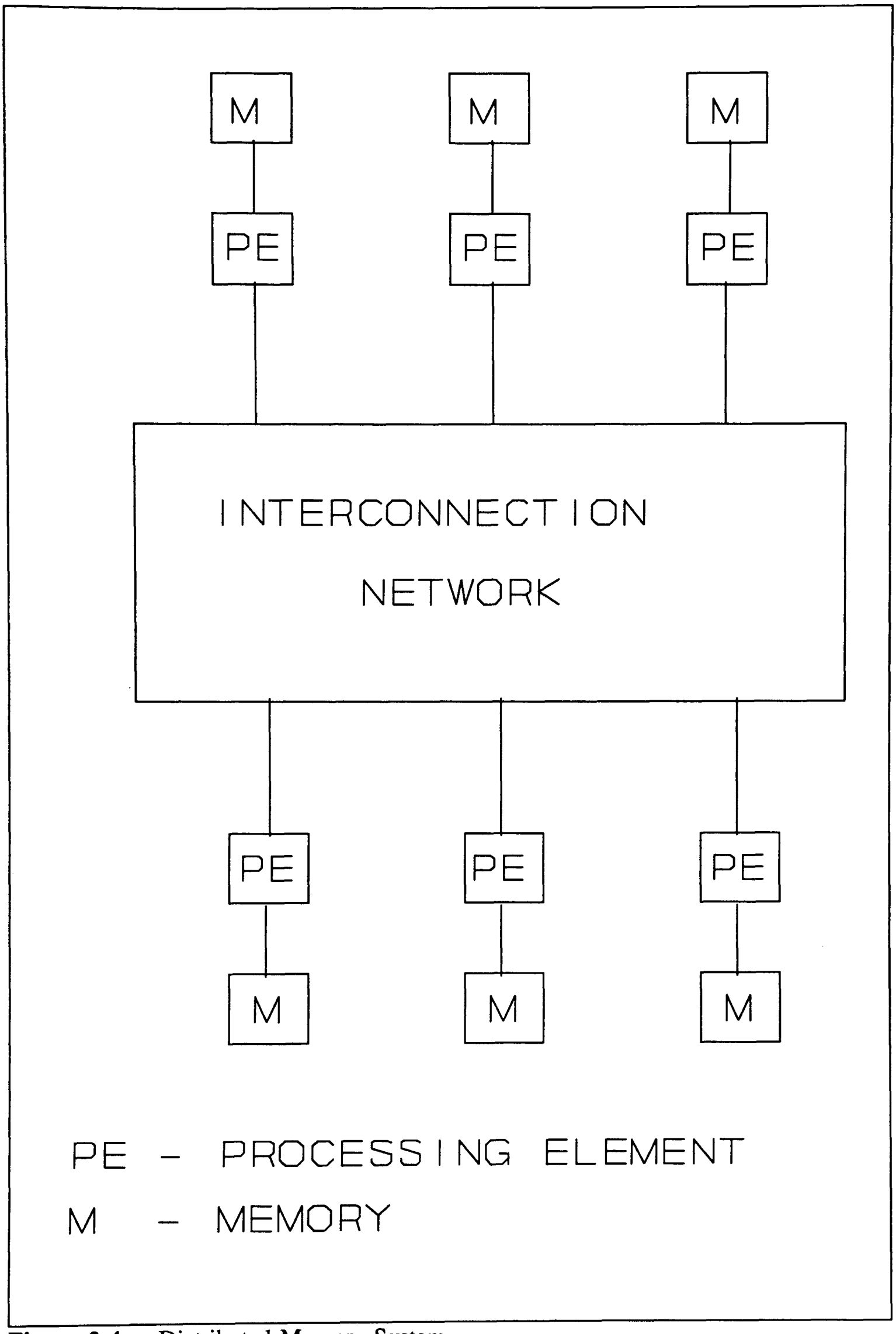


Figure 2-4 Distributed Memory System.

Other MIMD machines use a hybrid of the two previously described i.e. a distributed memory system where each processor consists of a small shared memory system or a shared memory system where each processing unit has a small amount of local memory.

2.4 The Transputer Architecture And Practical Use.

2.4.1 The Transputer Architecture.

The target machine in this work is a network of INMOS T800-20 Transputers. The T800-20 Transputer (see figure 2.5) consists of an integer processing element, a 32 bit floating point processing element, four communication links and four kilobytes of fast access RAM all on the same chip. The processing elements use reduced instruction set (RISC) technology to produce peak performances of 10 MIPS and 1.5 MFLOPS. In addition to the on chip RAM, external memory can be added, although accessing this RAM is slower than accessing the on chip RAM.

The communications links can be used to connect to other Transputers, to form networks of Transputers, or to input/output devices allowing access to file systems etc. The data transmission speed of the communication links is around 20 Mbits per second, with a small setup overhead in initialising a communication. Transmitting data between Transputers not directly connected requires a time proportional to the amount of data being sent and the number of individual Transputer to Transputer communications required.

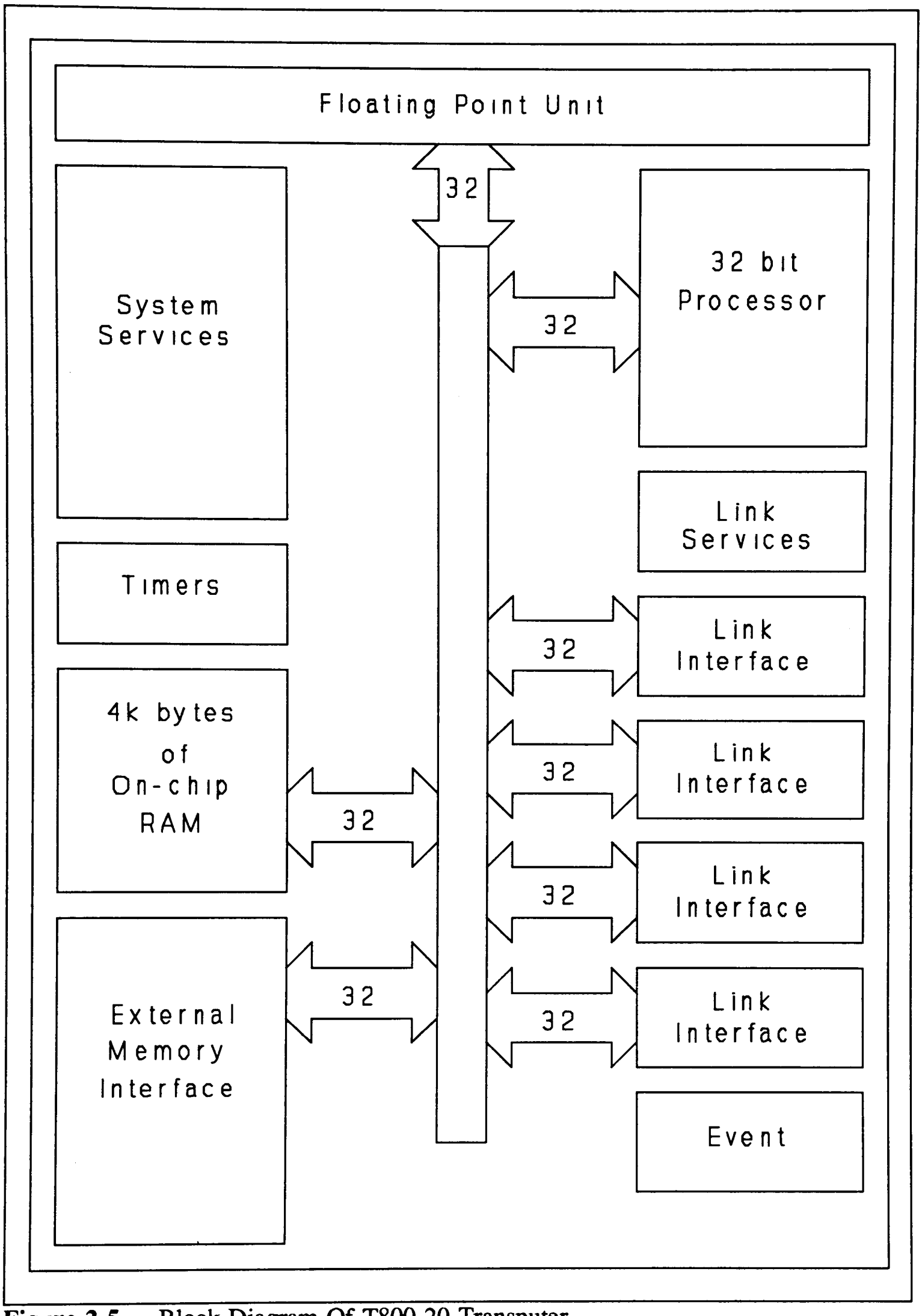


Figure 2-5 Block Diagram Of T800-20 Transputer.

2.4.2 Software Languages For Transputers.

A new language was developed specifically for use with Transputer networks with specialised features to allow simple exploitation of the Transputer's facilities. This language, OCCAM (INMOS 1988), although powerful, often proved a handicap to the Transputer's use, particularly for large numerical software which is traditionally written in FORTRAN. As a result, in recent years, parallel FORTRAN 77 versions with language extensions to support parallelism and other Transputer features have become available. A common way to incorporate these features into otherwise standard FORTRAN 77 is to provide a library of specialised routines. These routines can then be used to develop further useful utilities such as efficient routing facilities for communication data between processors not directly connected (i.e. a communications harness).

The software for such a Transputer network is a set of processes, each internally serial, that can be performed parallel to each other with the necessary interaction enabled by the use of communication routines. Such processes can be configured to execute on different Transputers to provide parallelism or on a single processor when the processes are separate for functional rather than parallelism reasons. The following two code structures, as used in 3L Parallel Fortran (3L 1988), can be used when appropriate. Other parallel Transputer Fortran dialects contain equivalent facilities, however, the 3L terminology is used throughout this work. These structures, Tasks and Threads, are discussed in the following sections :-

A **Task** is a process that has a completely separate memory area, thus data from other tasks must be physically transported into this task memory area. Tasks are used to provide parallelism by having their memory areas on separate processors, communication between

such tasks using the inter processor links. Several tasks can be placed in the same Transputer by allocating sections of that Transputer's memory uniquely to a task, communication between tasks on the same Transputer is made via memory locations not allocated to either task where these memory locations are treated as the Transputer links.

A **Thread** is a process that is spawned by a task and shares memory with the spawning task. Data can therefore be passed without physical movement of data although communications can still be made via internal channels (i.e. memory locations within the task data area which are treated as Transputer links). When threads are used, care must be taken when accessing the same data in two different threads, as with shared memory systems, locking semaphores are often required to ensure that only one thread accesses a memory location at a time.

Both tasks and threads allocated to a Transputer are executed in a time slice fashion where each process is allocated a section of the total execution time. A process involved in a communication where the other communicating process is not ready is de-scheduled (i.e made inactive using no execution time) until the other process is ready to communicate. Priorities can be given to different processes to force execution of some processes in preference to others. If one process has a higher priority than all others, it will execute until forced to de-schedule, waiting to complete a communication, all other processes will be idle when the high priority process can execute.

2.4.3 Strategies For Using Transputer Networks.

Three strategies are commonly used in the exploitation of Transputer networks. The choice

of strategy is dependent on the nature and implementation of the application being considered. The strategies are not exclusive thus a combination of them may well be the best solution for certain applications.

2.4.3.1 Algorithmic Parallelism.

Parallelism between different code sections is classified as algorithmic parallelism. In such a parallel code, different tasks may execute entirely different code sections, often, for example, exploiting parallelism between subroutines. The use of such a strategy often introduces large amounts of processor idle time since the execution time of each task will almost certainly not be the same. It is therefore rare for high efficiencies to be achieved with such a strategy.

2.4.3.2 Geometric Parallelism.

Geometric parallelism exploits parallelism between the same operations performed on different data areas. This generally occurs between iterations of loops, thus a task may consist of a number of iterations of a particular loop. Since the same code is executing on each processor, the idle time introduced is often small with all tasks completing at approximately the same time. The implementation of a code exploiting geometric parallelism can be achieved using a data partition, where data is divided into subsets and allocated to particular processors in the network, as appropriate.

2.4.3.3 Processor Farm.

A processor farm parallel code involves the splitting of the computational task into independent jobs. These jobs are then given to an arbitrary processor as and when that processor is ready to process it. Using this strategy, processors are kept busy as long as jobs remain, thus idle time is minimised. The use of a processor farm, however, incurs a potentially high communication cost since the allocation of a job to a processor requires all data for that job to be communicated to that processor from a controlling processor and the return of the results of that job to the controlling processor. The effectiveness of such a strategy is therefore dependent on the amount of data required by and returned from these jobs. In particular, the ratio of communication to computation must be heavily biased toward computation if efficient parallel code is to be produced. The operation of a processor farm is comparable to that of a shared memory system where the inter-processor communications perform a similar function to the communications bus in a shared memory machine.

2.4.4 Communications Harness.

A useful facility, available for Transputer networks, is a communications harness. The details of routing of messages between distant processors in a network of Transputers is handled by such a harness. To enable the transparent operation of such a communications harness, the use of an additional process on every Transputer is required. This allows through routing of data for communications between Transputers without altering the main computation process on intermediate Transputers. The computation process will be forced to idle until the communication in the other process is complete (unless the computational thread is allocated

execution time slices). The communication harness process is also required to handle communication calls from the computation process. Since physical data movement causes a time overhead, it is desirable to merely pass pointers to the data to the communications process, where that data is common to both processes. Therefore a communications harness performed as an execution thread is advantageous to the speed efficiency of the harness. However, a communications harness as a separate task is often used since one such task can perform communications for all other tasks on that processor.

The operation of a communications harness can use both synchronous and asynchronous communication. Synchronous communication requires the sending process of a communication to idle until the receiving process is ready. Asynchronous communication allows the sending process to continue by receiving the data into a data buffer held in the communications harness. This data is held until the request for it is received by the harness. Asynchronous communication has the advantage that it allows execution to continue on the sending process but the amount of buffering space required can be unpredictable and, when a buffer is full, synchronous communication must again be used.

The network of Transputers used can be in a number of different configurations. The configuration chosen should be influenced by the data access structure of the code concerned given the parallelisation strategy used. Use of a communication harness enables any configuration to perform an application, however the amount of communication time incurred can be minimised by a sensible choice of network configuration. Another requirement of the network is the ease of scalabillity, i.e. how extra Transputers can be added and used efficiently, consistent with the current configuration. Examples of possible configurations are

shown in figure 2.6.

2.5 Parallel Performance Measurement.

There are two main, often quoted, measures of performance of software on parallel systems. The first is speedup (Sp) which is defined as :-

$$Sp = \frac{\text{Time on a single processor}}{\text{Time on p processors}}$$

i.e the number of times faster the software executes on p processors as compared to the execution time on a single processor. There are, however, two possible single processor times that can be validly used, each conveying slightly different information about the software. If the single processor time is that of the best serial version, using optimal serial algorithms, the speedup indicates the benefit of using a parallel machine as opposed to a serial machine. The speedup figure produced may be reduced since the algorithms used for the parallel version may be different to those in serial, with serial performance sacrificed for the parallel nature of the new algorithm. The second single processor time that can be used is that of the parallel version run on a single processor. This speedup indicates the performance of a parallel machine as more processors are used and not performance over serial since any serial version should always use the best serial algorithms available.

The second measure of performance of software on MIMD machines is efficiency, a measure of how well an application uses the available computer power but again two versions can be

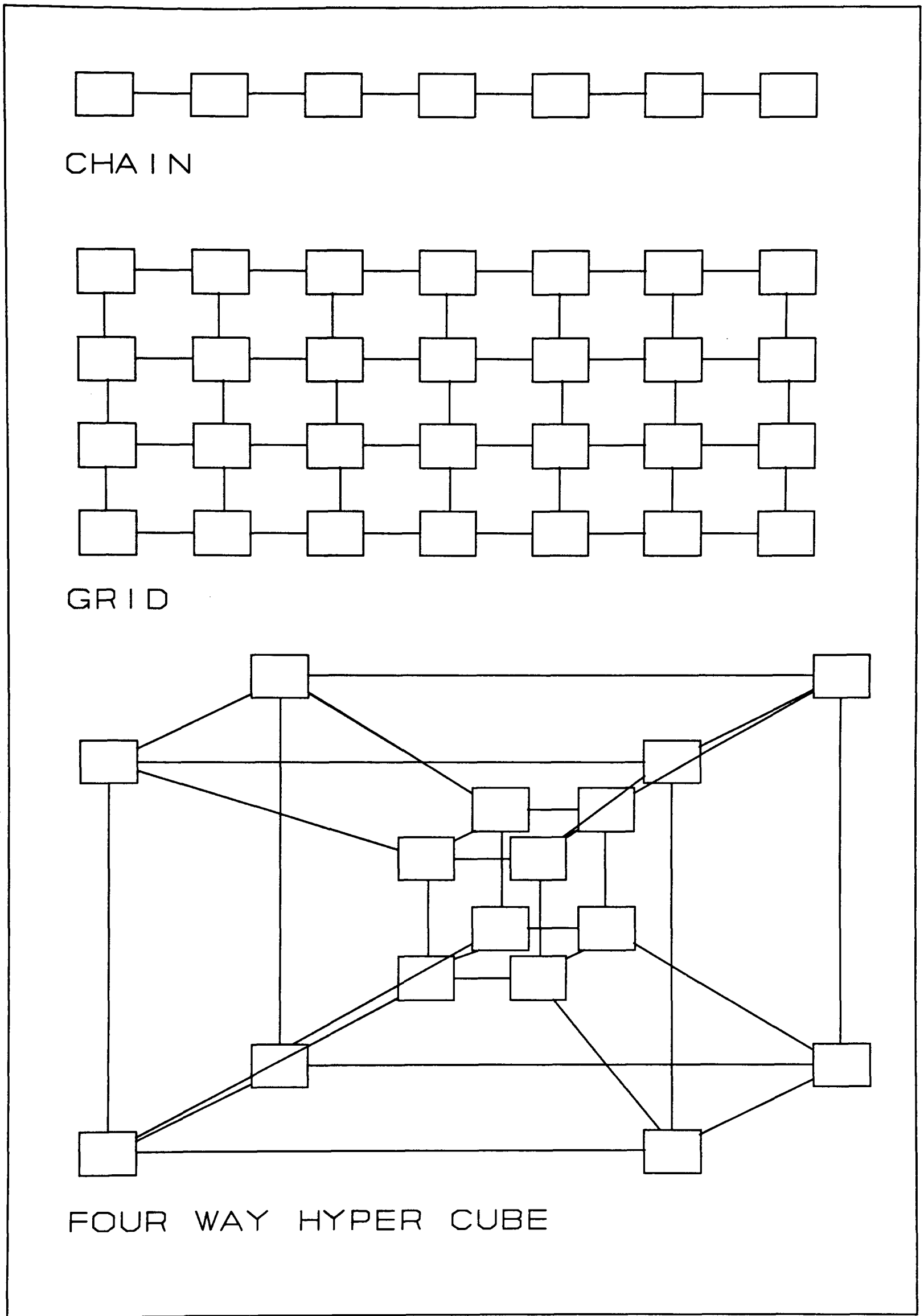


Figure 2-6 Example Transputer Network Configurations.

used. Firstly, an efficiency percentage (Ep) given by :-

$$Ep = \frac{Sp}{p} * 100 = \frac{\text{Speedup on p processors}}{p} * 100$$

where the speedup uses the best serial execution time, indicates the percentage of available processor time which has been beneficially used. The losses of efficiency here are mainly due to processor idle time, communication time and less efficient parallel algorithms.

The second efficiency is calculated using processor idle time i.e.

$$E'p = \left(1 - \frac{\text{Total Idle Time}}{\text{Total processor time}}\right) * 100$$

giving a measure of the percentage of time spent performing some form of operation.

In this work, speedup and efficiency values quoted are of the first type in each case.

2.6 Inherent Speedup Restrictions.

A simple analysis of limitations on parallel performance can be performed as follows. If f_s is the fraction of serial code in a given application then the parallel fraction is clearly given by $1 - f_s$. Assuming that the parallel fraction of the code can use all available processors and that no other overheads exist, the best possible parallel execution time (Tp^{min}) is

$$Tp^{min} = \left(f_s + \frac{(1-f_s)}{p} \right) * Ts$$

Where Ts is the serial execution time. This can then be used to give Amdahl's law (Amdahl 1967) for maximum possible speedup for an application i.e

$$Sp^{max} = \frac{Ts}{\left(f_s + \left(\frac{1-f_s}{p} \right) \right) * Ts} = \frac{1}{f_s + \left(\frac{1-f_s}{p} \right)}$$

Thus even if the number of processors tends to infinity, the maximum speedup is limited to $1/f_s$. As an argument against massively parallel machines, it has been claimed that since, in general, around ten percent of code is inherently serial, the maximum speedup is limited to around ten, thus large numbers of processors will not produce the hoped for speedups. In practice with MIMD machines, very large speedups have been achieved. This is because the serial fraction of many codes is insignificantly small since in most time consuming software, most of the execution time is spent inside loops with high iteration counts. Thus if those loops can be performed in parallel then almost the entire execution time will involve parallel computations.

It is clear that the serial fraction used in Amdahl's law is the fraction in terms of serial execution time and is therefore dependent on a particular instance of a program execution and not determinable at compile time, thus the law has only limited use as a characteristic of a given code and does not fatally prohibit speedup on MIMD machines. Amdahls law does, however, indicate that any code sections that cannot be parallelised (or where not all processors can be used) can have a significant limiting effect on speedup even when the

execution time of the serial sections is only a small percentage of the overall serial execution time. If it is to be worthwhile to use massively parallel machines for an application, the amount of serial code must be a minimum where any serial algorithms used must be replaced by parallel algorithms even if the parallel algorithms are very inefficient.

2.7 Maximum Speedup.

Consider a parallel code being performed on N Transputers. As mentioned in section 2.4, the parallel code will consist of a set of tasks possibly running on separate Transputers. Since the Transputer can perform many tasks on a single processor, it is possible to reconfigure the code to place all tasks on a single Transputer, dividing the Transputer memory as required, allocating a section to each task. This new code will perform the identical operations to the parallel code using time slices of the Transputers execution time. As a result the overall execution time of this serial code will be the sum of the execution times of all Transputers in the parallel version, i.e. at worst N times the parallel execution time. Although a small overhead in scheduling the tasks on the single Transputer will be incurred, this is offset by the inevitable processor idle time that occurs in parallel execution. In addition, communications in the serial version will involve internal data transfer from one memory location to another whilst the parallel version often required physical data transmission through the inter-Transputer communication links. The overhead time required in the serial version will therefore be significantly less than the communication time overhead in the parallel code.

Any parallel Transputer code can be configured as a serial code as described above.

Therefore, a serial execution time of at worst N times the parallel execution time can be achieved. This provides a maximum possible speedup of N for any code. Any speedup higher than the number of processors used is achieved either by using a non-optimal serial code for the comparison, producing an invalid speedup measure, or by, for example, the use of the distributed processor cache memories (i.e. keeping often used data items in the fast on chip memory for as long as possible) which, particularly with the small transputer cache and large data problems considered in this work, should not usually be significant.

2.8 Closure.

This chapter has broadly described the parallel computer architectures currently available. The use of such machines has also been discussed. Attention was focused on the Transputer as the major component in a parallel system and the effective exploitation of such a system. The success of use of Transputer networks will now be examined in the following chapters when the parallel solution to numerical computation problems is attempted on Transputer networks.

CHAPTER THREE

3. Mapping An Enthalpy Based Solidification Algorithm Onto A Transputer Network.

3.1 Introduction.

As a first example of using a Transputer network to perform a numerical task, an enthalpy based solidification algorithm due to Voller and Cross (Voller and Cross 1985, Voller et al 1987), using the control volume technique in three dimensions, was considered.

The algorithm was first coded and tested in serial to run on a single Transputer. The methods employed in the serial code were chosen and optimised for serial speed as well as low memory usage.

The parallel version was then created by converting the serial code with as few major alterations as possible. The success and simplicity of the parallel conversion could then be used to determine whether parallelism could feasibly be extracted from optimised serial code or if parallelism had to be considered as an early stage in the software design process.

3.2 The Enthalpy Algorithm For The Solidification Process.

3.2.1 Overview Of The Solidification Process.

The algorithm models the solidification of materials in solid, liquid or transitional states. For simplicity, the model ignores convection, being driven by conduction and the release of latent heat during solidifying only. As in most practical materials, the phase change from liquid to

solid occurs gradually over a temperature range, a mushy region thus exists which is neither solid nor liquid. In this mushy region, the latent heat held in the liquid is gradually released.

3.2.2 Governing Equations.

The governing equation of the three dimensional, transient, conduction in terms of enthalpy is :-

$$\frac{\partial}{\partial t}(\rho h) = \frac{\partial}{\partial x}\left(k \frac{\partial}{\partial x}\left(\frac{h}{c}\right)\right) + \frac{\partial}{\partial y}\left(k \frac{\partial}{\partial y}\left(\frac{h}{c}\right)\right) + \frac{\partial}{\partial z}\left(k \frac{\partial}{\partial z}\left(\frac{h}{c}\right)\right) - L\frac{\partial}{\partial t}(\rho f_l) \quad (1)$$

Where k is thermal conductivity

c is specific heat capacity

ρ is density

L is latent heat

h is sensible enthalpy

f_l is liquid fraction

Total enthalpy is the sum of sensible enthalpy and latent heat held in the material i.e.

$$H = h + \Delta H \quad (2)$$

Sensible enthalpy is given by

$$h = \int_{Tref}^{T} c \, dT \quad (3)$$

where T is current temperature and Tref is a reference temperature.

The enthalpy held in the material is the product of liquid fraction and latent heat :-

$$\Delta H = f_l L \qquad (4)$$

If we assume that the phase change occurs linearly within the mushy region temperature range then the liquid fraction is given by :-

$$f_l = \begin{cases} 1 & T >= T_L \\ \dfrac{T-T_S}{T_L-T_S} & T_S < T < T_L \\ 0 & T <= T_S \end{cases} \qquad (5)$$

Where T_L is the liquidus temperature for the given material (i.e. the material is entirely liquid if the temperature is at or above the liquidus temperature) and T_S is the solidus temperature.

3.2.3 Control Volume Discretization Of Governing Equations.

The numerical solution of the physical equations is achieved using the control volume technique as described by Patankar (Patankar 1980). The governing equations must be satisfied throughout the problem domain, thus if the domain was discretized into many finite sized subdomains, each subdomain must also satisfy the governing equations. Consider a uniform discretization creating many rectangular box cells, if we assume that all the physical variables are constant throughout each individual subdomain then it is possible to form an approximate form of the governing equation in the following way :-

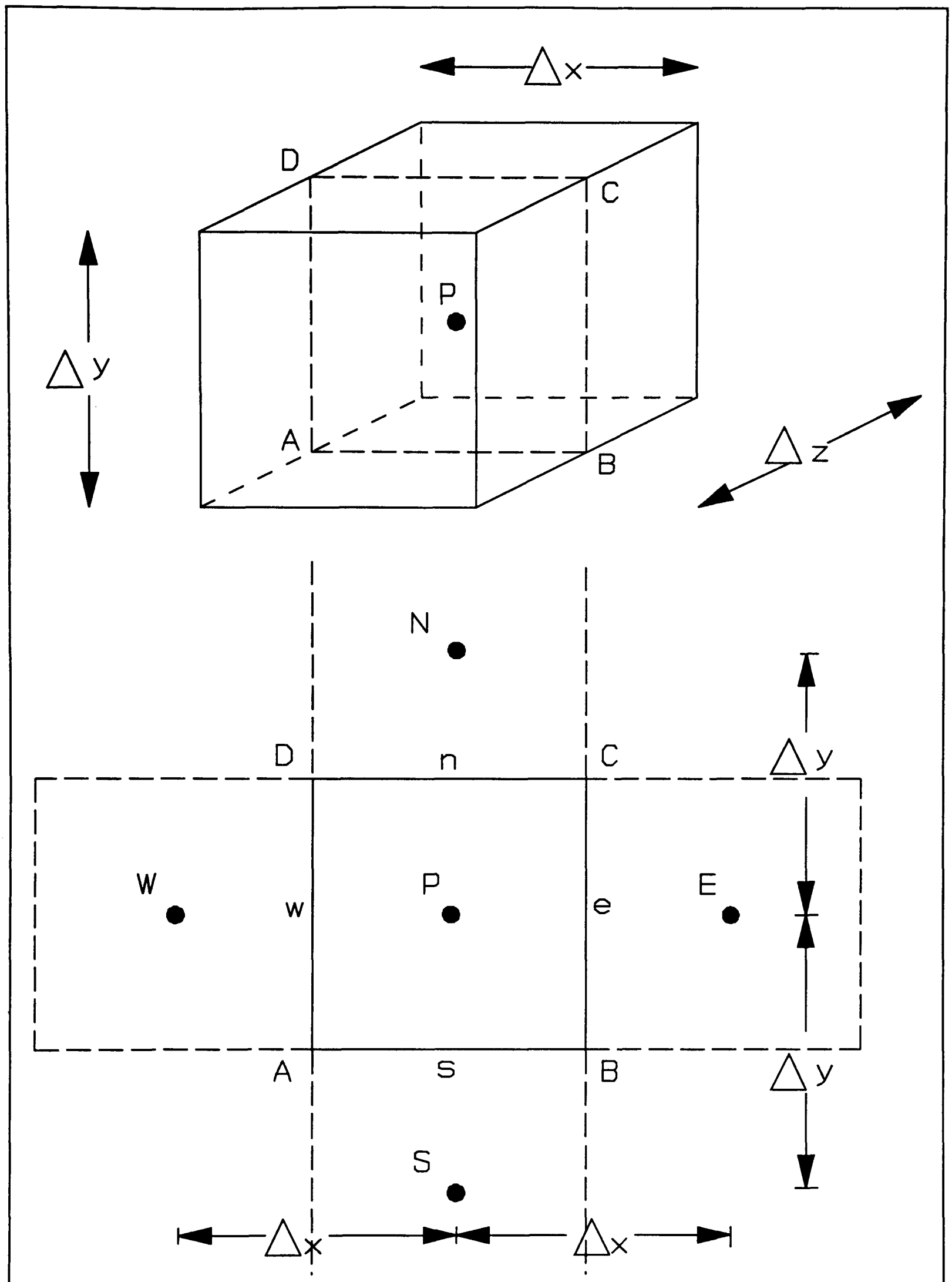


Figure 3-1 Control Cell And Cross-Section.

The diagram (figure 3.1) shows a two dimensional cross-section through a subdomain (control cell). The cell has dimensions Δx by Δy by Δz as does each neighbour cell therefore the distance between the centre of this cell and the centre of a neighbouring cell is Δx, Δy or Δz in the x,y and z directions respectively (since we have a uniform discretization).

Now an approximate form of the governing equation for the control cell P can be made by integrating the governing equation over cell P using approximations to derivative terms from gradients of variables from this cell centre to neighbouring cell centres.

The transient term is constant throughout the control volume thus :-

$$\int_{Vol} \frac{\partial}{\partial t}(\rho h)\, dV \approx \frac{\rho_P h_P - \rho_P^{(old)} h_P^{(old)}}{\Delta t} * \Delta x \Delta y \Delta z \qquad (6)$$

Where Δt is the time step and the 'old' variables are the values of those variables at the previous time step. The latent heat term of equation (1) is integrated in the same manner. Treating the derivative terms as mentioned above, the x-direction diffusion term gives :-

$$\int_{Vol} \frac{\partial}{\partial x}\left(k \frac{\partial}{\partial x}\left(\frac{h}{c}\right)\right) dV \approx \left\{ k_e \frac{\partial}{\partial x}\left(\frac{h}{c}\right)_e - k_w \frac{\partial}{\partial x}\left(\frac{h}{c}\right)_w \right\} * \Delta y \Delta z$$

$$\approx \left\{ k_e \left[\left(\frac{h}{c}\right)_E - \left(\frac{h}{c}\right)_P\right] - k_w \left[\left(\frac{h}{c}\right)_P - \left(\frac{h}{c}\right)_W\right] \right\} * \frac{\Delta y \Delta z}{\Delta x} \qquad (7)$$

and the y and z direction diffusion terms produce similar results.

This leads to a discretized form of the governing equation at control cell P in terms of

neighbouring cells :-

$$a_P h_P = \sum a_{nb} h_{nb} - b \qquad (8)$$

Where nb = E,W,N,S,T,B i.e. all neighbouring nodes.

The coefficients in equation (8) are obtained from equations (6) and (7) after dividing the equation through by cell volume.

$$a_E = \frac{k_e}{c_E \Delta x^2} \qquad \text{With } a_N \text{ etc. of similar form}$$

$$a_P = \Sigma a_{nb} + \frac{\rho_P}{\Delta t}$$

$$b = \frac{-\rho_P^{(old)} h_P^{(old)}}{\Delta t} + \frac{L}{\Delta t}(\rho_P f_l - \rho_P^{(old)} f_l^{(old)})$$

Control cells on the boundary of the domain produce similar coefficients but use half cell dimensions in the appropriate directions. Dirichlet, Neumann and mixed type boundary conditions can be used, each require a slightly different formulation of coefficients.

The value of all variables is required at the cell centre only, which is where they are stored, except thermal conductivity values which are required on cell faces, so these values must be evaluated by averaging the appropriate cell centre thermal conductivity values. The material properties k,c and ρ are all functions of temperature and material type allowing a piecewise set of functions of temperature for each material used in a simulation.

3.2.4 Solution Algorithm.

The above formulation of a discretized equation is used for every control volume in the discretized domain to produce a system of equations. These equations are non-linear since the latent heat term and the material properties are all functions of temperature which in turn is related to the solution variable, sensible enthalpy. The solution algorithm must therefore be of an iterative nature to use estimates of sensible enthalpy to produce new estimates of temperature and thus material properties to be used in an update of sensible enthalpy.

The solution algorithm runs as follows :-

1. Set up old sensible enthalpy, liquid fraction and density values using the provided initial conditions.

2. Set up coefficients of the discretized conduction equations and solve for sensible enthalpy from equation 8 using an appropriate method.

3. Evaluate temperature from sensible enthalpy (equation (3)).

4. Evaluate liquid fraction from temperature using equation (5).

5. Test for convergence of sensible enthalpy, if convergence not achieved return to step 2.

6. If another time step is required update old time step values using the newly calculated

values and return to step 2.

To ensure a solution will eventually be reached, it is necessary to under-relax the enthalpy and liquid fraction updates to damp out oscillations in their values that occur from iteration to iteration.

The methods used to solve for sensible enthalpy in step 2 can be such techniques as Jacobi updates, Gauss Seidel updates, Line based Gauss Seidel updates etc.

3.3 Conversion To Parallel Code.

Once the algorithm was written, tested and optimised in serial using a Gauss Seidel update method in step 2 of the solution algorithm, the conversion process could begin.

3.3.1 Parallel Code Requirements.

The criteria for a successful parallel version of the solidification code are that both the processing power and total distributed memory should be used with maximum efficiency. To achieve this, three constraints must be addressed :-

1. Ensure an equal computational load is given to every processor (i.e. balanced computations) so that no processor(s) are idle whilst awaiting communications from a processor still performing computations.

2. Ensure the program data is evenly distributed over every processor's local memory.

3. Minimise the time spent performing inter processor communications since this is an overhead not encountered in the serial code.

3.3.2 Data Partition Strategy

The computation load in the algorithm is approximately proportional to the number of control volumes being evaluated (extra computations involved in the mushy region are proportionally small). This indicates that an even distribution of control volumes on each processor would satisfy the balanced computations constraint at the same time as satisfying the memory usage constraint. Thus all that is now required is to partition the data and to place these partitions over the Transputer network in such a way as to minimise required inter-processor communications.

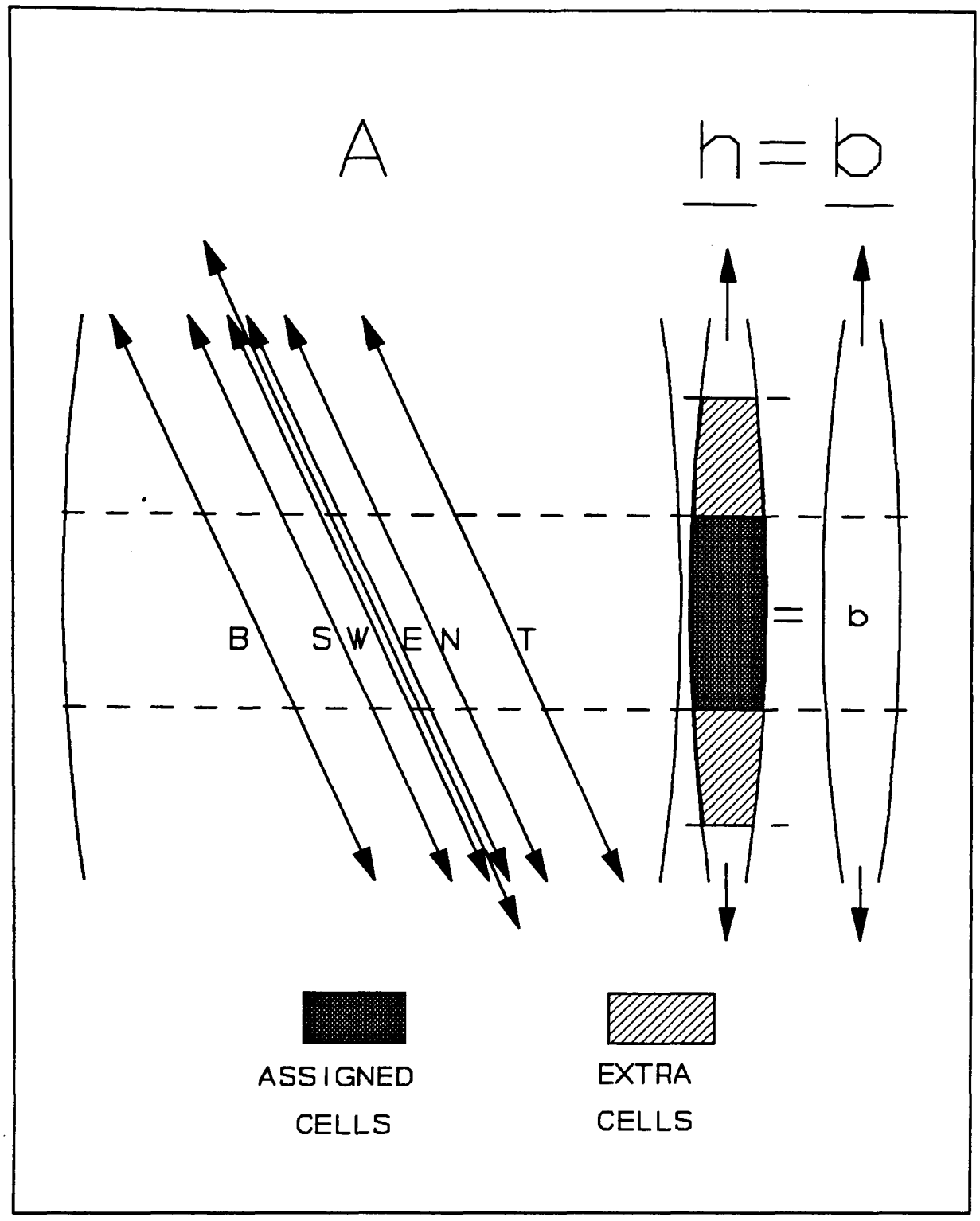


Figure 3-2 Section Of Matrix Equation.

To decide how to partition the data, consider the system of equations in step 2 of the solution procedure as a matrix system $A\underline{h} = \underline{b}$ (figure 3.2). The coefficient matrix A will be of diagonal form as shown if the order of control cells used is of a sensible form, eg. order of increasing x direction first, then y and finally z. If the consecutive sets of control cells indicated (i.e. rows of the matrix and vectors) are allocated to a processor then the enthalpy values required in the computation are those allocated to this processor and also those indicated in figure 3.2 in consecutive groups of cells adjacent to those allocated here. The extra required values are those of the nx * ny cells (nx being total number of cells in the x direction etc.) immediately before and immediately after the allocated set (i.e from the top cell of the first cell allocated to this processor to the bottom cell of the last cell allocated). In addition to extra enthalpy values being required, extra values for the material properties for the same sets of cells will be required in the coefficient calculation. Since all these material properties are functions of temperature, it is the temperature values that are actually required.

The extra cells will be allocated and assigned values on other processors, thus inter processor communication of the required values will have to take place. The values used in the calculations must be the most up to date values available from the assigning processor to provide a good update of the enthalpy values on this processor, therefore the communication must be performed within each iteration of the solution algorithm to send the values calculated in the previous iteration. Communicating these potentially large blocks of data represents a major, but essential, overhead in each iteration, thus fast execution of these communications is essential to the efficient operation of the parallel code.

The communication time is approximately proportional to the number of processors involved

in transferring the data from source processor to sink processor, so minimising this communications distance with nearest neighbour communication only is a desirable feature. To achieve this, the minimum number of cells that can be allocated to any processor is the number of cells involved in these communications (i.e. nx*ny cells). All values required in a block will then reside on a single processor and a simple chain of Transputers is sufficient to ensure these values will always be on a directly connected neighbour Transputer with the control volumes allocated along the chain of Transputers in order (see figure 3.3).

Each connected pair of Transputers in the chain require blocks of data from each other, thus the communication can be achieved by a data exchanging process. This process can be performed efficiently by allowing each pair of processors to exchange data in parallel, then letting the alternative pairs of processors exchange their data in parallel, thus completing the exchange process in the time of four individual block communications regardless of the number of processors involved (see figure 3.4).

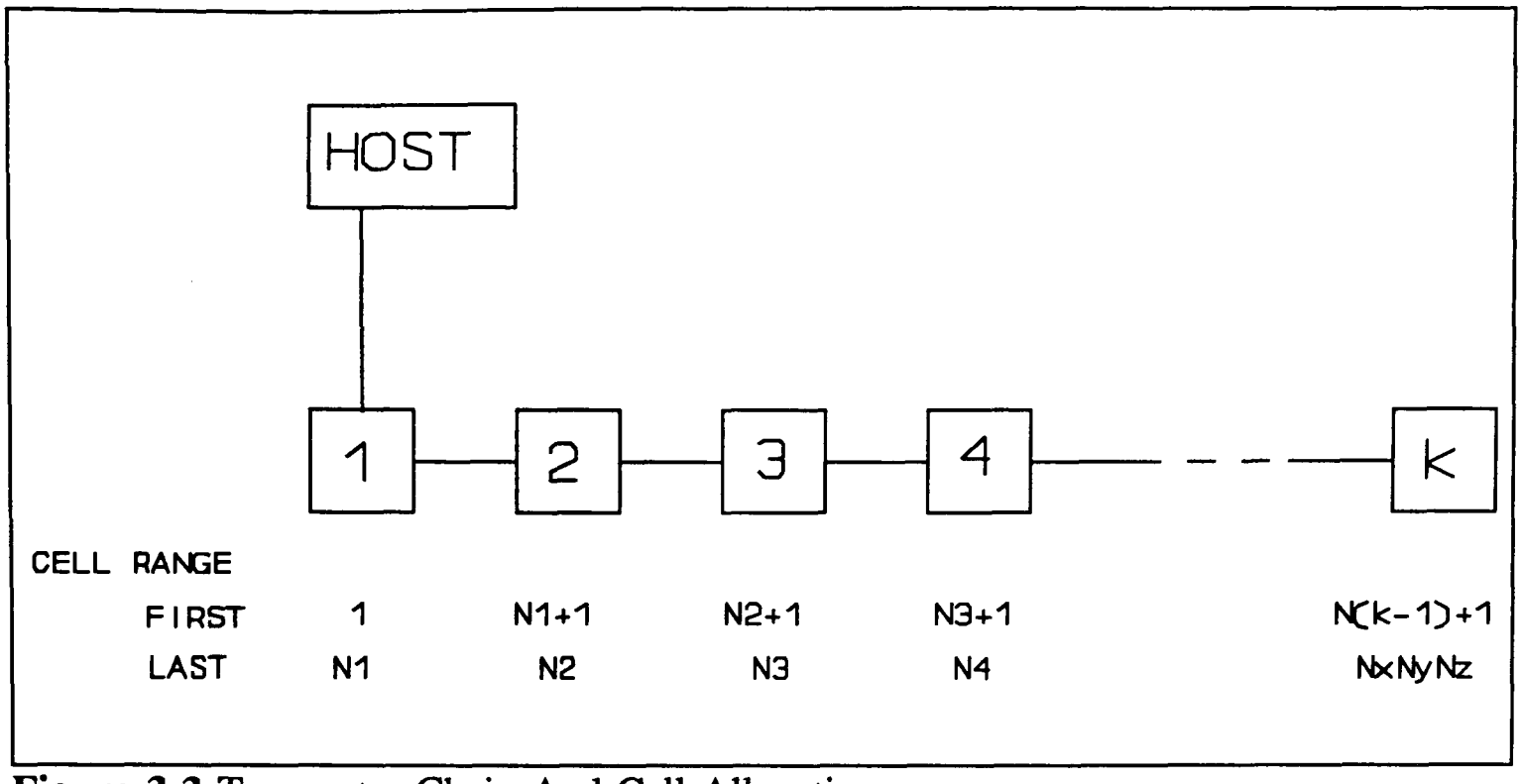


Figure 3-3 Transputer Chain And Cell Allocation.

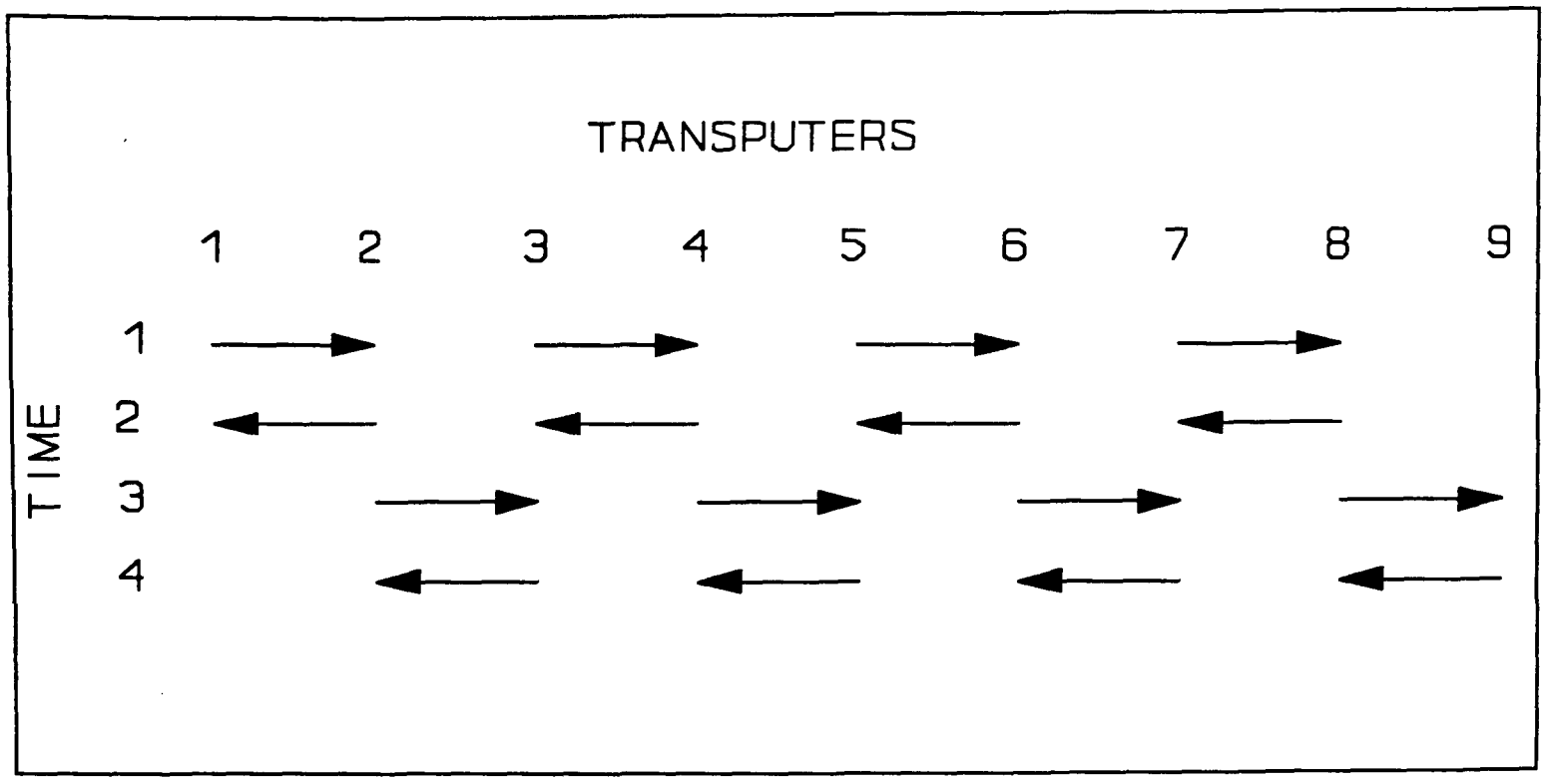


Figure 3-4 Exchange Procedure.

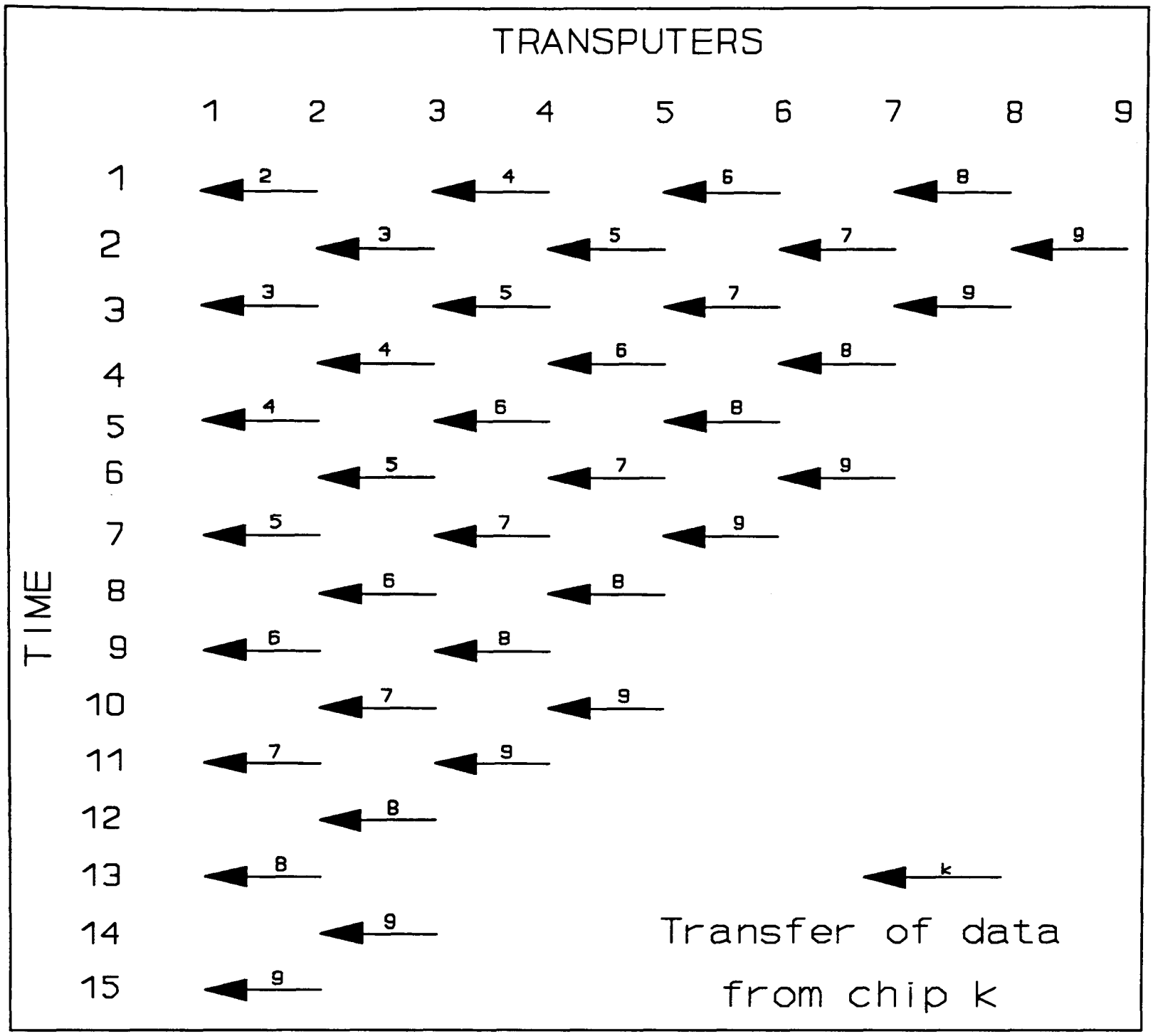


Figure 3-5 Data Transfer To Master.

Another communication required in the iterative process of the solution procedure is the transfer of residual values from each processor and the subsequent result of the global convergence test. Both these involve only a single word of data and thus require a proportionately insignificant amount of time, so again the chain configuration of Transputers will suffice.

The only other communication that can occur in the solution procedure is that of the converged fields of variables which may be sent to the host at the end of user selected time steps. Although this does not occur as frequently as the previously mentioned communications, it could potentially become a major overhead due to the large amounts of data involved. To allow fast completion of this data transfer, a pairwise parallel process was again used as shown in figure 3.5. To perform this process, every processor requires buffer space to store fields of converged solutions from processors further down the chain on route to the host. The even numbered processors require buffer space to store one field of converged solutions since they send data thus freeing the buffer before receiving more data, whilst the odd numbered processors require two such buffers since they receive before they can send on the data from the other buffer. Since we wish to maximise memory use for problem data and not for temporary buffer space, the buffers used are the arrays used to store the fields of variables old enthalpy and old liquid fraction (as used in transient terms). When a converged solution is produced at the end of a time step, the values in these arrays are no longer needed, thus they can then be used as buffer space before being updated ready for use in the next time step.

The major bottleneck in this process is the sending of data to the host and the host writing

this data to disk, thus although a more complex Transputer configuration may improve data movement in the Transputer network, it will suffer the same bottleneck and have little impact on overall speed (i.e. more processor idle time would result).

3.3.3 Overview Of Parallel Version.

Two separate programs constitute the parallel code. The first Transputer in the chain, called the master Transputer, is the only one connected to the host and therefore must perform all input output operations with the host system such as reading the input data and outputting the converged solutions to disk. The operation of the master Transputer in the algorithm is therefore different to all others and runs as follows :-

1. Read problem definition from host machine.

2. Evaluate the number of cells to allocate to each processor and determine which cells each processor shall be allocated.

3. Send allocation information and required problem definition information to processor number two.

4. Set up initial values for cells allocated to the master Transputer.

5. Use the data exchange procedure to exchange sensible enthalpy and temperature blocks with Transputer two.

6. Perform steps 2-4 of the solution algorithm for the control cells allocated to the master Transputer.

7. Receive a residual value from Transputer two.

8. Evaluate global residual from residual of cells on master processor and the received residual and perform convergence test.

9. Send convergence test result to Transputer two.

10. If not converged return to step 5.

11. If required, write the converged solution on this Transputer to disk, then receive converged solutions from Transputer two writing them to disk.

12. If another time step required, update 'old' variables and return to step 5.

The other Transputers all perform the same code (the slight differences in even and odd numbered Transputers operation being accounted for in software), known as the slave code. The operation of these Transputers runs as follows :-

1. Receive allocation information and problem definition from previous Transputer in chain.

2. Send this information on to the next Transputer in the chain.

3. Set up initial values for cells allocated to this Transputer.

4. Use the data exchange procedure to exchange sensible enthalpy and temperature blocks with neighbouring Transputers.

5. Perform steps 2-4 of the solution algorithm for the control cells allocated to this Transputer.

6. Receive residual value from next Transputer in chain.

7. Combine residual calculated on this Transputer with received residual.

8. Send combined residual to previous Transputer in chain.

9. Receive convergence test result from previous Transputer in chain and pass this flag on to the next Transputer in the chain.

10. If not converged, return to step 4.

11. If required, send converged solutions toward the master Transputer using the parallel procedure in figure 3.5.

12. If another time step required, update 'old' variables and return to step 4.

3.3.4 Parallel Code Development.

The original serial code was used as the start point for both master and slave programs. Communications code and other small code sections required were added and the input/output code was removed from the slave program.

The method used in step 2 of the solution algorithm, as mentioned earlier, is a Gauss Seidel style update using the most recent values of all variables including those evaluated earlier in the same iteration. As a result performing the iterations in parallel with identical computations to the serial code is not possible as many updates will require values not yet available. The parallel version therefore sacrifices the use of latest values for certain computations to gain parallel performance. Every Transputer simultaneously updates the enthalpy values allocated to it, with the set of extra values required from the block of cells immediately before the set of cells allocated to this processor having values from the previous iteration. Thus the first nx*ny cells allocated to a Transputer do not use latest values for the top neighbour cell with the consequence that the convergence rate of the solution algorithm is slowed, requiring a larger number of iterations to achieve the same accuracy as the serial version.

An invaluable technique in creating a successful parallel version was comparison with values from the correct serial code. Since the serial method used Gauss Seidel style updates and the parallel used only Local Gauss Seidel updates, the values in the serial and parallel code were not identical. To allow exact comparisons in the development process, the Gauss Seidel style updates were replaced by explicit Jacobi style updates in both versions of the code. Errors in the parallel version were then detected and corrected and when fully tested, the Local Gauss

Seidel updates were incorporated for the improved convergence performance they provide.

3.4 Test Case - Solidification Of AISI 1086 Steel.

3.4.1 Problem Specification.

A 40 cm cube of AISI 1086 Steel is initially at a uniform temperature of 1632 C. The solidus and liquidus temperatures of such steel are 1508 C and 1602 C respectively with the latent heat of solidification of 66 kcal/kg. The problem is driven by mixed type boundary conditions on all boundaries with a heat transfer coefficient of 4.7766E-3 kcal/m^2sec C and an ambient temperature of 25 C. The material properties are all linear functions of temperature defined as follows :-

Thermal Conductivity

$$k(T) = 1.203E\text{-}2 - 6.9647E\text{-}6 * T$$

Specific Heat Capacity

$$c(T) = 0.105 + 1.08667E\text{-}4 * T$$

Density

$$\rho(T) = 7853.08 - 0.3229 * T$$

The problem is run for a "physical" time of 320 minutes with a sixty second time step, by which time the entire steel cube has solidified. The tolerance for a converged time step

solution is 10^{-4} where the residual used is relative error using the infinity norm.

3.4.2 Timings Of Solution.

The timings presented here are from the original versions of the code using a point based update for enthalpy.

Nine Transputers, each with a single megabyte of external RAM were used configured into a simple chain as described earlier. Memory restrictions meant that the serial version was limited to a maximum problem size of around 35000 control volumes, whilst the nine Transputer version could solve for up to around 240000 control volumes indicating that the parallel version was fairly successful in terms of the memory usage requirement.
Figure 3.6 shows timings for a range of problem sizes with figure 3.7 showing the associated speedups for the problem size range that could be run using the serial code.

The speedup graph shows speedups less than the maximum of nine. There are several factors that caused this loss of speedup :-

1. The difference in convergence rate between the serial Gauss Seidel updates and the parallel local Gauss Seidel updates. This caused as small percentage increase in the number of iterations required to achieve the required accuracy in the parallel code, however, this increase had only a relatively small impact on the speedup produced.

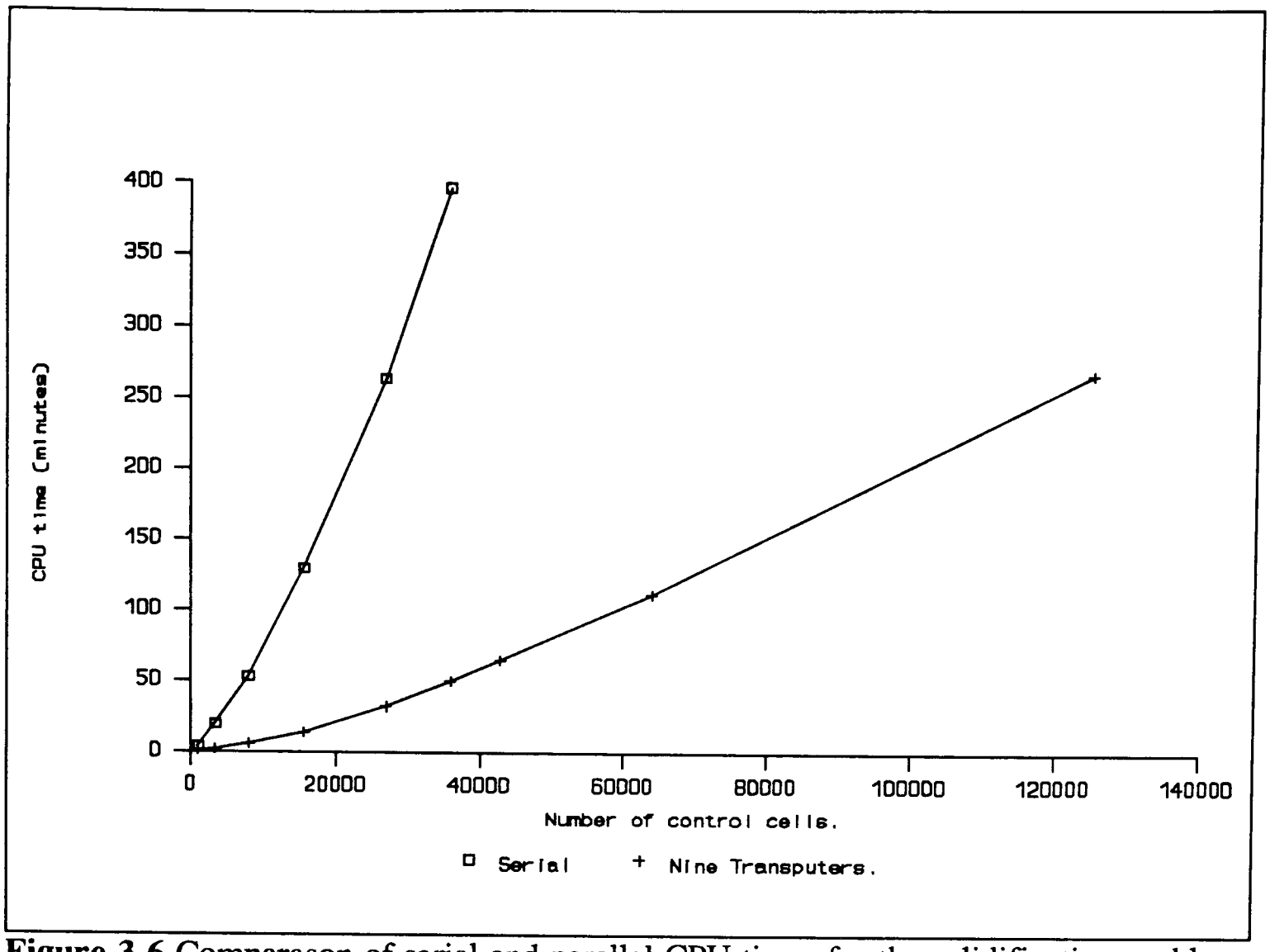


Figure 3-6 Comparason of serial and parallel CPU times for the solidification problem.

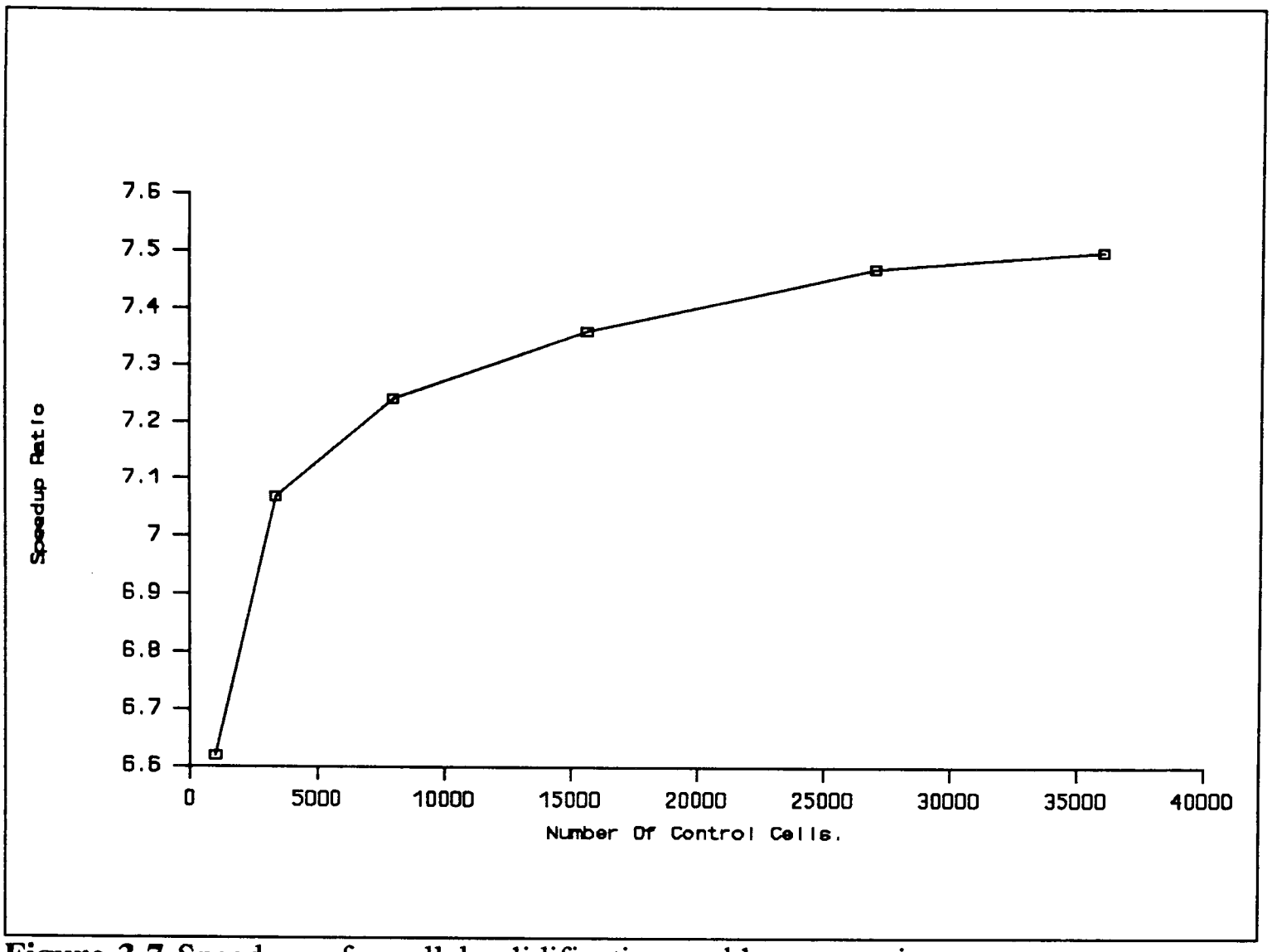


Figure 3-7 Speed up of parallel solidification problem on a nine transputer system.

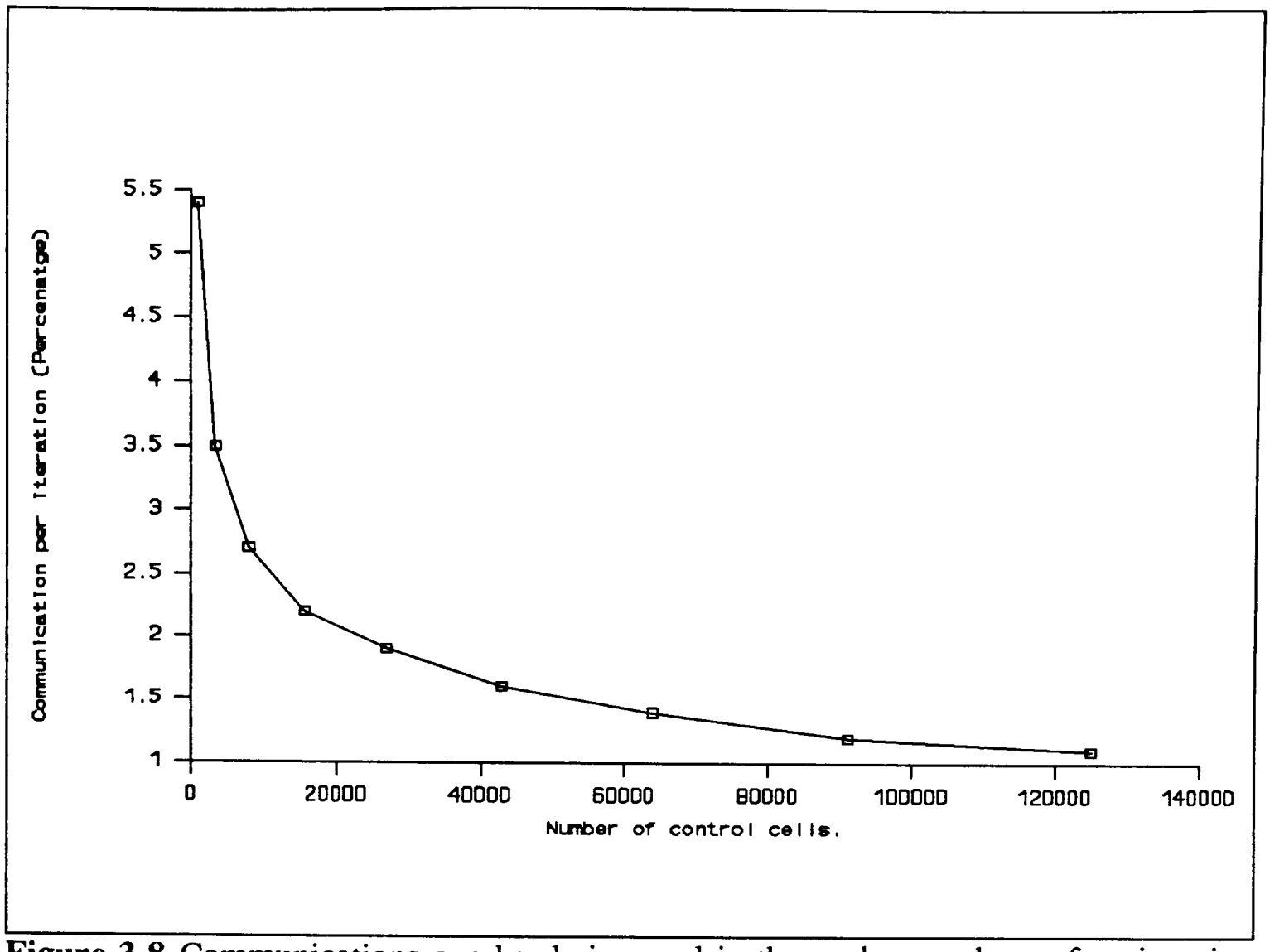


Figure 3-8 Communications overheads incurred in the exchange phase of an iteration of the solution procedure for the solidification problem

2. The inter iteration communication time could potentially represent a major overhead. Figure 3.8 shows the percentage of each iteration spent in the exchange process for a range of problem sizes. Clearly, as the grid is refined, this overhead becomes percentagly small and is thus not a major factor in the degradation of speedup. The exchange percentage graph does, however, give an indication as to why speedup improves as the grid is refined.

3. The time taken to transfer the converged solution to disk proves a significant overhead. This time, however, is largely taken in transferring the data to the host and writing that data to disk on the host. The excessive time taken to perform these operations are not inherent to the use of a Transputer network, new hardware with fast disk accessing capabilities using direct memory accessing have recently become available which will dramatically cut the time required. Since it is the parallel performance of the code that is being monitored and not the performance of external hardware devices, future results do not include disk access time.

The same test problem was then run on up to 18 Transputers with the disk access time not included. The timings produced for a range of mesh densities are shown in figure 3.9. The speedups produced with these parallel timings all relate to efficiencies aproaching 100%, clearly showing the success of the parallel implementation.

Further results, displaying similar speedup characteristics, for more complex problems can be found in other publications (Battle et al 1990).

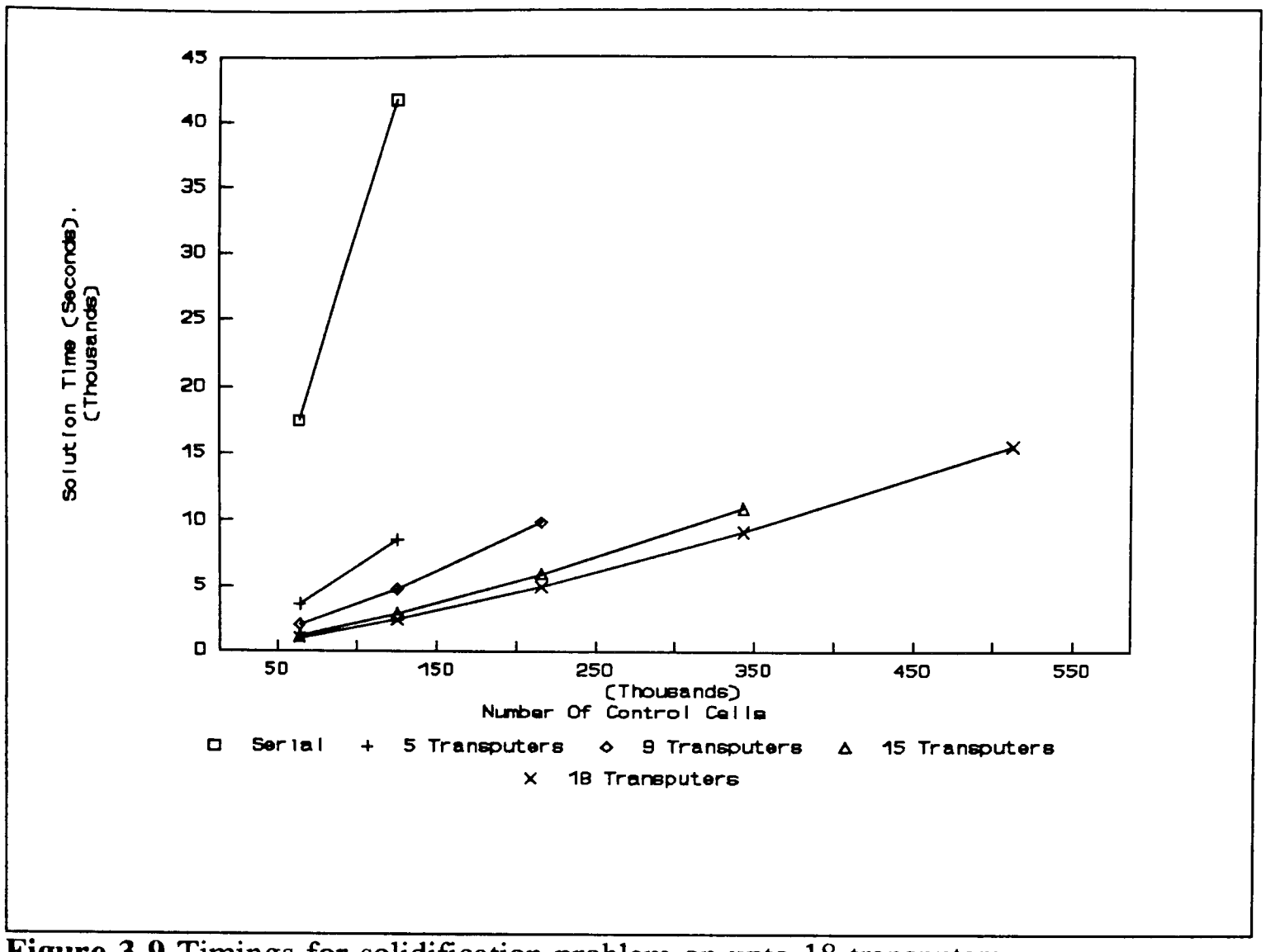


Figure 3-9 Timings for solidification problem on upto 18 transputers.

3.5 Closure.

The exploitation of geometric parallelism through a data partition has been demonstrated as a highly effective way of exploiting the benefits of a Transputer based parallel system. The nature of the solidification problem implemented only involved a subset of the full computational fluid dynamics features. The following chapter examines the use of a similar technique in a much broader application, where the serial code is a commercial product.

CHAPTER FOUR

4. Mapping Structured Grid C.F.D Codes Onto A Transputer Network.

4.1 Introduction.

The parallelism in the solidification code in the previous chapter demonstrated how the data partition technique with minor code alterations from the serial code can be highly effective. To demonstrate these features more generally, a commercial code written with no view to parallelisation was required, incorporating a wide range of fluid flow capabilities.

4.1.1 C.F.D Algorithms.

The Navier Stokes equations governing the physical behaviour of fluid are expressed in the form :-

$$\frac{\partial}{\partial t}(\rho\Phi) + \mathrm{div}(\rho\underline{v}\Phi - \Gamma_\Phi \,\mathrm{grad}\,\Phi) = S_\Phi$$

incorporating transient, convection and conduction influences where Φ is a general field variable. For simple three dimensional flow, four equations of this form exist, one representing velocities in each of the three dimensions and one to represent mass continuity. Similar equations exist representing turbulence values, enthalpy etc. Table 1 gives the values of the diffusion coefficients (Γ) and source terms (S) for the various variables that the general form of the equation can represent. Each equation is integrated over a control cell from the discretized domain as in the solidification example leading to a linear equation for each control cell. The coefficients of these linear equations involve the field variables thus forming

an implicit set of interrelated equations.

Φ	Γ_Φ	S_Φ
1	0	0 (continuity)
u	μ_{eff}	$-\frac{\partial p}{\partial x} + \frac{\partial}{\partial x}(\mu_{eff}\frac{\partial u}{\partial x}) + \frac{\partial}{\partial y}(\mu_{eff}\frac{\partial v}{\partial x}) + \frac{\partial}{\partial z}(\mu_{eff}\frac{\partial w}{\partial x})$
v	μ_{eff}	$-\frac{\partial p}{\partial y} + \frac{\partial}{\partial x}(\mu_{eff}\frac{\partial u}{\partial y}) + \frac{\partial}{\partial y}(\mu_{eff}\frac{\partial v}{\partial y}) + \frac{\partial}{\partial z}(\mu_{eff}\frac{\partial w}{\partial y})$ $-g(\rho-\rho_{ref})$
w	μ_{eff}	$-\frac{\partial p}{\partial z} + \frac{\partial}{\partial x}(\mu_{eff}\frac{\partial u}{\partial z}) + \frac{\partial}{\partial y}(\mu_{eff}\frac{\partial v}{\partial z}) + \frac{\partial}{\partial z}(\mu_{eff}\frac{\partial w}{\partial z})$
H	$\mu_{lam}/Pr_{lam} + \mu_t/Pr_t$	$\dot{q}$
k	μ_{eff}/Pr_k	$G_K - \rho\varepsilon + G_B$
ε	μ_{eff}/Pr_ε	$(\varepsilon/k)\ [(G_K+G_B)C_1 - C_2\rho\varepsilon]$

where

$$G_K = \mu_t \{ 2 [(\partial u/\partial x)^2 + (\partial v/\partial y)^2 + (\partial w/\partial z)^2] + [(\partial u/\partial z) + (\partial w/\partial x)]^2 + [(\partial w/\partial y) + (\partial v/\partial z)]^2 + [(\partial u/\partial y) + (\partial v/\partial x)]^2 \}$$

$$G_B = \frac{\mu_t}{\rho} g \frac{\partial p}{\partial y}$$

Table 1 : The Diffusion coefficients and Source Terms of the one-phase conservation equations.

The solution of these equations can be achieved by a solution procedure from the SIMPLE family of algorithms (Patankar and Spalding 1972). The pressure calculation equation is

derived from a rearrangement of the continuity equation producing the pressure correction equation. The basic form of such a solution procedure is :-

1. Guess velocities u, v and w, pressure p and all other relevant variables
2. Solve discretized momentum equations for an improved solution to u, v and w using latest guesses for all other variables
3. Solve dicretized pressure correction equation for p', the adjustment to be made to pressure to fit the continuity equation
4. Update velocities u, v and w and pressure p using p'
5. Solve for other variables such as turbulence etc.
6. If pressure solution not converged, return to step 2
7. If required, update variables for next time step and return to step 2

This is known as the SIMPLE algorithm. Variants of this algorithm also exist such as SIMPLE, SIMPLEC, PISO which involve minor alterations in the algorithm operation.

4.1.2 Description Of FLOW3D.

The code used in this work is FLOW3D, a fluid flow simulation package from AEA's Harwell laboratory. The features incorporated in FLOW3D include :-

Steady state or Transient behaviour

One, two or three dimensional domains

Regular or body fitted grids

Laminar or turbulent flow using k-ε model

Enthalpy and other scalar concentration variable transportation

A selection of solution algorithms from the Simple family are provided in FLOW3D. In the construction of coefficients in the discretized equations, special treatment of the convection terms is required. FLOW3D provides a selection of differencing schemes including the UPWIND, HYBRID and QUICK schemes to evaluate these convection terms.

FLOW3D was originally developed to execute on a CRAY computer, using the massive storage available and exploiting the CRAY's vector capabilities. As a result large amounts of storage are used to reduce re-calculation thus improving execution time. This large storage requirement has important implications in porting to a distributed memory parallel system. The efficient memory usage criteria is clearly vital in the production of a practical and useful parallel code.

4.2 Conversion To A Parallel Code.

4.2.1 Requirements Of Parallel Code.

The requirements of the parallel FLOW3D version are similar to those for the solidification example. The transparency of the parallel version is clearly essential to allow the users of the code to use existing input files and create new input files without any consideration to whether a parallel or serial version of FLOW3D is to be used. The maximum usage of computational power is also a vital requirement and the efficient use of distributed memory,

as previously mentioned, is absolutely essential.

Additionally, however, the minimum alteration of the serial code in construction of the parallel is a fourth vital requirement. Since FLOW3D is an ever evolving package, it is important that corrections and improvements can easily be incorporated in the parallel version as well as the serial. This should ideally be performed by the original code authors/maintainers without the requirement of any knowledge of the details of the parallel conversion. As such, a routine in the original serial code should have an equivalent routine in the parallel version with only minor alterations whenever possible. Clearly, the best way to achieve easy maintenance is to hold only one version of the code with either runtime decisions to determine execution of extra processes for the parallel version, or by the use of a source code preprocessor. The initial version, however, is implemented as a separate source version with the feasibility of integrating parallel and serial versions being considered in future work.

4.2.2 Data Partition Strategy.

As with the solidification example, the local calculation nature of the control volume technique suggests a suitable data partition will provide an effective parallel code. Figure 4.1 shows the 7 point stencil of values required to update a variable representing a value at control cell P. An investigation of the code reveals that the stencil represents the maximal set of values required in all calculations except for the calculation of convection coefficients when a second order differencing scheme is used (i.e. QUICK, HUW and CCCT). As a result, only the three first order schemes are implemented in the parallel code (although the second

order schemes could be incorporated in later versions by, for example, including extra

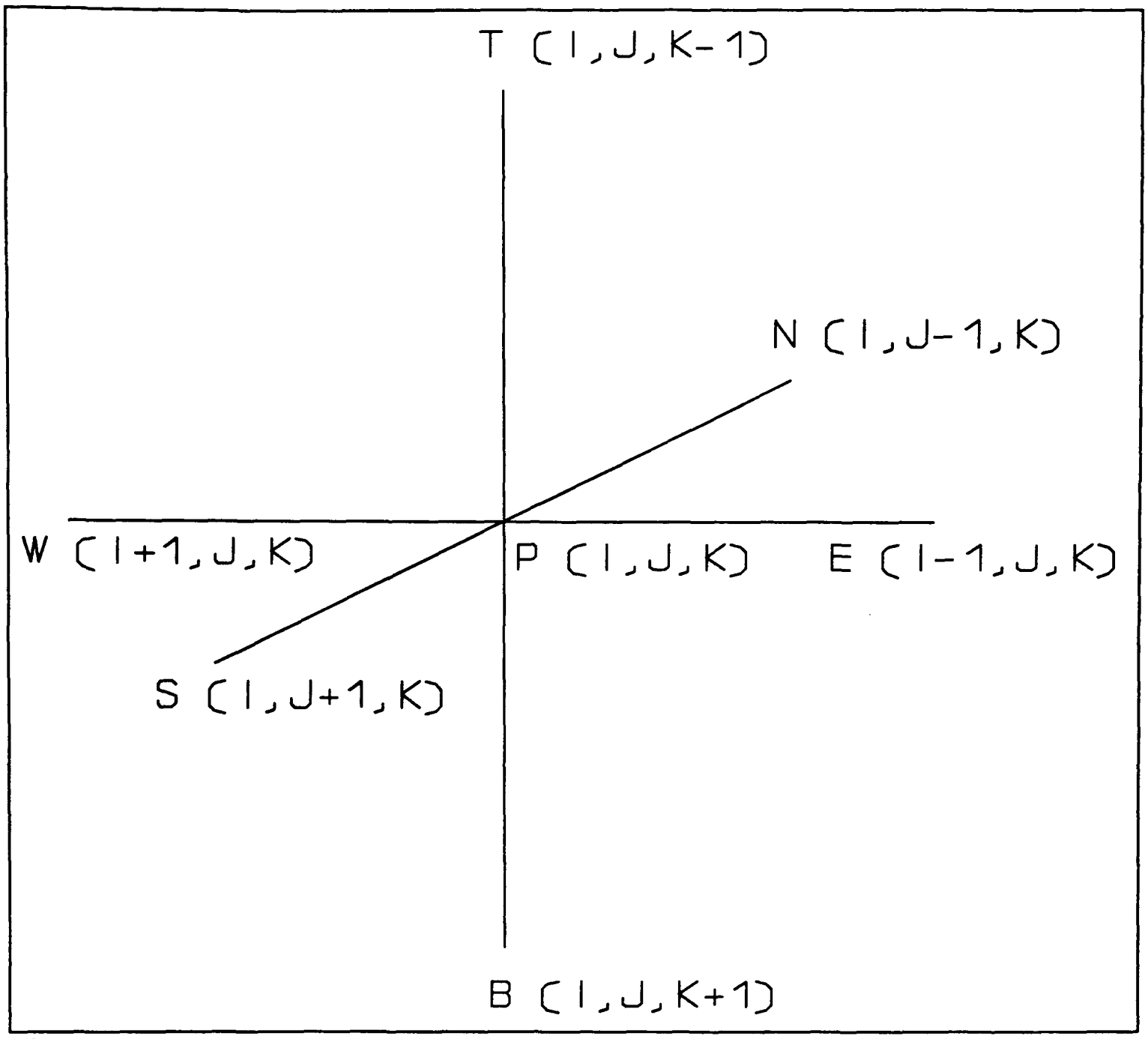


Figure 4-1 Seven Point Stencil For 3D Control Volume Calculations

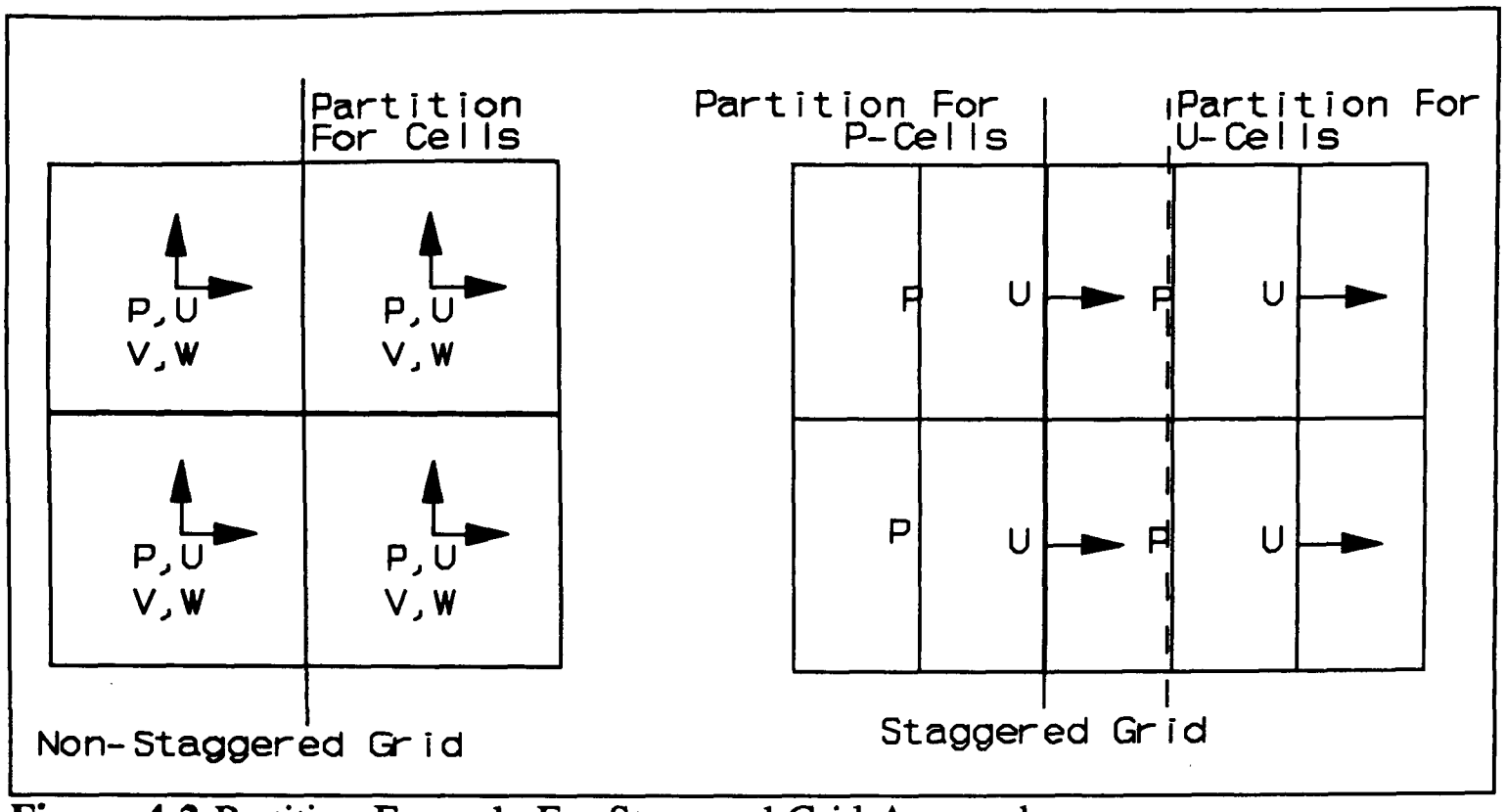


Figure 4-2 Partition Example For Staggered Grid Approach.

communications).

The FLOW3D code uses the Rhie and Chow formulation (Rhie and Chow 1983) which requires pressure values to be stored at cell centres (as with all other variables). This is an alternative to the staggered grid approach used by many codes where the velocity values are stored at cell walls to avoid numerical problems encountered when using first order differential approximations. The inclusion of the cell centred pressure version in a data partition clearly presents no problem whilst figure 4.2 demonstrates that the staggered grid technique can also be simply incorporated in a data partition.

Three data partition strategies arise naturally from an investigation of the code (see figures 4.3 to 4.5). The first possibility is the three dimensional partition, where a decomposition into cuboidal subset of control cells is used. The second alternative is the two dimensional partition, where a decomposition into long rectangular boxes of control cells containing a subset of cells in two dimensions with all cells in the third dimension is used. Finally, a one dimensional decomposition into sets of slabs of control cells can be used.

Each of these partition strategies could produce effective parallel code, the choice between them being made by a number of factors. The three dimensional partition allows for massive parallelism (i.e. the use of thousands of processors), however, to maintain nearest neighbour communications on the processor network would require six communication links per processor whilst only four are available on the current generation of Transputers. Both the three and two dimensional partitions require large, disjoint sets of data to be exchanged between neighbour processors, requiring a set of either multiple communication calls or gather

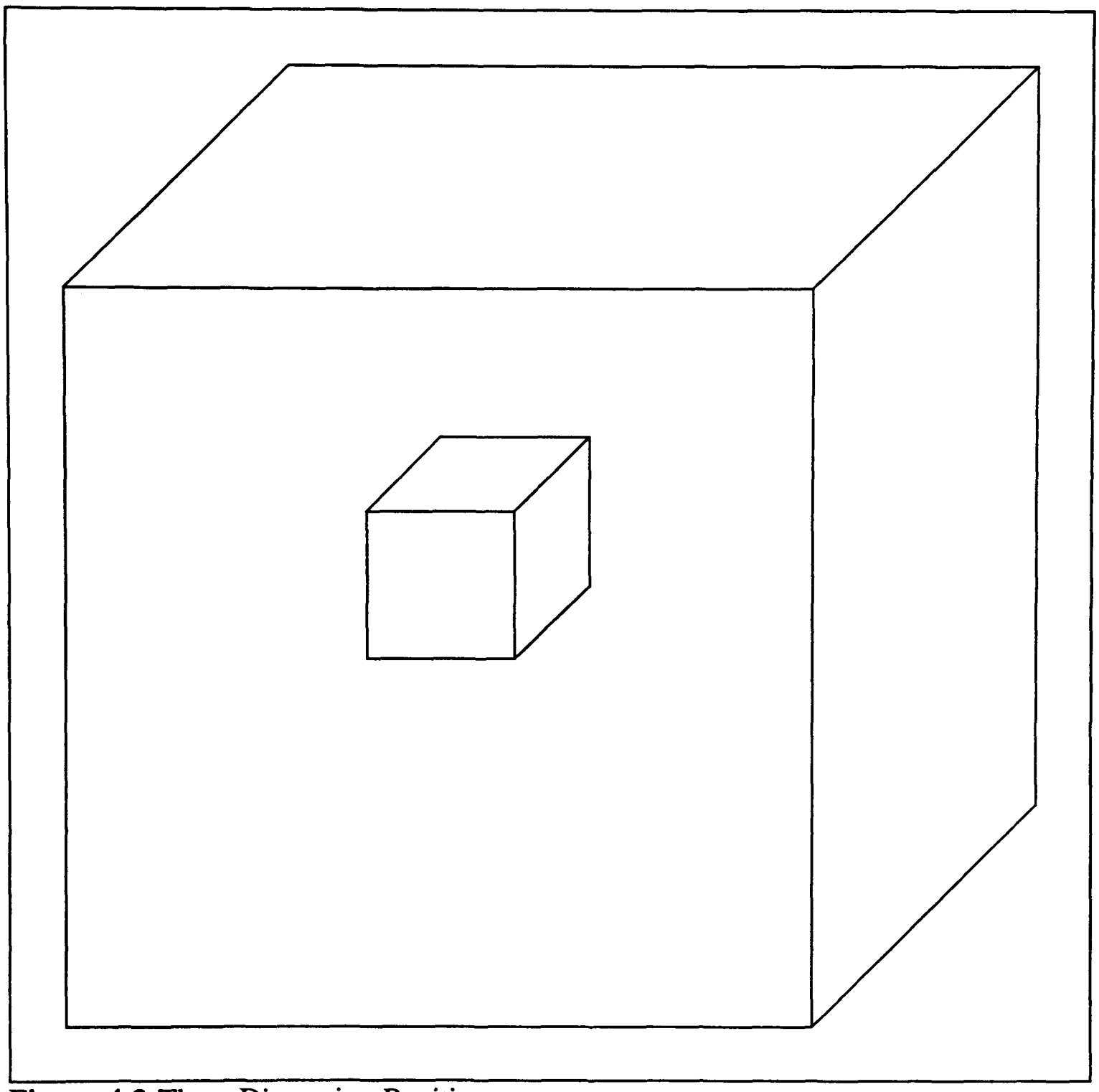


Figure 4-3 Three Dimension Partition.

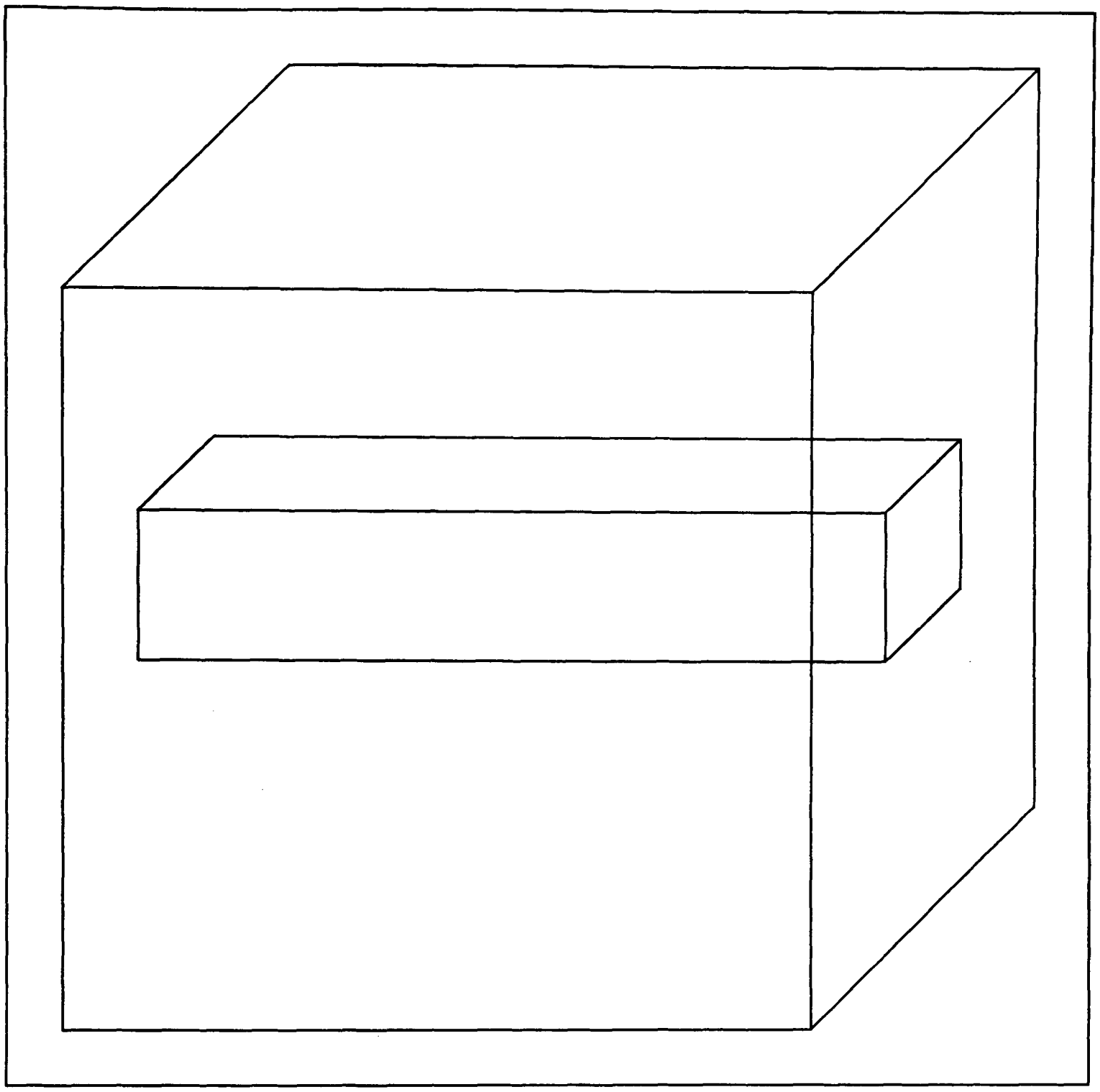


Figure 4-4 Two Dimensional Partition.

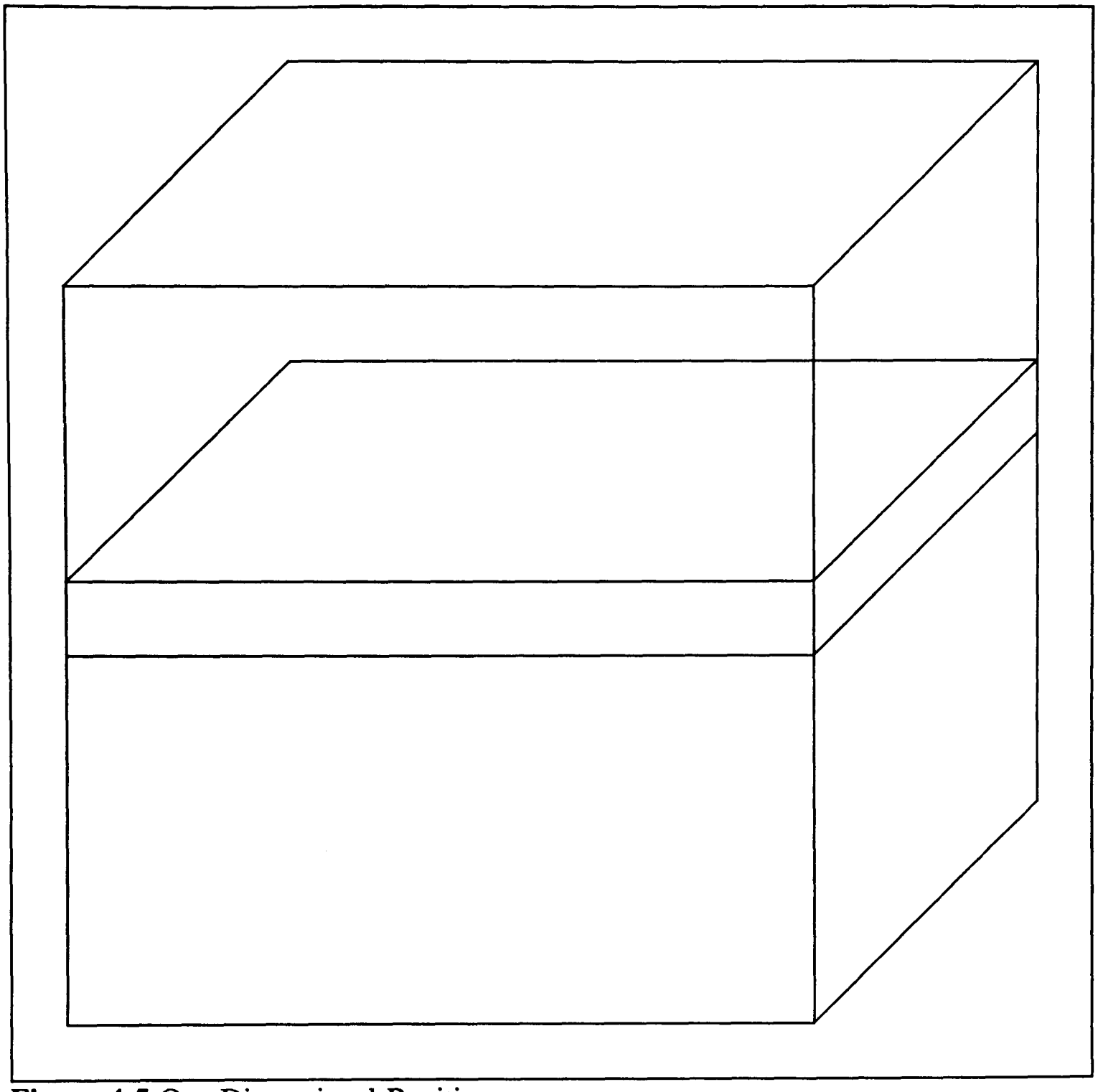


Figure 4-5 One Dimensional Partition.

and scatter operations with a single communication. The one dimensional partition requires only a set of single bulk data communications since the required data is stored in consecutive memory locations. Conversely, the one dimensional partition is restrictive in that the maximum number of processors that can be utilised is the number of slabs in the problem to be solved. The processor configuration required to achieve a one dimensional partition is a chain of processors as used in the solidification example.

In the scope of this work, the one dimensional partition has been implemented to demonstrate the effectiveness of the data partition technique since the amount of code restructuring required is less than for the other strategies. Another major reason for this choice was that the target systems contained small to medium numbers of Transputers (i.e. up to around 50). The amount of communication and potential algorithm replacement that may be required in the three and two dimensional partitions may well cause a significant performance degradation over that attainable through the one dimensional partition. For larger parallel systems, the other partition strategies may prove essential. For completeness, the implication of each partition strategy will be considered in the following sections where possible.

The partition and overlap areas for the one dimensional partition are shown in figure 4.6. Unlike in the solidification example, here the allocation areas are a set of complete slabs. This restriction is necessary to minimise the code alteration since many of the domain computations involve three nested loops, one for each dimension, where adjusting one of the loops only suffices for the partition implementation. As a result, balanced loads on each processor cannot be guaranteed unless the number of slabs is divisible by the number of processors used. The importance of any uneven partition clearly diminishes as the problem

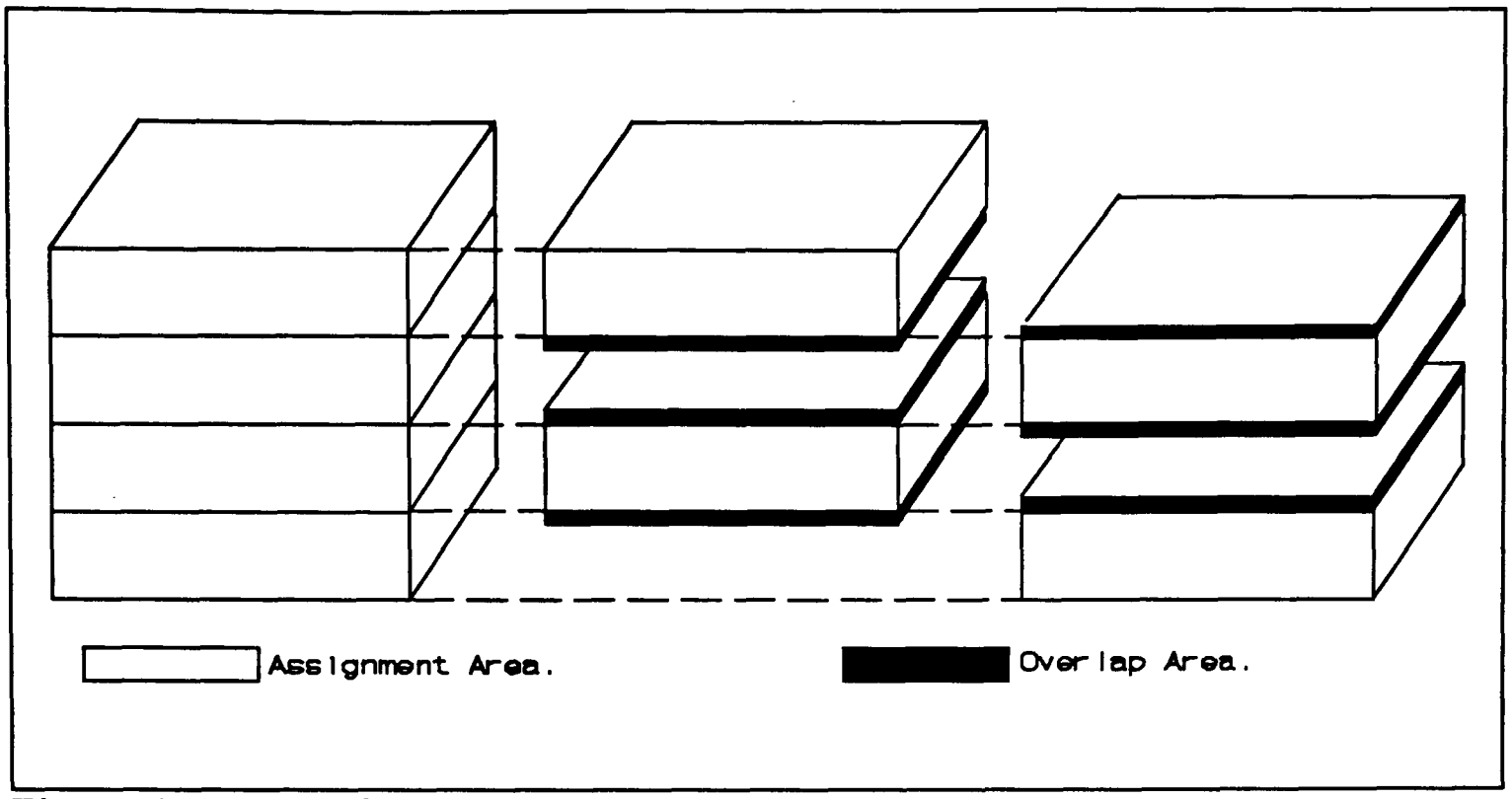


Figure 4-6 Details Of One Dimensional Partition.

size (i.e. number of slabs) increases, i.e. as the unbalancedness reduces proportionately.

4.2.3 Code Alterations.

Although a knowledge of the nature of the methods used in the code aided the decision on how the parallelisation should proceed, the actual code alteration process was performed purely by an inspection of the original code. The aim of the code alterations, as with the solidification, is to limit each processor to only perform computations involved in assigning data in that processor's assignment area.

For the vast majority of the code, the alterations involved adjustment of loop limits and array declarations. For the one dimensional partition, only the z dimension loop and declarations need be adjusted. In some cases, such as when several different array elements are assigned in a single iteration, masked assignments are required. This is where a conditional statement is added to control the execution of the array assigning statement, the value of the conditional only being true if the array element to be assigned is allocated to this processor's allocation area. Another case where care is required in FLOW3D is when loops start from the second slab. To implement such loops in the parallel implementation, the lower limit of such a loop operating in the z direction must thus be two on the first processor, but start from the first slab allocated on all other processors. Other similar cases exist, all requiring attention to ensure semantically correct parallel code.

The setting up of the partition and initialisation of all required data on each processor had to be achieved according to the order of data read from the FLOW3D input file, bearing in mind

the memory limitation that no single processor is allowed to store more than the amount of data to be allocated to it. As a result, data read from the host must immediately be sent to the processor(s) on which it is required and not buffered on the first processor.

The first stage in the setup is the reading of problem and work space size information from the input file. From this the one dimensional data partition can easily be calculated. This information is then sent to all processors where they allocate memory for the required arrays and await data sent later. The next stage involves the setting up and distribution of geometric information, whether this information is read from a file or calculated from information in the input file. When the geometric information for a processor is received/evaluated on the first processor it is immediately sent to the processor concerned before the next processor's geometric information is processed. The final phase in the setup is to distribute the boundary condition information across the processor network. Boundary conditions in FLOW3D are preprocessed and converted into an internal representation. A significant amount of code is required to perform this preprocessing so, rather than duplicating this code on every processor wasting valuable memory, it is only held on the first processor. The preprocessing, however, must be performed in two stages since geometric information is required at one point. Thus the boundary conditions are preprocessed on the first processor without the incorporation of the geometric information, then sent to the appropriate processor(s). When received, the geometric information is incorporated.

4.2.4 Improved Exchange Procedure.

An exchange of slabs of data between neighbour processors is required frequently to ensure

the semantics of the serial code are preserved. Whenever any of the solution variables are updated an exchange must be performed before that variable can be used on a neighbour processor. Since the full C.F.D algorithm involves far more variables than the conduction only solidification example, further attention was given to improving the speed of performing the slab data exchange.

The improved exchange procedure uses three subsidiary threads along with the main thread, each performing one of the four required individual communications (see figure 4.7). The subsidiary threads are started by initialisation calls at the start of the parallel FLOW3D code. They then deschedule themselves on reaching a receive statement (i.e. use no CPU cycles), waiting until data is sent through the appropriate internal communication channel from the main thread. When a call is made to the exchange procedure in the main thread, the appropriate start address of the data to be exchanged by a particular thread is sent through the appropriate internal communication channel to that thread. The reception of this data starts up the thread which then sends/receives one slab of data from the address received from the main thread. On completion of its communication, each thread sends a signal to the main thread indicating completion before returning to the receive statement, again descheduling itself. When all subsidiary threads have informed the main thread of their completion (and the main thread has performed its inter-processor slab communication), the main thread continues, returning from the call to the exchange procedure.

4.2.5 Parallelising The Linear Equation Solvers.

The entire solution procedure has been parallelised using the one dimensional data partition.

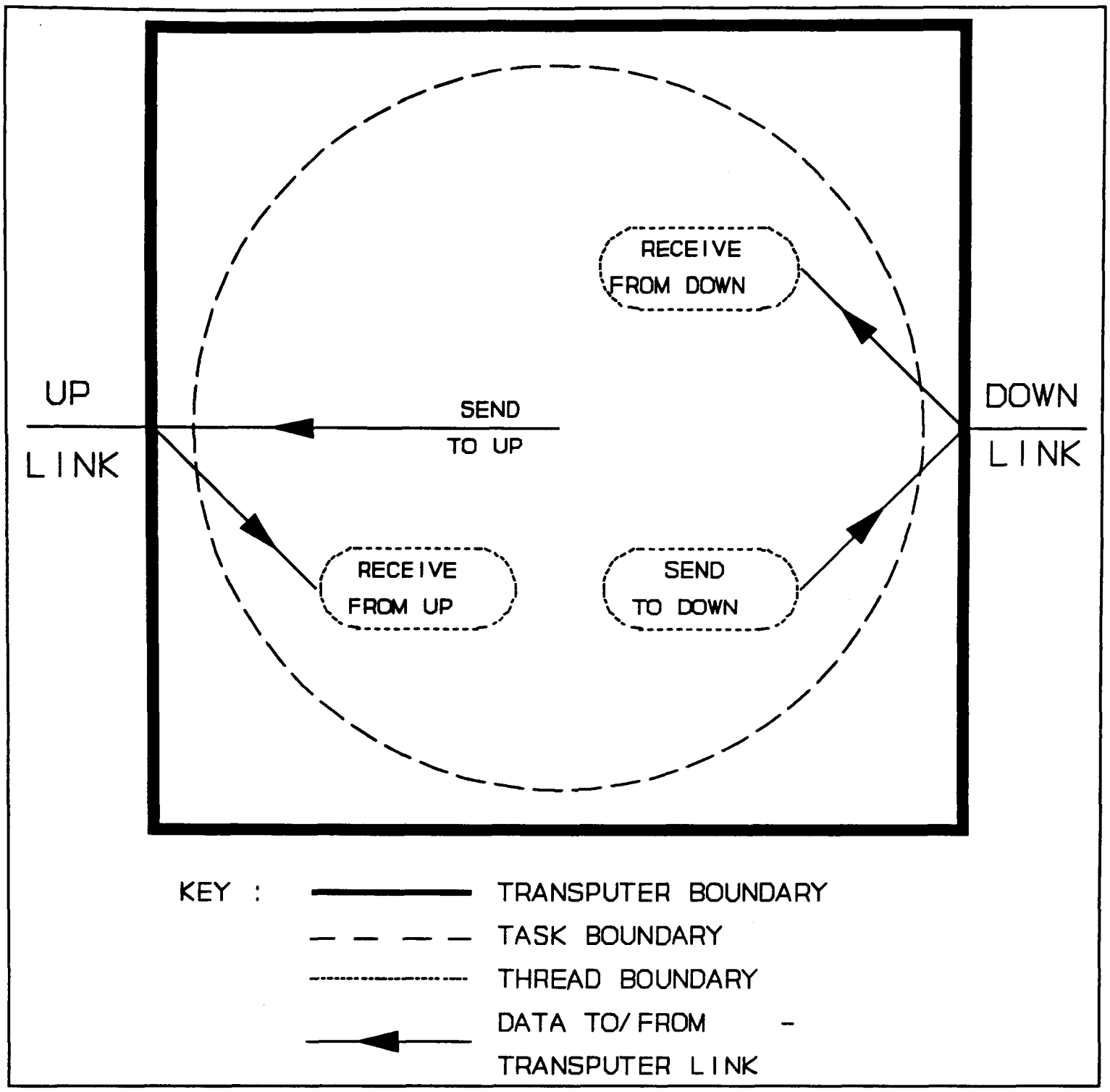


Figure 4-7 Diagramatic representation of the structure of the exchange procedure.

The explicit nature of most of the operations concerned, such as the coefficient setup and boundary condition incorporation, allowed most of the parallelism in the code to be fairly easily exploited using the code alterations mentioned earlier. The linear equation solvers, however, solve an interrelated set of equations where the effective solution algorithms often prove implicit. Extra attention was therefore required in parallelising these solvers, particularly since the percentage of serial execution time spent in these routines is very high.

FLOW3D incorporates six linear equation solvers of which three have been implemented in this work. The remaining three solvers can be parallelised using techniques used for the solvers that have been parallelised.

4.2.5.1 Stone's Implicit Procedure.

The Stone's implicit procedure uses recurrences to calculate coefficients in the Stone matrix. Recurrences also exist in the forward elimination and backward substitution phases of the procedure. In each case, three nested loops are used, the inner two loops passing over matrix diagonals and the outer loop passing over slabs. An investigation of these recurrences indicates that the outer slab loop is inherently serial whilst parallelism can be extracted from the inner loops.

To exploit this parallelism for the chosen partition, loop interchange was required. The outer slab loop was transferred to become the inner loop of the nest. The semantic validity of such a transformation and the adjustment of the scalar index variables to maintain semantics were performed by careful manual inspection of the code. To achieve parallelism, the now inner

slab loop is performed in a pipeline over the chain of processors with the addition of communications calls before and after the loop enforcing the pipelining as shown below :-

Serial Code

```
DO ISLAB=1,NSLAB
    DO J=. . .
        DO K=. . .
            . . .
        END DO
    END DO
END DO
```

Parallel Code

```
DO J=. . .
    DO K=. . .
        Receive value(s)
        DO ISLAB=NLOW,NHIGH
            . . .
        END DO
        Send value(s)
    END DO
END DO
```

I.e the outer two loops perform in a DOACROSS fashion as described in chapter 5. For the first iteration of the two now outer loops, the inner loop performs assignments for the first set of slabs on the first processor only whilst all other processors remain idle. When the first processor completes the inner slab loop, the last value evaluated is sent to the second processor, initiating its computation. The first processor then commences the second iteration of the middle loop in the nest, performing the associated iterations of the inner loop in parallel with the second processor. This process continues, eventually allowing all processors to be working in parallel.

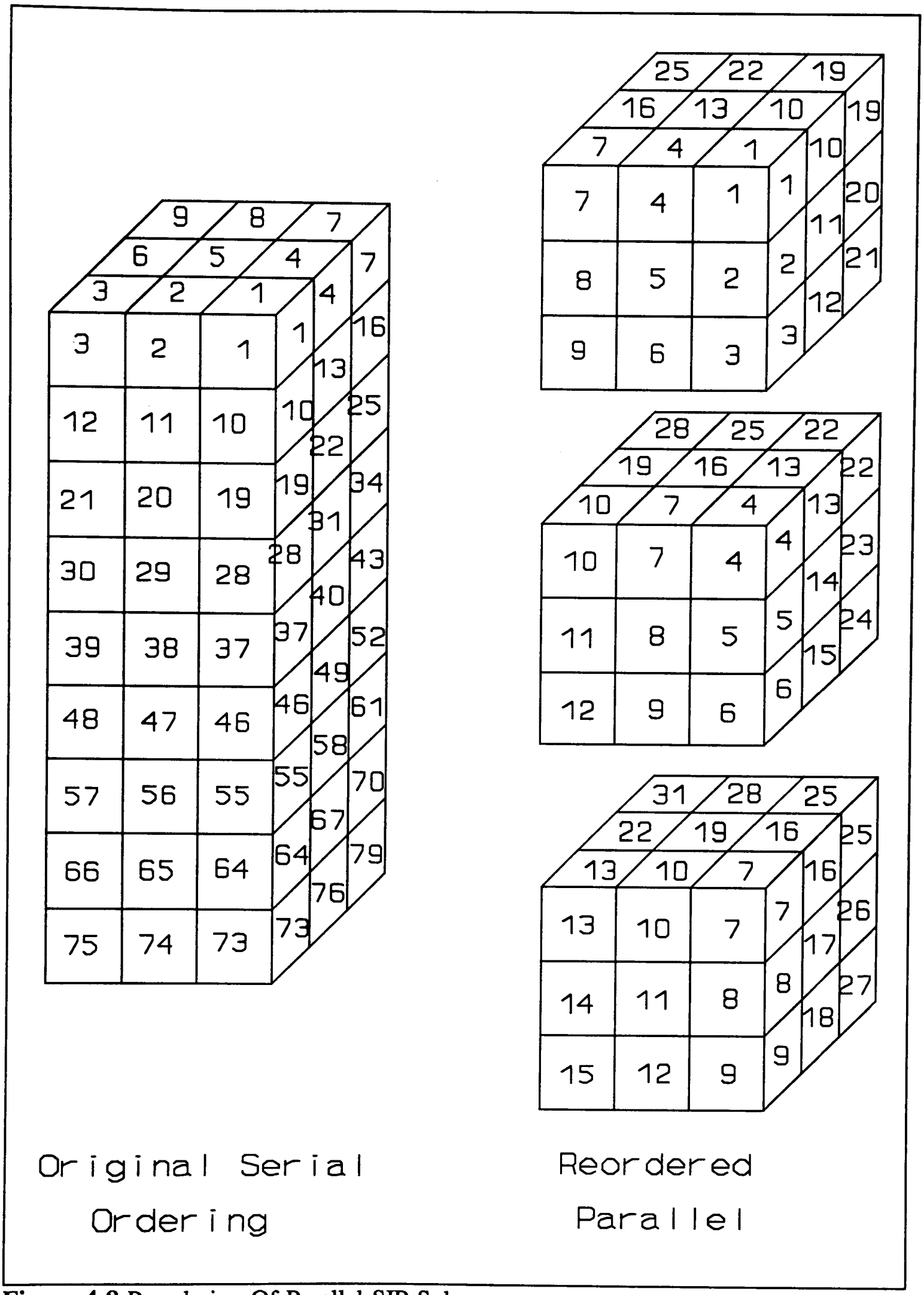


Figure 4-8 Reordering Of Parallel SIP Solver.

Figure 4.8 shows an example of a 3 x 3 x 9 grid. The numbers in the control cells indicate the execution order of the evaluation of values in those cells. The parallel execution order is shown for a three processor system using the above scheme. The semantic validity of the parallel code is assured since when any cell values are updated in the parallel version it will use the same set of previously updated neighbour cells as in the serial version in the calculations. The parallel version shown in figure 4.8 indicates that 18 units of processor time are idle and 81 units involve computation (a unit being the time taken to perform a single cell computation). The total serial execution time is 81 units whilst the parallel execution time is 33 units, thus the maximum possible speedup is around 2.5. Clearly, the communication time for the 18 individual cell value communications required in the parallel version will cause an extra overhead, reducing this speedup. Although this speedup figure is somewhat disappointing, as the problem size increases the startup and shutdown time of the process will proportionately reduce compared to the full system utilisation time.

Using this technique with the two and three dimensional partitions may also be possible. However, in both cases, significant code alterations will be required (breaching one of the parallel code requirements), and introduce far more processor idle time than for the one dimensional partition. As a result, the Stone's implicit procedure may not be as feasible for two and three dimensional partition parallel versions.

4.2.5.2 Line Relaxation Solver.

The line relaxation solver uses bi-directional line sweeps across slabs in the problem domain. The main sweep direction is set by the user according to the nature of the problem. The line

sweeps are then performed alternately in the two directions perpendicular to the main sweep direction. With the one dimensional partition, the main sweep direction is fixed to be in the z direction. This allows the alternate sweep direction feature to be preserved in the parallel version since this will operate over the complete slabs allocated to each processor. For the two dimensional partition only a single direction for the line sweep is practical without major algorithm change or the introduction of excessive processor idle time. The three dimensional partition does not suit a line based approach, thus algorithm change will be required (assuming excessive idle time is unacceptable) as discussed in chapter 7.

Two alternative approaches for the parallelisation of the line relaxation solver can be used :-

a) Create a pipeline of line updates across slabs creating a DOACROSS/pipeline system as for Stone's implicit procedure

b) Allow 'old' values to be used for certain updates reducing convergence rate as with the solidification example

Since method a) uses a similar approach to that used for Stone's implicit procedure, the second approach has been used. The convergence behaviour of the parallel version as compared to the serial is a major influence of the effectiveness of the parallel version.

4.2.5.3 Conjugate Gradient.

The conjugate gradient method itself uses only explicit calculations thus parallelisation of the basic method is fairly straight forward for all three partition types. To use the method

successfully, a matrix preconditioner is often required. Many preconditioners are supplied in FLOW3D, however, only the diagonal preconditioner is at present implemented which involves only explicit operations. The other preconditioners use matrix operations involving forward elimination and backward substitution and could thus be parallelised using a similar approach to that employed in the Stone's implicit procedure parallelisation.

4.3 Performance Of Parallel FLOW3D.

To test the effectiveness of the parallel implementation of FLOW3D, two test problems were used which involve simple flow (i.e. velocities and pressure variables only). Both problems are derived from fairly standard two dimensional test problems adapted to have simple three dimensional geometry and three dimensional flow characteristics. The tests have been performed for a range of problem sizes and a range of Transputer numbers with several combinations of linear equation solvers (the effectiveness of the solvers being a major influence on effectiveness of the parallel version). The combinations of solvers used are :-

Stone's implicit procedure for all solutions

Stone's implicit procedure for all solutions except the pressure correction which uses diagonally preconditioned conjugate gradient

Line relaxation solver for all solutions

Line relaxation solver for all solutions except the pressure correction which uses diagonally preconditioned conjugate gradient

The conjugate gradient solver in FLOW3D requires a symmetric matrix system. Since only

the pressure correction equation system provides such a matrix, only the solution of this equation can use this solver.

The tests were carried out on two Transputer systems. The smaller scale tests were performed on a Transputer system at Thames Polytechnic whilst the larger scale tests were performed on the Meiko computing surface at Edinburgh University. The equality of the two systems was verified, giving identical timings for the same problem.

All the results shown are for balanced loadings i.e. the number of processors used is a factor of the number of slabs in the computational grid.

4.3.1 Backward Facing Step Problem.

This problem models the flow through a square cross-sectioned duct with a restricted inlet (see figure 4.9). The inlet velocity of the flow was set to cause recirculations just downstream of the inlet to complicate the flow patterns.

Figures 4.10 to 4.13 show the speedup produced for a range of processor numbers. Each curve shows the speedup for a particular density of control cells. All solver combinations showed an improvement in speedup for a particular number of processors as grid density increases with a degradation as processor numbers increased.

The Stone's implicit procedure alone proved fairly successful attaining a speedup of 32 on a 50 processor system (64% efficient) for the most dense grid (40000 cells). The loss of speedup is due largely due to the high inter-processor communications required. For smaller

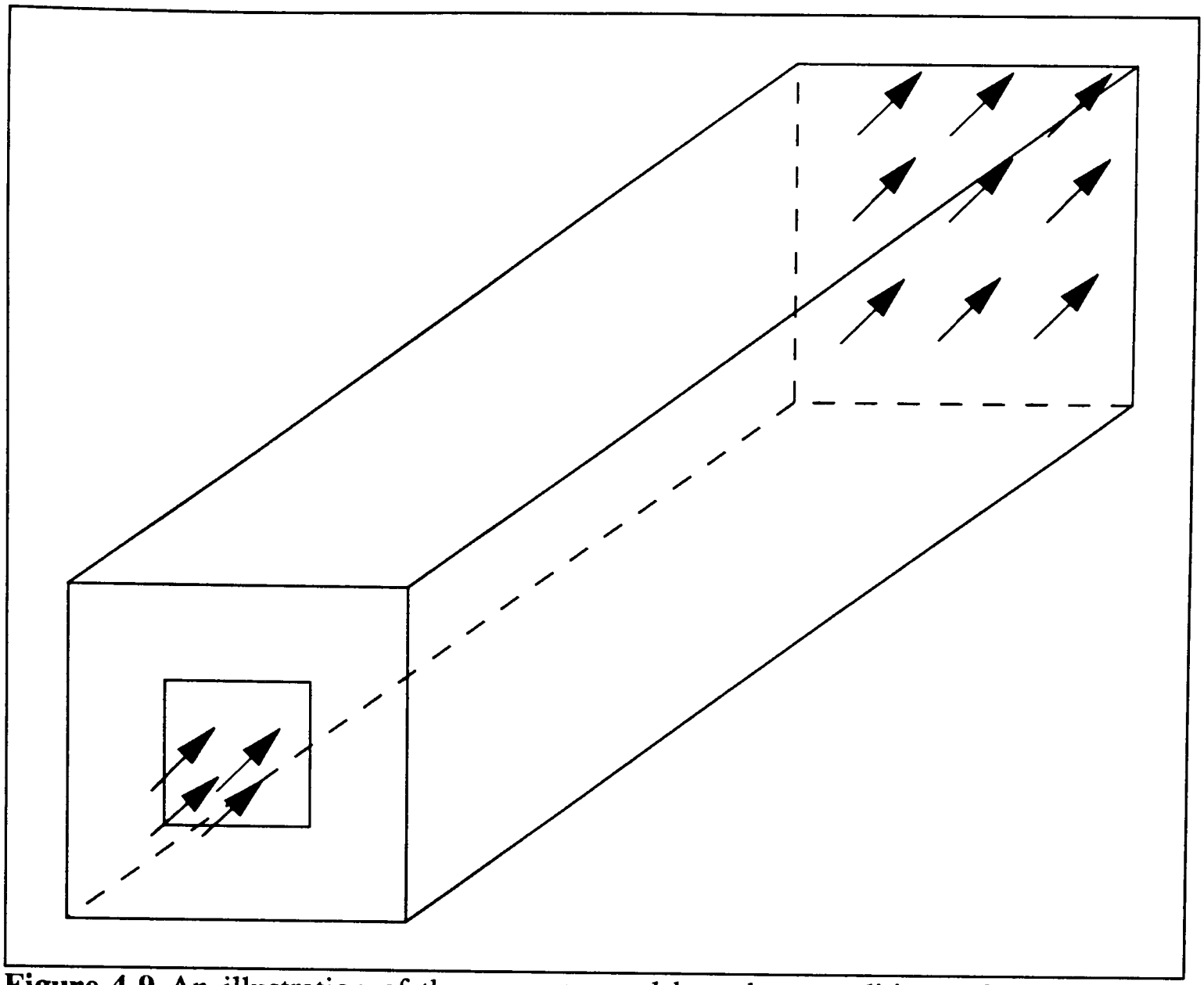


Figure 4-9 An illustration of the geometry and boundary conditions of the backward facing step problem.

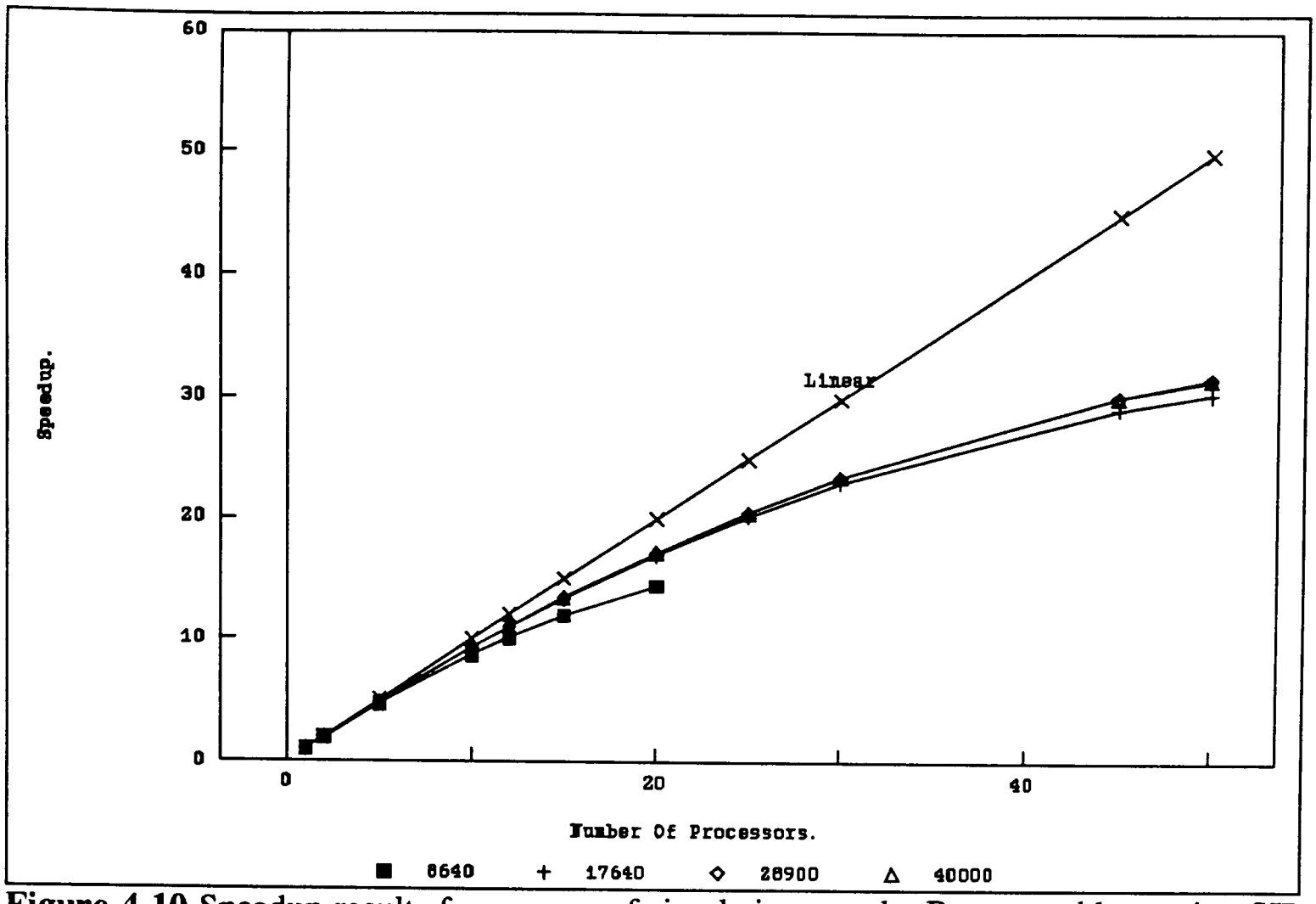


Figure 4-10 Speedup results for a range of simulations on the B step problem using SIP as the solver for all variables.

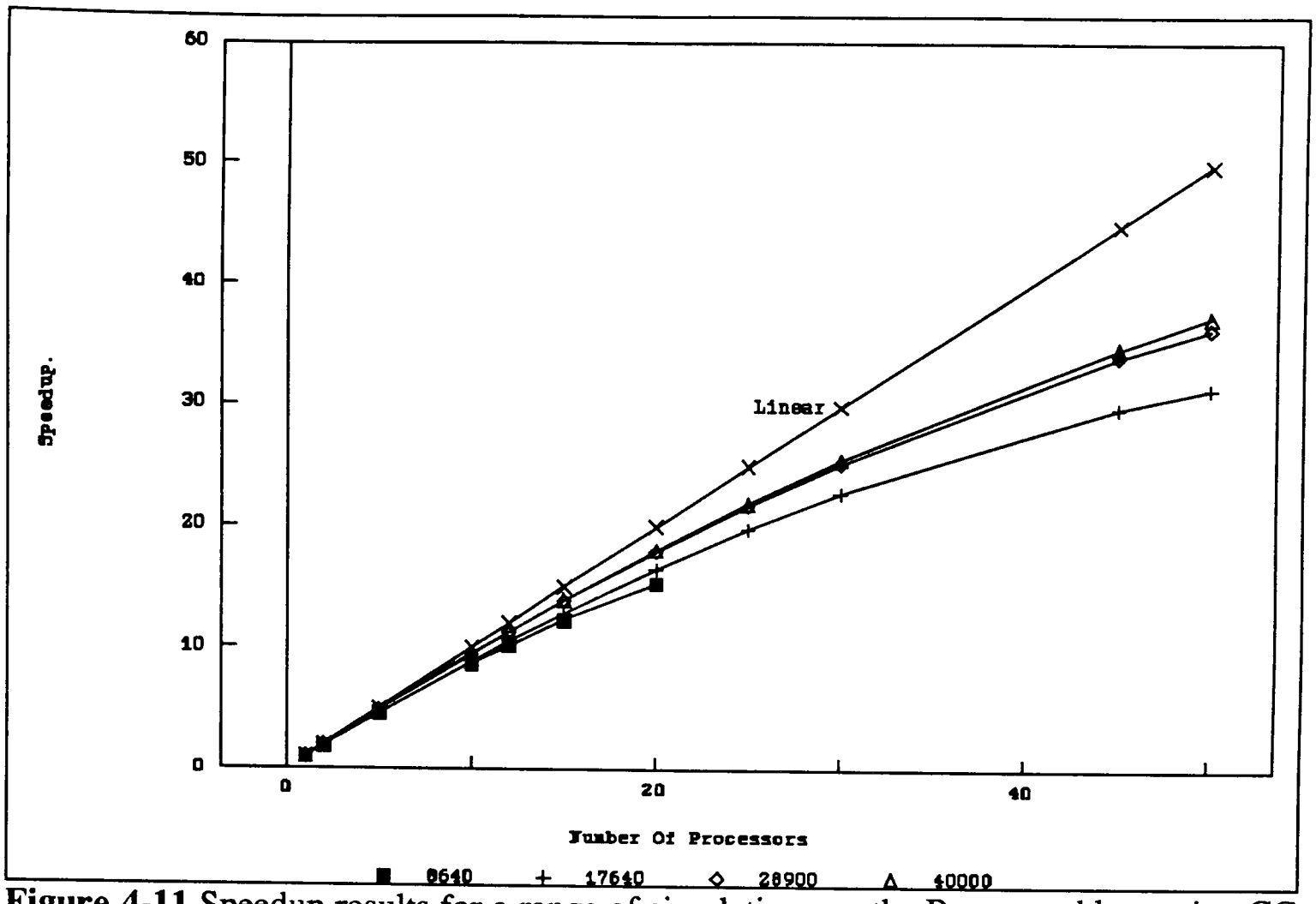


Figure 4-11 Speedup results for a range of simulations on the B step problem using CG for pressure solution and SIP for all other variables.

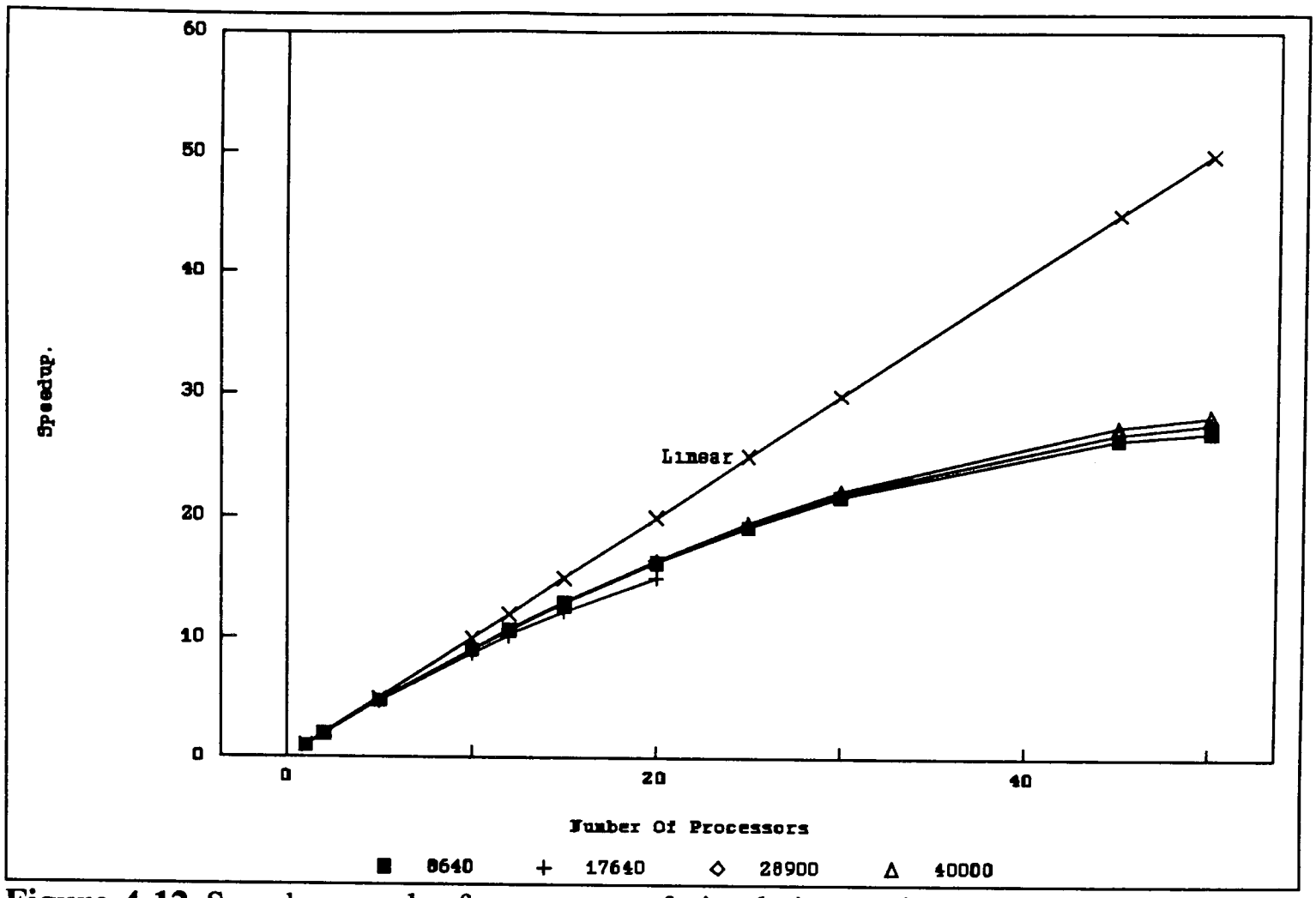


Figure 4-12 Speedup results for a range of simulations using the line solver for the solution to all variables.

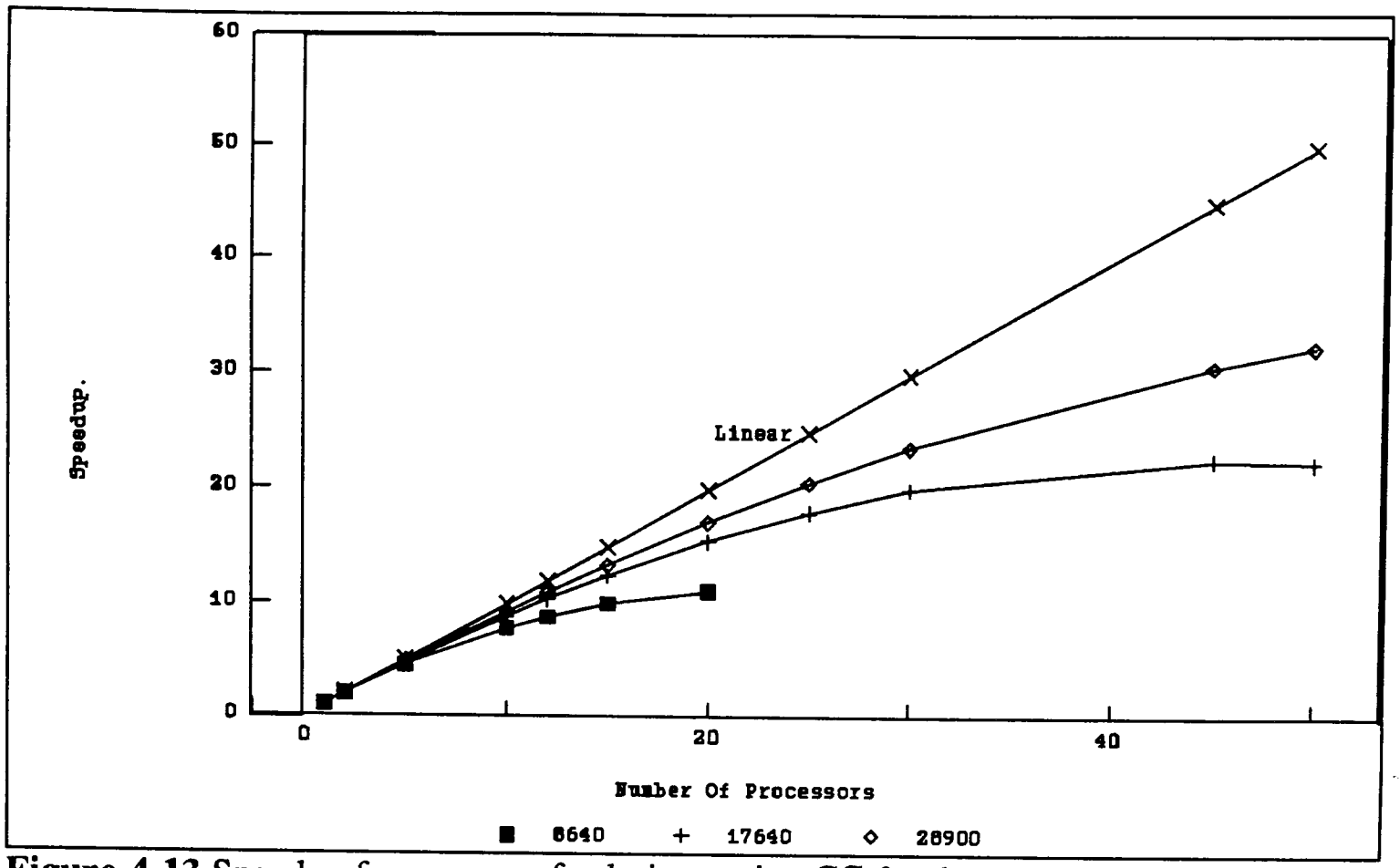


Figure 4-13 Speedup for a range of solutions using CG for the pressure solution and line solver for all other variables.

processor numbers typical efficiencies were 85% and above. When the pressure correction solution is achieved using the conjugate gradient method, the speedups exhibited improved (37.5 on a 50 processor system giving 75% efficiency). This occurred since the pressure correction solution constitutes a very significant proportion of the solution time (i.e. far more linear equation solver iterations are dedicated to pressure correction solutions than all others), where the communication overhead incurred in the conjugate gradient solution is far less than that of Stone's implicit procedure. The gradient of the speedup curve for 50 processors indicates that a larger number of processor could be effectively exploited using this solver combination.

The use of the parallel line relaxation solver for all solutions gave fairly poor speedups. On the larger processor number examples efficiencies of less than 50% were produced. The poor performance is caused by the increase in line solver iterations required on the larger systems. Since the first slab on all but the first processor uses old values for the 'top' value, as the number of processors increases the number of 'old' values used in updates increases, slowing convergence. Figure 4.14 shows the percentage increase in line iterations to reach convergence for the 8640 cell problem. For 15 processors, around a 15% increase in the number of iterations is required with a gradient indicating that this percentage will increase as more processors are used. When the conjugate gradient method is used to solve the pressure correction equation the speedups produced for the 28900 cell problem improved over those for the line solver only solution. The use of the line solver for the velocity components, however, caused inconsistent convergence behaviour, with no converged solution being produced for the 40000 cell problem. Figure 4.14 shows the percentage increase in iterations required to reach a converged solution for the 8640 cell problem. The increase in line solver

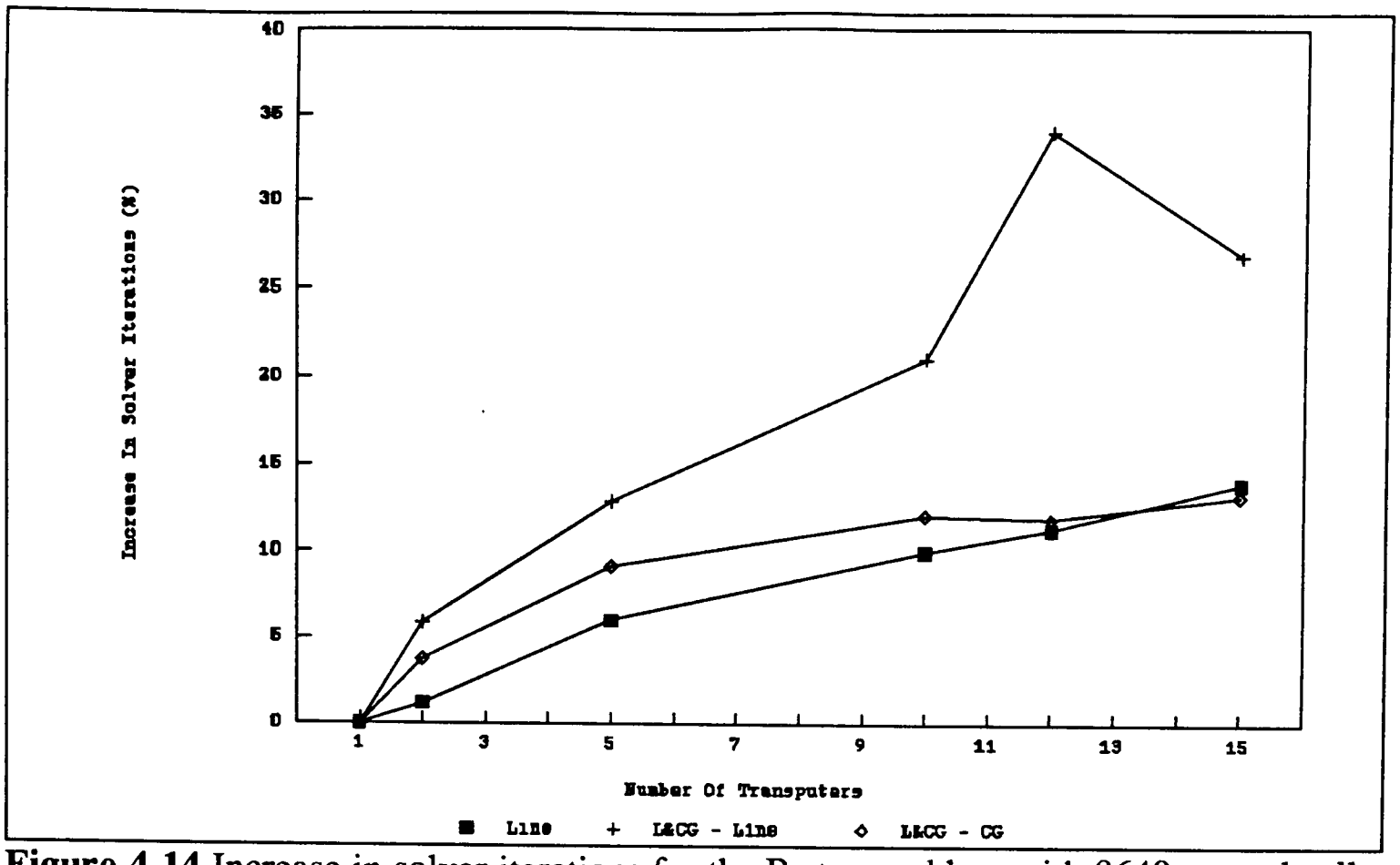


Figure 4-14 Increase in solver iterations for the B step problem with 8640 control cells.

iterations is higher than for the line solver alone, however, since the conjugate gradient solution time dominates the overall time, this increase has a small effect. The number of conjugate gradient iterations required is fairly consistent as processor number increase in the given example, thus for larger processor numbers the effectiveness of this solver combination is greater than the line solver alone solution for certain problems.

It should be noted that the serial execution time used for speedup calculation in the larger problems is estimated since the memory requirement prohibited serial execution on the systems available. The estimate was made assuming linear efficiency degradation as numbers of processors increase. This assumption is made in accordance with the efficiency graph shown in figure 4.15.

4.3.2 Moving Lid Cavity Problem.

This problem models the flow in a three dimensional box where the flow is induced by the lid of the box moving in one direction with constant velocity (see figure 4.16). The flow is further complicated by wall effects making it truly three dimensional.

Figures 4.17 to 4.20 show the speedups produced on up to 18 processors for several grid densities.

The Stone's implicit procedure, when used for all solutions, produces a speedup of just over 13 on 18 processors for the highest grid density (i.e. 72% efficiency for 46656 cell grid). When the conjugate gradient method is used for the pressure correction solution, the speedups

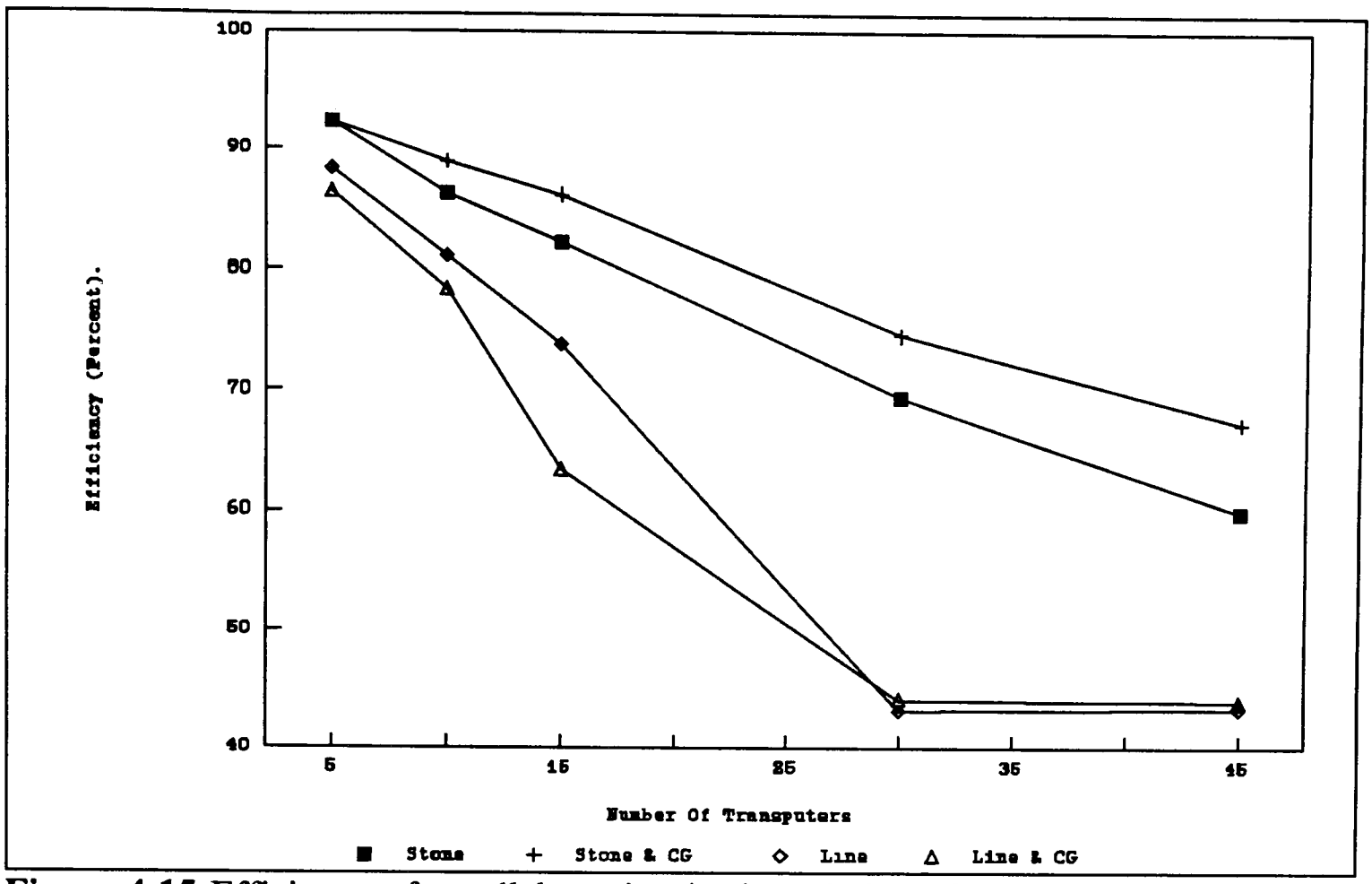


Figure 4-15 Efficiency of parallel version in the solution of the B step problem with 17640 control cells.

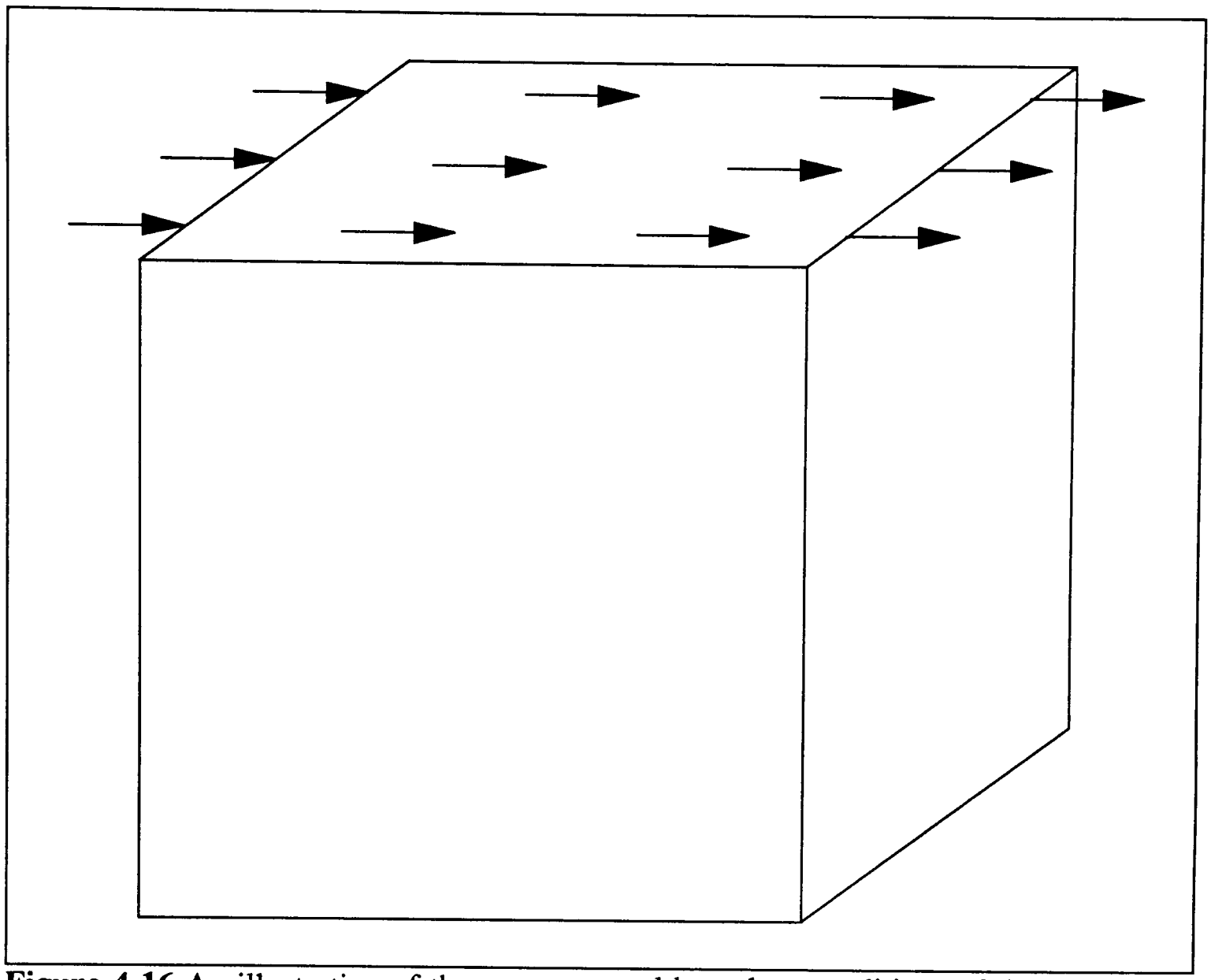


Figure 4-16 An illustration of the geometry and boundary conditions of the moving lid cavity problem.

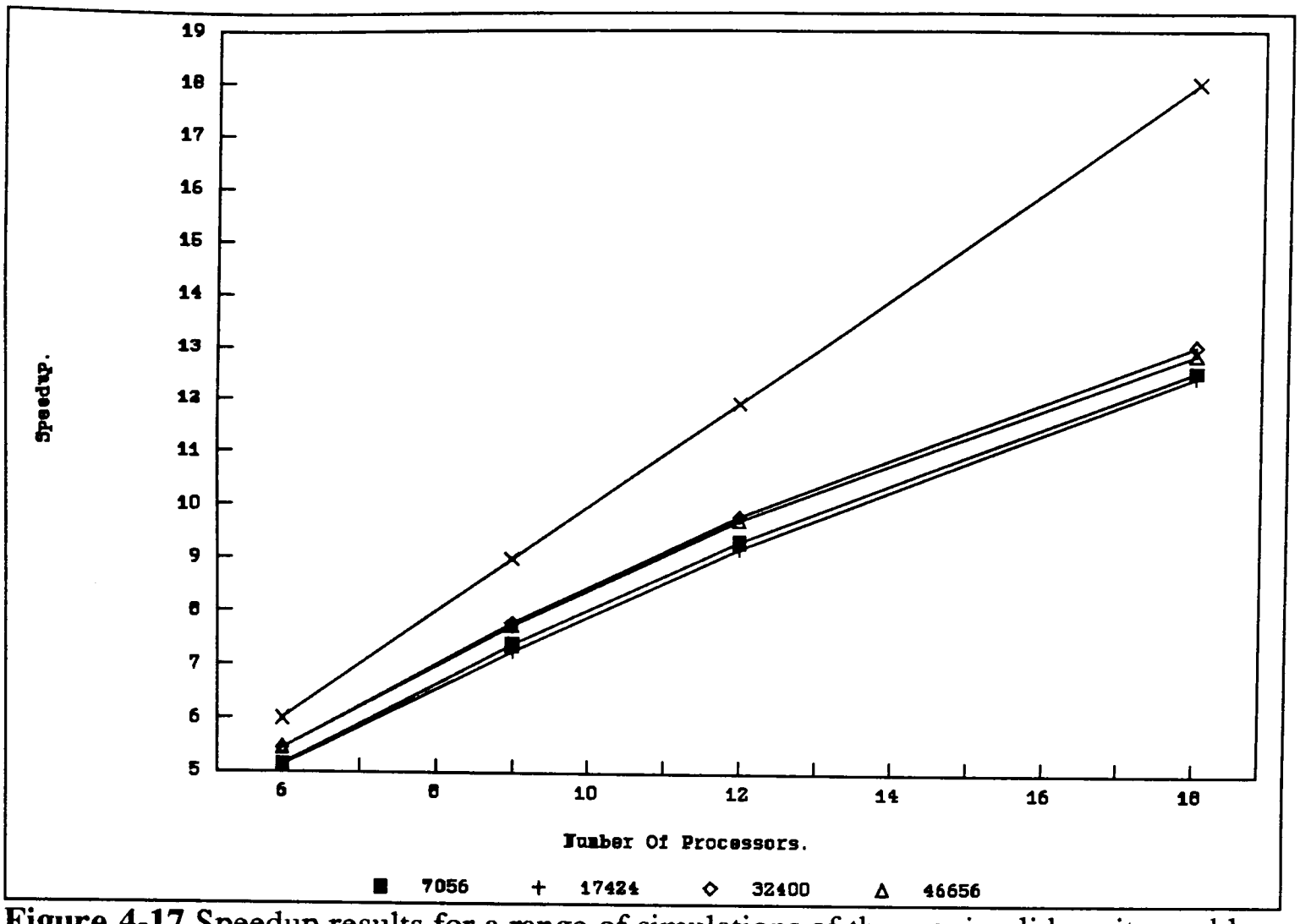


Figure 4-17 Speedup results for a range of simulations of the moving lid cavity problem using SIP for the solution to all variables.

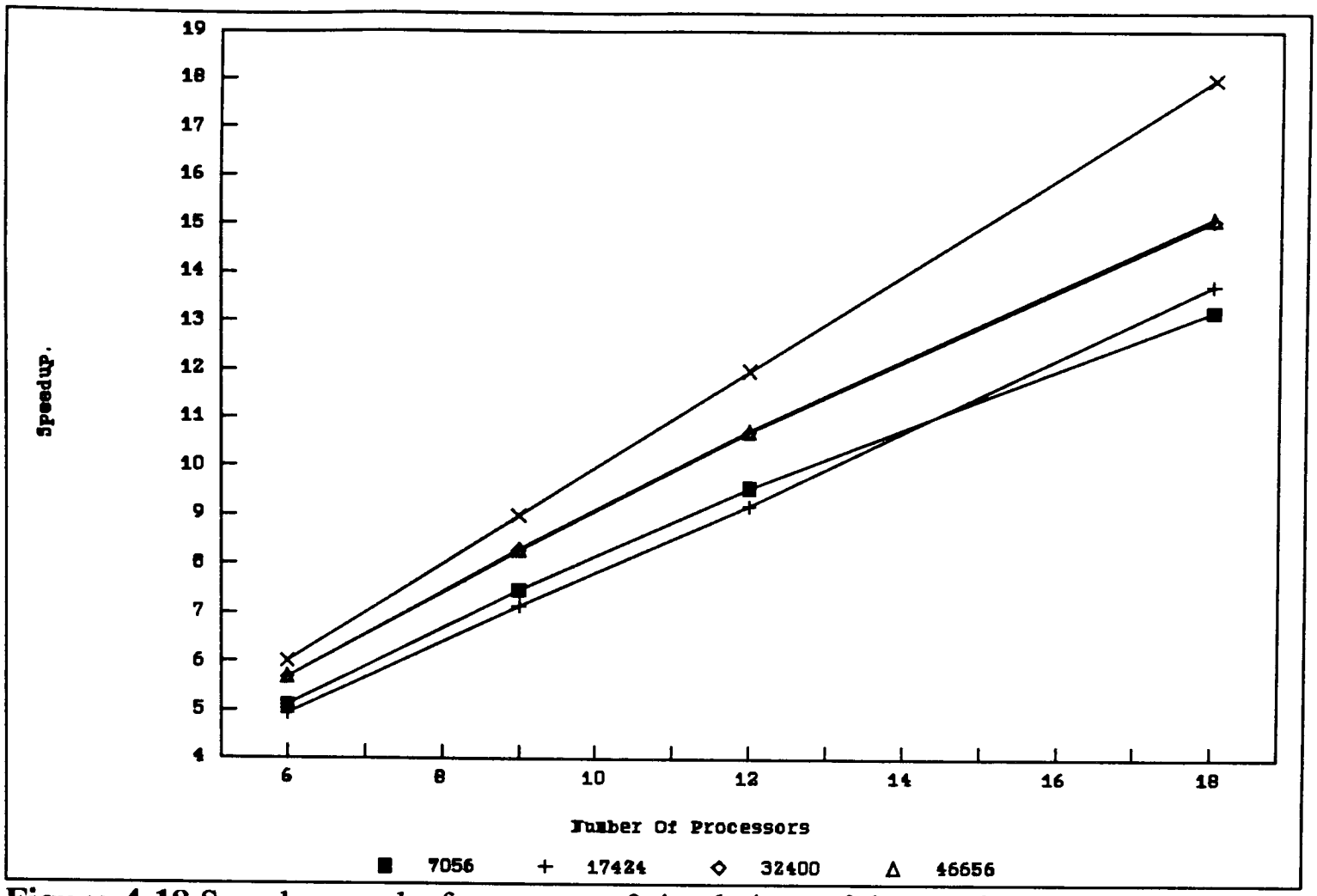


Figure 4-18 Speedup results for a range of simulations of the moving lid cavity problem using CG for the pressure solution and SIP for the solution of all other variables.

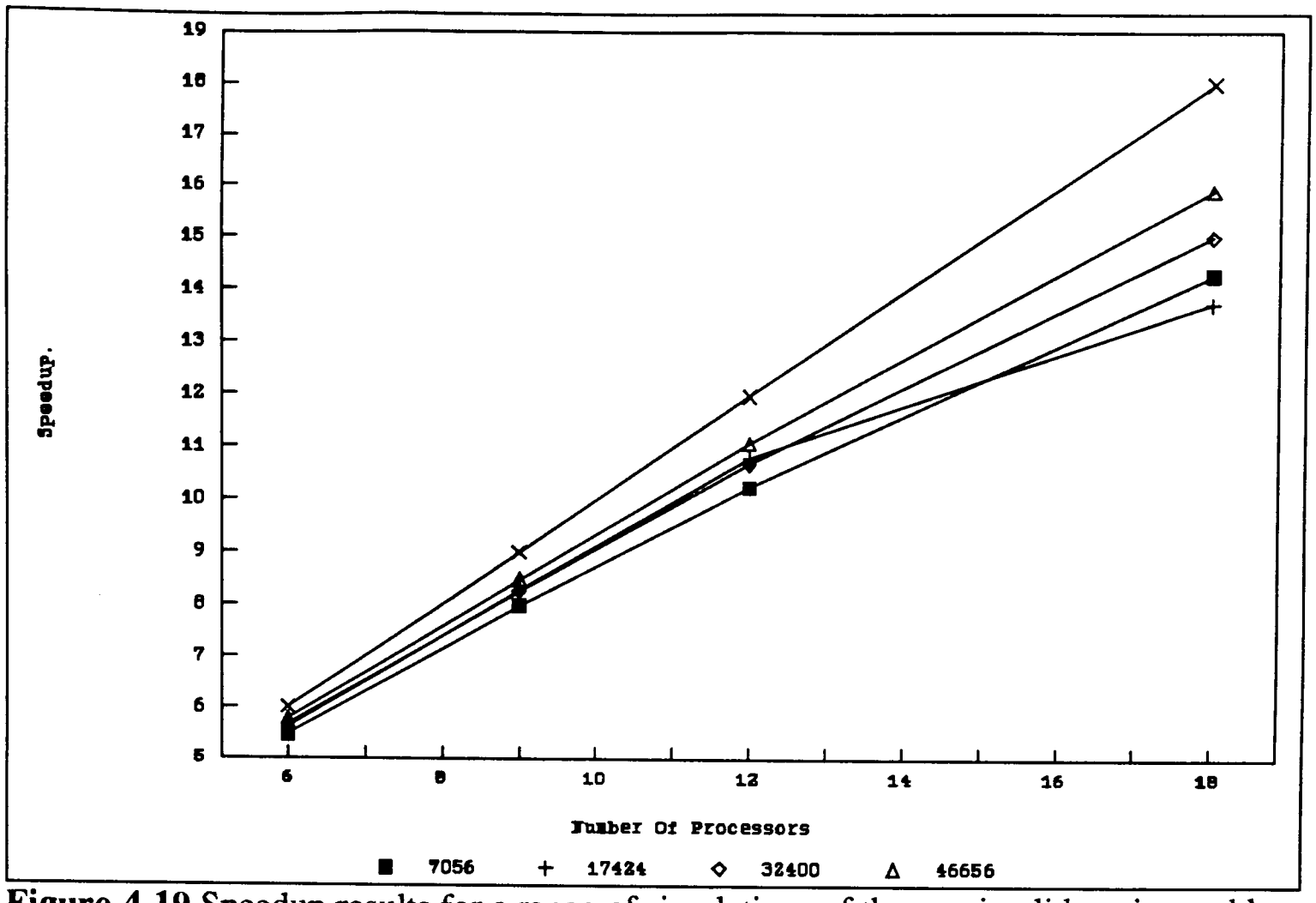


Figure 4-19 Speedup results for a range of simulations of the moving lid cavity problem using line solver for the solution to all variables.

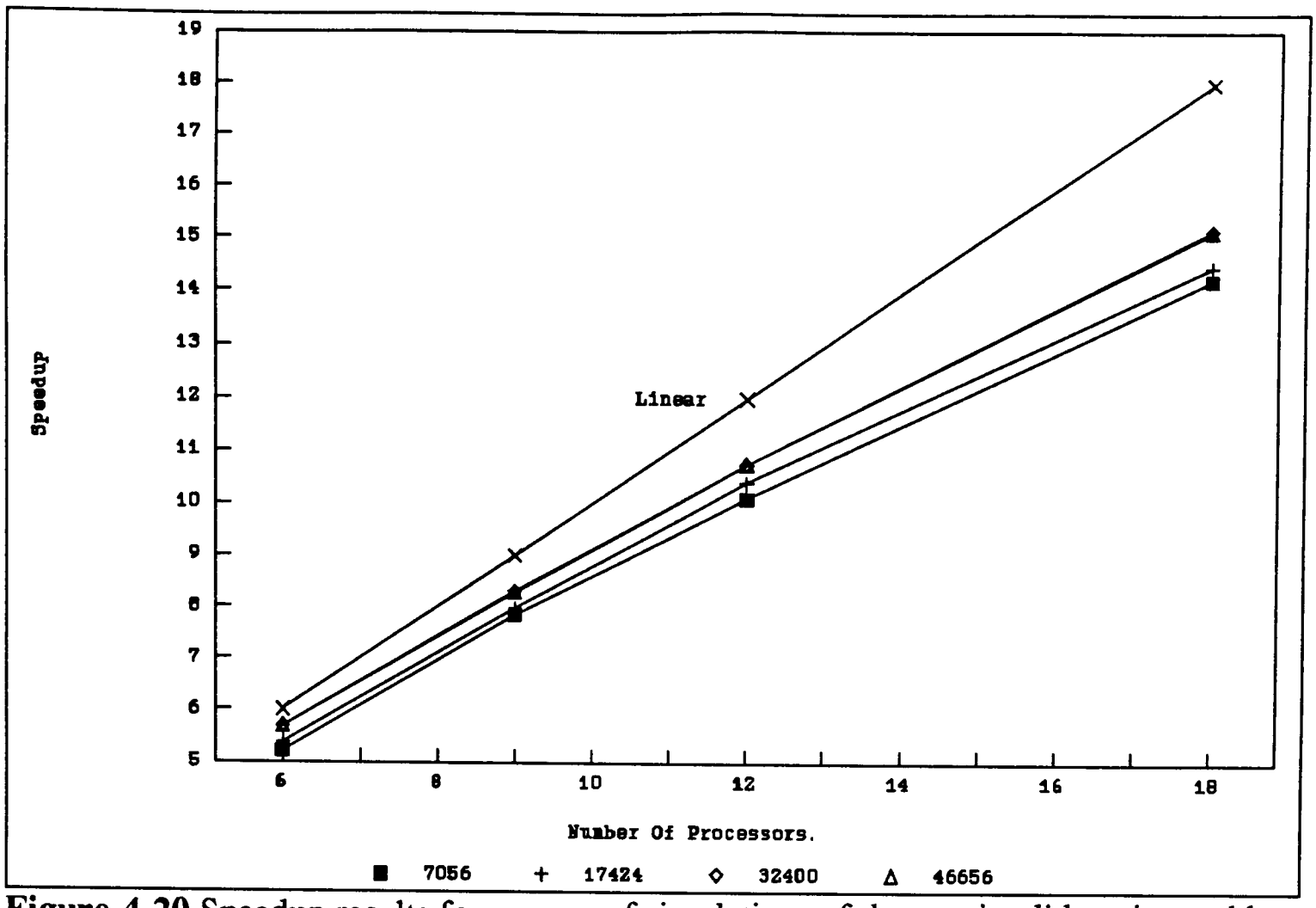


Figure 4-20 Speedup results for a range of simulations of the moving lid cavity problem using CG for the pressure solution and line solver for the solution to all other variables.

produced increase since, similar to the backward step problem, the amount of idle time and inter-processor communication time significantly reduce proportionately to total time.

When the line relaxation solver is used for all solutions, the speedups produced are an improvement on both the speedups produced using Stone's implicit procedure. A speed up 16 on 18 processors is achieved for the highest density grid (i.e. 88% efficiency for 46656 cell grid). When the conjugate gradient solver is used for the pressure correction solution, the speedups produced are slightly reduced. This is explained by figure 4.21 which shows the percentage increase in iterations required in parallel over serial for the lowest density grid. Although the line only solution requires a higher percentage increase in iterations, these iterations are relatively computationally cheap. The increase in conjugate gradient iterations, caused by using the line relaxation solver for velocity solutions, although percentagely lower than the line solver only increase, each iteration is computationally expensive as compared to line solver iterations, therefore the resultant speedup is degraded.

Again, as with the backward step problem, the serial times were estimated for the higher grid densities, assuming linear efficiency degradation as shown in the efficiency graph of figure 4.22.

4.3.3 Comparison Of Test Problem Performance.

The two test problems indicated different relative performance of the solution method combinations. The solutions gained by employing Stone's implicit procedure for the backward step problem were superior in speedup to those that used line relaxation based solutions.

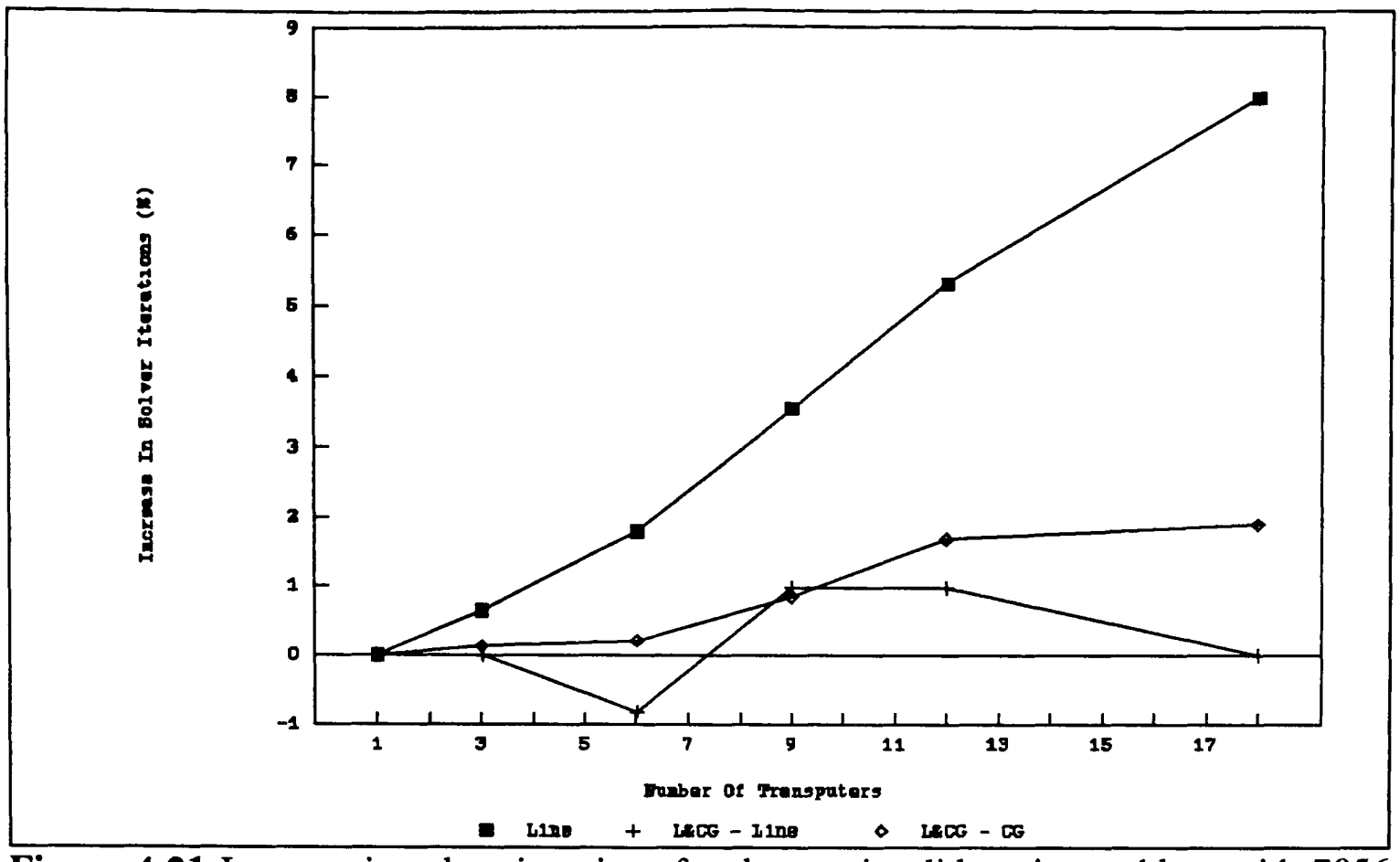


Figure 4-21 Increase in solver iterations for the moving lid cavity problem with 7056 control cells

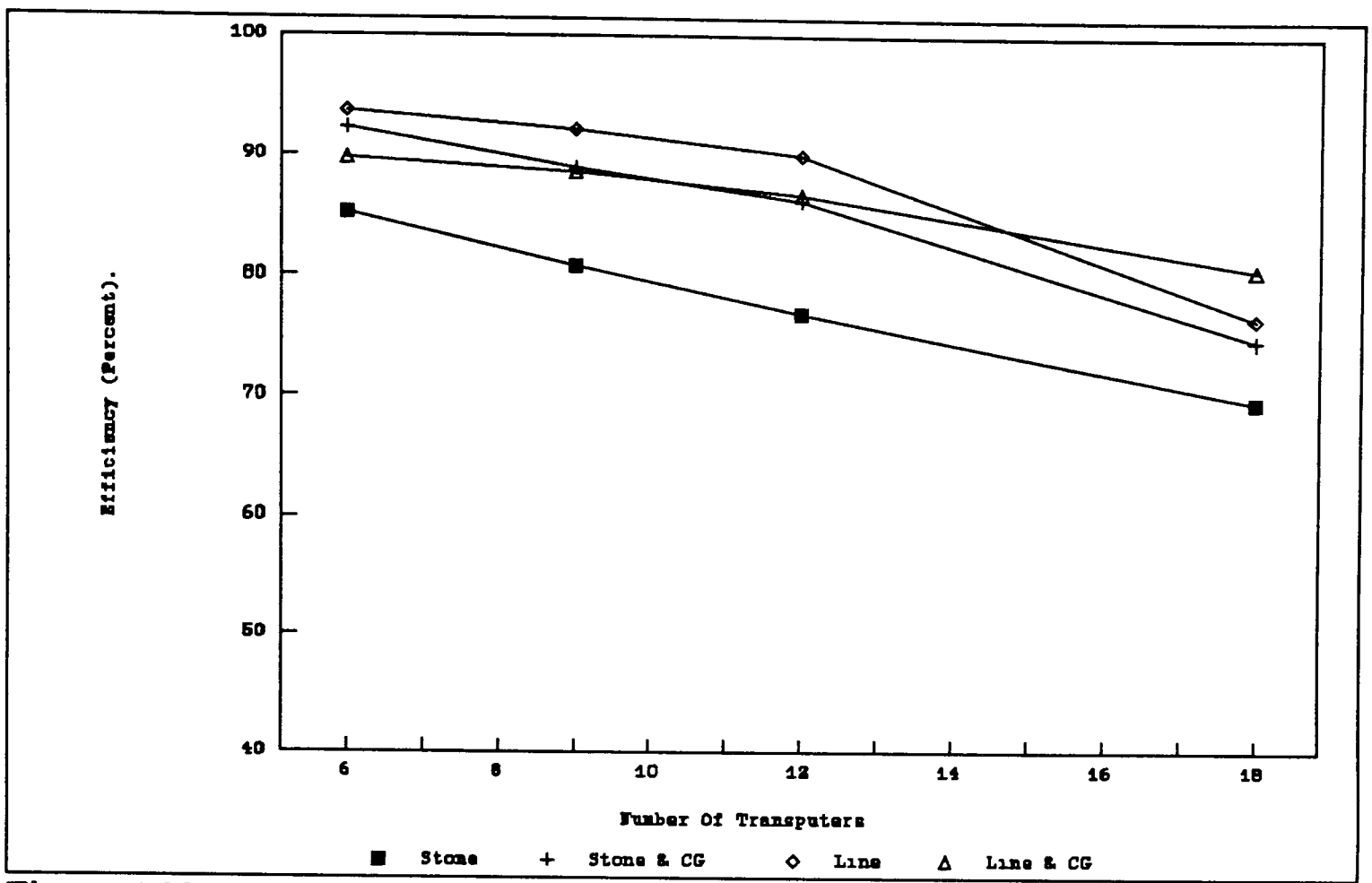


Figure 4-22 Efficiency of the parallel version for the moving lid cavity problem with 17424 control cells.

Conversely, the line relaxation based solutions were superior for the moving lid cavity problem.

The relatively poor performance of the line relaxation solver in the backward step problem is due to the parabolic (i.e. predominantly uni-directional) nature of the problem. The serial version, using S.O.R., propagates information from inlet to outlet in a single iteration whilst the parallel version will require a number of iterations equal to the number of processors before the influence of the inlet is directly felt at the outlet. The disjoint nature of the parallel version, using old values as the 'top' values for the first slab on all but the first processor, causes this phenomena. In the moving lid cavity problem, the flow is essentially elliptic, therefore the flow has no predominant direction to be exploited by line sweeps in serial. As a result, the increase in iterations required, and thus the speed up produced, are not so greatly affected.

The Stone's implicit procedure requires a cell by cell inter-processor communication of an entire slab of cells between each adjacent pair of processors. In the backward step problem, since the geometry is basically a pipe flow, the slabs consisted of cross sections of the pipe and thus contain a relatively small proportion of the total number of cells. The moving lid cavity problem, however, is a geometric cube and thus the slabs will contain a higher proportion of the total cells than in the backward step problem. This higher proportion of cells clearly requires a significant amount of extra inter-processor communication for iterations of the solver in the moving lid cavity problem than the backward step problem for the same number of control cells. This extra overhead therefore degrades the speedup produced.

4.3.4 Investigation Of Loss Of Speedup.

The performance degradation experienced in all cases is caused by several important factors :-

1. the communication overhead inherent in any distributed memory program
2. imbalance in the computational loads between processors
3. any code duplicated on several processors (including the evaluation of mask functions etc.)
4. any increase in work required in the parallel version such as an increase in iterations

The influence of these factors is examined for the backward step problem when the Stone's implicit procedure is used for all solutions. Clearly, no increase in iteration numbers is experienced using this solver. The communication and idle time per processor are combined since idle time will be experienced on waiting for at a receive or send statement on a processor. Figure 4.23 shows the time per processor spent performing communication or waiting on a communication statement (i.e. idle time), along with an estimate of time spent in duplicated code for a range of processor numbers. The duplicated code execution time is estimated from the remainder of execution time used after subtracting the serial execution time and the communications and idle time. The amount of time required per processor for both values remains fairly consistent. Since these times are spent concurrently on every processor, the additional time added to parallel execution time is fairly constant as processor numbers increase. Figure 4.24 shows the percentage the communications and idle time represent of the overall parallel execution time. This percentage increases as processor numbers increase. This is clearly due to the reduction in computation time (all other factors

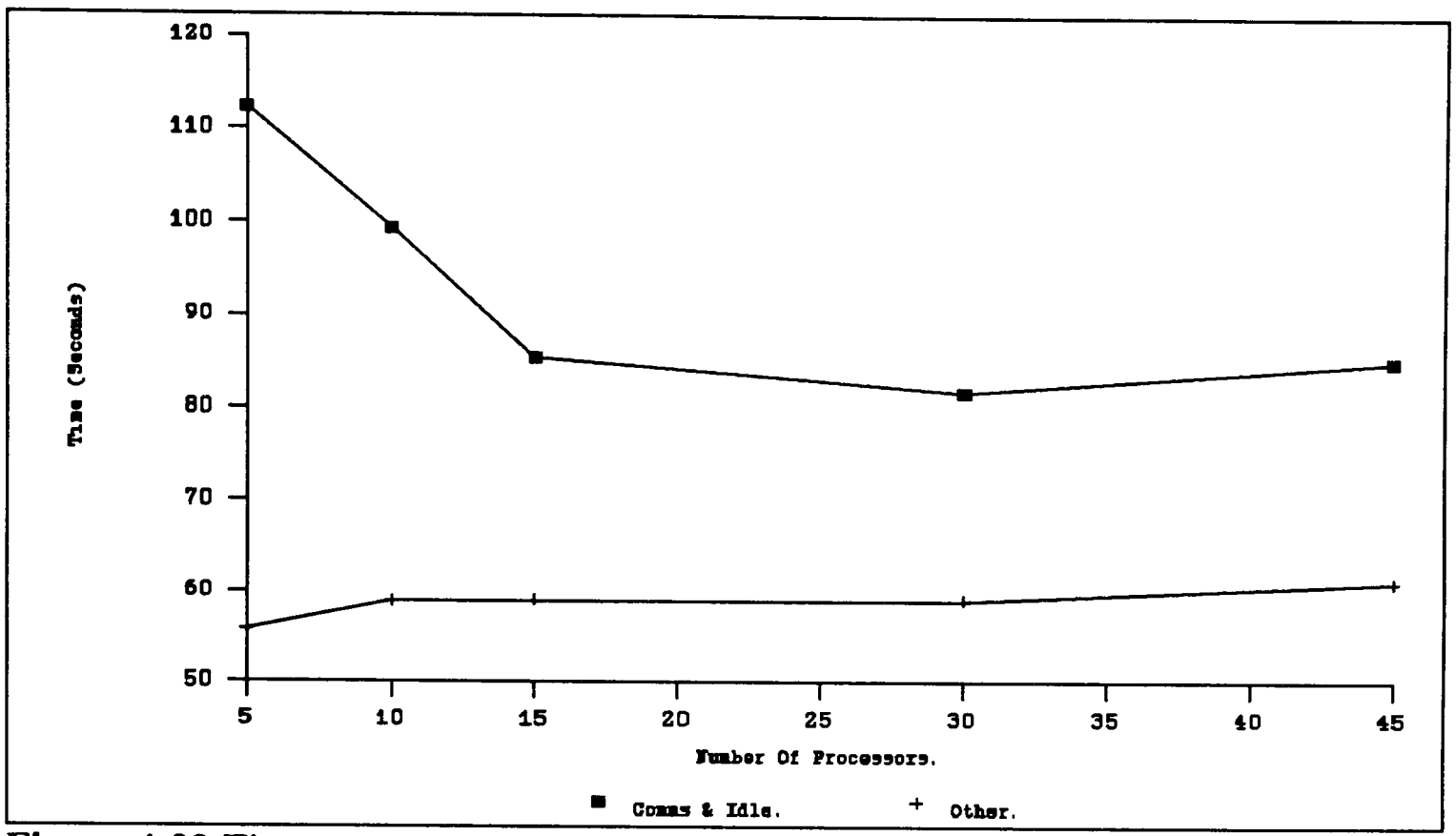


Figure 4-23 Time per processor spent in overheads for the B step problem with 17640 control cells using SIP for all variable solutions.

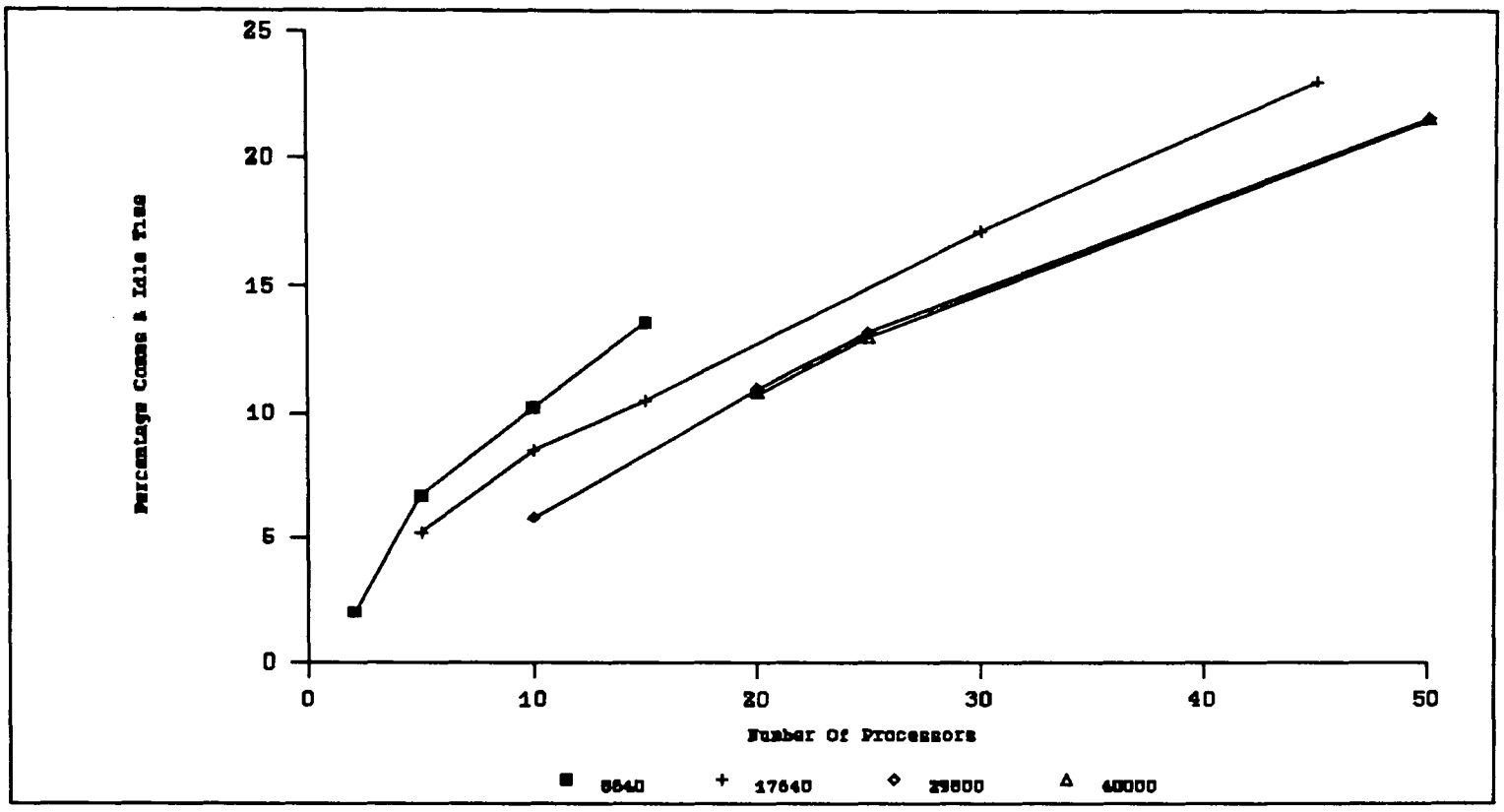


Figure 4-24 Percentage of overall time spent performing communications or awaiting communication.

remain fairly constant).

This observation represents a minimum limit to the parallel execution time regardless of the number of processors used. This limit must also include the solution time for a single slab of control cells since each processor must contain a minimum of one slab.

Clearly, to minimise this limit and thus the communication overhead, the dimensions that produce the minimum number of cells in a slab should be set as the x and y directions. This also aids the scalabillity of the parallel systems that can be used since the dimension with the maximum number of control cells will be the z direction, allowing the maximum number of processors to be used.

4.4 Closure.

This chapter has demonstrated the effectiveness of the data partition technique on a commercial, broad ranging code. The ease of use and encouraging results achieved show that, from the user perspective, the parallelisation was successful. Similarly, from the code authors point of view, the similarity of the serial and parallel codes enables simple extension of maintenance to the parallel version.

The opportunities for the exploitation of distributed memory architectures for a whole class of numerical software packages have been clearly shown. The conversion process to construct the parallel versions is, however, an enormous, time consuming task.

Much of the conversion process involves analyzing the serial code to determine where parallelism exists. Similarly, the vast majority of the code alteration and restructuring performed, involved simple adjustments or standard transformations. As such, the potential for software tools to analyze the serial code for parallelism, test the validity of code alterations etc. and perform semantically correct transformations is clear. The following chapters pursue this theme with a view to devising effective, useable software tools.

CHAPTER FIVE

5. Background To Parallelising Tools.

5.1 Introduction.

The task of software development for serial computers is often a difficult, time consuming process. Software development for parallel machines, particularly MIMD machines, adds an extra dimension to the complexity of the task. Since the amount and quality of software currently available for parallel machines is extremely restrictive and new improved hardware is continually being developed, techniques to aid, improve quality and speed of development of parallel software are urgently required.

The previous chapters have shown how using an efficient, correct serial code as a starting point can greatly aid parallel code development. An initial investigation of the serial code indicated the suitability of the code for parallel conversion, with unsuitable algorithms being replaced or restructured prior to the actual parallelisation. Creating parallel codes from the serial code required a large amount of fairly mechanical code alterations with many of the optimisations used to enhance parallelism being of a standard form. In both these cases, there are great possibilities for software tools to perform and/or aid the majority of the work.

Most of the effort put into software tools to aid development of parallel code has been aimed at parallelising compilers. These compilers take an input serial program, perform an analysis of the code and use that analysis to generate object code for the target architecture. Many of the internal decisions made during these processes are conservative assumptions since the creating of a semantically equivalent parallel code is the overriding objective. These

assumptions mean that much potential parallelism exhibited in the code is not detected. Similarly, code generation often uses heuristic decisions when determining how to generate code to best use the available hardware. These decisions can lead to potential parallelism not being exploited. The inadequacy of the 'black box' compiler is partly due to restraint on compiler execution time. An indepth examination of the serial code would require prohibitively large execution times, reducing the attractiveness of the compiler to the commercial market.

The popularity of the compiler approach to creating parallel codes is largely due to the success of early compilers to adapt code to exploit SIMD architectures - known as vectorising compilers. To produce vectorised code, a relatively simple analysis of the inner loops of the code, followed by a replacement of single array assignment statements and the inner loop surrounding them with a vector statement, was sufficient to achieve often impressive results.

For MIMD machines a more accurate analysis and more complex code generation algorithms are required. The conservative assumptions that are made in the analysis phase have a dramatically detrimental effect since parallelism in outer loops can best be exploited, where these loops often contain many statements increasing the risk of an assumption forcing the serialisation of the loop. The code generation phase is not as well defined as the simpler case of vectorising compilers, many algorithms have been suggested, all of which rely on heuristic decisions and which cannot guarantee optimal parallel code.

An alternative to the compiler route for developing parallel code is to use the same techniques as are embedded in the compilers, but allow user participation in the process. This can be

done in two ways :-

1. By allowing the user to decide if conservative assumptions must be made in the analysis phase and by accepting user guidance on the heuristic decisions made during code generation in an otherwise automated process.

2. By providing the phases of the compiler as autonomous tools where the results of each phase are relayed to the user who then continues the parallelisation process manually.

These approaches allow far more efficient code to be produced due to the invaluable knowledge of the user, however they are generally commercially less attractive than black box compilers since they require a degree of user understanding, if not expertise.

User interaction in the automated process approach can be achieved by the addition of compiler directives in the source code instructing the compiler to produce parallel code sections. Similarly, the addition of comments by the compiler about conservative assumptions made, to an output source version of the code can indicate what information the compiler requires to determine more accurately the potential parallelism of a code section. This approach has been used in many commercial parallelising compilers such as the CRAY parallelising compiler.

Another method to introduce user interaction is to add an interactive question and answer phase to the process where the compiler can ask the user questions aimed at removing assumptions and the user can inquire as to why certain code sections were assumed inherently

serial, forcing the code to be parallel if false assumptions were made (i.e PTOOL parallelisation environment, Allen et al. 1986).

Several tools to aid manual parallelisation exist including FORGE and MIMDIZER from Pacific Sierra Research and the SUPERB tool (Zima et al. 1988) developed as part of the SUPRENUM project. These tools perform some of the components of the parallelisation process but rely heavily on the user to make all major decisions.

The remainder of this chapter describes the basic serial code analysis of the parallelising systems and discusses the code generation and code optimisation phases. The analysis is based on work by Kuck and his group at the University of Illinois (Kuck et al. 1980) and extensions to this work by Kennedy and co workers at Rice University (Allen and Kennedy 1987).

5.2 The Dependence Graph.

The analysis performed on the serial code by the parallelising compilers and tools is based on the identification of dependencies.

Definition 5.1

A dependence between two statements implies a definite order in which they must be executed to maintain semantic equivalence of the initial code and any subsequent output code.

Definition 5.2

A dependence graph is a graph where each node is a statement and each edge is a dependence

relation between two statements. Every edge has a single direction indicating that the source of the dependence must be executed before the sink.

The graph thus displays all the potential parallelism (at the statement level) in the code since any statement execution ordering, parallel or not, can be generated and is valid as long as it does not violate any edges in the dependence graph. The choice of a statement level graph (as opposed to routine level, operation level etc.) was made to allow the graph to display sufficient parallelism (routine level parallelism being insufficient), whilst not requiring excessive amounts of computer memory to store the graph (as may be required by operator level graphs). Most workers in this field concentrate on statement level dependence graphs.

Much of the following is concerned with the construction of such a dependence graph.

5.2.1 Types Of Dependence.

A dependence is any relationship between two statements that implies an ordering whether or not that ordering is caused by actual data flow, control flow or is just due to the way the input code uses variables. Four different types of dependence exist (Kuck 1978) :-

Definition 5.3

1. True dependence - A statement assigns a value to a variable that is used in another statement which is reachable from this statement. Thus the second statement cannot execute until the first statement has completed execution.

S_1 A = ...

S_2 = ... A ...

Thus $S_1 \delta S_2$

2. Control dependence - The execution of a statement is controlled by the result of a previous statement. i.e if the first statement is an IF statement and the second statement is in the true or false block of that IF. Thus the second statement cannot execute until the conditional statement has completed.

S_1 IF Condition THEN

S_2 .....

ENDIF

Thus $S_1 \delta^c S_2$

3. Anti dependence - A use of a variable occurs before a subsequent redefinition of the same variable. Thus the use of the variable in the first statement must be performed before the required value is lost in the second statement.

S_1 = ... A ....

S_2 A = ...

Thus $S_1 \bar{\delta} S_2$

4. Output dependence - Two statements assign to the same variable, thus the second definition must follow the first.

S_1 A= .....

S_2 A= .....

Thus $S_1 \delta^o S_2$

Anti and output dependencies are known as pseudo dependencies since they do not involve the flow of information but merely a reuse of a memory location, often used to reduce memory requirements.

5.2.2 Depth Of Dependence.

An important source of parallelism in software is that between iterations of loops (i.e the possibility of performing every iteration of the loop simultaneously). In a nest of loops it is often possible to perform several of the loops in parallel with great potential gains in speedup. To enable detection of parallelism for a particular loop in a nest, dependencies are given a depth to indicate between iterations of which loop, if any, the dependence exists.

Definition 5.4

If a dependence exists within a single iteration of all surrounding loops then the dependence is defined as loop independent (defined as at the infinity level for convenience). If the dependence exists between iterations of the outermost loop surrounding the statements involved, the dependence is defined as at level one. A dependence between iterations of the next loop in the nest (during a single iteration of the outermost loop) is defined as at level

two and so on.

In this way, if a loop in a nest is forced to be a serial loop then all dependencies between iterations of this loop (i.e. at the appropriate level) have been satisfied and can be ignored in the dependence graph. The parallelism detection and code generation algorithms shown later all make use of this feature.

5.3 Dependence Testing.

Essential to efficient parallelism detection is the construction of an accurate dependence graph. Any uncertainty about the existence of a dependence must result in a dependence being assumed to ensure a semantically correct graph. Dependencies involving scalar variables are always detectable by reference only, however arrays represent a more difficult problem. Early parallelising systems assumed that any reference to an element of an array was a reference to the entire array as far as the dependence testing was concerned. This very conservative assumption was later replaced by using the tests outlined below.

Definition 5.5

An instance of a scalar variable in the source code is defined as a loop variable or nonloop variable as follows :-

> It is a loop variable if and only if this instance of the variable is nested within a loop where the same variable is the loop iteration counter of that loop. All instances of variables not classed as loop variables are classed as nonloop variables.

5.3.1 Preliminary Code Transformations.

5.3.1.1 DO Loop Normalisation.

Before developing the dependence tests, the DO loops in the code can be normalised to run from a lower limit of one with steps of one. This is done to simplify the dependence tests which follow. A typical DO loop such as

DO I = L,U,S

thus becomes

DO I^* = 1,(U-L)/S+1,1

where any reference to the loop variable I is adjusted to reference the new variable using

$I \equiv (I^*-1)S + L$

Do loop normalisation simplifies all the following and is thus used throughout this work. Formulations of the following tests without the requirement of DO loop normalisation are given in Zima (Zima 1990).

5.3.1.2 Induction Variable Substitution.

The tests detailed in the following sections use information about the iteration counters of loops. To enhance the information available to these tests, a code transformation known as induction variable substitution, can be used.

Any assignment of a scalar variable within a loop where every iteration of the loop invokes

an instance of the statement which increments the scalar variable by a constant amount (i.e. a constant or a variable whose value is not altered within the loop concerned) can be transformed into a linear function of the loop counter variable. Consider the following example :-

```
                                    Kinitial = K
DO I = 1 , N                        DO I = 1 , N
      K = K + C                           K = Kinitial + I * C
      ....                                ....
END DO                              END DO
```

An algorithm for the identification of induction variables from source code is given in Aho et al. (1986). The method used in this work examines the scalar dependence graph to identify valid induction variables, completing the transformation before the dependence analysis of arrays is commenced (see section 6.19). Further discussion of the use of knowledge about variables that fit some but not all the requirements of strict induction variables is given in chapter 7.

5.3.2 Statement Model.

The array dependence tests consider linear functions of loop variables as the form of array indices. The following is a general example of nested statements between whom dependence calculation is required :-

L_1 DO $I_1 = 1 , U_1$

L_2 DO $I_2 = 1 , U_2$

. . . . . .

L_c DO $I_c = 1 , U_c$

L_{c+1} DO $I_{c+1} = 1 , U_{c+1}$

. . . . . .

L_m DO $I_m = 1 , U_m$

S_1 $A(f(I_1,I_2,..,I_c,I_{c+1},..,I_m)) = ...$

END DO

. . . . . .

END DO

L'_{c+1} DO $I'_{c+1} = 1 , U'_{c+1}$

. . . . . .

L'_n DO $I'_n = 1 , U'_n$

S_2 $..=..... A(g(I_1,I_2,..I_c,I'_{c+1},..,I'_n)) ....$

END DO

END DO

END DO

END DO

END DO

Where L_k indicates a loop at level k

I_k indicates the loop variable for the loop at level k

U_k indicates the upper limit of the iterations of the loop at level k

and the iteration lower limit and inter iteration step are set to one by DO loop normalisation

Statements S_1 and S_2 are both nested in loops L_1 to L_c. S_1 is additionally nested in loops L_{c+1} to L_m and S_2 is additionally nested in loops L'_{c+1} to L'_n

Now consider two instances of statements S_1 and S_2. Let x_1 to x_m be values of the loop variables surrounding statement S_1. Let y_1 to y_n be values of the loop variables surrounding statement S_2. The linear functions f and g are defined as :-

$$f(x_1,x_2,...x_m) = a_0 + a_1x_1 + a_2x_2 + ... + a_mx_m$$

$$g(y_1,y_2,...y_n) = b_0 + b_1y_1 + b_2y_2 + ... + b_ny_n$$

Where a_0 to a_m and b_0 to b_n are integer constants.

Dependencies between S_1 and S_2 can exist at the infinity level as well as between iterations of the loops L_1 to L_c (i.e. only the loops commonly surrounding both statements).

If we are testing for a dependence of statement S_2 on statement S_1 between iterations of the k^{th} loop surrounding both statements (i.e $S_1\ \delta_k\ S_2$) then we know the following :-

$$x_i = y_i \quad i = 1(1)k\text{-}1$$

$$x_k < y_k$$

i.e. between iterations of the k^{th} loop, we are in the same instance of loops 1 to k-1 thus the values of these loop variables in f and g are the same and since we are testing for a dependence of S_2 on S_1, S_2 must be in a later iteration than S_1 therefore, with the loop

normalised, the value of this loop variable must be greater in statement S_2.

If we are testing at the infinity level (i.e. $S_1\ \delta_\infty\ S_2$), all common loops are in the same iteration i.e.

$$x_i = y_i \qquad i=1(1)c$$

A dependence of statement S_2 on statement S_1 thus exists if any instance of $(x_1,x_2,....,x_m)$ and $(y_1,y_2,....,y_n)$ exists given the above constraints where the functions f and g are equal.

The equation we require to solve is thus

For a level k dependence :-

$$f = g \text{ --> } (a_1 - b_1)x_1 + ... + (a_{k-1} - b_{k-1})x_{k-1} + a_k x_k .... + a_m x_m - b_k y_k .... - b_n y_n = b_0 - a_0$$

For an infinity level dependence :-

$$f = g \text{ --> } (a_1 - b_1)x_1 + ... + (a_c - b_c)x_c + a_{c+1}x_{c+1} .... + a_m x_m - b_{c+1}y_{c+1} .... - b_n y_n = b_0 - a_0$$

5.3.3 The Greatest Common Divisor Test.

The equations for determining dependence are linear diophantine equations. Such an equation has a solution if, and only if, the constant term is divisible by the greatest common divisor of the coefficients of the variable terms (Griffin 1954). i.e.

For a level k dependence :-

$$gcd(a_1 - b_1\ ,\ ....\ ,\ a_{k-1} - b_{k-1}\ ,\ a_k\ ,\ ....\ ,\ a_m\ ,\ b_k\ ,\ ....\ ,\ b_n)\ |\ b_0 - a_0$$

For an infinity level dependence :-

$$gcd(a_1 - b_1\ ,\ ....\ ,\ a_c - b_c,\ a_{c+1}\ ,\ ....\ ,\ a_m\ ,\ b_{c+1}\ ,\ ....\ ,\ b_n)\ |\ b_0 - a_0$$

If the gcd test proves false then f - g can never equal zero and the statements are definitely independent. If the gcd test proves true then a dependence could exist but a further test is used to again try to disprove a dependence.

5.3.4 Banerjee's Inequality.

A solution to the f - g equation only exists if a solution can be found given the constraints indicated earlier and the upper limits of the normalised loops. Let the normalised upper limit of the i^{th} loop surrounding S_1 be M_i and the upper limit of the i^{th} loop surrounding S_2 be N_i. Thus since the first c surrounding loops are common to both statements

$$M_i = N_i \qquad i=1(1)c$$

By calculating the maximum and minimum values of every term in the equation by examining the signs of coefficients, the following inequality due to Banerjee (as described in Allen and Kennedy 1987) can be derived :-

For level k dependencies

$$-b_k - \sum_{i=1}^{k-1} [(a_i - b_i)^-(M_i - 1)] - (a_k^- + b_k)^+(M_k - 2) - \sum_{i=k+1}^{m} [a_i^-(M_i - 1)] - \sum_{i=k+1}^{n} [b_i^+(N_i - 1)]$$

$$<= \sum_{i=0}^{n} b_i - \sum_{i=0}^{m} a_i <=$$

$$-b_k + \sum_{i=1}^{k-1} [(a_i - b_i)^+(M_i - 1)] + (a_k^+ - b_k)^+(M_k - 2) + \sum_{i=k+1}^{m} [a_i^+(M_i - 1)] + \sum_{i=k+1}^{n} [b_i^-(N_i - 1)]$$

For infinity level dependencies :-

$$-\sum_{i=1}^{c}[(a_i - b_i)^-(M_i - 1)] - \sum_{i=c+1}^{m}[a_i^-(M_i - 1)] - \sum_{i=c+1}^{n}[b_i^+(N_i - 1)]$$

$$<= \sum_{i=0}^{n} b_i - \sum_{i=0}^{m} a_i <=$$

$$+\sum_{i=1}^{c}[(a_i - b_i)^+(M_i - 1)] + \sum_{i=c+1}^{m}[a_i^+(M_i - 1)] + \sum_{i=c+1}^{n}[b_i^-(N_i - 1)]$$

Where $a^+ \begin{cases} a & a >= 0 \\ 0 & \text{otherwise} \end{cases}$

$a^- \begin{cases} -a & a <= 0 \\ 0 & \text{otherwise} \end{cases}$

If either side of the inequality concerned can be proved false then no f - g solution exists in the required constraints and the statements are independent, otherwise a dependence exists or must be assumed.

5.3.5 Multi - Dimensional Arrays.

Now consider the array involved in the dependence as a multi-dimensional array i.e.

S_1 $A(f_1,f_2,...,f_N)$ = ...

S_2 .... = ... $A(g_1,g_2,....,g_N)$ ....

Where f_i and g_i are linear functions of the surrounding loops loop variables.

For a dependence to exist, equality of every pair of indices must be possible i.e.

$$f_i = g_i \qquad i=1(1)N$$

Thus the GCD and Banerjee tests can be used for each pair of indices. Any result indicating no solution possible for a pair of indices indicates that an overall dependence between the two statements does not exist and thus they are independent.

5.4 Code Generation Strategies.

Once a complete dependence graph has been constructed, it can be used to detect and generate parallel code for the target machine. The discussion shall concentrate on loop level parallelism since this is the most profitable source of parallelism available. Other parallelism is exhibited by the dependence graph and other approaches (such as critical path analysis) can be used in these cases.

The different machine types require different code generation strategies. The basic test for a parallel loop, common to all machine types, is whether a dependence carried by this loop exists. If such a dependence exists, a definite order of iteration execution exists, forcing serial execution of the loop. The following sections give a brief overview of code generation

strategies that can be used.

5.4.1 Vectorised Code Generation.

Vectorised code is code generated for SIMD machines. The target languages are the standard languages with the addition of vector operation statements which may be calls to low level routines (as used by MASSCOMP), or language extensions as in the specification for FORTRAN 8x.

Vector code is generated through single statements being performed in parallel in their innermost surrounding loop. The vector code generation algorithm shown, due to Allen and Kennedy (Allen and Kennedy 1987), uses loop distribution to attempt to split the statements in the innermost loop into separate statements, each with an individual copy of the loop. This transforms any dependencies carried by this loop into loop independent (infinity level) dependencies, thus allowing parallel execution of the loop. Loop distribution for a statement is valid if the statement is not involved in a dependence cycle with other statements. Such a cycle implies that the order of execution of the statements involved requires that each instance of this statement must be followed by instances of the other statements in the cycle before the next instance of this statement can occur. Thus all statements in a cycle must be nested in a single common loop to enforce this ordering, making loop distribution for those statements illegal.

The algorithm first examines the dependence graph, identifying strongly connected regions in the graph.

Definition 5.6

A strongly connected region is a sub graph of the original graph where every node is reachable from every other node through dependence edges. Only edges linking nodes within this strongly connected region are included.

For each strongly connected sub graph, the outermost surrounding loop (i.e. the level one loop) is generated in serial, thus allowing all dependencies marked as at level one to be ignored since they are all satisfied. This new subgraph is a level two dependence subgraph which can then, in turn, be examined for strongly connected regions with the process continuing in this recursive fashion.

Statements not involved in any strongly connected regions can legally be used with loop distribution on all surrounding loops not yet generated. These loops can be generated in serial except the innermost loop which is executed in parallel through incorporation with the statement to form a vector statement.

The order of processing strongly connected regions at any sub graph level is determined by topological sort (Knuth 1973). Every strongly connected region forms a single node (called a pi-block) with all statements not involved in such regions being allocated an individual pi-block. The pi-blocks are connected through dependencies between statements that are now in separate pi-blocks. Clearly, since any cycle is placed in a single pi-block, all external dependencies do not form cycles, thus the pi-graph is acyclic, allowing a simple topological sort to determine processing order.

The Allen and Kennedy algorithm proceeds as follows :-

procedure CODEGEN(G,k) - G is the input dependence subgraph, k is the current level of this subgraph

Split G into strongly connected subgraphs using Tarjan's algorithm (Tarjan 1972) considering only dependencies of level k and above.

Create pi-graph by allocating each strongly connected region to a pi-block and every other statement not already in a block to an individual pi-block. Identify all inter pi-block dependencies.

Use topological sort to order the pi-blocks in the pi-graph consistent with the inter block dependencies.

For each pi-block in order do

If pi-block is not a strongly connected subgraph then

Generate serial DO loops for loops k to n-1 surrounding the statement in this pi-block (n being the total number of surrounding loops)

Generate this statement in vector form if n > k-1

Generate n-1 down to k continue statements

else

Let G' be the sub graph of statements and dependencies entirely within this pi-block

Generate level k DO loop in serial

Recursivly call CODEGEN passing in the subgraph G' and the level k+1

Generate the level k continue statement

The initial call to CODEGEN provides the complete dependence graph and inputs k as 1.

Generation of all code is guaranteed since cycles and strongly connected regions can only be caused by loops and thus must involve loop carried dependencies. Thus the serial generation of all loops will leave only loop independent dependencies which can only form an acyclic graph which can then be processed merely by the topological sort.

A demonstration of the working of the algorithm is given in Allen and Kennedy 1987.

For some SIMD machines, several nested loops can be performed in parallel, where the vector statements can accommodate code generation of this type. All the loops surrounding a statement that is not contained in a strongly connected region can legally be generated in parallel.

5.4.2 Parallel Code Generation For Shared Memory MIMD Machines.

MIMD machines can theoretically exploit all potential parallelism in a code if the required number of processors are available. For realistic systems however, not all parallelism can be exploited and much more cannot profitably be exploited. As a result, most code generation algorithms concentrate on loop level parallelism since the scalabillity to exploit large numbers of processors and the potential gains this brings make them the most attractive source of parallelism.

A code generation algorithm due to Allen, Callahan and Kennedy (Allen et al. 1987) uses a recursive approach similar to that of the vectorising algorithm of Allen and Kennedy. In the parallel code generation algorithm, code is generated for the outmost parallel loop in a nest, with all other loops generated in serial. The algorithm also generates the necessary synchronisation statements to ensure that all processors complete their assigned iterations before dependent code can start on any processor(s).

Exploitation of several parallel loops in a nest as well as parallelism between nests of loops can lead to far greater gains in performance than exploiting only a single loop at a time. Polychronopoulos (Polychronopoulos et al. 1986,Polychronopoulos and Banerjee 1987) has

developed a heuristic processor assignment algorithm which attempts to minimise processor idle time by assigning variable numbers of the available processors to each parallel loop. Unknown loop iteration limits at compile time often present a problem for such an algorithm since the number of required iterations is an essential requirement of the algorithm. Thus the algorithm will usually have to be performed during run time when all the required values are known representing a potentially large execution time overhead.

Parallel DO loops can be classified into two types, DOALL loops and DOACR loops.

A DOALL loop is a completely parallel loop with no dependencies carried by this loop as is required for the above algorithms.

A DOACR (do across) loop is a loop where iterations can overlap in execution i.e the next iteration can start after the required statements in this iteration have completed. DOACR loops do cause loop carried dependencies, however, not every statement in the loop is in a dependence cycle (i.e. strongly connected region) thus the loop is not entirely serial. DOACR loops are restrictive in that they can only efficiently use a small number of processors depending on the time proportion of each iteration of the loop spent in overlapped and non overlapped code. Inter iteration synchronisation is required to ensure correct execution and two methods have been suggested with associated algorithms. Cryton (1986) suggested using a time delay in each iteration calculated via iteration number to enforce the required synchronisation, whilst Midkiff and Padau (1986) use semaphore mechanisms via the global memory.

Certain special cases exist where the dependence graph indicates a serial loop is required although parallel code is possible. If the dependence cycle that causes the loop to appear serial involves loop carried true dependencies on a statement, where that statement is of a commutative nature, parallel code may be possible. Consider a summation of values stored in an array i.e.

```
DO I = 1,N
        SUM = SUM + A(I)
END DO
```

To evaluate SUM, any order of inclusion of the array elements is valid since the operation is commutative. Each processor involved in this calculation can assign to a separate memory location the local sum for the iterations of the I loop allocated to that processor. After all iterations have been completed the local sums can be combined into a global sum and stored in the program variable SUM.

Many similar circumstances exist such as minimum and maximum calculations etc. all of which can be handled in a similar way to the summation. The result of such a parallel evaluation and the original evaluation can be different due to rounding error differences of the partial sums. This may, in extreme circumstances, prove significant to the overall performance of the code. Any code that suffers from this problem must use a very sensitive algorithm which may well experience the same problem on different serial machines indicating that the algorithm used is inadequate. As such the danger of reordering commutative operations is ignored in parallel code generation.

Other techniques for preserving semantics whilst exploiting parallelism involve locking

semaphores where when the variable is accessed a flag is set to disable access by any other processor.

Polychronopoulos gives a detailed treatment to many aspects of shared memory MIMD code generation in Parallel Programming And Compilers (Polychronopoulos 1988).

5.4.3 Parallel Code Generation For Distributed Memory MIMD Machines.

Both geometric and processor farm techniques, as described in section 2.4.3, can exploit parallelism between iterations of loops. The suitability of loops for parallel execution using the processor farm technique can be determined in a similar way to that used for shared memory systems (since the controlling processor in the farm has in effect a global memory). The use of a data partition exploiting geometric parallelism, however, has significantly different rules in parallelism detection.

The code generation for distributed memory MIMD machines considered here assumes the strategy introduced in chapters 3 and 4. The data is partitioned over the local memories in the processor network with any assignment to a partitioned array element being performed on the processor to which that array element was allocated. Overlap areas for data used on a processor but not assigned are also used when required.

Many of the decisions such as which arrays to partition, how to partition those arrays and processor network configuration are at present made by the user. The following sections give

an overview of the identification of parallelism in loops for use in the above strategy assuming the partition etc. has been performed by the user.

Since the memory is distributed, parallelism detection for code executing on different local memories and code ordering for execution on a single processor with one memory use different rules.

Code ordering (see optimisations in section 5.5) concerns a semantically correct code order on a single processor and as such all dependence types must be considered.

In parallelism detection, since the array elements have a unique serial assigning processor, the source and sink of any output dependence will refer to the same memory location on that processor. As such, an output dependence is irrelevant to parallelism detection. The relevance of anti dependencies for parallelism detection is influenced by the nature of variable involved in that dependence. Variables can be classified into two classes to show their influence on the relevance of anti dependencies in parallelism detection :-

Firstly 'pre-fetched' arrays in a parallel loop are arrays where all values of that array that are required from other processors are received before the loop starts execution. Since the required values from other processors are stored in a buffer on this processor, any subsequent redefinition of those variables on their assigning processor in the loop will not alter the value used on this processor. Thus for this type of variable, loop carried anti-dependence is irrelevant. The examples in chapters 3 and 4 all used 'pre-fetch' arrays except in the Stone's Implicit Procedure section. Pre-fetching is valid if the variable is not involved in a loop

carried true dependence, but is only practical for organised data access patterns or when repeating the index calculation used in the loop in the pre-fetch phase represents a relatively small overhead.

The second array type is 'in-fetched' arrays. These receive the values required from other processors when actually used within the loop. Therefore the data is sent from the assigning processor after a possible redefinition has occurred, causing anti-dependencies to be important in parallelism detection.

The requirement of a communication is indicated by a true dependence in the dependence graph. However the location of the required data is often not determinable at compile time since the partition will usually be dynamically performed at run time. To receive the required data thus uses run time utilities, one to determine where the data lies and the other to initiate the necessary communications to receive the data if it is assigned to another processor. The communications used here will often have to be between distant processors, therefore a communications harness utility will be required on every processor to perform the required through routing.

A data partition is not prohibited by serial or DOACR loops. These loops will be serialised by the addition of communication calls within iterations. The serial loop in the Stone Implicit Procedure section in chapter 4 uses a serial loop nested within parallel loops. The serial loop is transferred to be the innermost loop and then run as a serial pipeline, iterations of the loop being performed on each transputer in turn, initiated by the sending of the required data.

As with shared memory machines, commutative operations can be performed in parallel. The variable causing the serialising dependencies will be local to each processor performing the loop iterations, thus all that is required is for these local values to be combined in an appropriate way and the result broadcast to all processors that require the global value. This was used extensively in the examples of chapters 3 and 4, particularly in summations and maximum value calculation used to evaluate global residual values.

With the current state of the art, the code generation solution used in this work is to present all available information to the user from the analysis phase, including all parallel loops. The user then determines which parallelism can most effectively be exploited and manually develops a parallel code with tools to aid the more mechanical operations in such a conversion process.

5.5 Optimisations To Increase Parallelism.

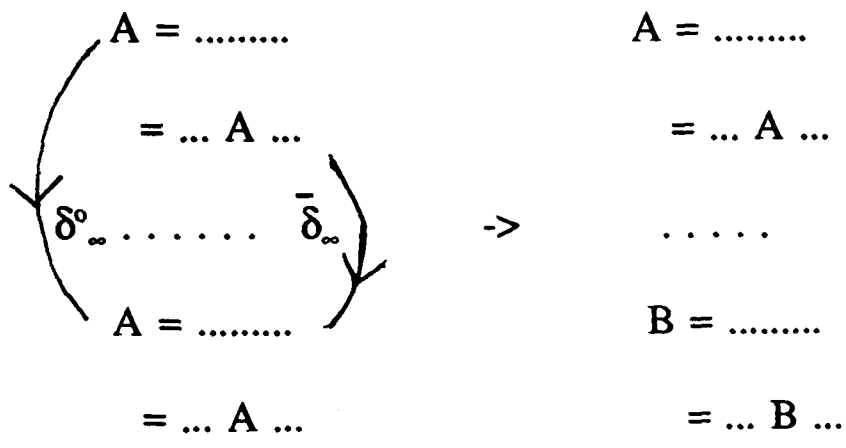
Many optimisations are used to improve execution time of the compiled software even in serial compilers. Many of these optimisers use a crude dependence analysis to recognise legal and beneficial optimisations. Similarly with parallelising compilers and tools, optimisations have been developed which aim to extract as much parallelism as possible from a serial code, converting it into a form where it can be exploited. Some of these optimisations uncover a small amount of extra parallelism whilst others are essential to code generation of efficient parallel code.

The following sections describe some useful optimisations.

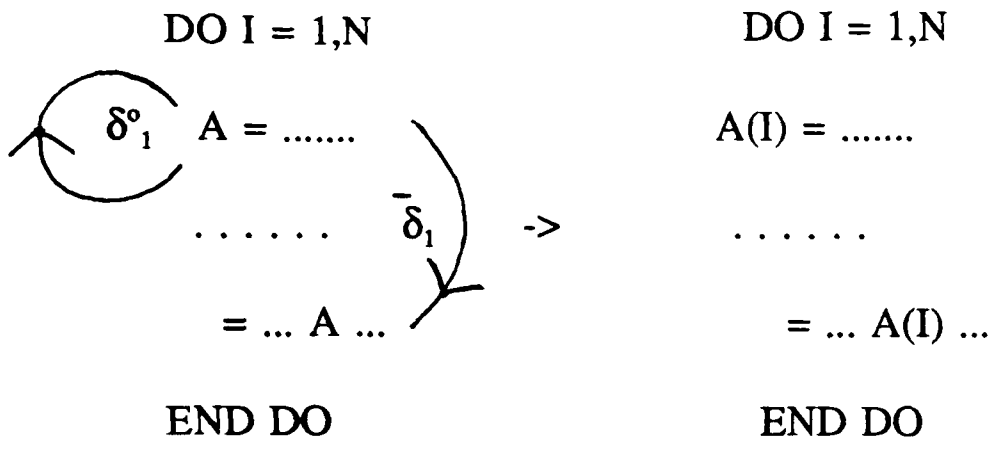
5.5.1 Elimination Of Pseudo Dependencies.

The first set of optimisations increase parallelism by removing pseudo dependencies (Kuck et al. 1981) which, as mentioned earlier, are often caused as a side effect of memory conservation. As a result all these optimisations incur an increase in memory requirements as a cost of increased parallelism.

Renaming variables can remove pseudo dependencies when the same variable is used in two separate code sections with no direct true dependence link. i.e.

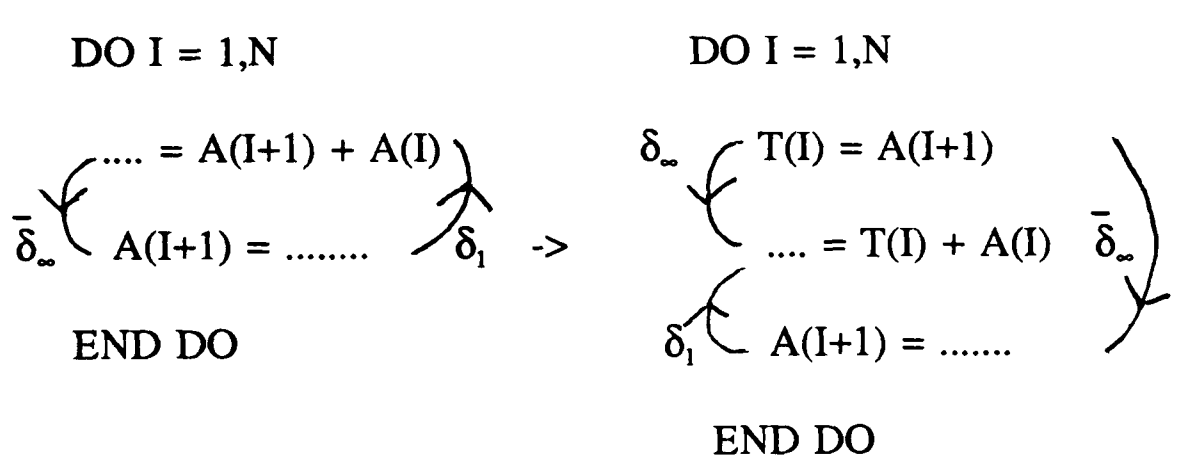


Scalar expansion transforms a scalar variable used in a loop into an array with the loop variable as the array index. This is valid if no loop carried true dependence involving the scalar variable exists. The loop carried pseudo dependencies caused by this scalar are removed by this transformation. Scalar expansion is very useful in vectorisation or shared memory machines.



The same technique can also be applied to arrays, increasing the dimension of the array.

Node splitting stores values of a particular variable in a temporary array, using the temporary array instead of the original variable in usages in the loop. This moves any anti dependence that may have existed due to use and redefinition of this variable, placing the anti dependence on the newly introduced statement. This is particularly useful when the original anti dependence is a crucial link in a dependence cycle. The use of pre fetch array in distributed memory has the same effect as this optimisation.

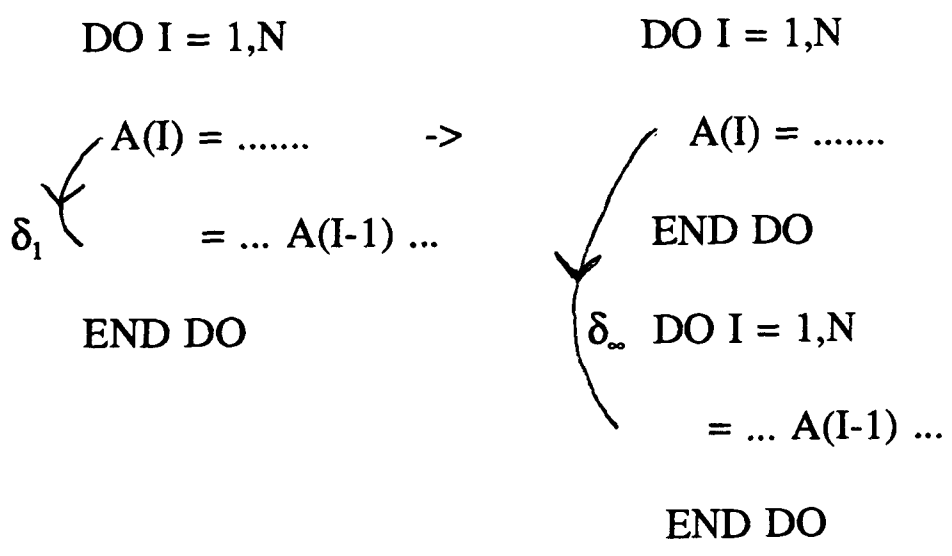


Using this transformation, the dependence cycle in the original code is broken.

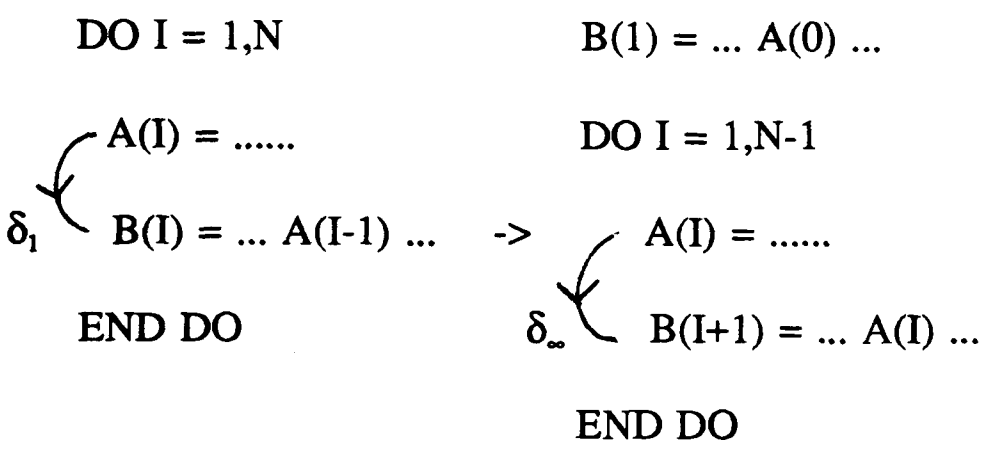
5.5.2 Converting Loop Carried Dependencies Into Loop Independent Dependencies.

Loop carried dependencies usually serialise a loop, thus if they can be converted to be loop independent (i.e. at the infinity level) then the loop(s) in the converted version can be executed in parallel.

Loop splitting (Kuck et al. 1981) transforms a single loop into two separate loops. If a dependence carried by the original loop exists between statements which are in separate loops after loop splitting then the dependence must now be at the infinity level. Loop splitting is valid if the two sets of statements which are split have no strongly connected components (see definition 5.6) of the dependence graph which contain statements from both groups. If no such component of the graph exists then the two new loops are ordered depending on the acyclic dependencies that join them, if any such dependencies exist. This optimisation is invaluable to code generation for all machine types and forms the basis for the vector code generation algorithm in section 5.4.1 (the loop distribution which is essential to the algorithm is an extreme case of loop splitting).



Loop alignment (Zima 1990) changes the indices of arrays in a statement, ensuring an evaluation consistent with the original, but performs the evaluations in offset iterations to the original. This subtle adjustment can thus adjust a dependence previously involving variable instances in two different iterations to be between two variable instances in the same iteration making the dependence loop independent. This optimisation has fairly restricted use for fairly simple code structures only and requires more complex code alterations even for the simplest case :-



5.5.3 Loop Interchange.

Vectorisation exploits parallelism in the innermost loop of a nest of loops whilst MIMD parallelisation best exploits parallelism in outer loops in a nest. Loop interchange (Padua and Wolfe 1986, Allen and Kennedy 1987) allows two loops in a nest of loops to be interchanged and thus is useful to move parallel loops 'inward' for vectorisation or 'outward' for MIMD parallelisation as well as more subtle uses in breaking dependence cycles etc. It can also be used to move the most advantageous loop in a nest (perhaps the one with the highest iteration count) in the appropriate direction.

Loop interchange is not always a valid transformation, it can only be performed if no dependence relation is violated by the change in execution order caused. Consider the following example of nested loops :-

```
        DO I = 1,M
          DO J = 1,N
S           .....
          END DO
        END DO
```

Where S is any block of statements

The execution order of the statements S in terms of loop variables (where S(I,J) indicates execution for those particular loop variable values) is :-

S(1,1),S(1,2),S(1,3),..,S(1,N),S(2,1),S(2,2),S(2,3),..,S(2,N),S(3,1),....,S(M,N)

If the loops were interchanged (i.e. the J loop became the outer loop and the I loop the inner) then the execution order would be :-

S(1,1),S(2,1),S(3,1),..,S(M,1),S(1,2),S(2,2),S(3,2),..,S(M,2),S(1,3),....,S(M,N)

Now consider between which executions of the original loops a dependence occurs where that dependence is violated in the new order (e.g. where a use now precedes an assignment where a true dependence existed in the original loop). i,e,

S(1,2) δ S(2,1)

S(1,2) δ S(3,1)

. . . . . .

S(1,2) δ S(M,1)

S(2,2) δ S(3,1)

S(2,2) δ S(4,1)

. . . . . .

S(2,2) δ S(M,1)

Expressed generally, the violated dependencies are any dependence

S(I,J) δ S(I',J')

where I < I' and J > J'.

This loop interchange preventing dependence can be tested for by formulating a test in the same way as the Banerjee test in section 5.3.4, using I < I' and J > J' as additional constraints for the two loops concerned. The dependencies that potentially cause an interchange preventing dependence can easily be seen for the new constraints. The J > J' constraint is a dependence of an earlier iteration of the J loop on a later one, so clearly this cannot be a dependence carried by loop J. Thus only dependencies of the I loop need be additionally tested for loop interchange preventing. If any test for such a dependence proves true then loop interchange is invalid.

Loop interchange was used in chapter 4 to parallelise the Stone's Implicit Procedure. This example demonstrated that the use of this optimisation, although powerful, can require many subtle alterations of other related code, particularly that nested in only one of the two loops.

5.6 Closure.

This chapter has introduced a background to commercial parallelising compilers. The analysis described forms the basis for the parallelising tools developed in this work. The basic tests alone do not constitute a sufficient core to an analysis system. The following chapter shows how these tests have been strengthened and introduces extra information to enable an accurate dependence graph to be constructed.

The practical implementation of code generation and use of parallelism extracting optimisations is discussed in chapter 7 in the context of a suite of parallelising tools. Many of the current parallelising compilers automatically apply the optimisations, however, this is unsuitable in the context of parallelising tools due to the major role of the user.

CHAPTER SIX

6. Description Of Parallelising Tools.

6.1 Introduction.

The aim of this chapter is to describe the work to date on a serial code analysis toolkit. The majority of the work is concerned with the creation of an accurate dependence graph where the time required to perform the analysis of the serial code is not an important consideration. As such, the algorithms used extract as much information as possible and use that knowledge to eliminate as many dependencies as possible.

The motivation for the sections of this work was twofold :-

- Firstly, the extension of the dependence tests described in the previous chapter by providing more information and allowing a wider class of index expressions to be admissible. As described in section 5.3, the basic tests require the index of an array to be a linear function of the loop variables of the surrounding loops. This restrictive case is extended to allow symbolic index expressions and extra work in equality of scalar variables etc. has been included.

- Secondly, the analysis algorithm, at a stage in its development, would prove inadequate for some more complex code segments in the test cases (see chapter 7) where dependencies were assumed, which on a detailed inspection did not, in fact, exist. The extra features were included to enable the analysis algorithm to eliminate these dependencies by exploiting the same knowledge inherent in the code as was used

in the manual inspection. These features prove essential for a useful, useable tool to be created.

The following sections describe the components of the toolkit in chronological order of their inclusion in the PASCAL code developed in this work. The first sections are those required for a basic dependence graph building code with the remainder being the extensions and improvements incorporated.

The extracting of control information from the input serial code is detailed in sections 6.3 to 6.7 where accurate control dependencies and identification of loops in the code are performed. Sections 6.8 to 6.10 detail the actual dependence tests. Sections 6.11 to 6.13 involve additional features to reduce the number of dependencies set by incorporating the concept of exact dependence, interprocedural scalar information and symbolic forward substitution of scalar information. Section 6.14 shows the condition for equality of scalar variable references which is then used in the subsequent sections in the extra facilities which aim to aid the dependence testing and reachabillity stages. These involve extending Banerjee's inequality to involve scalar variables, general inequality testing and the use of a knowledge base of inequality and Boolean information. Finally, section 6.18 describes the interprocedural array analysis.

6.2 Language Interface Frontend.

The first stage in the process is the reading of the serial source code and its conversion to an internal representation which is then used throughout the code. The FORTRAN 77 source is

read and a parse tree created where each node in the tree is a pointer to the symbol table. Each routine in the source creates a separate parse tree and symbol table. The development of this section was based on Bornat (1979).

6.3 Call Graph And Routine Ordering.

Definition 6.1

A call graph is a graph where every node is a particular routine joined to other nodes by single direction edges. Each edge represents a routine call where the source is the calling routine and the sink is the called routine.

The graph is constructed by identifying all call statements and functions references in the parse trees of every routine and matching the symbol name with the header names of each routine.

This graph is then used to enforce a strict order on the routines in which they will be processed in all further sections of the analysis code. This order is imposed by performing a depth first search from the top node of the call graph (i.e. the main program), a routine being added to the end of the strict order list only after all routines referenced by this routine have been added. The call graph is assumed to be acyclic since FORTRAN 77 is a non recursive language, although it is possible for a non recursive cycle to exist. Any such cycle can be eliminated by duplicating the routines involved, but since a cyclic call graph is extremely rare, they are not considered further here.

The strict ordering ensures that the analysis is performed on a routine only after all the routines it references have been processed. Thus information from the called routine can be used in the dependence calculations in the calling routine.

6.4 The Control Flow Graph.

The control flow graph displays all possible routes that control can flow from routine start to routine end (Aho et al. 1986).

Definition 6.2

A basic block is a sequence of consecutive statements of the source code where control flow must start at the first statement passing through all statements once before leaving after the final statement.

Definition 6.3

A control flow graph is a graph where the nodes are basic blocks and the edges are single direction edges indicating a possible passing of control from one basic block to another.

For control passing by a binary if statement, the edges are additionally marked true or false. In later discussions, a father block is a block that can pass control to the current block whilst a child block is a block that the current block can pass control to. The use of basic blocks, rather than individual statements, minimises the number of nodes required in the control flow graph.

The following is an example code section :-

```
S1      10      CONTINUE
S2              IF Condition THEN
S3                      . . . . .
                ELSE
S4                      . . . . .
                ENDIF
S5              DO 20 I = 1,N
S6                      . . . . . .
S7                      DO 30 J = 1,M
S8                              . . . . . .
        30              CONTINUE
S9                      . . . . . .
        20      CONTINUE
S10             . . . . . .
S11             IF Condition GOTO 10
```

The basic blocks in this example are :-

Block 1	S_1 S_2
Block 2	S_3
Block 3	S_4
Block 4	S_5
Block 5	S_6
Block 6	S_7

Block 7 S_8

Block 8 S_9

Block 9 S_{10} S_{11}

The control flow graph for this code section is shown in figure 6.1.

6.5 Pre And Post Dominator Trees.

Definition 6.4

A predomination relationship between two nodes in a graph exists when to reach a node A from the start node of the graph, node B must have been passed through (i.e. no route exists from start to A that does not pass through B). In this case node B predominates node A.

Calculation of predominators can be achieved by performing depth first search from the start node, marking all reachable nodes but not passing through a blocking node of which the nodes it predominates are required. All unmarked nodes after the search completes are predominated by the blocking node.

Definition 6.5

An immediate predominator of a node A is the unique node B that predominates node A and itself is predominated by all other predominators of A. An efficient algorithm to calculate immediate predominators is given by Aho and Ullman (Aho and Ullman 1977).

Definition 6.6

The predominator tree is an acyclic graph where the nodes are the basic blocks in the control

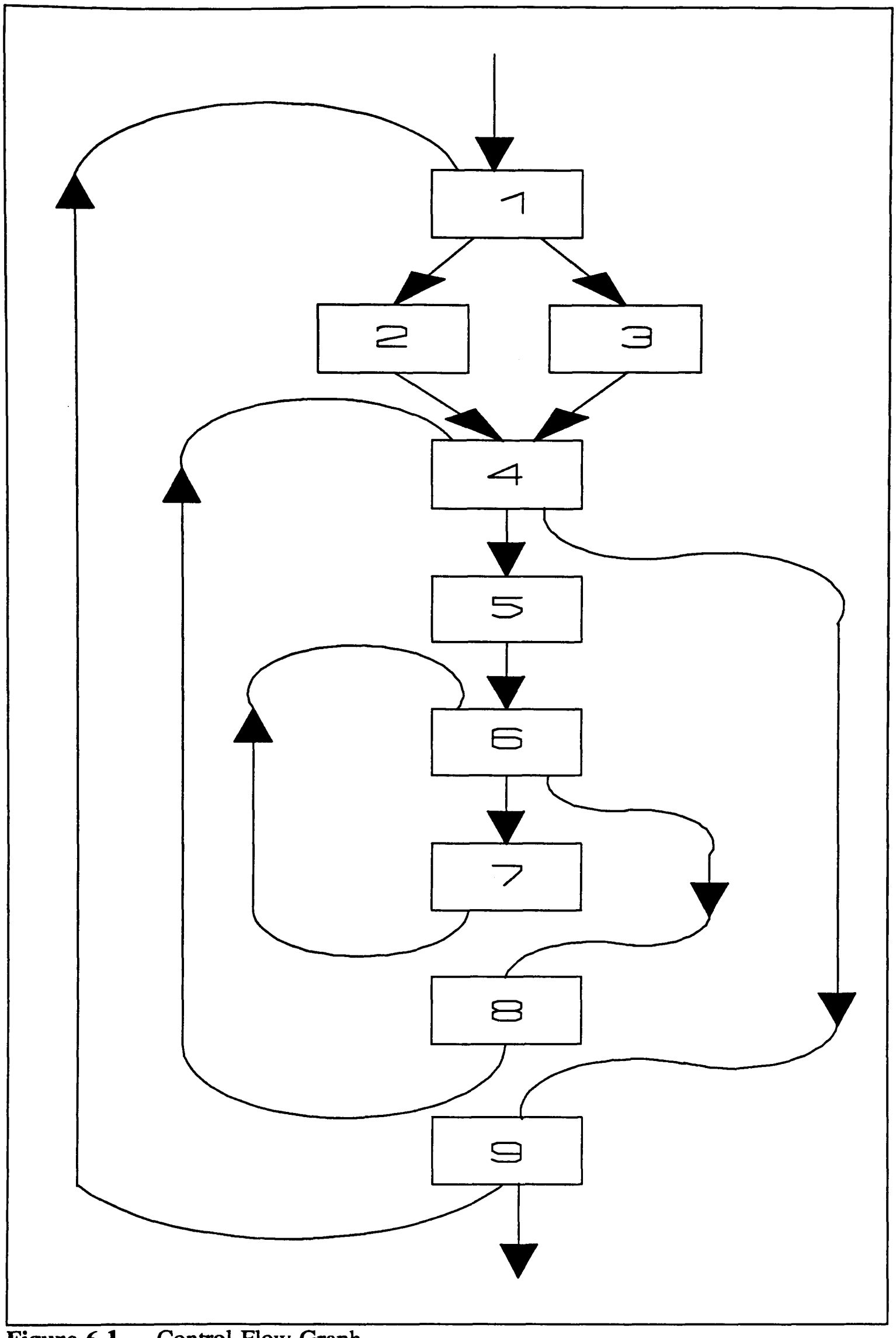


Figure 6-1 Control Flow Graph.

flow graph and the edges are the single direction relationships between basic blocks where the sink block is the immediate predominator of the source block in the control flow graph.

The postdominator tree is constructed in the same way as the predominator tree except that the reverse control flow graph is used where the searches commence from the end node traversing up control flow edges from sink to source.

Figure 6.2 shows the pre and post dominator trees for the example in figure 6.1.

6.6 Control Dependence Calculation.

To calculate accurate control dependencies of every statement, the post dominator tree is used.

Definition 6.7

The statements in a basic block are control dependent on the final statement in a directly connected father block in the control flow graph if the father block is not postdominated by this block.

This is true because the execution of the statements in the child block is dependent on which control flow edge is taken from the father block. One or more control flow paths exist from the father block to the end block in the control flow graph that do not pass through the child block.

The following algorithm is based on Ferrante,Ottenstein and Warren (Ferrante et al. 1987) and is used to calculate control dependencies :-

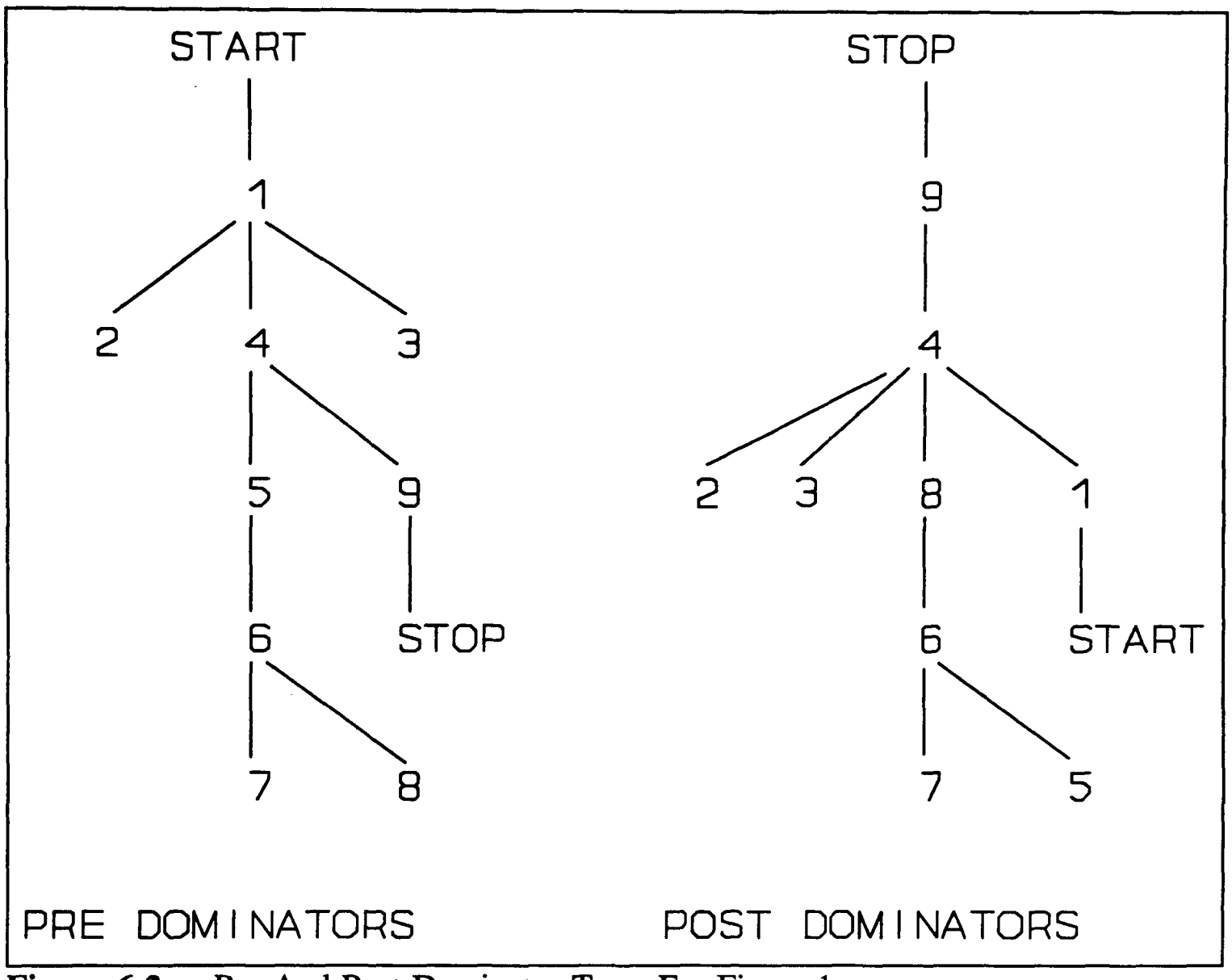


Figure 6-2 Pre And Post Dominator Trees For Figure 1.

For every basic block with two or more child basic blocks in the control flow graph, except DO loop heads do

Mark all basic blocks in the postdominator tree from the father block to the end block.

For each edge from the father block to a child block in the control flow graph do

If the child block is not marked (i.e child block does not post dominate father block) then

mark all blocks from the child block, travelling up the post dominator tree (i.e. mark the statements in these blocks) as control dependent on the last command in the father block, continuing until

a) a marked block is reached (i.e. a block postdominated by the father block and thus not control dependent on it)

or b) a predominator of the father block is reached (this case is included as it refers to the head of a loop as defined in the following section)

The algorithm produces a list of control dependencies for each statement. These dependencies are combined using logical OR operators to indicate the control set of the statement. If the controlling statement (i.e. the statement at the source of a control dependence) is itself control

dependent on other statements, the overall control of the original statement is formed using logical AND operators between all such control dependencies. A control dependence on a loop entry statement (see section 6.7) implies a control dependence on the entire loop concerned.

For those edges in the control flow graph marked true or false, the control dependencies are marked as true or false.

The workings of the control dependence calculation algorithm are demonstrated for the example in figures 6.1 and 6.2 :-

Consider the link (1,2). Block 2 does not postdominate block 1 thus a search up the postdominator tree is made, from block 2 until a postdominator of block 1 is found. Since block 4 is a postdominator of block one, only block 2 is marked as control dependent. Now consider link (4,5). Block 5 does not postdominate block 4. A search up the postdominator tree from block 5 passes through blocks 6 and 8 before block 4 itself is reached. Therefore blocks 5,6 and 8 are marked as control dependent. The full control dependencies are :-

Block 2 is control dependent on block 1

Block 3 is control dependent on block 1

Blocks 5, 6 and 8 are control dependent on block 4

Block 7 is control dependent on block 6

Note that block 7 is control dependent on block 4 in a transitive nature via block 6 and that

in the algorithm, blocks 4 and 6 are DO loop heads and thus are not processed.

6.7 Loop Setting.

An assumption about the input code is that it is well structured and, in particular, all loops must be single entry loops i.e the entry block (known as the head block) of the loop must predominate all basic blocks contained in the loop. Although the majority of parallel performance is gained through DO loops, other iterative loop types are also considered for completeness.

Definition 6.8

A loop in the control flow graph is identified by an edge where the sink block of the edge predominates the source block. Such an edge is called a backlink (Aho and Ullman 1977).

If several backlinks have sinks at the same head block then one or more loops must be generated. If the required control dependence information exists then one loop will suffice, however, this is not always the case.

Consider two backlinks to the same head block. Let second backlink source be a child block of the first backlink source. Thus if both blocks are to be contained in the same loop there must be a control dependence on the second block to indicate that control only reaches it if the condition for taking the first backlink route is false. The control dependence algorithm will only set such a control dependence if the second block does not post dominate the first, thus a single loop is invalid for the postdominating case.

A post dominator of the first block must also be a postdominator of the loop head since the first block has a direct link to the loop head which by definition cannot reach the end block in the control flow graph without passing through this postdominator. The loop setting algorithm must therefore generate nested loops if a block postdominates the loop head and is contained in some, but not all, cycles formed by backlinks to this loop head.
The nesting order of these loops is then determined by examining each individual backlink cycle to find which postdominator blocks of the loop head are contained in this cycle. The number of these blocks contained determines the loop nesting, the cycles containing the fewest blocks forming the inner loop where cycles with the same number of blocks included are used to form a single loop.

The postdominators of the loop head represent a single path in the flow graph through the loops with that loop head. Similarly, the predominators of a block in such a loop must form a path that passes through the loop head. The two paths intersect as the loop head is approached with predominators and postdominators used in the paths being the same basic blocks (i.e. if all routes from the head down the flow graph pass through a block then all routes up the flow graph from a child of this block to the loop head must also pass through this block). Thus a search is made from the loop tails (backlink source blocks) up the predominator tree, until a loop head postdominator is reached. The reached postdominators position in the post dominator tree determines the loop nesting.

The algorithm used must also determine the correct nesting for loops with different loop heads, placing the outermost loop first and innermost last. Thus the basic blocks that are examined to determine if they are loop heads are tested in strict order, determined by a depth

first search of the control flow graph, considering child blocks first. The loop nesting lists are added to all basic blocks contained in the loop maintaining the nesting order.

The algorithm proceeds as follows :-

Commence at the start block and perform a depth first search.

When all child blocks in the control flow graph have been processed do

For each control flow edge with this block as a sink do

If this block predominates edge source block then list source block as a loop tail, mark edge as a backlink and consider this block as a loop head

If only one loop tail has been found, generate this loop

If more than one loop tail has been found then

Mark from loop head up postdominator tree

For each postdominator from the loop head in order do

For each tail not yet added do

Search up the pre dominator tree from this loop tail until a

marked block is reached. If marked block is the postdominator of the loop head currently being considered then

Include this loop tail in current loop information record

If any tails have been added to this loop record then generate this loop

Loop generation adds the new loop nesting information to the blocks in the appropriate loops as follows :-

For each loop tail do

Perform depth first search up the reverse control flow graph from this loop tail (also traverse backlinks of the current loop head that have already been included in deeper nested loops). Add the current loop to the start of the reached block loop nesting list.

The loop nesting and control dependencies provide all the necessary control flow information required when DO loop exit edges are included in the loop information. Iterative loops (i.e. not DO loops) are generated as REPEAT style loops with unique entry and exit statements. This provides a great deal of flexibility in the statement ordering used in code generation where no GOTO statements except loop exit GOTO's are required (i.e. statements can be prefixed with a conditional mask statement evaluating the control dependencies, with the statement in the assigned loop nest).

As a first example of the loop setting algorithm, consider the code section used in figures 6.1 and 6.2 :-

Backlinks are (7,6), (8,4), (9,1)

Since no two backlinks have a common sink, they each produce one loop.

Let L1 be the loop with head 6 containing blocks 6 and 7

Let L2 be the loop with head 4 containing blocks 4,5,6,7 and 8

Let L3 be the loop with head 1 containing blocks 1,2,3,4,5,6,7,8 and 9

This gives the loop nesting

Block 1 is nested in L3

Block 2 is nested in L3

Block 3 is nested in L3

Block 4 is nested in L3 and L2

Block 5 is nested in L3 and L2

Block 6 is nested in L3, L2 and L1

Block 7 is nested in L3, L2 and L1

Block 8 is nested in L3 and L2

Block 9 is nested in L3

Now consider a more complex case as shown in figure 6.3 :-

Backlinks are (3,1), (4,1), (5,1)

All three backlinks have the same sink block so the first postdominator block of the loop head block contained in the cycles formed by these backlinks determines the number and nesting of loops formed :-

First postdominator of loop head reached in search up the predominator tree from block 3 is block 1.

For block 4, first postdominator reached is block 1.

For block 5, first postdominator reached is block 1.

Therefore, since they all reach the same postdominator block, enough control information exists to allow a single loop to be generated.i.e.

Loop is 1,2,3,4 and 5

Control dependencies are 2 on 1, 3 on 2, 4 on 2 and 5 on 3.

Backlinks are (3,1), (4,1), (5,1)

Finally, consider the example in figure 6.4:-

Again all three backlinks have the same sink so search up the predominator tree to find first postdominator of loop head reached :-

From block 3, first postdominator reached is block 3.

From block 4, first postdominator reached is block 2.

From block 5, first postdominator reached is block 5.

Therefore, since different postdominator blocks have been reached, three different loops must be generated. The order in the post dominator tree from the loop head up of the reached blocks is 2 followed by 3 and finally 5. The three loops are thus created as follows :-

Loop L1 contains blocks 1,2 and 4.

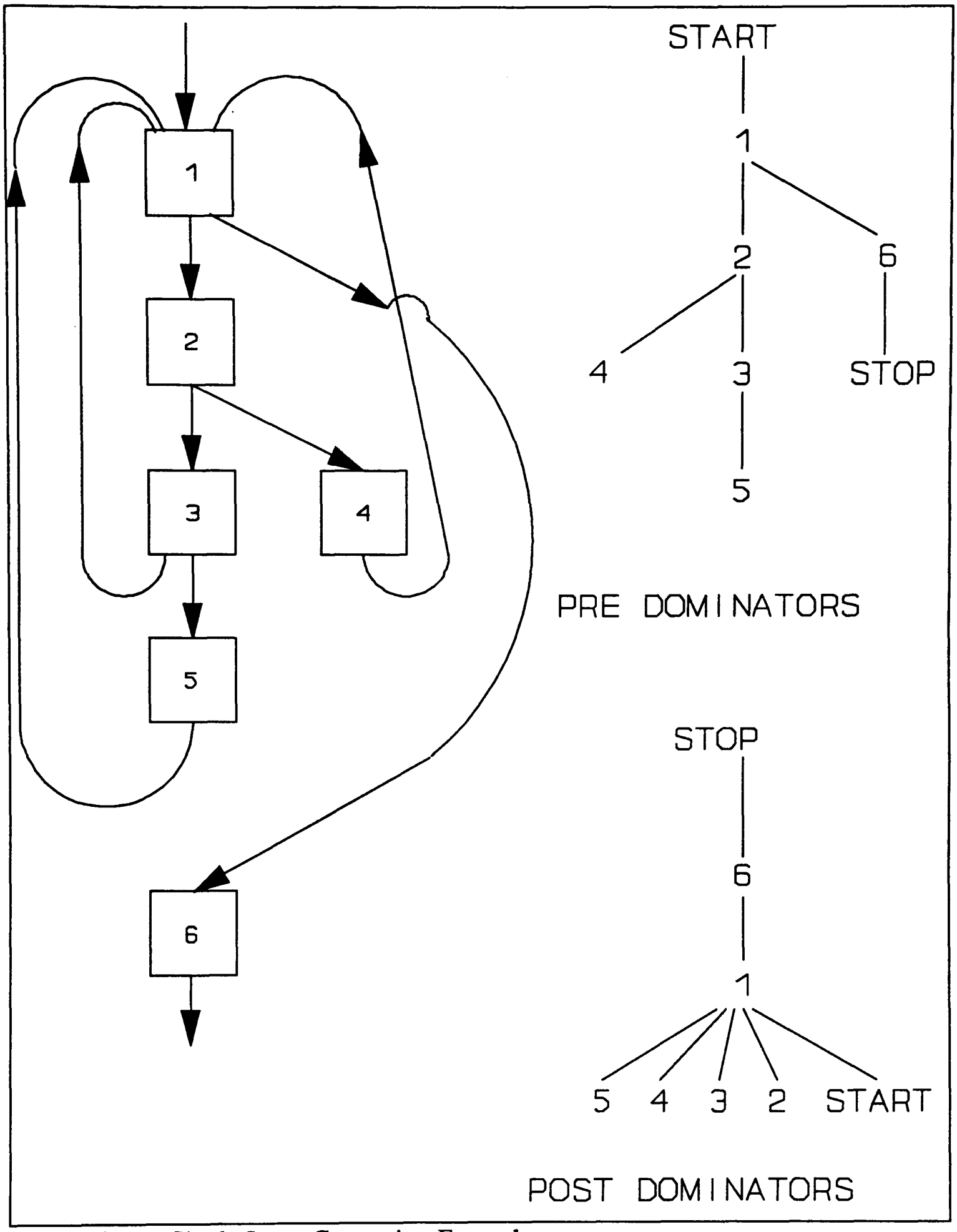


Figure 6-3 Single Loop Generation Example.

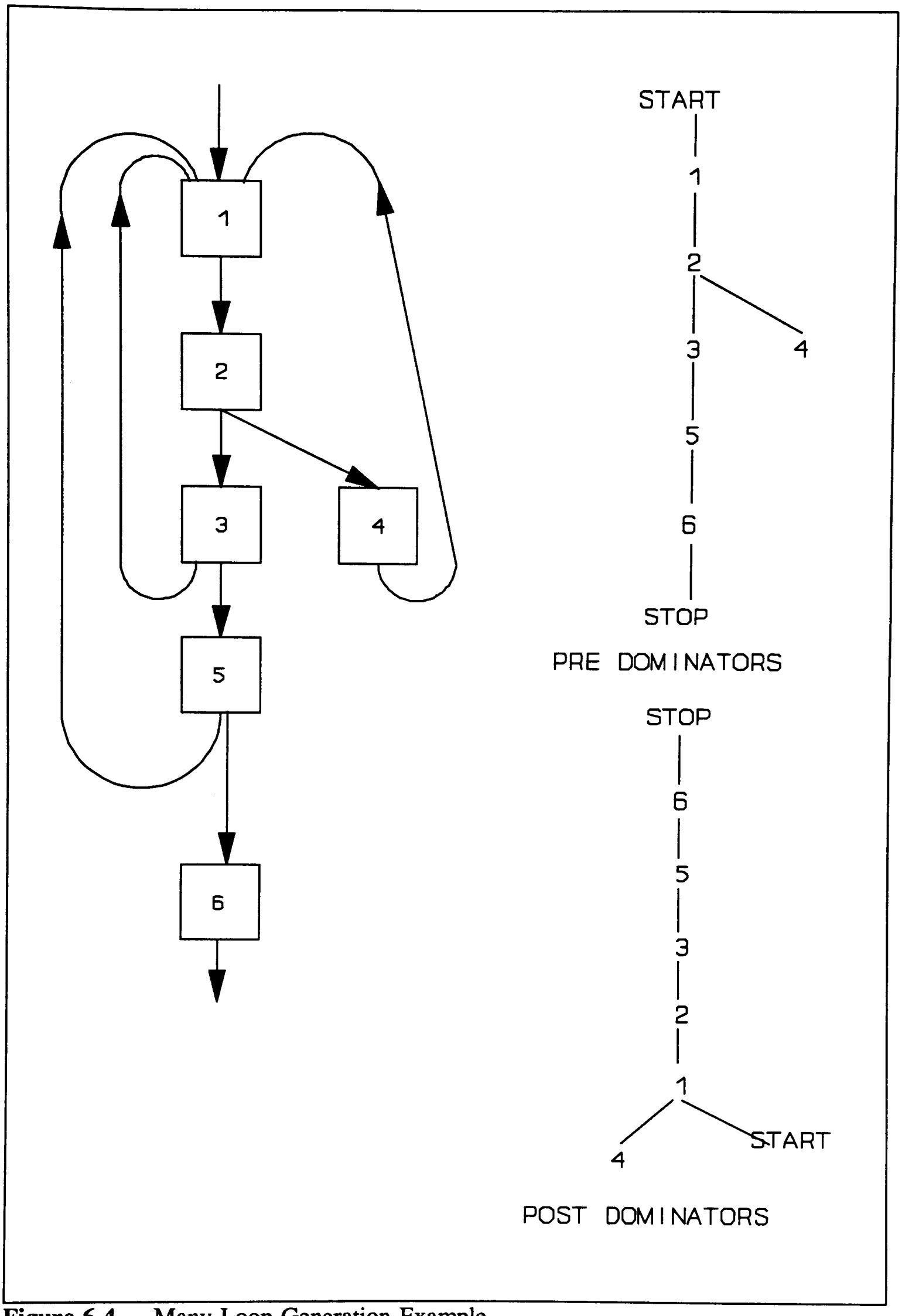


Figure 6-4 Many Loop Generation Example.

Loop L2 contains blocks 1,2,3 and 4.

Loop L3 contains blocks 1,2,3,4 and 5.The nesting of blocks in order of outermost surrounding loop to innermost surrounding loop is :-

Block 1 is nested in L3, L2 and L1.

Block 2 is nested in L3, L2 and L1.

Block 3 is nested in L3 and L2.

Block 4 is nested in L3, L2 and L1.

Block 5 is nested in L3.

Several loops are clearly necessary since the only control dependence found in this case is 4 control dependent on 2.

6.8 Reference Information.

The first stage in the dependence analysis process is to produce a list of variables referenced for every statement.

Definition 6.9

The set of variables referenced by a statement consists of two subsets. Firstly, the set of all variables whose values are used by the statement and, secondly, the set of all variables assigned values by the statement.

For scalar variables, the symbol referenced is stored and marked to indicate if the variable is used and/or assigned. For call statements and function references, the reference information is obtained from the called routine by examining its dependence graph (See section 6.12).

References of variables in the call parameter list and in common blocks active in the current routine are marked as used or assigned as with other references. Some variables accessed in called routines may, however, be passed through common blocks which may not be active in the current routine. To ensure that dependencies between such variables in different call instances are detected, when an inactive common block variable is found in a called routine, the common block is temporarily added to the common block list of this routine. Since the common block is in a different routine, all symbols will be located in the unique symbol table of that routine, therefore no clashes with other variables in the current routine can occur. Dependencies will then be detected as carried by the common block variable in the first instance found, providing the necessary code ordering information. The added common block is removed after dependence analysis is complete. i.e.

```
SUBROUTINE Caller
COMMON/c1/. . .
. . . . . .
CALL Called1(A)
. . . . . .
CALL Called2(A)
END
SUBROUTINE Called1(B)
COMMON/c2/X
X = . . . .
B = . . . .
END
SUBROUTINE Called2(C)
```

```
      COMMON/c2/Y
      PRINT*,Y,C
      END
```

The references for the call to Called1 are

X Assigned and A Assigned.

The references for the call to Called2 are

X Used and A Used.

with the common block c2 temporarily added to the common block list of subroutine Caller.

For arrays, the array symbol is stored along with the access information as for scalar variables. In addition, each index of the array is examined and listed in two parts. Firstly a list of coefficients of the loop variables surrounding the statement in the order of loop nesting is produced, each coefficient being an integer constant and a list of variables other than loop variables (referred to as nonloop variables). The second part of the index is a list of nonloop variables in the index which are not coefficients of loop variables. Nonlinear combinations of loop variables are included in the second list. If an index expression contains an array or a function, that index is marked as unavailable to the dependence testing routines.

6.9 Reachabillity Of Statements.

Definition 6.10

To create a dependence, there must be a path from source statement to sink statement in the control flow graph.

In addition to the actual reachabillity determination, the level of the dependence (ie loop carried by which loop or loop independent) that can exist is determined by monitoring which backlinks have been taken in the current path.

Every statement is considered a dependence source statement. To find the dependencies on this source statement, a depth first search from this statement down the control flow graph is used. Tests are made at the infinity level (i.e. loop independent) between the original statement and the statements in the reached blocks until a backlink is taken in the path. After a backlink has been taken, the dependence tests are made at the level of the loop that the backlink forms (i.e. loop nesting level over the source statement). A backlink is only taken if the source statement is nested within the associated loop. Once a backlink is taken, no further backlinks can be used until an edge where the sink block is not nested in this loop in the control flow graph is taken. After taking such a loop exit edge, the dependence test level returns to the infinity level. As long as the loop heads predominate the blocks in those loops, the full reachable set of blocks is found for each statement.

6.10 Dependence Testing.

The dependence graph building algorithm works in two passes through the routine list in strict call graph determined order.

Scalar analysis is first performed with all scalar references for a routine listed. A dependence exists between two references of the same scalar variable if the sink statement is reachable from the source statement and the usage and assignment information of the two references

involved fit one of the true, anti or output dependence forms (i.e. at least one assignment).

Array dependencies can exist when two accesses to the same array variable are reachable in the same way as scalar variables, except extra tests on array indices can prove no dependence exists. The dependence disproving test are the GCD and Banerjee tests introduced in chapter 5. The level of the test used is determined in the reachabillity process. The tests proceed as follows :-

If no index information is available from either statement then assume dependence (i.e. a full array reference in a read or write statement).

For each pair of indices in turn until all have been processed or independence is proved do

If either index is unavailable then examine next pair of indices

If all nonloop variables can be eliminated from the equation, use the GCD test (section 5.5.3). If no solution to equation possible then independent

Use extended Banerjee test (see section 6.15) based on Banerjees inequality (section 5.5.4). If no solution possible then independent

If independence not proved then dependent

6.11 Exact Dependence.

Definition 6.11

A dependence is defined as an exact dependence if every usage (or the assignment in the source statement for output dependence) of a dependence is matched by an assignment of the same memory location in the other statement.

For true dependence $S_1\ \delta\ S_2$, usage in S_2 must be a subset of assignment in S_1. For anti-dependence $S_1\ \bar{\delta}\ S_2$, usage in S_1 must be a subset of assignment in S_2. For output dependence $S_1\ \delta^{\circ}\ S_2$, assignment in S_1 must be a subset of assignment in S_2

For scalar variables, since only one memory location is used, all scalar dependencies are exact. In the case of array dependencies, a test is used for a restrictive, but never the less important, set of index functions. Any index not in this set must use a conservative assumption and thus cannot be considered exact. The set considered is defined using the statement model from section 5.3.2 :-

For level k dependencies

$a_i - b_i = 0 \quad i = 1(1)k\text{-}1 \qquad , \qquad a_k = b_k = 0$

$a_i = 0 \quad i = k+1(1)m \quad i \neq A \qquad , \qquad b_i = 0 \quad i = k+1(1)n \quad i \neq B$

For Infinity level dependencies

$a_i - b_i = 0 \quad i = 1(1)c$

$a_i = 0 \quad i = c+1(1)m \quad i \neq A \qquad , \qquad b_i = 0 \quad i = c+1(1)n \quad i \neq B$

Thus all loop variable coefficients in the f - g equation are zero except for the A^{th} surrounding loop of S_1 and the B^{th} surrounding loop of S_2 where these surrounding loop must be DO loops not common to both statements (or in different instances of the same loop for loop carried dependencies). These coefficients must also be integers only as must the constant terms a_0 and b_0.

The test then ensures that every value in S_1 is covered by a value in S_2 considering the coefficients and appropriate loop limits. Since only one loop variable with an integer coefficient is admissible, the requirement for an exact dependence can be stated as follows :-

For all $x \in [1,N_A]$ there must exist a y value ($y \in [1,N_B]$), where N_A and N_B are the loop upper limits of loops A and B respectively, such that

$$a_A x + a_0 = b_B y + b_0$$

Therefore

$$y = \frac{a_A}{b_B} x + \frac{a_0 - b_0}{b_B}$$

x and y are loop variables therefore they must be integers. For the solution y for a given x to be an integer, both the coefficient of x and the constant term must also be integers i.e.

a_A must be divisible by b_B

$(a_0 - b_0)$ must be divisible by b_B

y must be in range $[1,N_B]$

Where all executions of statement S_2 in the loop B are guaranteed to be performed.

The tests used thus are :-

1. Test $a_A \bmod b_B = 0$

2. Test $(a_0 - b_0) \bmod b_B = 0$
3. Test high and low extremes of the index values to ensure S_2 extremes encompass S_1 extremes.
4. Ensure no DO loop exits exist in the loop B
5. Ensure S_2 predominates loop tail of loop B

For multi dimensional arrays, the loop B for each index of S_2 must be a different loop in the nest to ensure that all combinations of indices will be executed. Every pair of indices must be exact and satisfy the above requirement.

These exact dependencies are then used to simplify the dependence graph in two ways, one involving transitive ordering and the other involving actual data flow.

Firstly, during the depth first search to determine reachabillity, an exact output dependence can indicate a barrier to the assignment in the first statement. For scalars or arrays where all coefficients of loop variables are zero, the second statement definitely redefines the assignment made in the first statement in this control flow path. If an exact array dependence where non-zero coefficients were found exists then on leaving the loop B (or the outermost loop B of all such loops found for the different indices of a multi dimensional array) in the control flow path, the first statements influence can be ignored.

This is legal since any further dependencies that would have been found in this control flow path on the first statement will now only be found for the second statement. If these dependencies are true dependencies, then this is clearly correct (i.e. the values used are those

assigned in the second statement and not the first), or if output dependencies are found then the ordering of the first statement to the statement dependent on the second statement is still imposed transitively through the second statement.

Similarly anti-dependence ordering can be enforced transitively. If an anti-dependence is exact then the references in the source are a subset of those in the sink, thus any further anti-dependencies on the first statement will also be output dependent on the second statement. The anti-dependence is therefore enforced by an anti-dependence followed by an output dependence.

This treatment of anti-dependencies is clearly only valid if output dependencies are considered when the dependence graph is used. Thus using the code generation strategy for distributed memory machines described in section 5.4.3, anti-dependencies must be determined directly, not relying on transitive dependencies involving output dependencies that are ignored in parallelism detection.

The second use of exact dependencies is when a true dependence involving an array is found to be exact. After the complete dependence graph has been constructed, any true dependence, where the route from defining statement to using statement must pass through the defining statement (or appropriate loop surrounding that statement) of an exact true dependence on the same using statement, can be removed since the values used are all assigned in the exact true dependence source statement.

6.12 External Influences Of The Dependence Graph.

Definition 6.12

The external influences of a routine are the variables whose values are passed into the routine and the variables assigned and passed back to the calling routines.

After constructing the dependence graph for a routine, extra start and stop nodes are added. All statements with variables used that can have a value from outside this routine and are included in the routine parameter list or a common block are joined in the dependence graph to the start node. Similarly all statements that define variables that are passed out of this routine are connected to the stop node.

These start and stop nodes form the basis for the interprocedural information of call statements etc. mentioned earlier. To account for indirect variable accesses through calls in the called routine, a depth first search of the call graph from the called routine is used, listing all common block variables accessed in routines where the common block is active in the original calling routine. The appropriate variables are found by examining all variables connected to the start and stop nodes.

6.13 Forward Substitution Of Scalar Information.

The scalar analysis is performed before the array analysis to allow scalar information to be used to aid the more complex array test.

A scalar variable can be a loop variable where the value of the variable is always assigned in the DO loop statement. All other scalar variables are nonloop variables whose values are evaluated in previous statements which can be determined through true dependencies in the scalar dependence graph. To increase the information available to the tests, the true dependence paths up the graph are followed, substituting an original variable with the set of variables used to evaluate it.

Definition 6.13

A dependence indicates a definite definition of the value of a variable if and only if it is an infinity level true dependence and is the unique true dependence involving this variable with a sink on the current statement.

This guarantees that the source of such a dependence is the statement that was used to evaluate the variable concerned. A depth first search up the scalar dependence graph only using the dependencies described substitutes the new variables, continuing until an inappropriate statement is reached (i.e. a read statement etc.), or until no unique true dependence is found. When the search path cannot continue, the set of variables currently held are used with the defining statement of those variables set to be the last statement reached.

The search continues through assignment statements, performing further searches for every variable used in the statement.

If the search encounters a call statement then it continues from the stop node of the called

routine with the appropriate name alteration through the parameter list or common blocks. The calling routine and call statement concerned are listed to indicate that control will return to that statement in that routine if this routine is exited.

If the start node of a routine is reached, the variable is assigned in a calling routine. If a unique calling routine and a unique call statement exists or if a routine and statement are listed (from passing into a called routine earlier in the search), then the search continues from the call statement in the calling routine. If no such call can be found the start node of this routine is set as the defining statement of the current variable. The call statement may pass the value into the current routine through an expression parameter, in which case it is treated as the right hand side of an assignment statement. Alternatively, it could be passed through a common block which may or may not be active in the calling routine. For an active common block, a search from the call statement is used to continue the process, whilst for a non active common block the process continues from the start node of the calling routine.

This forward substitution converts variables to a more standard set of defining variables, allowing a comparison between statements to eliminate these unknown variables. No information is lost by the search since if two variables reach the same statement in the graph with the same variable they will continue up identical paths, producing the same set of defining variables.

6.14 Nonloop Variable Storage And Equality.

A nonloop variable is stored in two parts, the variable symbol and the defining statement of

this instance of the variable. Without forward substitution, a used nonloop variable defining statement would be the statement that the usage is in. Thus two separate statements accessing the same nonloop variable would always be considered as not equal. The forward substitution enables such variables to be traced back to a defining command and thus giving the possibility of proving equality.

An expression is broken down into loop and nonloop variables where all nonloop variables in the lists of nonloop variables are added together (i.e. the nonloop variables must form a linear combination). Thus nonlinear terms must be stored as a single nonloop variable which itself is a product of individual nonloop variables each with a symbol and defining statement. The terms of a nonlinear nonloop variable are stored in alphabetical order of the symbols they reference to simplify the comparison process.

Two nonloop variables are considered the same only if it is guaranteed both will have the same value. To test this, both variables must be a combination of the same individual nonloop variables where each associated pair of these variables is guaranteed to have the same values.

Definition 6.14

Two individual nonloop variables have the same value if they reference the same symbol and they are both defined by the same instance of the same defining statement.

Definition 6.15

To ensure that the same instance of the defining statement produced both values, the number of loops common to both statements involved in the original nonloop variable reference and

the defining statement, must be less than the level of the dependence test being performed. In addition, the common nesting of the defining statement and either of the other statements must not be greater than the common nesting of the two nonloop using statements.

This last check is only relevant when the defining statement is in a repeat style loop since this is the only way a unique, definitely executed, defining statement can be found contained in loops not also common to the nonloop variable using statement (i.e. since other loop types do not guarantee execution).

At the infinity level, the same defining command assures the same instance of the defining statement was used. The following example demonstrates the equality of nonloop variables :-

```
          DO I = 1,10
S_D         N = .....
          DO J = 1,10
S_1           = ... N ...
S_2           = ... N ...
          END DO
        END DO
```

S_D is the defining statement of nonloop variable N after forward substitution is performed on both statements S_1 and S_2. At the infinity level the values of N in S_1 and S_2 are clearly the same. At level 1, between different iterations of the I loop, two different instances of the

defining statement S_D will provide the values of N used in S_1 and S_2, thus they cannot be considered equal. At level 2, between iterations of the J loop for the same iteration of the I loop, the same instance of S_D is used for both S_1 and S_2 and thus the values of N are definitely equal.

6.15 Extended Banerjees Test.

In order to perform the GCD test (see section 5.3), all nonloop variables must be eliminated from all the coefficients of the equation (including constant terms). Even with nonloop variable equality and possible elimination from terms, this can severely restrict the use of the GCD test.

The Banerjee inequality can be used without any such restriction. Many of the terms in the equality use the positive or negative parts of coefficients only, thus for any coefficient that involves nonloop variables the sign of the coefficient may have to be assumed. For independence to be proved, every possible combination of signs for every term involving nonloop variables must be tested and all must prove independence. The process is therefore recursive with every term in the Banerjee inequality whose positive or negative parts are required that involves nonloop variables having a sign assumed positive, negative and zero, each assumption being followed by a recursive call for the next term in the inequality, eventually leading to a Banerjee test for the set of assumptions made.

Consider the following example :-

```
        DO I = 1,M
            DO J = 1,N
S               A(N*(I-1) + J) = . . . .
            END DO
        END DO
```

The Banerjee test for statement S to test for a loop carried output dependence on itself at level 1 has the coefficients :-

$$a_0 = b_0 = -N \quad , \quad a_1 = b_1 = N \quad , \quad a_2 = b_2 = 1$$

The Banerjee inequality thus is :-

$$-N - (N^- + N)^+(M - 2) - (N - 1) <= 0 <= -N + (N^+ - N)^+(M - 2) + (N - 1)$$

To evaluate the truth of the test, the signs of all the superscripted terms must be known or assumed. Three such terms exist, N, $(N^- + N)$ and $(N^+ - N)$, thus if each is to assume a positive, negative and zero value, 27 tests will be required in total, all of which must prove false. To simplify this, consider the J loop surrounding S. For S to be in a dependence this loop must have been entered, thus we know that $N > 0$. Using this information in the Banerjee inequality reduces it to :-

$$-N - N(M - 2) - (N - 1) <= 0 <= -N + (N - 1)$$

$$0 <= -1$$

The inequality is thus proved false and no level 1 output dependence is set.

The final inequality tests will often still involve nonloop variables, thus further tests are required to try to evaluate the truth of inequalities with nonloop variables. Similarly, the determination of signs that must be assumed for each term can be made to eliminate combinations where a contradiction exists between the assumptions. The following section describes the algorithm used.

6.16 Inequality Tests With Nonloop Variables.

The algorithm described in the following section tries to prove false an inequality involving nonloop variables. The applications of this algorithm are numerous since nonloop variable array indices and nonloop variable DO loop maximums are common, requiring all algorithms to make as much use of symbolic information as possible.

The information used in this test is extracted from several sources. In the Banerjee test sections, all previous assumptions about the sign of terms involving nonloop variables can be used. The upper limits of all surrounding loops of both statements must be greater than or equal to one (since the loops are normalised). This is true since, if the statements are to execute, all surrounding loops must be entered. An upper limit less than one will cause the loop to be skipped, thus control cannot reach the nested statement and therefore it cannot be involved in a statement. If we are testing for a loop carried dependence there must be at least two iterations of the loop concerned (since the dependence is between different iterations), therefore the upper limit of this loop must be at least two. Information given by the user can

also be added such as equalities or inequalities relating to certain variables. Control information can also be included as described in section 6.17. All the equations used are adjusted to give information relative to zero.

This information is only useful to the inequality we are testing if it can be used to eliminate all unknowns from the original inequality. The process therefore continues by trying to find a linear combination of the known expressions with the same set of nonloop variables as the original inequality. Every inequality used indicates the sign of the combination (since the inequalities are relative to zero), thus the linear coefficient of a known expression and the sign of the expression indicate the sign of this term in the combination. For a less than zero test, a combination is required that has a definitely positive value. Thus every expression sign and coefficient product used in the combination must be greater than or equal to zero to enable the elimination of unknowns and possible disproof of the original test i.e.

Let the original test be

NONLOOPS + C < 0

Where NONLOOPS is a linear combination of nonloop variables and C a constant term.

A linear combination of known expressions gives

NONLOOPS + K >= 0

Where NONLOOPS is the same set of nonloops as in the original test and K is the linear combination of the constant terms in each inequality

Therefore

NONLOOPS + C < 0 <= NONLOOPS + K

Cancelling nonloop variables gives

$$C < K$$

Thus if C is greater than or equal to K the original test is false.

For an original test of less than or equal to zero, each term in the combination must again be positive or zero. If any term is definitely greater than zero (i.e. a greater than or less than expression used with a non-zero coefficient in the combination) then the disproving test is again C >= K. If no such term is included, the test must be C > K. Tests of greater than zero work in the same way but require negative combinations of known expressions.

The algorithm to perform the process described so far runs as follows :-

1. Set up a vector $\underline{b}$ of coefficients of the nonloop variables in the inequality being tested.

2. Set up a matrix A where each column represents the coefficients of nonloop variables in a particular inequality in the known information set, the rows corresponding to the nonloop variables already placed in $\underline{b}$, two nonloop variables being considered the same as described in section 6.14. Any nonloop not in $\underline{b}$ is allocated a new row with a zero placed in the appropriate element of the $\underline{b}$ vector.

3. Set up two further lists C and S, C to store the constant terms for the inequalities and S to indicate the sign of the inequality (i.e. the set (<,<=,=,>=,>) represented by the integers (-2,-1,0,1,2)). The order of these lists corresponds to the column order of the

known inequalities in A.

4. Solve the system $A\underline{k} = \underline{b}$ to provide $\underline{k}$ the coefficient vector used to form the linear combination of known inequalities that give the same set of nonloop variables as in the original test.

5. If a unique solution is found then ensure that all combinations $k_i * S_i$ have the required sign. If so test the constant in the original test against the constant from the linear combination $\sum (k_i * C_i)$.

This algorithm is demonstrated in the following example :-

Test N + M >= 0

Use known inequalities N <= 0 and 2*M + 1 <= 0

Thus $A = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$ $\underline{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ Solves to $\underline{k} = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix}$

Constant list $\underline{C}$ = (0 ,1) ; Sign list $\underline{S}$ = (-1 ,-1)

$S_1 * k_1 = -1 * 1 <= 0$; $S_2 * k_2 = -1 * 0.5 <= 0$

Thus the final test is possible :-

N + M >= 0 >= N + M + 0.5 implies 0 >= 0.5

This is false, therefore the original test inequality is false.

The $A\underline{k} = \underline{b}$ system is solved using Gauss Elimination and backsubstitution. The matrix A however is not usually a square matrix (i.e. the number of known equations and unknown

nonloop variables are different), thus some systems will have no solution and others an infinite number of solutions. The elimination phase attempts to convert the matrix into an upper triangular echelon matrix, if this is possible then solutions may exist. If any diagonal elements are zero, an infinite number of solutions may be possible (after ensuring a consistent set of equations remain), with variables involved in the solution for the $\underline{k}$ vector.

Consider an echelon matrix as produced by the forward elimination phase with the associated vectors shown below :-

$$\begin{bmatrix} a_{1\,1} & \cdots & a_{1\,m} \\ & \ddots & \vdots \\ & & a_{n\,n} \cdots a_{n\,m} \end{bmatrix} \begin{bmatrix} k_1 \\ \vdots \\ k_m \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Where m inequalities and n unknowns are involved.

The backward substitution algorithm introduces variables into the solution vector $\underline{k}$ whenever possible, these variables legally taking any value. These variables are denoted by the x variables introduced in the algorithm, with U representing the number of unknowns introduced. The algorithm runs as follows :-

$k_i = x_{m-i+1}$ $\quad i = n+1(1)m$

$U = m-n$

For i = n downto 1 do

If $a_{i\,i} = 0$ then

If $\sum_{j=i+1}^{m} a_{i\,j}\, k_j = b_i$

$U = U + 1$

$k_i = x_U$

else

If $\exists$ p s.t. $a_{i\,p} <> 0$ & $k_p = f(x_V)$ then

Set $x_V = (b_i - \sum_{j=i+1}^{m,\, j\neq p} a_{i\,j}\, k_j) / a_{i\,p}$

Substitiute for unknown x_V in all solutions.

$k_i = x_V$

else

No Solution Possible.

endif

endif

else

$k_i = (b_i - \sum_{j=i+1}^{m} a_{i\,j}\, k_j) / a_{i\,i}$

endif

This produces solutions to $\underline{k}$ where each k_i is of the form :-

$$k_i = d_{i\,0} + d_{i\,1}\, x_1 + d_{i\,2}\, x_2 + \ldots\ldots + d_{i\,U}\, x_U$$

If the solution contains variables (i.e. $U > 0$), the algorithm attempts to find the 'best' solution given the requirements of disproving the original inequality. To achieve this the Simplex optimisation algorithm is used with the variables x in the $\underline{k}$ solution being the basic variables in the Simplex procedure.

The constraints used for the Simplex algorithm are formed to ensure every inequality and coefficient product has the correct sign to produce a final combination of the correct sign. ex.

If $S_i * k_i$ must be >= 0 then

If $S_i > 0$ then

$$d_{i\,0} + d_{i\,1}\, x_1 + d_{i\,2}\, x_2 + \ldots\ldots + d_{i\,U}\, x_U >= 0$$

else If $S_i < 0$ then

$$d_{i\,0} + d_{i\,1}\, x_1 + d_{i\,2}\, x_2 + \ldots\ldots + d_{i\,U}\, x_U <= 0$$

endif

endif

A requirement of the Simplex algorithm used is that all variables must be greater than or equal to zero. In the back substitution phase, when a variable is introduced it alone forms the solution for one particular k_i and is thus the coefficient of an inequality used. Therefore, to ensure the variable is always positive, a check of the sign of the inequality concerned is performed, if this k_i coefficient must be negative to ensure the correct $S_i * k_i$ sign, then the variable is redefined throughout the $\underline{k}$ solution to be $-x_i$. i.e.

If k_D introduces x_V and k_D must be less than or equal to zero then

$$d_{i\,V} = -d_{i\,V} \quad i = 1(1)m$$

The cost function for the Simplex algorithm is the resulting constant term that will be used in the final constants test. This constant term is itself made up of a constant and a linear combination of the Simplex variables. i.e.

$$\text{Constant} = \sum_{i=1}^{m} (k_i * C_i)$$

$$= \sum_{i=1}^{m} \{(d_{i\,0} + d_{i\,1}\, x_1 + d_{i\,2}\, x_2 + \ldots\ldots + d_{i\,U}\, x_U) * C_i\}$$

Thus the final cost function is :-

$$\sum_{i=1}^{m} d_{i\,0}\, C_i + (\sum_{i=1}^{m} d_{i\,1}\, C_i)\, x_1 + (\sum_{i=1}^{m} d_{i\,2}\, C_i)\, x_2 + \ldots\ldots + (\sum_{i=1}^{m} d_{i\,U}\, C_i)\, x_U$$

If the inequality combination must be negative then the cost function is maximised to find the maximum constant for a negative combination to provide the most profitable final test. Similarly, a minimum positive value of the cost function (found by maximising the negative cost function) is required, if the inequality combination must be positive.

If an optimal, feasible solution is found , the final test is performed using the optimal value of the cost function.

6.17 Knowledge Acquisition And Inference.

A desirable and essential requirement for an efficient dependence graph to be constructed is the use of all knowledge that can be gained from the source code and additionally from the user. This knowledge can be used to supply extra information to the dependence tests and reachabillity sections described earlier.

The reachabillity of other statements from a dependence source statement is determined by depth first search of the control flow graph (see section 6.9). When an IF or DO statement is encountered on the search path, previously acquired information can be used to determine if a definite result of the statement enables only one of the exit edges in the control flow graph to be taken or, if no definite result can be found, both paths must be taken. If both exit edges are taken then the taking of an edge and the subsequent paths further down the flow graph implies that an assumption of the result of the statement has been made. This assumption can therefore be used in later decisions along this path, to try to determine which path to take in the control flow graph and it can also be used in dependence tests made.

The original entry in the knowledge base (except for user provided information) is the truth of the control set of the unique calling statement of this routine, if such a call exists. As the search continues, all assumptions made about the path taken in the control flow graph are added to the knowledge base for the recursive call to continue the depth first search of the control flow graph, with this information removed when a return is made. If a definite exit edge can be determined the information that could be added to the knowledge base must already be encompassed in the current knowledge base for the definite decision to be made,

thus no knowledge can be extracted from such an occurrence.

The construction of a general knowledge base and other related areas is detailed in many texts such as Frost (1986). The knowledge base consists of two lists, a known fact list and a known expression list.

The known fact list stores Boolean expressions in clausal form where a list of lists is stored. Each element of the inner list is a Boolean variable or expression (a literal) each being involved in an OR relationship with each other. The outer list has an AND relationship between each of these inner lists. For every literal, a Boolean flag is stored to indicate the truth of the literal.

An example of a known fact list is :-

(A OR B OR C) AND (~C OR D OR ~E) AND (F) ...

The known expression list is a list of expressions whose truth value is known. The expression is stored in terms of loop and nonloop variables with the equality or inequality being altered to be relative to zero.

The process of using this information to evaluate the result of an IF statement is shown below :-

1. Copy the tree of the Boolean expression from the parse tree of the IF statement.

2. Restructure the parse tree into clausal form using the DeMorgan and Distributive law transformations.

3. Build a clause list from the parse tree.

4. For every literal in every 'OR' list do

    If literal is a not equals with a TRUE flag or an equals with a FALSE flag then replace with the set less than OR greater than, both being given a TRUE flag.

    If literal is definitely TRUE then this entire 'OR' list is TRUE and can thus be removed from the 'AND' list.

    If literal is definitely FALSE then remove this literal from the current 'OR' set. If no literals remain in the 'OR' set then this set cannot be true and overall result must be FALSE.

5. If 'AND' list is empty then overall result is definitely TRUE.

6. For every 'OR' list do

    Pass 'OR' list into inference engine to use current known fact and known expression lists to try to create an empty set, indicating overall test FALSE

7. If no definite result found, store remaining list as TRUELIST.

8. Make another copy of the Boolean expression parse tree in the IF statement, add a NOT node to the top of the parse tree.

9. Repeat stages 2-6 using the negated parse tree, any definite TRUE or FALSE result is negated.

10. If no definite result found store remaining list in FALSELIST.

The process terminates as soon as a definite result is found. The process is performed twice to allow TRUELIST and FALSELIST to be evaluate as these contain the information to be added to the knowledge base when a path is taken and an assumption is made. The two passes also allow the inference engine in stage 6 to determine either a FALSE result for the actual Boolean expression or, in the second pass, a FALSE result for the negated expression, thus indicating the truth of the original expression.

The inference engine in stage 6 uses the current knowledge base to try to prove FALSE the input 'OR' list. The inference engine works by combining the 'OR' lists from known fact list with the input list in all possible ways to try to form an empty list, thus indicating that the input list is FALSE. Two 'OR' lists can be combined if one literal from each list contradict each other, forming a resolution. The contradictory literals are then removed and the remaining list joined.i.e.

(A OR B OR C) forms a resolution with (~A OR D OR E)

Where A and ~A are contradictory literals giving the new list

(B OR C OR D OR E)

A pair of expressions are considered contradictory when, by adding the expression from the 'OR' list in the known fact list to the known expression list, the literal expression we are testing can be proved false in the inequality test described in section 6.16. This is valid since the contradictory literal technique uses the fact that one or other of the literals must be false, thus if it can be proved that all other literals in the two 'OR' sets are false then one of the two 'OR' sets must be false. The proving false of the literal being tested also indicates that either this literal or the literal from the knownfact list must be false i.e.

P -> ~Q therefore ~P or ~Q

and the remaining literals in the 'OR' sets can be combined in the same way as contradictory Boolean literals.

The addition to the knowledge base is made when an assumption is made in taking an edge in the control flow graph. The information to be added is either the TRUELIST or the FALSELIST built during the test of the IF statement. The process of adding this information runs as follows :-

1. Remove any Boolean expression literals from the list where that literal is the only literal in an 'OR' set, adding the expression to the known expression list (since every 'OR' set is assumed true).

2. Add the remaining list to the known fact list.

Information can also be extracted from a DO loop. Whenever a DO loop head is reached, two alternative paths can be taken, one entering the loop, the other skipping the loop entirely. When the level of testing during reachabillity determination is not at the level of the current loop then a test is made on the loop upper limit. If the upper limit is greater than or equal to one then the loop is definitely entered, thus no information is added and only the one route is taken. Similarly if the loop high limit is definitely less than one then only the loop skipping route is taken. At the level of this loop, control has been passed from inside this loop, thus the original value of the upper limited forced loop entry, but now in subsequent iterations, both routes are possible.

When a test is required, the inequality testing algorithm is used. However, the nesting of the DO loop statement is temporarily altered with the last element of the nesting list which refers to this loop removed. This must be done to stop the inequality test using the information that we are in this DO loop already, thus using the inequality that is being tested as known information. If the inequality test does not produce a definite result then both paths must be taken, the appropriate assumption about the value of the loop upper limit being added to the known expression list.

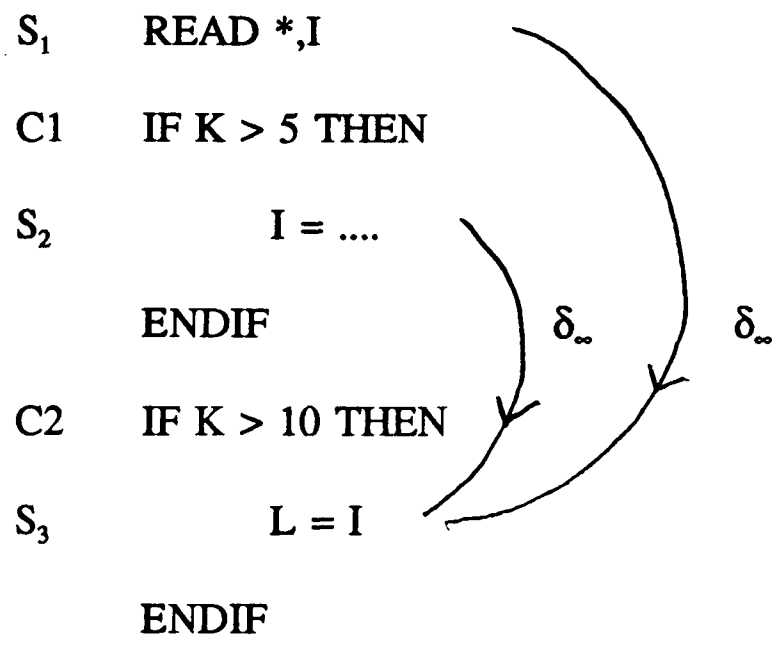
As mentioned in section 6.16, the known expression list can be used in the inequality testing routine to improve future route decision tests and dependence tests.

The use of the knowledge base clearly requires the construction of the scalar dependence

graph, thus the scalar dependence graph is originally constructed without the use of the knowledge base. To improve the quality of the scalar graph before it is used in array dependence analysis, control information is used to determine pre and post domination relationships using the control sets of statements concerned and thus allowing redundant dependencies to be identified and eliminated.

If the control set of a dependence sink statement is a subset of the control set of the dependence source statement then control passing through the source implies that control will pass through the sink. Let C1 and C2 be the control sets of two statements linked by a dependence. To test if C2 is a subset of C1, ~C1 is added to the knowledge base and C2 tested. A false result indicates the C2 is a subset of C1.

Consider the following example :-



The control set C2 is a subset of control set C1 i.e. if C1 is true then C2 must be true (i.e. assuming C1 to be false with K <= 5 proves C2 is false since K < 10). Therefore statement S_2 predominates statement S_3 with the value of I used in S_3 always being that defined in S_2.

As a result the true dependence $S_1\ \delta_\infty\ S_3$ does not exist and can be eliminated.

6.18 Inter-Procedural Dependence Analysis.

To extract more information from the source code, interprocedural analysis can be used to break down routine boundaries. As mentioned in section 6.12, scalar variables referenced in call statements are marked as used or assigned for dependence testing by a simple check of the start and stop nodes of a called routine. Similarly, as shown in section 6.13, scalar variables are searched up the scalar dependence graph, through routine boundaries, to replace them with the set of variables used to evaluate them. Thus scalar variables do not have routine boundaries as a major problem.

A similar approach could be applied to arrays referenced in a called routine, with the entire array being accessed if any element is accessed. A far more effective approach, however, is to extract the index information from the called routine, adjusting that information to match the array declaration in the calling routine.

Consider the model of such an array access in a called routine shown below :-

SUBROUTINE Caller (...)

DIMENSION $A(c_1:c'_1,c_2:c'_2, \ldots\ ,c_m:c'_m)$

. . . . . .

CALL Called (.... $,A(x_1,x_2, \ldots\ ,x_m)$, ...)

. . . . . .

END

SUBROUTINE Called (.... ,B, ...)

DIMENSION $B(k_1:k'_1,k_2:k'_2, ... ,k_n:k'_n)$

. . . . . .

$B(y_1,y_2, ... ,y_n)$ = ...

. . . . . .

END

To convert the indices of the access to array B in routine Called to be used as an access of array A in routine Caller, the following is used :-

If $n = m$ and $c'_i - c_i = k'_i - k_i$ $\quad i = 1(1)m-1$ then

If A has indices in CALL statement (i.e. if x's exist) then

$$Index_i = y_i - k_i + x_i \qquad i = 1(1)m$$

else

$$Index_i = y_i - k_i + c_i \qquad i = 1(1)m$$

endif

else {dimensions are not identical, map to one dimensional array}

If A has indices in CALL statement (i.e. x's exist) then

$$Index = \sum_{i=1}^{n} \{(y_i - k_i) \prod_{j=1}^{i-1}(k'_j - k_j + 1)\} + \sum_{i=1}^{m} \{(x_i - c_i) \prod_{j=1}^{i-1}(c'_j - c_j + 1)\}$$

else

$$Index = \sum_{i=1}^{n} \{(y_i - k_i) \prod_{j=1}^{i-1}(k'_j - k_j + 1)\}$$

endif

endif

If the declarations in the two routines are declared with the same number of dimensions and

all but the last pair of dimensions have an equal size, then the indices can be directly adjusted, maintaining the multi dimensional form, thus allowing separate dependence tests on each index pair. If these declarations are different, the conversion maps the index onto a one dimensional array, using the memory address relative to the first element in the array as the index. Any array reference to the same array in the calling routine that is to be tested against a reference mapped to a one dimensional form must itself be converted to one dimensional form, with only one pair of indices to be tested. Since the dependence tests can deal with nonlinear expressions, the product required in the one dimensional mapping does not in itself present a major problem. However, the mapping can lose information as many tests are replaced by a single test.

The conversion process can be used repeatedly to convert all array references in a called routine and all the routines descending from it, the mapping being performed between each adjacent pair of routines in the call path. If at any stage a one dimensional mapping is used then the array is assumed one dimensional for all subsequent conversions.

The loop nesting of the subsequent CALL statements and the loop nesting of the referencing statement can be added to the nesting of the original CALL statement, ensuring that the correct nesting order is maintained. This makes an accurate dependence test possible with the full loop nesting. The path of routines passed through to reach a reference is listed and used in the forward substitution of section 6.13 to provide the unique calling routine of each listed routine. Thus nonloop variables can be traced back to the routine currently being processed in the dependence analysis, potentially allowing their elimination with nonloop variables in other indices or loop maximums and with another reference in a CALL statement.

As with the scalar interprocedural information, external influences of arrays can be found by examining dependencies on the start and stop nodes of routines. The recursive processing of CALL statements in a called routine is only made when a dependence on a start or stop node involving the array is linked to a CALL statement.

6.19 Overview Of Analysis Algorithm.

All the components of the scalar code analysis tool have been described. The application of each component in the overall analysis is detailed in the following algorithm :-

(* Initialisation and Control analysis *)

For each routine in the input file do

Construct internal representation of code (see section 6.2)

Construct control flow graph (see section 6.4)

Construct pre and post dominator trees (see section 6.5)

Calculate control dependencies (see section 6.6)

Identify and mark loops in the routine (see section 6.7)

Construct routine by routine call graph hence evaluating routine order (see section 6.3)

(* scalar analysis *)

For each routine in strict order do

List scalar variable access information for each statement in this routine (see section

6.8)

For each statement do

For all statements reachable from this statement do

If any variable reference fits one of the dependence types then set a dependence link in the scalar dependence graph

For each dependence set do

If control sets of statements are contradictory (see section 6.17) then eliminate dependence

If control sets produce pre or post domination then remove any dependencies made redundant by this fact

Determine the external influences of this routine, creating the start and stop nodes in the dependence graph (see section 6.12)

(* array analysis *)

For each routine in strict order do

List all array variable reference information (see section 6.8) using forward substitution for all scalar variables used in the indices (see section 6.13)

For each statement do

If it is a CALL statement then repeat the following for all relevant array references, converting indices appropriately (see section 6.18)

For all statements reachable from this statement (see section 6.9) using and acquiring information (see section 6.17) do

If reached statement is a CALL statement then repeat the following for all relevant array references, converting indices appropriately (see

section 6.18)

Perform array dependence analysis for any array references that fit one of the dependence types (see section 6.10)

Determine external influences of this routine (see section 6.12)

This algorithm is the current implementation of the analysis tool. Future versions may alter the algorithm by removing certain features from the body of the analysis, instead incorporating them after the initial analysis is complete. This alteration may be required to reduce the execution time of the analysis with a view to including further extensions to the analysis system (see section 7.3).

6.20 Closure.

In this chapter, the features of the serial code analysis toolkit have been described. The additional features introduced enable a thorough analysis of the serial code to be performed extracting and using a large amount of information implicitly exhibited by the code. Further extensions to the serial code analysis system will be discussed in the next chapter, including the essential addition of user interaction.

Demonstrations of the power of the analysis are given in the following chapter, as well as an example of the analysis of a larger scale application code.

The construction of an accurate dependence graph allows the code generation and optimisation application as described in chapter 5 to be used effectively. In particular, the criteria that the

use of computer aided parallelisation should not detrimentally effect the efficiency of the parallel code produced is a more realistic goal with the more accurate dependence graph provided. The generation of parallel code and other relevant issues are also discussed in chapter 7.

CHAPTER SEVEN

7. Test Case For Serial Code Analysis Toolkit And Future Work.

7.1 Introduction.

In this chapter, the serial code analysis tool described in the previous chapter is demonstrated on FORTRAN source codes. The first set of test cases show small examples explicitly illustrating code sections where the extensions to the standard analysis system are essential. The second section shows the application of the analysis tool to a package developed with no view to parallelism.

The code generation algorithm used is only simple adjustment of DO loops identified as parallel in an examination of the dependence graph, to DOALL loops. As such, the first set of examples have been carefully selected, where a key dependence which would serialise the loop is eliminated when the analysis extensions are used. In the application program example, the number of dependencies detected are given.

The remainder of this chapter is devoted to a discussion of the future extensions of the analysis tool, the building of code alteration/restructuring tools and other issues relevant to using computer aided parallelisation.

7.2 Test Cases Of The Serial Code Analysis Tool.

The analysis code has been tested for a range of test cases, some of which are discussed below.

7.2.1 Simple Code Section Examples.

The following test cases demonstrate parallelism detection that required the use of one or more of the extensions to the basic serial code analysis system. In each case, the decisions involved in eliminating the key dependencies will be examined in the context of the analysis features that enabled that decision.

7.2.1.1 Example 1 - Interprocedural Dependence Testing.

The following example has two nested loops, but in different subroutines. The dependence analysis must pass through the routine boundaries for an accurate analysis.

```
      DIMENSION A(*)
      READ*, M , N
      K = N
      DO 10 I = 1 , M
10          CALL Sub(A(N*(I-1)),K)
      END
      SUBROUTINE Sub(B,L)
      DIMENSION B(*)
      DO 20 J = 1 , L
20          B(J) = . . .
      END
```

The test must determine if loop 10 can be performed in parallel. The test must thus be for a dependence between calls to routine Sub. The interprocedural analysis described in section

6.18 tests instances of the assignment to variable B in routine Sub. B is mapped to variable A with the appropriate adjustment in index. The nonloop variable L in routine Sub is substituted by the variable N with the READ statement as the defining statement (see section 6.13). Also the loop 20 is added to the end of the nesting information of the CALL statement. The resulting dependence test is then the same as that in the example in section 6.15, thus the extended Banerjee test gives a no dependence result.

This result indicates that the assignments performed by each call to Sub access separate sections of the array A. If the value of K were set to N+1 then the sections of array A accessed in routine Sub would not be separate (i.e. the last element accessed in one call is the first accessed in the next call). This should, and does, serialise the loop when the analysis is repeated.

This example demonstrates that the interprocedural analysis removes many of the barriers introduced by routine calls in dependence testing.

7.2.1.2 Example 2 - Control Information And Exact Dependence.

The following example displays the characteristics of a code section in a finite element code. The inner loops perform calculations for either quadrilateral or triangular elements in two dimensions. The outer loop is the element loop which is where the most profitable source of parallelism should be found.

```
      DO 10 I = 1 , N
      IT = TYPE(I)
      IF (IT.EQ.TRIANGLE) THEN
            DO 20 J = 1 , 3
S1                A(J) = . . .
 20           CONTINUE
      ELSE
            DO 30 J = 1 , 4
S2                A(J) = . . .
 30           CONTINUE
      ENDIF
      . . . .
      IF (IT.EQ.TRIANGLE) THEN
            DO 40 J = 1 , 3
S3                . . = . . A(J) . .
 40           CONTINUE
      ELSE
            DO 50 J = 1 , 4
S4                . . = . . A(J) . .
 50           CONTINUE
      ENDIF
 10   CONTINUE
```

where TYPE is an array of values representing element types and TRIANGLE is set in a parameter statement.

A dependence analysis of this code without the extensions introduced in chapter 6 would, for example, set a dependence of S_4 on S_1 with true dependence carried by the outer loop (level 1). Clearly, this dependence cannot exist since the values used in S_4 are all assigned in S_2 in the same iteration.

This assumed dependence is removed by the extended features in the following way. Firstly, the loop independent true dependence of S_4 on S_2 is marked as exact in the dependence test (see section 6.11), this implies that the values used in the sink of the dependence are assigned in the source. Secondly, use of the knowledge base is made to determine that the control set of the sink of the dependence is a subset of the control set of the source (clearly true since the control sets are equal). This indicates that the values used in S_4 must have been assigned by S_2 in this iteration. With this information, all other true dependencies with a sink on statement S_4 can be removed. Other assumed dependencies are eliminated in the same way, thus removing all true dependencies carried by the outer loop, and many pseudo dependencies.

Parallel execution of the loop is now possible, although some loop carried pseudo dependencies still exist. In the case of shared memory systems, variable expansion (see section 5.5.1) is required, making 'A' a two dimensional array, in order to make parallel execution legal. For distributed memory machines, 'A' can be considered as a 'pre fetch' variable (see section 5.4.3), since it is not involved in a true dependence carried by the outer loop, and thus pseudo dependencies do not inhibit parallelism.

7.2.2 Application Code Example.

The code used for this example is a two dimensional conduction simulation code, where the solution method is a combination of finite element and control volume techniques. It was developed with no view to parallelism and exhibits many common code characteristics. It consists of 57 routines with over 3000 lines of code. The call graph for the code is shown in figure 7.1 with a close-up of routine CONSYS and its descendants in figure 7.2.

The code generation in the current version of the analysis code is fairly crude, only being required for limited inspection of the dependence graph through the generation of DOALL loops. As a result, for the testing of the effectiveness of the analysis system, the number of dependencies detected is used to judge the success of the analysis features. Clearly, the fewer dependencies set the greater the flexibility in code generation and parallelism detection etc. The reduction in dependency numbers is due to assumed dependencies being proved non-existent by the extended analysis features introduced in chapter 6.

The extensions are used in various combinations to display the effectiveness of each method and how they interact. The extensions are grouped into five categories :-

Knowledge	-	see section 6.17
Disproofs	-	see section 6.16
Interprocedural	-	see section 6.18
Exact	-	see section 6.11
Scalar Equality	-	see section 6.14

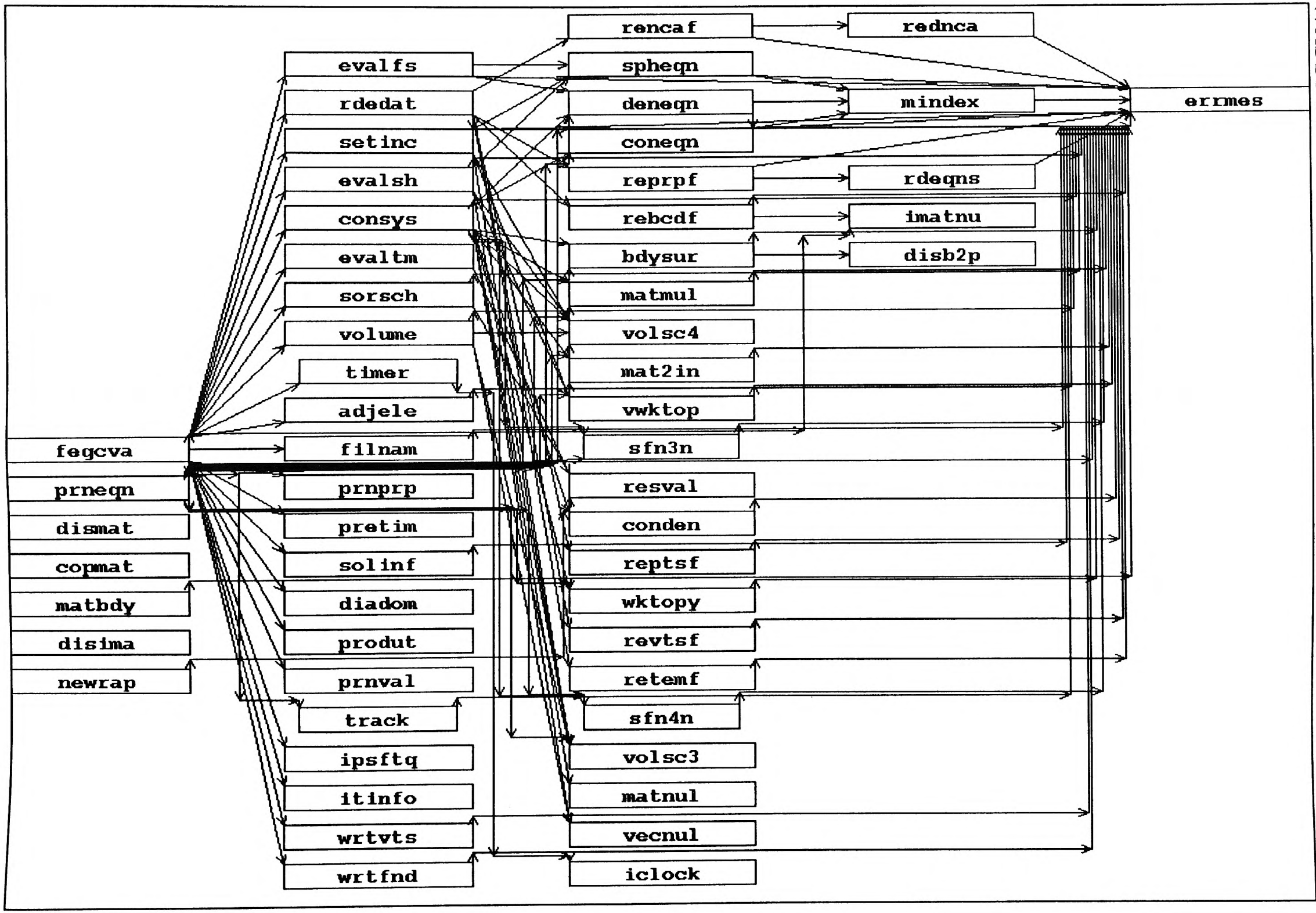


Figure 7.1 Call graph for Control Volume/Finite Element Code (main proram is FEGCVA)

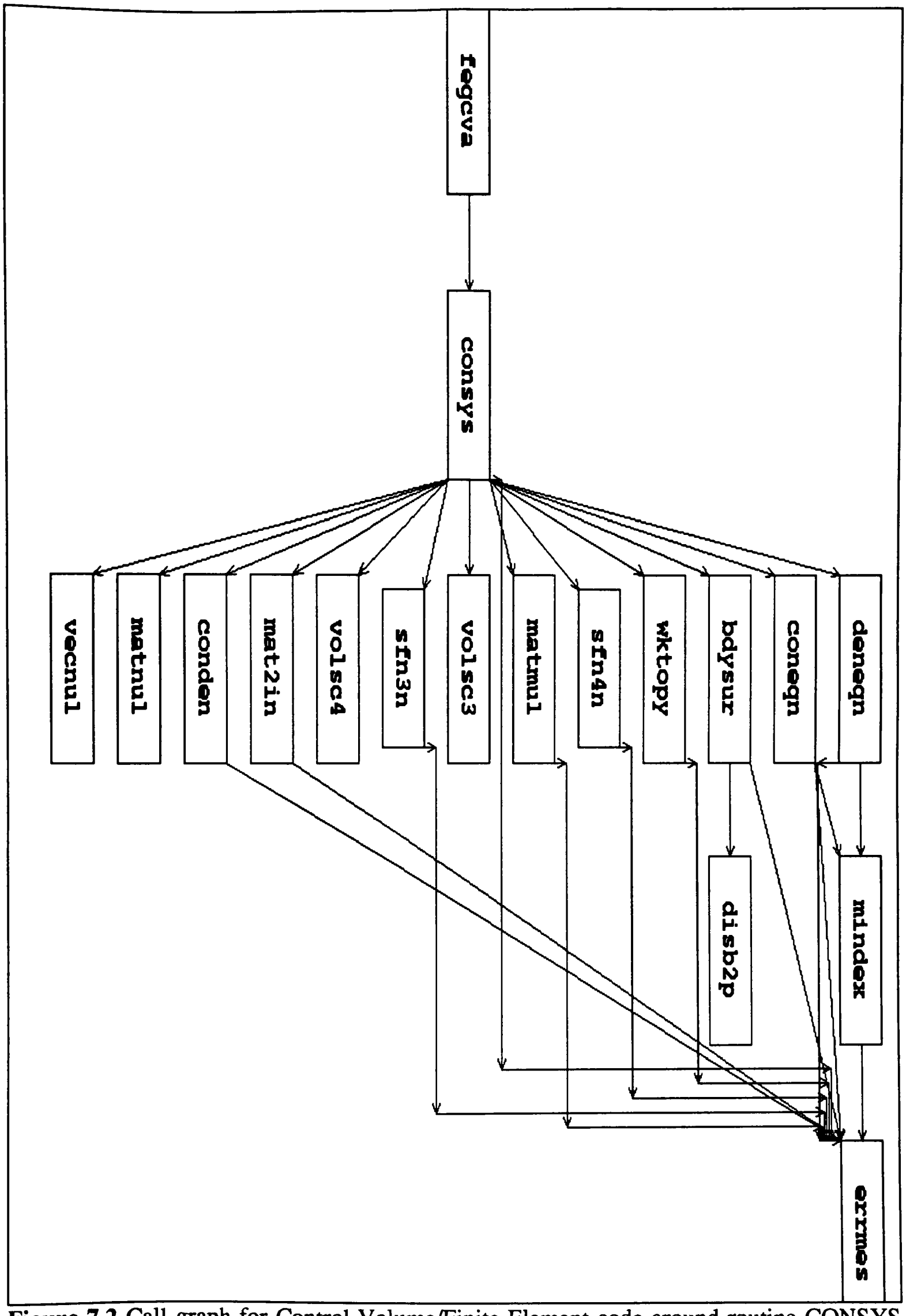


Figure 7.2 Call graph for Control Volume/Finite Element code around routine CONSYS

The use of scalar equality alone does not eliminate any assumed dependencies. The use of exact dependence evaluation alone produced a 4% reduction in the number of dependencies. This can mainly be attributed to determining exact dependencies between array accessing statements where the indices are purely loop variables (as is common in the application code), thus eliminating now redundant dependencies. The combination of scalar equality and exact dependence detection achieved an extra 0.7% reduction. Clearly, all the dependencies eliminated are array dependencies since all scalar dependencies are always marked as exact. The routines where dependencies were eliminated were predominantly the larger routines with the more complex code structures.

All the other extensions rely heavily on scalar equality and are therefore only used in combination with it.

The use of equation disproofs in addition to scalar equality eliminates no assumed dependencies. Also, the addition of exact dependence calculation to this combination produced only a very slight gain over the 4.7% reduction achieved before.

Using the knowledge routines with scalar equality and exact dependence, as with equation disproofs, produces only a negligible reduction in dependence numbers.

A far more significant impact is produced when the knowledge base and equation disproof extensions are used in conjunction with the scalar equality and exact dependence tests. This combination produced a 9.6% reduction in the number of dependencies set. This is predominantly caused by two effects, firstly, by the control flow information in the knowledge

base being used in dependence tests through the equation disproof facility, and secondly, by the improved accuracy of the reachabillity determination by the use of the equation disproof facility to compare control flow statement Boolean expressions involving integer comparisons etc. The removed dependencies are between both scalars and arrays and are mainly in the main routine of the solution phase of the code (routine CONSYS in figure 7.1 and 7.2).

Finally, the interprocedural feature was added to incorporate all extensions in the analysis. This gave a reduction of 12.8% in the number of dependencies set. The dependencies removed by adding the interprocedural feature were largely array dependencies from the main program (FEGCVA), since this routine calls all others.

The total number of dependencies eliminated from the dependence graph with all the extensions active is 495 out of an original total of 3869. This significant reduction in the total number of dependencies allows for, as hoped, more flexibility in both code generation and parallelism detection. It also makes any human inspection and interpretation of the dependence graph far easier, which, as discussed later in this chapter, is an important requirement of the parallelisation system.

The use of total dependence numbers may, however, have underestimated the impact of the analysis extensions. A manual inspection of the code determined that one routine, routine CONSYS in figure 7.1, was central to any parallelisation of the code since it contained calls to all the main sections of the finite element/control volume calculations. This routine, along with its descendants, also dominated the runtime profile of execution time for the code, indicating its importance in parallelisation. As mentioned in chapter 5, true dependencies have

the largest influence on the parallelism that can be extracted for use on distributed memory machines, thus the number of true dependencies is a very important measure of the usefulness of the analysis. The number of true array dependencies set in routine CONSYS were reduced by over 30% (53 dependencies out of a total of 172). The true significance of this is shown by the number of dependencies preventing parallelism in the element processing loop which contains almost all the code (including routine calls) in CONSYS. Using the analysis extensions removed twenty of these serialising dependencies allowing some actual serialising dependencies to be identified (where these dependencies require code alterations to allow parallelism).

The use of the analysis extensions does incur an increase in analysis execution time, rising from around 2 minutes with no extensions, to around 20 minutes when all extensions are employed, with the use of knowledge and inequality disproofs together incurring the most significant time cost.

7.3 Further Extensions To Analysis System.

The analysis techniques shown in chapter 6 provide a powerful analysis of a serial code producing a fairly efficient dependence graph. The following section indicates the extensions that are to be included in the system to enable more information to be extracted from the source code and used to produce a more efficient dependence graph.

Loops other than DO loops (i.e. Repeat and While style loops) are identified as described in section 6.7. Unlike DO loops, these loops do not have a loop counter variable, thus any

induction variables in these loops are considered as nonloop variables. To allow more accurate tests, non DO loops can have an iteration counter variable explicitly added, running from one in increments of one. This variable can then be used to convert induction variables into functions of the new loop counter allowing dependence tests to treat the loop counter in the same way as a DO loop variable but with an unknown upper limit.

Extra information on variable changes between iterations of a particular loop can be used in the inequality tests of section 6.16. When a test is made between statements for a loop carried dependence, the scalar variables in the indices of one of the array references are in an earlier iteration of the loop concerned than those in the array indices of the other statement. A variable common to both indices will be treated as a different variable in each case if a common defining statement for the variables is nested inside a surrounding loop. If a relationship between the two instances of the scalar variable can be found then this may provide the information required to perform a successful test. This relationship could be that the variable is increasing or decreasing in this particular loop (i.e. the variable definitely increases in every iteration or definitely decreases), the minimum size of the inter iteration change can also be profitably used if available. i.e.

```
      J = 5
      DO I = 1,N
            J = J + 1
            K = K + J
S           A(K) = . . . . .
```

Let K be the value of K in an earlier iteration and K' be the value of K in a later iteration.

The test for a self output dependence of statement S carried by the I loop will test K against K'. Without any information a dependence must be assumed, however examining the code reveals the following relationship :-

$$K' - K >= 6$$

i.e. the value of K in each iteration is at least 6 less than the value in the following iteration. This information can then be incorporated in the matrix set up in the inequality test (see section 6.16) and used to prove that no dependence exists in the above example.

Such information can be inherited from one statement to another i.e. in the previous example, since J is initially positive and increasing, K must also be increasing. Work on using this type of information about statements in loops involving loop variables has been performed by Brandes (1988).

Non-linear variables are stored as single nonloop variables as described in section 6.14. When such a variable occurs in an inequality test either directly from the given expression or through forward substitution, these variables may create a problem in the inequality test. More information can be extracted from the existing known expression set in the testing routines by combining several known inequalities to form a new inequality containing the non linear variable we wish to eliminate from the original inequality. For example:-

Combine $N + M + 2 <= 0$

and $N + 1 <= 0$

Giving $N^2 + NM + 3N + M + 2 >= 0$

The combinations form new equations which are included as columns in the matrix formed in the inequality test of section 6.16. They do, however, often introduce new not required non linear variables into the matrix which cannot be eliminated, causing the combination to be of no use unless further combinations are included for this new variable. An infinite number of combinations exist thus the process must be restricted to introduce the minimum number of new variables with, possibly, a maximum total number of combinations allowed. The process of adding inequalities and combinations of inequalities to the matrix can proceed in an iterative manner where an inequality is only added when some of the variables it includes are already present in the matrix. The identification of all relevant combinations of inequalities can be achieved by a recursive procedure examining all possible combinations.

The forward substitution described in section 6.13 is an essential feature of the analysis. The current implementation only performs substitution when a unique true dependence is found, thus conditional assignments form a barrier to substitution. The ideal system would pass through such barriers but require multiple tests for all possible values of variables involved in conditional assignments. The mechanics of such a substitution must be carefully considered to test all possible combinations of variable substitutions, eliminating all those that contain contradictory control sets.

Such a process would be extremely time consuming but has the major advantage of often substituting variables back to READ statements since these are the only statements that represent an inherent barrier to substitution. Information required from the user to aid

dependence tests (as described in the next section) will then usually involve questions about variables in READ statements which are program input variables with which the user will often be familiar (as opposed to internal program variables).

7.4 User Interaction.

User interaction is an essential component of a useful analysis system. The user input into the system is required during the initial analysis phase when unknown variables prevent a decision in the inequality test or when an array is used as an index to another array. A dependence test can be proved definitely true or definitely false, however, if no such certain result can be obtained, additional information about the variables that remain causing the uncertainty can be requested. Index arrays often consist of distinct values (i.e. as in finite element codes). This information can be used to disprove dependencies since if the indices of the index array in the two statements are definitely different then the values of the index array are definitely different and a dependence of the outer array is disproved.

The user information can be provided before the analysis starts if the user considers that the information may prove important and be useful in many cases. The second set of user information can be requested by the system after the initial analysis. The number of requests for information must be restricted to a reasonable number, however, many uncertainties may be caused by one variable, allowing questions to be ordered from the potentially most useful information down to information on variables only used in one statement. Some assumed dependencies do not represent a major problem, particularly pseudo dependencies that do not inhibit parallelism, thus requests for information to eliminate such dependencies need not be

performed.

Requests could be made for parallel preventing dependencies displaying the result of the Banerjee inequality in as simple form as possible. The user can then instruct the system as to the truth of the inequality (i.e. true, false or uncertain) or provide information about a single variable or subset equation of the inequality.

User information can be stored in the known expression list of the knowledge base (section 6.17) and be used in the matrix set up in the inequality test (section 6.16). Assumed dependencies can be re-tested at any stage as new information is acquired.

User directed interaction can be used to examine serial loops and inquire as to why a loop has been indicated as serial. This may be performed before or after the optimisation phase to identify dependencies that cause a loop to be serial and examine each of these dependencies to try to prove some of them false.

7.5 Optimisation application And Data Partitioning.

The analysis phase can be automated to a very large extent with a minimum of user interaction without a degradation in the quality of the resulting dependence graph. The second phase in the parallelisation process involves parallel code generation using the information held in the dependence graph. This inevitably requires a large number of the code transformations shown in section 5.5 to be applied and many other decisions to be made about how best to exploit the parallelism identified. Although the user's role in this process can be minimised using heuristic techniques (Polychronopoulos 1988, Sarkar and Hennessy 1986) it is desirable for the user to play a major role if efficient parallel code is to be the result.

The extreme options for optimisation application are fully automated application and fully manual. The problems with each extreme are highlighted below.

For fully automated application, many optimisations will be used to eliminate cycle forming dependencies. Without user control, a large proportion of loops will be altered in an effort to enable their parallel execution, whether or not the final code can exploit this parallelism. The optimisations will often be applied many times to a single code segment causing the final code to be unrecognisable to the user. This may inhibit the interrogation of the system by the user after the optimisation phase to inquire why certain loops have remained serial. The fully automated algorithms for vector and shared memory code shown in chapter 5 perform loop distribution and code re-ordering which can create code vastly different form the original. The code produced often requires extra user investigation to improve its parallel performance but since the code is unrecognisable to the user it may be necessary to relate the new code to the

original (possibly by indicating the optimisations used) before the required information can be provided.

The other extreme of fully manual optimisation will usually be a very time consuming process. Without tools to perform the mechanics or to check the validity of optimisations, the resulting code may be incorrect due to human error. Even with such tools, the number of optimisations required in large software still requires a considerable amount of user time.

Since both extremes have undesirable features, a combination of them should be used. An effective system could present the user with every serial or DOACR loop identified in the code and ask if that loop is desired to run in parallel. A response indicating parallelism is desired could lead to either an automated optimisation process to attempt to produce a parallel loop or a semi automated process, guided by the user, to select a set of the optimisations that can validly be performed.

For distributed memory systems using a data partition, the partition chosen has a major influence on the optimisation process. For a given partition, only certain loops in a nest will be able to exploit parallelism between the partitioned data sets thus only the appropriate loops need be considered during the optimisation phase. This suggests that the details of the data partition should be determined before the optimisations commence. Alternatively, the choice of a partition may be influenced by the loops which can be executed in parallel, the partition being chosen to enable exploitation of that parallelism. This requires parallelism to be known prior to partition definition, thus optimisations to extract parallelism should be performed before the partition is chosen.

This indicates that the choice of a data partition and the application of optimisations are interrelated. The choice of a partition is, however, also a process that can benefit from information provided by the parallelisation system.

Loops which assign to large arrays are clearly candidates to exploit any parallelism that they may contain if the array(s) concerned is partitioned appropriately. The index of such an array will often be a linear function of loop indices, thus each iteration of a surrounding loop may assign a unique element of the array. Loops not containing such code can, perhaps after user consultation, be eliminated from the optimisation process (at least at early stages), since it cannot exploit any data partition to produce parallelism.

When optimisations of the selected loops are completed, information on which loops have been successfully converted to parallel form and information on which arrays need to be partitioned to exploit this parallelism can be presented to the user to aid the final data partition decision. If a log of optimisations has been kept it may then be possible to "undo" optimisations which were used to extract parallelism from loops which cannot be exploited for the chosen partition.

7.6 Practical Code Considerations.

The identification and extraction of parallelism at compile time is a major procedure in the full parallelisation process. To create actual parallel code, hardware considerations have a further important influence.

7.6.1 Vector Code.

For vector code, vector statements involve vectors of a length usually determined at run time. The hardware that performs the vector operation may have a maximum vector length that can be processed in one instantiation of the operation due to memory limitations or the number of processing elements available.

To facilitate large vectors to use such a vector processor a process known as strip mining can be used. A single vector operation is replaced by a loop containing the same vector operation with the maximum vector length for the operation allowable in the vector processor. The loop iterates over consecutive sections of the original vector where each of these sections is of the maximum vector length. When only a number of elements of the vector less than the maximum length remain the loop is exited and a final vector statement for the remaining elements is performed. Strip mining can be applied as a program transformation involving a large increase in code size or it can be incorporated in high level vector routines which are called in the code instead of low level vector processor routines (Ierotheou 1987).

Another machine dependent parameter that influences when a vector statement can be profitably used is the minimum vector length for a given operation required to produce speedup. Vectors with a length smaller than this minimum may produce slowdown since the small amount of parallelism exploited may easily be more than offset by the overhead of setting up the vector operation. To overcome this difficulty, a run time decision is required. The final section of the vector that remains after the strip mining loop can be tested against the minimum vector length required for the operation concerned, if the remaining length is

greater then a vector computation is used, if not a scalar computation is used. This test can easily be incorporated in high level routines thus not requiring the user/code generator to have specific knowledge of the characteristics of the operation and machine concerned.

7.6.2 Shared Memory Systems.

The shared memory accessible by autonomous processor's enables a great deal of flexibility in exploiting parallelism. Any code section can be executed on any processor determinable at run time with all compile time unknown variables set. A control system can easily allocate any code section to be performed on any processor with any data set. This is possible since the code is stored in global memory thus to force a particular processor to execute a given code section all that is required is to set that processor's program counter to the start address of the code section. This system is used by Polychronopoulos (1988) to allow all code sections that are determined as ready for execution (from the dependencies on other code sections) to be listed in a pool of ready tasks. As soon as a processor becomes free it can collect another task from the pool thus idle time is eliminated as long as enough ready tasks are available at all times.

7.6.3 Distributed Memory Systems.

The code and a large amount of the data a processor will use in its execution are stored in the processor's local memory. If a similar system of allocating both tasks and associated data areas as used in shared memory systems is to be used then both the code and the data may have to be transferred through a potentially long interprocessor path. Even if all code is held

on every processor with a mechanism for dynamically determining which code sections execute, the data will still need to be transferred,often producing excessive communication times (i.e. as in a processor farm). As a result the flexibility exploited by the shared memory systems is usually not practical in distributed memory systems. With both code and data placement being predominatingly decided at compile time it is essential to have an efficient analysis of the code to extract parallelism which will be able to keep all processors active. Load balancing can then be achieved by dynamically adjusting the partition as described in the following section.

7.7 Practical Considerations Of A Data Partition.

An ideal application for a data partition would contain loops where the indices of the arrays involved are simple linear functions of loop variables with no loop conditional code contained (loop conditional code being code where the truth of the condition that controls its execution can be different in each iteration) exists. With this type of application, efficient parallel code may be possible with a static placement of partitioned data on each processor. The placement of partitioned data over the processor network may be decided by examining the usage of variables in the parallel loops and/or by knowledge of the nature of the application (ex. the control volume computational molecule).

In the case of finite element code, the partition can be made between sets of elements. The data placement then requires elements to be placed over the processor network to minimise the required communication in the finite element analysis. This involves placing each element on a processor where all directly connected elements are either placed on the same processor

or on a near neighbour in the processor network. To do this, the elements can be sorted into a topological order of their interconnections before partitioning, then partitioning and placing the elements in a manner determined by the ordering. This process can be performed after the initial input phase of the application with the partition information and relevant data sent to processors before initial field data etc. has been input and before the finite element computation commences. Although the pre-sort and efficient placement operations are not essential, the communication times involved in a randomly partitioned and placed finite element code will be excessive with many communications involving distant processors.

If the partition does not produce balanced computation throughout the processor network the resulting execution will be inefficient. This can easily occur in loop conditional code which cannot be accounted for in the compile time partition. To overcome this, the partition can be allowed to dynamically adjust the assignment areas on processors to redistribute the processing load during program execution.

The unit of partition must be identified to allow a dynamic partition. In the line based partition used in chapter 3, a line of control cells can be considered as a unit of partition, in chapter 4, a slab of control volumes can be used, whilst in finite element codes, an individual element can be used. A redistribution of the computational load can then occur by transferring one or more of the units of partition from one processor to another with all assignments to those units now being performed on the new assigning processor only. The choice of which units of partition to transfer should be made to maintain the minimised communication feature of the original partition.

The decision of when unbalanced computation is occurring can be made by timing appropriate phases of the calculation (ignoring idle time) on every processor. If the difference between the largest and smallest times is above a given tolerance (related to the calculation time of a unit of partition) then partition redistribution is required. The calculation phases that are timed can be outer loops of the code where each iteration performs a major computational task (for example, an iteration of the SIMPLE algorithm in FLOW3D). The transfer of units of partition and subsequent re-organisation of data on all processors involved can be achieved by additional routines, however, for efficient operation, it may be necessary to redesign the data structures used in the code.

A dynamic partition of this type is being implemented by Leggett (1990) for adaptive finite element analysis based on a SIMPLE type algorithm. An initial partition is determined for a coarse grid which is then automatically refined. Elements are created on every processor and transferred as required. An application of this code where the dynamic partition should prove invaluable is a solidification model. A fine mesh of finite elements will be required around the solidification front with a coarser grid sufficient in many other areas of the domain. Therefore as the solidification front moves the computational effort moves with it thus a continual redistribution as elements are created and destroyed will be required.

7.8 Use Of Distributed Memory Systems For Other Code Types.

To examine code that can exist that does not display the parallelism or does not fit the local calculation feature best exploited on a distributed memory system with a data partition, consider the following examples of matrix type operations.

7.8.1 Tridiagonal Matrix Systems.

Consider a matrix equation $A\underline{x} = \underline{b}$ where we wish to find the solution to the x vector. If A is a tridiagonal matrix then the Thomas algorithm can be used to obtain a solution. This algorithm is, however, inherently serial since it uses recurrences in its operation which therefore create true dependence cycles in the dependence graph.

In the line based solutions in chapters 3 and 4, tridiagonal matrix systems could be solved in parallel (if Jacobi style updates were used). The parallel operations were performed between sections of the matrix referring to different lines of control cells in the control volume mesh. Since each such matrix section has a first and last cell, the row referring to these cells will contain a zero in the appropriate off diagonal element (i.e. the east end cell has no influence from the east thus the east coefficient in the matrix will be zero). These zero elements have the effect of breaking the recurrence in the Thomas algorithm in both elimination and back substitution phases allowing the algorithm to perform independently for every matrix section representing a line of control cells.

The serial coding of the Thomas algorithm could have been performed without exploiting these breaks in the recurrence. Such code would not be amenable to parallelism detection in an analysis system but the system could indicate to the user that the code is inherently serial then relying on the user to recode the algorithm exposing the potential parallelism.

For general tridiagonal matrix systems, the parallel version would require the solution to be achieved using a method more amenable to parallel execution such as an iterative solution,

cyclic reduction (Hockney 1965) etc.

7.8.2 Full Matrix Multiplication Computation.

A matrix - matrix multiplication of full matrices inherently contains a large amount of potential parallelism. The evaluation of each element in the result matrix involves the same calculation time therefore balance computations can easily be achieved by dividing the result matrix evenly over processors. The problem arises since the calculation is not local with every element of each matrix being combined at some point in the calculation. Regardless of the data partition strategy used for each matrix, a large amount of communication will be required, much of it between distant processors.

The communication can be achieved using a communications harness transferring individual elements of the matrices as required, however, this will entail an extremely large amount of communication. This communication can be reduced for the partition used by carefully considering what data to communicate at what time. If a block row partition is used for the first matrix and for the result matrix (with the equivalent rows being partition for both matrices) then the assigning processor for a row of the result matrix will also contain the row of the first matrix required in the calculations, thus only the second matrix need be received. The mechanics of an efficient implementation of this type can, for example, broadcast (i.e. pass the data to all processors from the assigning processor) each column of the second matrix in turn to every processor, the calculations providing the result matrix values for that column being performed on all processors before the next column is broadcast.

7.8.3 Gaussian Elimination Computation.

In Gaussian elimination, all matrix elements are involved in adjusting all others, thus producing potentially high communications times. To produce balanced computations, the operation of a parallel implementation must be carefully considered.

For a row based partition, balancing can be achieved by ensuring all processors hold an approximately equal number of elimination rows (i.e rows already used to eliminate others), thus holding similar numbers of rows still requiring adjustment (Chu and George 1986). This requires some exchanging rows between processors, inevitably involving communications to maintain this balancing, but this is often essential if even moderately efficient parallel code is to be achieved. The mechanics of the algorithm can then be performed by broadcasting the relevant elimination row to all processors, where each processor then performs elimination on the rows still requiring adjustment that it holds.

7.8.4 Efficient Distributed Algorithms.

Much research has been aimed at developing algorithms for parallel machines. Some achieve parallelism by slight adaptation of the serial algorithm such as the Gaussian Elimination mentioned above. Other parallel algorithms use methods that would be inefficient in serial but exhibit sufficient parallelism to provide speedup when sufficient processors are used. In both cases, the parallel code requires inclusion by the user in the parallel code since although the use of parallel optimisations and communications harnesses can allow automated parallel conversion, the code produced may well be highly inefficient in both communication time and

load balancing.

An alternative to writing specific parallel code when a serial algorithm is unsuitable is to use a routine from a library of parallel algorithms such as those being developed at Liverpool University. Versions of these algorithms have been produced to use a particular transputer configuration with given data partition strategies. If such a routine fits the configuration and partition which was suitable for the majority of the overall code then it can be incorporated into the parallel version.

7.9 Closure.

In this chapter, the power of the current serial code analysis code has been demonstrated. When the future developments are incorporated, along with the user interaction facilities, the analysis should give an accurate analysis of most, if not all, codes. This should allow the analysis tool to be used to build a dependence graph which can then be used to develop the parallel code with no sacrifice in efficiency being afforded in the produced code due to deficiencies in the dependence graph.

The development of code alteration/restructuring tools is clearly essential if the efficient dependence graph built is to be of any use. These tools can be constructed using the dependence graph, however, the method of their application to the input code (as discussed in section 7.5) requires further examination. At present, the most attractive solution is to supply all choices to the user, rather than forcing a decision on the user.

Despite the use of computer aided parallelisation, recoding by the user will almost certainly be required at times. The parallelisation system must therefore, also incorporate editing facilities if is to be a comprehensive parallelisation environment.

CHAPTER EIGHT

8. Conclusions

In this work, various issues in the creation of software to efficiently exploit parallel computer architectures have been addressed. The requirements to achieve such software, a technique to produce such software and demonstrations of its effectiveness for a certain class of codes, and aids to the parallel software creation process have been detailed.

Despite enormous efforts over recent years, the use of parallel computers has been extremely restricted by the lack of applications software able to exploit it to even a fraction of its full potential. The incentive for this work was to start to accelerate the rate of arrival of parallel software and thus help to introduce parallel computers to a far wider commercial audience than it could currently service.

The requirements that direct the creation of successful parallel software arise for both the user perspective and the original code developer. The code user requires fast execution, maximised usage of all available memory and also total transparency of the parallel execution. In addition to these requirements, the original scalar code developer must be able to maintain the parallel code along with any scalar versions, requiring close similarity between the scalar and parallel versions, and also portability of the parallel code over a range of parallel computers, thus restricting the use of specialist processor capabilities. The ultimate goal for the code author is a single code, maintainable by the code author, with facilities to generate parallel versions for a range of parallel computers.

The creation and success of parallel software, generated under the above requirements, has

been demonstrated for a class of code, that is structured mesh computational fluid dynamics codes, for distributed memory machines. This class of code was chosen since it encompasses a large number of codes with very high computer power and storage requirements which could benefit from parallel execution. The distributed memory class of parallel computer was chosen since it provides cost effective parallel processing, making it financially accessible to a wide audience, and since software creation for this class of machine has been extremely slow due to the added complexity of the codes required.

The first structured mesh C.F.D. code was an in-house, conduction only solidification modelling code. This code was parallelised using a data partition technique in an effort to show that all the above requirements for parallel code could be achieved by this technique. The speedup achieved, the size of problem that could be solved, the identical user input as for the scalar code and the close similarity of the parallel code to the original scaler code clearly demonstrated success. This result gave the incentive to parallelise a large scale, commercial C.F.D. code, using a similar technique, to enforce the conclusions made.

The code used was the HARWELL - FLOW3D fluid flow prediction package. This code incorporates a wide range of facilities, representing a far sterner test for the data partition strategy. As with the solidification code, the code alterations needed to generate the parallel code did not significantly disguise the original scalar code and again the other requirements for successful parallel code were satisfied. Thus the conclusions of the solidification example have been enforced, showing that the data partition technique is highly effective for parallelising structured mesh style codes.

The creation of efficient parallel software involves a time consuming process consisting of many stages. This process can sometimes be bypassed by the use of currently emerging parallelising compilers. These compilers can sometimes achieve moderate success, dependent on the code being parallelised, and, although a market for them doubtlessly exists, the requirements of code users of parallelised applications software demand highly efficient parallelisation, a demand usually beyond the range of fully automated, heuristically driven, compilers. The basic theory behind these parallelising compilers is the dependence graph and it is this theory which has been adopted in this work.

The construction of a dependence graph must clearly use conservative assumptions with any uncertainty about the existence of a dependence leading to that dependence being set to ensure the construction of a semantically correct dependence graph. These conservative assumptions may well cause parallelism to be hidden, thus causing any parallel code developed with the aid of this graph to probably be inferior to a purely manually developed parallel code. This inferiority is considered an unacceptable sacrifice, thus extensions to the analysis system creating the graph were essential if its use were to be practical.

The dependence determining tests used in the parallelising compilers are powerful. However, a great deal more information can be extracted form the code enabling many assumed dependencies to be removed. This extra information has been extracted from control flow and loop nesting information and effectively exploited by the use of knowledge inference facilities and symbolic inequality processing algorithms. The set of dependence eliminating tests and other related extensions to the analysis system developed in this work,although not in themselves certain to uncover more potential parallelism, will make the task of parallelism

detection and code transformation far easier since far fewer dependencies have been falsely set. In particular, human inspection of the dependence graph is significantly simplified with reasons for inherently scalar code not disguised by non-existent dependencies. The extra analysis components require a significant increase in execution time. However, the significant reduction in the number of dependencies set and the decreased time therefore required to make use of the more accurate dependence graph justifies this extra analysis time cost.

To gain the full potential from the new analysis features, user interaction enabling extra information to be added to the knowledge base is required. The incorporation of this information into the current analysis system is fairly straightforward, user information being treated in a similar fashion to information automatically extracted from the code in the present version.

The essential use of heuristics in the code generation process of the parallelising compilers also restricts the quality of the produced codes. This, as with the assumed dependencies, is considered unacceptable as inferior code to that manually created will result. Additionally, the development of automated parallelisation strategy determination for distributed memory systems is at an early stage, and, since this decision and many code specific code generation problems must be successfully tackled, fully automated code generation is impractical for the foreseeable future. This leads to the conclusion that, to generate parallel software fulfilling all the requirements mentioned, the code alteration and restructuring transformations should be implemented as a set of tools with, at least initially, full user control. These tools should enable parallel software to be created in a fraction of the current manual creation time, achieving the original goal of greatly increasing the software able to exploit parallel machines.

The current work is by no means complete. Many more analysis extensions could be developed to further improve the accuracy of the dependence graph produced. The proving of strategies to parallelise other classes of software is required, and subsequent development of code generation facilities for the structured mesh data partition strategy and the other strategies developed are needed. The research in this field could continue indefinitely and will probably lead to software creation of code far superior to that developed purely manually.

References.

3L Limited, 1988, Parallel Fortran Reference Manual. 3L Ltd. Livingston, Scotland.

Aho A. V. and Ullman J. D. 1977, Principals Of Compiler Design. Addison-Wesley, Reading, Mass.

Aho A. V., Sethi R. and Ullman J. D. 1986, Compilers. Principals, Techniques and Tools. Addison-Wesley, Reading, Mass.

Allen F. E., Cocke J. 1972, A Catalogue Of Optimising Transformations. pp 1-30 in Design And Optimisation Of Compilers (Rustin R. ed.), Englewood Cliffs NJ, Prentice-Hall.

Allen J. R., Kennedy K. 1982, PFC A Program To Convert Fortran Programs To Parallel Form. in Proc. IBM Conf. On Parallel Computation And Scientific Computations.

Allen J. R., Baumgartner D., Kennedy K. and Porterfield A. 1986, PTOOL : A Semi-Automatic Parallel Programming Assistant, pp 164-170 in Proc. 1986 Int. Conf. Parallel Processing, IEEE Computer Society.

Allen J. R., Kennedy K. 1987, Automatic Translation Of FORTRAN Programs To Vector Form. pp 491-542, ACM Transactions On Programming Languages And Systems, 9.

Allen J. R., Callahan D. and Kennedy K. 1987, Automatic Decomposition Of Scientific

Programs For Parallel Execution. pp 63-76 in Conf. Rec. 14th ACM Symposium on Principals Of Programming Languages.

Amdahl G. 1967, The Validity Of The Single Processor Approach To Achieving Large Scale Computing Capabilities. pp 483-485 in AFIPS Conference Proceedings. Vol 30.

Arnold C.N. 1983, Vector Optimisation On CYBER-205. pp 530-536 In Proc. 1983 International Conf. On Parallel Processing, IEEE Computer Society.

Battle T. P., Johnson S. P., Chow P., Wade K. C. and Cross M. 1990, Hardware/Software Strategies For Solidification Modelling, Presented at IASTED Internantional Symposium On Modelling, Simulation and Optimisation, Montreal.

Bernstein A. J. 1966, Analysis Of Programs For Parallel Processing. pp 757-762 In IEEE Transactions On Electronic Computers, 15.

Bornat R, 1979, Understanding And Writting Compilers. Macmillan Educational Limited, London.

Brandes T. 1988, Determination Of Dependencies In A Knowledge Based Parallelization Tool. pp 111-119 in Parallel Computing 8, North-Holland.

Chu E. and George A 1986, Gaussian Elimination With Partial Pivoting And Load Balancing On A Multiprocessor. pp 54-65 in Parallel Computing, 5 (1987), North-Holland.

Cryton R. G. 1986, Doacross: Beyond Vectorisation For Multiprocessors (extended abstract). pp 226-234 in Proc. 1986 Int. Conf. Parallel Processing, IEEE Computer Society.

Ferrante J., Ottenstein K. J. and Warren J. D. 1987, The Program Dependence Graph And Its Use In Optimisation. pp 319-349 in ACM Transactions On Programming Languages And Systems, 9.

Frost R. A. 1986, Introduction To Knowledge Based Systems. Collins Professional And Technical Series, London.

Flynn M. 1966, Very High Speed Computing Systems. pp 1901-1909 in Proc. IEEE, 54.

Griffin H. 1954, Elementary Theory Of Numbers. McGraw-Hill, New York.

Higbee L. 1979, Vectorisation And Conversion Of Fortran Programs For The CRAY-1 CFT Computer, Publication 2240207, Cray Research Inc. Mendota Heights MN.

Hockney R. W. 1965, A Fast Direct Solution Of Poisson's Equation Using Fourier Analysis. pp 95-113 in Journal Of The ACM, 12.

Hockney R. W. and Jesshope C. R. 1981, Parallel Computers. Adam Hilger, Bristol.

IBM 1988, Parallel Fortran, Language And Library Ref., First Edition, International Business Machines Corp.

Ierotheou C. S. 1987, High Level Subroutines For The Masscomp MC5400 Vector Accelerator. Thames Polytechnic, London.

INMOS 1988, OCCAM 2. Reference Manual, Prentice Hall International Series In Computer Science, New York NY, Prentice Hall.

Jones I. P., Kightly J. R., Thompson C. P. And Wilkes N. S. 1985, FLOW3D A Computer Code For The Prediction Of Laminar And Turbulent Flow And Heat Transfer. Release 1, Harwell Report AERE-R11825.

Karp A. H. And Babb R. G. 1988, A Comparison Of Twelve Parallel Fortran Dialects.,pp 52-67 In IEEE Software,5.

Knuth D. E. 1973, The Art Of Computer Programming, Vol 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass.

Kuck D. J. 1978, The Structure Of Computers And Computations, Vol 1, Wiley, New York.

Kuck D. J., Kuhn R. H., Leasure B. and Wolfe M. 1980, The Structure Of An Advanced Vectoriser For Pipeline Processors. In Proceedings Of The IEEE Computer Society Fourth International Computer Software And Applications Conference.

Kuck D. J., Kuhn R. H., Leasure B., Padua D. A. and Wolfe M. 1981, Compiler Transformations Of Dependence Graphs. pp 207-218 in Conf. Rec. 8th ACM

Symposium On Principals Of Programming Languages.

Leggett P. F. 1990, Private Communication, Thames Polytechnic, London.

McGraw J. R., Skedzielewski, Allan S., Oldehoeft R., Glauert J., Kirkham C., Noyce B. And Thomas R. 1985, SISAL - Stream And Iteration In A Single Assignment Language, Language Reference Manual, M-146 Ver 1.2 Rev 1, Lawrence Livermore National Lab.

Mehrota P. And Van Rosendale 1985, The BLAZE Language, A Parallel Language For Scientific Programming. Report 85-29, Institute For Computer Applications In Science And Engineering, NASA Langley Research Centre, Hampton VA.

Mikdiff S. P. and Padua D. A. 1986, Compiler Generated Synchronisation For DO Loops, pp 544-551 in Int. Conference Of Parallel Processing, IEEE Computer Society.

Padua D. A. and Wolfe M. J. 1986, Advanced Compiler Optimisations For Supercomputers. pp 1184-1201 in Communications Of The ACM.

Patankar S. V. and Spalding D. B. 1972, A Calculation Procedure For Heat And Momentum Transfer In Three Dimensional Flows. pp 1787-1808 in International Journal Of Heat And Mass Transfer, 15.

Patankar S. V. 1980, Numerical Heat Transfer And Fluid Flow, Hemisphere.

Polychronopoulos C. D., Kuck D. J. and Padua D. A. 1986, Execution Of Parallel Loops On Parallel Processor Systems. In Proceedings Of The 1986 International Conference On Parallel Processing.

Polychronopoulos C. D. and Banerjee U. 1987, Processor Allocation For Horizontal And Vertical Speedup Bounds. In IEEE Transactions On Computers, Vol C-36.

Polychronopoulos C. D. 1988, Parallel Programming And Compilers, Kulwer, Boston, Mass.

Rhie C. M. and Chow W. L. 1983, Numerical Study Of Turbulent Flow Past An Aerofoil With Trailing Edge Separation, pp 1525-1532 In AIAA.

Sarkar V. And Hennessy J. 1986, Compile-Time Partitioning And Scheduling Of Parallel Programs. pp 17-26 in

Shore J. E. 1973, Second Thoughts On Parallel Processing. pp 95-109 in Computer An Electrical Engineering, 1.

Tarjan R. E. 1972, Depth First Search And Linear Graph Algorithms. pp 146-160 In SIAM Journal Of Computing, 1.

Voller V. R. and Cross M. 1985, Applications Of Control Volume Enthalpy Methods In The

Solution Of Stefans Problems. pp 245-276 in Computational Techniques In Heat Transfer, Pineridge Press, Swansea, U.K.

Voller V. R., Cross M. and Markatos N. C. 1987, An Enthalpy Method For Convection/Diffusion Phase Change, In International Journal Of Methods In Engineering, 24.

Zima H. P., Bast H. J. and Gerndt M. 1988, SUPERB: A Tool For Semi Automatic MIMD/SIMD Parallelisation. pp 1-18 in Parallel Computing, 6, North-Holland.

Zima H. P. 1990, Supercompilers For Parallel And Vector Computers. ACM Press, New York.