Parallel Implementation of The Finite Element Method on Shared Memory Multiprocessors

M.Pakzad

Ph.D. Thesis

Department of Computing Science
University of Newcastle upon Tyne

1995

Abstract

The work presented in this thesis concerns parallel methods for finite element analysis. The research has been funded by British Gas and some of the presented material involves work on their software. Practical problems involving the finite element method can use a large amount of processing power and the execution times can be very large. It is consequently important to investigate the possibilities for the parallel implementation of the method. The research has been carried out on an Encore Multimax, a shared memory multiprocessor with 14 identical CPU's.

We firstly experimented on autoparallelising a large British Gas finite element program (GASP4) using Encore's parallelising Fortran compiler (*epf*). The parallel program generated by *epf* proved not to be efficient. The main reasons are the complexity of the code and small grain parallelism. Since the program is hard to analyse for the compiler at high levels, only small grain parallelism has been inserted automatically into the code. This involves a great deal of low level synchronisations which produce large overheads and cause inefficiency. A detailed analysis of the autoparallelised code has been made with a view to determining the reasons for the inefficiency. Suggestions have also been made about writing programs such that they are suitable for efficient autoparallelisation.

The finite element method consists of the assembly of a stiffness matrix and the solution of a set of simultaneous linear equations. A sparse representation of the stiffness matrix has been used to allow experimentation on large problems. Parallel assembly techniques for the sparse representation have been developed. Some of these methods have proved to be very efficient giving speed ups that are near ideal.

For the solution phase, we have used the preconditioned conjugate gradient method (PCG). An incomplete LU factorization of the stiffness matrix with no fill-in (ILU(0)) has been found to be an effective preconditioner. The factors can be obtained at a low cost. We have parallelised all the steps of the PCG method. The main bottleneck is the triangular solves (preconditioning operations) at each step. Two parallel methods of triangular solution have been implemented. One is based on level scheduling (row-oriented parallelism) and the other is a new approach called *independent columns* (column-oriented parallelism). The algorithms have been tested for row and red-black orderings of the nodal unknowns in the finite element meshes considered.

The best speed ups obtained are 7.29 (on 12 processors) for level scheduling and 7.11 (on 12 processors) for independent columns. Red-black ordering gives rise to better parallel performance than row ordering in general. An analysis of methods for the improvement of the parallel efficiency has been made.

Acknowledgements

I would like to express my sincere gratitude and acknowledge my debt to the following:

- Dr.J.L.Lloyd for his patient supervision
- Dr.C.Phillips for his invaluable help and guidance
- Dr.P.K.Jimack for his helpful suggestions
- Mr.Luiz E. Buzato, Mr.Ehsan Mesbahi and Mr.Saeed Taghavi for their help in preparing the text
- British Gas for providing the funding

Publications

The work described in this thesis is entirely my own. Some material related to that described in chapter 7 of this thesis, concerning the implementation of parallel solvers for finite element-type problems on distributed memory systems conducted jointly with Dr.R.Cook and Dr.C.Phillips has been published in the following:

- Cook,R.,Pakzad.,M.,and Phillips,C.,*Parallel Implementation of Conjugate Gradient-Type Methods*,presented at The Sixth International Conference on Scientific Computing,University of Benin,Nigeria, 24-28 Jan.1994.
- Cook,R.,Pakzad.,M.,and Phillips,C.,*Parallel Preconditioners for the Conjugate Gradient Method*,Tech.Rept.No.467,Dept. Comput. Sci., Univ.Newcastle upon Tyne,1994.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview of Chapter

This chapter firstly describes the aims of the research and motivates the need for parallel computation. The various parallel architectures available are then described. Since the work is to be carried out on a shared memory multiprocessor (Encore Multimax), more detail is given about such computers than other parallel systems. Some terminology related to parallel programming is then given and we discuss some strategies for parallel program design. The layout of the thesis is given in the final part of this chapter.

1.2 Aims

Many practical engineering calculations involve the solution of partial differential equations on regions with complex geometrical configurations. Analytical solutions are generally not possible for such problems and numerical techniques are often used to obtain approximate solutions. The finite element method (FEM) is one such technique and is used for modelling systems whose behaviour is governed by partial differential equations.

The aim of the research described in this thesis is to investigate the potential for the parallelisation of the finite element method and to design parallel algorithms implementing this numerical technique on shared memory multiprocessors. We aim to derive from this general recommendations concerning the implementation of similar computations on shared memory multiprocessors.

The types of engineering problems which are likely to benefit from using the FEM in their analysis are diverse. These include structural analysis problems, heat transfer problems and analysis of electromagnetic fields. The problems essentially involve the evaluation of unknowns such as displacements or temperatures across an area or volume subjected to forces, heat or other forms of energy. The partial differential equations governing these systems can be solved numerically by the FEM leading to the evaluation of the desired unknowns at discrete points in the domain under consideration.

The FEM gives rise to complex and time-consuming calculations. As an example, one large British Gas program used for pipe stress analysis (GASP4), discussed in a later chapter, takes 20 minutes to run on a VAX 8300 for each kilometre of pipe. Real problems involve many kilometres of pipework.

Finite element simulations of a system under conditions of different loadings and configurations may be required together with graphical display of the results, perhaps in real time. The processing power required for such applications can become very large and faster computing is therefore desirable, if not essential, for the solution of large scale problems. The design and use of parallel computers and algorithms is an important approach to gain speed and power.

In the finite element method the domain of the problem is divided into a number of subregions (the elements) and each element is modelled independently. The behaviour of the whole domain is modelled as the summation of the elements' models according to their adjacency (see chapter 2). Each element is defined by a number of nodes and represented by an element matrix. The matrix representing the whole structure is formed by assembling the element matrices into one overall *stiffness matrix*. The element matrix entries at common nodes are summed together and the result is the coefficient matrix of a linear system of equations used for obtaining a set of unknowns in the model. These can be values like displacements or temperatures at the nodes of the problem domain.

The stiffness matrix represents the material characteristics and geometry of the problem domain (see chapter 2). The right-hand side vector (RHS) represents the applied forces (for a problem of structural analysis) and is calculated according to the elements' properties (see section 2.3.2). The resulting displacements are the unknowns in this system of equations. The finite element process consequently consists of the assembly of the stiffness matrix and RHS vector and the solution of a set of simultaneous linear equations. This system of equations typically has a sparse set of coefficients. There is scope for integrating the two phases as in frontal methods [45].

Our aim has been the parallelisation of the finite element method which involves parallel assembly of the stiffness matrix and subsequent parallel solution of the resulting equations. Sparse representations of the stiffness matrix are used. FE matrices are typically of such high orders that it is not practical to store them as dense. Sparse representations can reduce the amount of storage substantially due to the high sparsity of these matrices. For example, in our model program (see chapter 5), a FE mesh with 10201 nodes would require over 100 million storage units if stored fully. The actual number of entries in the stiffness matrix is 90601 ie. it is only around 0.1

percent full, making sparse storage very convenient.

Efficient parallel sparse techniques are difficult to design compared to the better understood dense case. We have investigated the problems associated with designing efficient parallel algorithms for sparse data structures. The aim has been to design algorithms for parallel assembly of the stiffness matrix and parallel solution of the resulting equations.

The parallel assembly methods developed have proved to be very efficient (see chapter 6). The solution method used is the preconditioned conjugate gradient method (PCG). The reason for using this method is that it is particularly suited to the solution of large sparse positive definite systems (see section 2.3.4) ie. the type of matrices that define our model problems. The preconditioner is an incomplete factorization of the stiffness matrix designed to conform with the parallel solution schemes used (see chapter 7).

The solution methods consist of one row-oriented parallel solution scheme (level scheduling) and a novel column-oriented approach to parallel solution which we have called *independent columns*. The ordering of the unknowns has a significant effect on the efficiency of such algorithms. We have tested our implementations with row and red-black orderings. We have also made a detailed study of the ways in which the performance of the solution algorithms can be improved.

Some work has also been done on using parallelising compilers to speed up finite element programs. This work has led to conclusions regarding the writing of code suitable for such compilers.

1.3 Types of Parallel Computers

We can divide parallel computers into three architectural configurations [44]:

- Pipeline computers (vector processors)
- Array processors
- Multiprocessor systems

We shall briefly discuss the first two classes of parallel computers below and then give a more detailed account of multiprocessor systems and shared memory architectures which are of particular interest to us.

1.3.1 Pipeline Computers

The process of executing an instruction involves instruction fetch, instruction decode, operand fetch (if necessary) and execute. In a non-pipelined computer, all four of these steps must be completed before the next instruction can be issued. In a pipelined computer, successive instructions are executed in an overlapped fashion. Similarly, individual instructions (eg. multiplication) may be pipelined at a lower level. Vector pipelines are specially designed to handle vector instructions over vector operands. Examples of such computers are the Cray-1 and VP-200 [44].

Vector computers are used extensively for numerical applications through vectorising compilers and/or explicitly vectorised code and produce attractive speed ups on problems with many vector operations on long vectors. Their performance on short vectors is less satisfactory due to start-up delays. Finite element applications give rise to matrices with sparse rows. In order to assign long sparse rows to vector pipelines some packing and unpacking of the rows is generally necessary. This causes a large amount of overhead, making vector computers not suitable for such computations. Such machines are, however, used for sparse systems in spite of the mentioned overhead [77].

1.3.2 Array Processors

An array processor is a synchronous parallel computer with multiple arithmetic logic units that can operate in parallel in a lock-step fashion. The processing elements are synchronised to execute the same instruction at the same time. Each processing element consists of an Arithmetic Logic Unit (ALU) with registers and a local memory. The processing elements are connected by a data-routing network. Array instructions are broadcast to the processing elements for distributed execution over different components of the array fetched directly from the local memories. The processing elements are under the supervision of a control unit.

Examples of such computers are the Burroughs Scientific Processor, the AMT DAP (Distributed Array Processor) and the Connection Machine. The range of numbers of processors varies widely. Many array processors have fewer than one hundred processing elements. This number is much larger for the DAP (1024 or 4096 processors) but the processors are very elementary. The Connection Machine has approximately 65k processors. Another example of array processors are butterfly networks [43]. These are made of rows of processors containing nodes which are interconnected according to the required application.

Parallel processing algorithms have been developed by many computer scientists for array processors. Examples of these are algorithms for matrix multiplication, fast Fourier Transform and solving partial differential equations [65].

1.3.3 Multiprocessor Systems

A multiprocessor organisation basically consists of two or more processors of approximately comparable capabilities. The processors may share access to common sets of memory modules, I/O channels and peripheral devices. The entire system is controlled either by a single integrated operating system or several communicating operating systems. These provide interactions between processors and their programs at various levels. Besides the shared memories and I/O devices, each processor may have its own local memory and private devices. Inter-process communications can be done through the shared memories, an interrupt network or by message passing. Some available interconnection structures between the memories and processors are:

- Time-shared common bus
- Cross bar switch network
- Multiport memories

Multiprocessors can be divided architecturally into two groups: tightly coupled systems and loosely coupled systems. In the latter type of multiprocessor, each processor has a set of input/output devices and a large local memory where it accesses its instructions and data. The degree of coupling in such a system is very loose and the modules communicate through a message transfer system. An example of such an architecture is the CM* computer [43]. Distributed memory systems also fall into this category of multiprocessors. Such systems provide decentralised computing networks which share common resources. Loosely coupled systems are best suited to low degrees of interaction due to the high costs of interprocessor communication. When a higher degree of interaction between processors is required without significant deterioration in performance, tightly coupled multiprocessors are used.

Shared Memory Architectures - Tightly coupled multiprocessor systems typically contain a global shared memory (see figure 1.1). As mentioned earlier, one way to provide interprocessor communication is through a common bus. If such an interconnection mechanism is used, the speed of data transfer on the bus limits the total power that can be provided by the system. The number of processors must be balanced against the bus speed. The typical maximum number of processors sharing a common bus is between 20 and 30 depending on processor speed, bus bandwidth, etc. Cray machines usually have a smaller number of processors. In some systems the processors may themselves be a vector processing unit providing additional scope for parallel processing.

Each processor may possess a small local memory (cache) which acts as a fast buffer. Caching reduces the amount of access to global memory and bus usage. Associated with caching is the cache coherence problem ie. avoiding the use of inconsistent copies of data. This problem is resolved by the use of cache coherence protocols [44]. Figure 1.1 outlines the configuration of a bus-connected shared memory system with n processors.

Most shared memory systems use identical CPU's and are symmetric. This means that all CPU's can run the operating system, run user code and receive interrupts. Examples of such architectures are the Encore Multimax and the Sequent Symmetry [44].

Shared memory multiprocessors (SMM's) have been used for many numerical applications and have produced encouraging results. The use of such machines for numerical problems is an active area of research. The work described here represents a further contribution in this area.

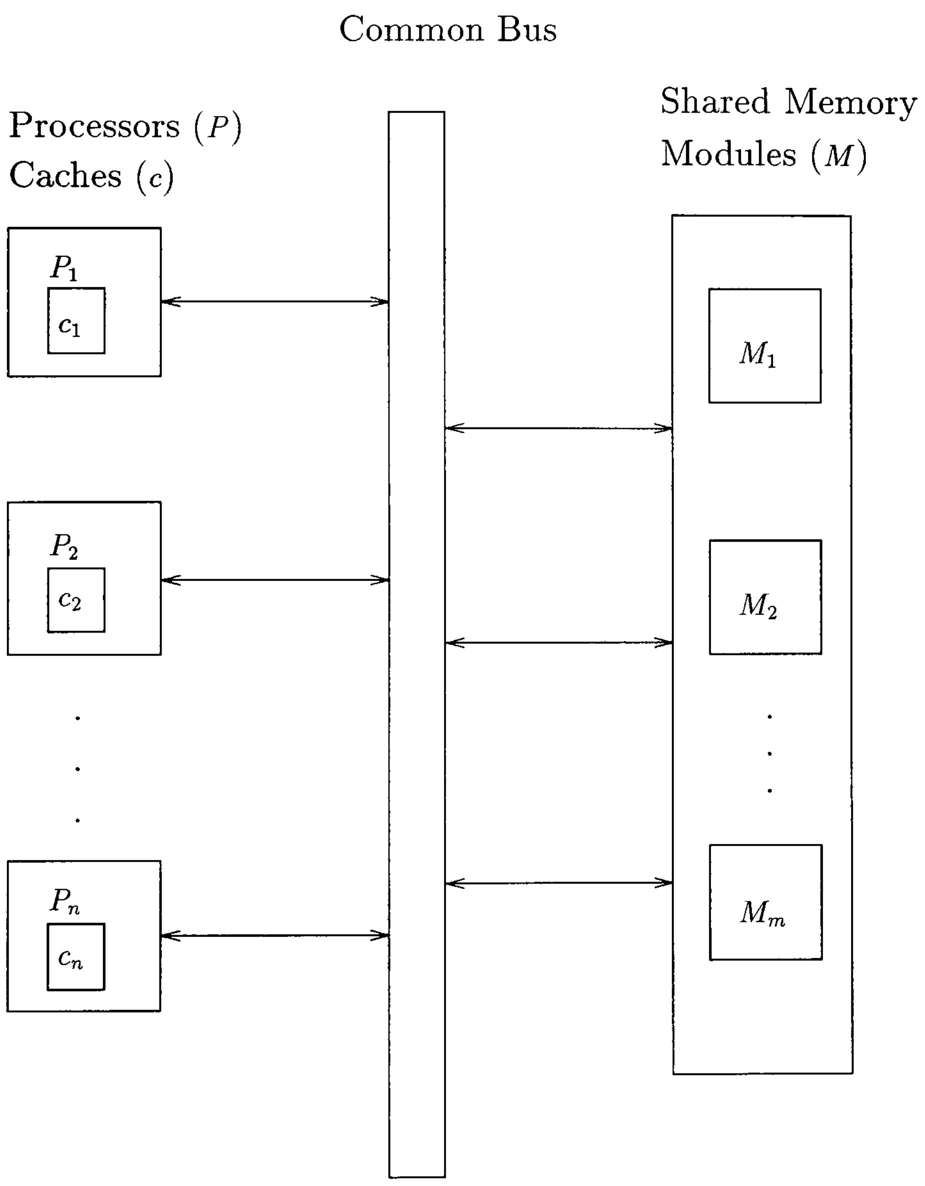


Figure 1.1: A typical shared memory architecture

1.4 Using Shared Memory Machines

Let us consider the Encore Multimax as a typical shared memory machine. The Multimax machines run Encore's version of UNIX (UMAX) as their operating system. Parallel processing is provided by execution units called *processes* which are managed by the operating system. Since the use of processes is too costly for obtaining efficient user-level parallelism, the *task* abstraction [52] is used which involves a smaller overhead. Tasks are capable of running user code and can be started, idled and restarted by processes. In this way, several tasks can be assigned to the same process. Figure 1.2 illustrates how the 14 NS32532 processors in the Multimax machine used for this work are utilised by parallel tasks through a hierarchy of parallel process management modules.

The most important issue concerning program design for shared memory machines is the avoidance of inadvertent access to shared data by different processes. Since two or more processors may attempt to update the same memory segment at one time, it must be ensured that any such write access to shared memory data is done under *mutual exclusion.* This means that any part of memory which could be incorrectly updated if accessed simultaneously by different processors must be protected. This is done using synchronisation constructs (see section 4.4.1) provided by the compiler or the operating system.

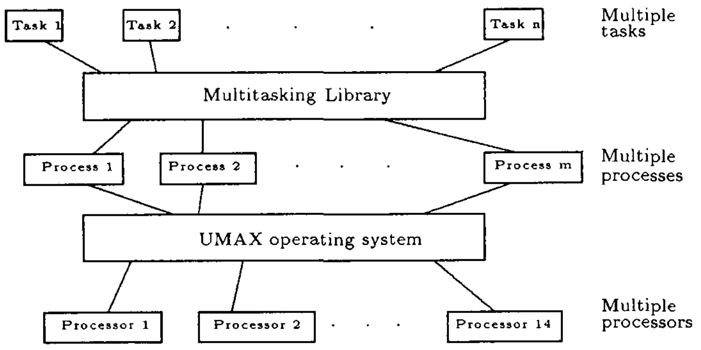


Figure 1.2: Task/Process/Processor Relationships

For some shared memory machines, parallelising compilers are available which make an analysis of sequential code to produce parallel programs. The Encore parallelising Fortran compiler, *epf*, is one such example and is discussed in detail in chapter 4. Such autoparallelised programs are usually not sufficiently efficient, however, mainly due to the complex structure of the sequential code which makes it difficult for the compiler to parallelise efficiently (see section 4.5).

The compilers used for shared memory machines also provide various constructs which can be used for parallel programming. These are synchronisation primitives such as LOCK's, SEMAPHORE's and BARRIER's which are used for intertask communication to achieve goals such as mutual exclusion. LOCK's and similar synchronisation variables are stored in shared memory and are tested by parallel tasks in order to determine whether or not they are allowed to proceed. Some programming language extensions such as the Encore DOALL mechanism for spreading DO loops are also used. Section 4.1 contains descriptions of some of the parallel programming constructs available on the Multimax.

Design tools are also available to aid parallel program design [76]. These offer the programmer a set of macros which abstract out lower level synchronisation detail and provide an environment in which effort can be put into obtaining an efficient design. An example of such tools is the *Force* (see section 3.2.1 and [47]) which provides a set of macros to facilitate parallel program design.

Programs designed for shared memory machines can be used on distributed memory machines by the use of virtual shared memory. This means that the distributed memory machine is run such that at user level it can perform as a shared memory machine. The shared memory model is consequently important because its algorithms can be transported to a distributed memory environment [44].

1.5 Parallel Program Design Strategies for Shared Memory Machines

It is better to write parallel programs with parallelism in mind at the design stage than to parallelise existing algorithms or programs. This is because we can aim to minimise the inherently sequential fraction of an algorithm by thinking about parallel processing at the start of our design.

The cost of programming SMM's is related to the synchronisation cost. In order to achieve good speed ups we need to aim to have a large computation to communication ratio. This means that parallelism should be implemented at high levels by setting off parallel tasks to execute large numbers of independent operations before needing to communicate. Processors' local memories can be used to store and accumulate each task's contribution to shared values. This idea should be used to minimise synchronisation points.

If autoparallelisation is to be performed, the sequential program must be written with simple and clear data dependencies. This allows the parallelising compiler to analyse the program more easily and detect possibilities for parallelisation more readily. We require a well structured program which can be parallelised at high levels. It is important to avoid the insertion of large amounts of low level synchronisation by the compiler which will be the case if the program is hard to analyse at high levels. By taking the above considerations into account at the design stage, the ability of the compiler to parallelise the program efficiently will be enhanced. Autoparallelisation is discussed in detail in chapter 4.

Let us now expand on the synchronisation cost issue. Synchronisation primitives such as LOCK's and EVENT's (see section 4.4.1) have implementation costs which must be taken into account for the design of efficient parallel algorithms. If an algorithm involves the use of a large amount of synchronisation, the cost of setting up and managing the primitives may become large. This overhead will have the effect of degrading the parallel efficiency. The degradation can be substantial if parallelism is implemented at low levels since many synchronisation points will be required.

When designing parallel algorithms for SMM's we must aim to minimise the synchronisation overhead. This should be done by having as few synchronisation points as possible and implementing parallelism at higher levels as far as possible. There may also be cases in which it is not beneficial to use parallelism. Examples of such cases are loops which perform small amounts of computation. If such loops are parallelised by spreading the loop index among the available processors, we will not necessarily gain any speed up. This is because the cost of the parallel loop management synchronisations may not be sufficiently compensated for by a gain due to parallel processing. If it is necessary to insert more than trivial synchronisations into the loop, the performance may become even poorer.

One further important issue is load balancing. Even if we manage to design an algorithm (or parallelise a program) such that parallelism is implemented at high levels, we will not necessarily obtain maximum efficiency unless we make sure that the computational load is spread evenly among the available processors. If this is not achieved, then a significant amount of processing power may be wasted due to idle processors waiting for other processors to finish their respective jobs.

In summary, the important issues for efficient parallel program design for SMM's are:

- A high computation to communication ratio for each parallel task
- Good load balancing
- Only parallelising what will be beneficial.

The above issues are discussed as they apply to the performance of parallel finite element algorithms in sections 4.5.1, 6.7, 7.5.2 and chapter 8.

1.6 Some Terminology

It is useful at this stage to make some definitions of terms used frequently when discussing parallel models.

Granularity A measure of the number of instructions in a parallel computation between synchronisation points [65]. Shared memory machines usually have synchronisation intervals in the order of 100's of instructions, which is medium grain. Array processors implement very fine grain parallelism and network distributed systems are often very coarse grain.

Speed up The speed up achieved by a parallel algorithm running on p processors is the ratio between the time taken by a single processor of the same type executing the fastest sequential algorithm and the time taken by the same parallel computer executing the parallel algorithm on p processors [65].

Parallel Efficiency The efficiency of a parallel algorithm running on p processors is the speed up divided by p [65].

Amdahl's Law Let f be the fraction of operations in a computation that must be performed sequentially. The maximum speed up achievable on p processors is then $\frac{1}{f+(1-f)/p}$ [65].

1.7 Organisation of Thesis

In chapter 2 we discuss the finite element method in terms of what it is used for and how it works. Some application programs using the method are mentioned and we discuss some possibilities for parallelising the method. Chapter 3 reviews recent research work related to the thesis.

The parallelisation of a large British Gas finite element program using the Encore autoparallelising Fortran compiler is discussed in chapter 4. Some comments are made on our experiences of hand-parallelising the program. We also make some suggestions about writing sequential programs suitable for such compilers in this chapter.

In chapter 5 the finite element model program used for testing our parallel algorithms is described. These algorithms are for parallel assembly of the stiffness matrix (chapter 6) and parallel iterative solution of the resulting equations (chapter 7). Timing results are also given in these chapters. Chapter 8 contains our conclusions regarding the parallel design of finite element code together with some recommendations for future work based on the thesis material.

Chapter 2

The Finite Element Method and Linear Equation Solvers

2.1 Overview of Chapter

In this chapter the finite element method is described in terms of why and how it is used to analyse physical problems in fields such as structural mechanics. Details of the mathematical formulation and various stages of finite element analysis are given together with an explanation of how the method could lend itself to parallelisation. We have also described some real applications of the method to show how costly it can be in terms of processing time. This motivates later chapters on the parallelisation of the method.

2.2 The Finite Element Method

2.2.1 Uses of the Method

Scientists and engineers are often faced with practical physical problems whose solution by conventional analytical methods is either too difficult or even impossible. In structural mechanics, for example, there are many cases in which the complex geometrical configurations that practical problems have make it exceedingly difficult to obtain *exact* solutions. The analyst must consequently use numerical techniques to solve such problems. The Finite Element Method (FEM) is one such method and is used for the solution of partial differential equations on regions with complex geometrical configurations.

2.2.2 The Method

The finite element method is a general technique for constructing approximate solutions to differential equations [5]. The method involves dividing the domain of the solution into a finite number of simple subdomains (the finite elements) and using variational concepts to construct an approximation of the solution over the collection of finite elements.

In a problem of structural analysis, for example, a body is considered to be actually broken up into a number of elements. The elements are interconnected by means of *nodes* and the body is replaced by the system of finite elements and the nodes connecting them [61]. The nodes are used to identify points on the elements.

Each element is represented by an *element matrix* which models its individual properties and behaviour. The whole body is represented by the summation of all element matrices into a *stiffness matrix*. This matrix consists of the coefficients of the set of equations which can be used to evaluate the nodal unknowns.

The finite element method consequently consists of the following stages:

- Discretisation of the domain
- Evaluation of the element matrices
- Formation of the equations ie. assembly of the overall stiffness matrix
- Solution of the equations.

The operations associated with the assembly and solution phases are independent of the type of problem at hand. Characteristics such as problem type and geometry are relevant only to the element matrix evaluation stage. The solution phase usually dominates the processing time for finite element analysis. The above steps are described in detail in section 2.3.

We shall now give some details of the mathematical formulation of the FEM.

2.2.3 Mathematical Formulation

Let us consider the solution of differential equations of the type

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + c(x,y)u = f(x,y) \quad in \;\; \Omega \tag{2.1}$$

subject to the following boundary conditions

$$u = u_0(x,y) \quad on \quad \partial\Omega_1 \quad \text{Dirichlet}$$

$$\frac{\partial u}{\partial n} = v_0(x,y) \quad on \quad \partial\Omega_2 \quad \text{Neumann}$$

(see figure 2.1).

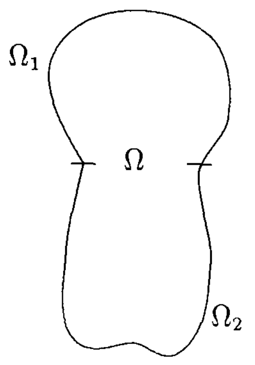


Figure 2.1: The domain of the problem and its boundaries

A *variational* statement of the problem defined by (2.1) can take the form

$$I(u) = \int_\Omega \left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + c(x,y)u^2 + 2f(x,y)u \right) d\Omega - 2\int_{\partial\Omega_2} v_0 u \; ds \tag{2.2}$$

subject to $u = u_0(x)$ on $\partial\Omega_1$ (see [38]). We can see that the Neumann boundary condition appears in the functional in (2.2) but not the Dirichlet boundary condition.

If u is a minimum point of I, then $I(u) \leq I(v)$ for any v. In particular, if $v = u + \varepsilon w$ (for any small ε and any w which satisfies (2.1)) and using integration by parts, it follows that if u is a minimum point of I then u must satisfy (2.1) [38]. This means that if we determine the function $u(x, y)$ which minimises (2.2) we have also obtained the solution to (2.1). The reason for seeking a numerical solution for (2.2) (and hence (2.1)) rather than the analytical one is that in many practical situations the data given in a problem are not smooth. In such cases some derivatives of the solution may not exist and therefore the solution will not satisfy (2.1) at all points in Ω.

More precisely, in the case of (2.1) we are looking for a solution in the space of all twice-differentiable functions which satisfy (2.1) and the boundary conditions associated with it (ie. Dirichlet and Neumann). In the case of (2.2), however, we are seeking a function in the space of all once-differentiable functions which minimises (2.2) and satisfies the Dirichlet boundary condition. Since minimising (2.2) is equivalent to solving (2.1), we are now seeking a solution to the latter equation in a more restricted domain.

Variational methods seek an approximate solution in the form of a linear combination of suitable approximating functions. The result is the minimisation of a functional related to the problem at hand (ie. (2.2)) with respect to a suitable reduced function space spanned by the approximating functions. The FEM can be thought of as a special case in which the chosen functions are piecewise continuous with respect to the mesh.

The first step in variational methods is the expression of the solution $u(x,y)$ as

$$u(x,y) \approx \sum_{i=1}^{n} a_i \; N_i(x,y) \tag{2.3}$$

where each N_i is called a *shape*, *test* or *basis* function and the a_i's are constants. Our aim here is consequently the determination of the coefficients $a_i - a_n$ since the shape functions are already known. The coefficients a_i are actually the values of the solution corresponding to the nodal unknowns in a finite element mesh (see section 2.3.1). The choice of shape functions depends on the problem under consideration and they usually take the form of simple linear or quadratic functions of the independent variables.

The expression of the solution in the form of (2.3) means that we are dealing with a finite dimensional subspace spanned by the basis functions. In this subspace we now seek a function which minimises (2.2) and satisfies the Dirichlet boundary condition. The choice of a suitable subspace is vital in finding basis functions which yield values of (2.2) close to its minimum. Since we require only first derivative continuity, linear splines can be used here as basis functions. Typically the basis functions will have local support ie. will be zero outside a small subdomain. The solution to the variational form of the original problem is a projection of the actual solution onto the finite dimensional subspace.

Once we have expressed the solution in the form of (2.3) we can substitute the latter into (2.2) and minimise with respect to the a_i's. This gives rise to a set of linear equations from which the a_i's can be determined (see section 2.3.2). Alternatively, a variational method such as the *method of weighted residuals* can be used to minimise the functional. In this method, the parameters are determined by setting the integral (over the domain) of a *weighted* residual of the approximation to zero, ie.

$$\int_\Omega \Psi_i(x,y)\ E(x,y,a_j)\ d\Omega = 0 \quad i = 1,2,\ldots,N \tag{2.4}$$

where Ψ_i are *trial* or *weight* functions and E is the *residual* defined by

$$E = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + c(x,y)u - f(x,y). \tag{2.5}$$

The weight functions need not be the same as the basis functions mentioned above (ie. N_i's). The Galerkin Method is a special case of the method of weighted residuals in which $\Psi_i = N_i$ ie. weight functions equal to the shape functions are used giving rise to integrals of the form

$$\int_\Omega N_i(x,y)\ E(x,y,a_j)\ d\Omega = 0 \quad i = 1,2,\ldots,N. \tag{2.6}$$

We obtain the same linear system in the a_i's from both the straight minimisation mentioned earlier and the evaluation of (2.6). The coefficients of this linear system are computed by numerical integration (see section 2.3.2). In the case of the FEM these coefficients form what is called the *stiffness matrix* (see section 2.3.1) and the solution of the system of equations yields the nodal unknowns ie. the a_i's in (2.3) (see section 2.3).

Nonlinear Problems

Sometimes the differential equation to be solved is nonlinear in u and its solution is somewhat complicated due to the presence of nonlinear terms. For example, consider the nonlinear problem defined as (2.7) in which the RHS is also dependent on u ie.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + c(x,y)u = f(x,y,u). \tag{2.7}$$

In these cases one can linearise the equation by means of the introduction of an iteration into the solution algorithm. For example, (2.7) can be solved by a sequence of iterative steps such as

$$\frac{\partial^2 u^{(n+1)}}{\partial x^2} + \frac{\partial^2 u^{(n+1)}}{\partial y^2} + c(x,y)u^{(n+1)} = f\left(x,y,u^{(n)}\right). \tag{2.8}$$

We can start with an initial guess for the solution, $u^{(0)}$, and compute the RHS (ie.$f^{(0)}$) corresponding to this value for u. The next iteration is a linear problem involving the calculation of $u^{(1)}$. This is used in turn to compute $f^{(1)}$. The iterations proceed in the same manner until a satisfactory estimate for u is found. Other nonlinearities are possible in (2.1) such as a $c(x, y, u)$ term or having an extra function of u multiplied by the derivative terms. These nonlinear problems can be solved by the same technique mentioned above as well as others (eg. Quasi-Newton method).

The model program described in chapter 5 uses the PCG method (see section 2.3.4) for the solution of the resulting linear equations. If the problem to be solved involves any form of nonlinearity, part of the stiffness matrix will be different at each PCG outer iteration. We must consequently recompute the varying part of the stiffness matrix at each step before the rest of the PCG operations are performed. This can be done by means of the introduction of an outer loop round each PCG iteration which recomputes the necessary coefficients prior to each iteration.

2.3 Steps Of The Method

This section contains details of the steps of finite element analysis. Each step is treated separately and the aim is to form a basis for the topics discussed in the following chapters.

2.3.1 Discretisation

One-dimensional bodies are subdivided into finite elements by means of nodes. Lines and planes are used for the subdivision of two- and three-dimensional bodies. In one-dimensional bodies the resulting finite elements may have unequal lengths, while in two and three dimensions they may have unequal sizes as well as unlike shapes.

In all cases, however, the aim is to break up the body into a number of finite elements which cover as much of the body as possible. It may not be possible to cover the whole body due to the irregularity of its boundaries. Also, the greater the number of nodes in the mesh, the greater will be the number of points in the mesh where the unknowns can be evaluated.

Figure 2.2(a) shows the finite element model of a beam fixed at both ends. Figure 2.2(b) shows the forces and displacements at the extremities of an element. f's refer to forces and d's refer to displacements. Each of the displacements is called a *degree of freedom* [61]. The degrees of freedom may be quantities such as slopes (derivatives) as well as displacements in other problems.

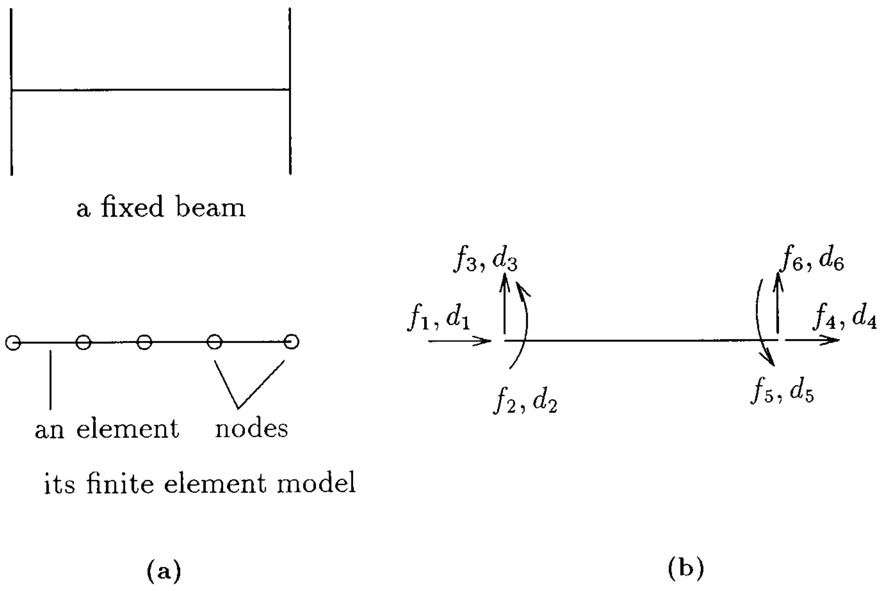


Figure 2.2: Finite element modelling of a beam

As explained later in this chapter, the method leads to systems of equations of the form

$$Kd = f$$

where f is the *force vector*, K is the *stiffness matrix* and d is the *displacement vector*. K often has the form of a band or sparse matrix. The above system of equations can be solved to determine displacement unknowns, stresses and strains. The method extends to other applications such as temperature/pressure distributions.

Figure 2.3 illustrates how a plane body can be subdivided into rectangular and triangular elements. The curved boundary is approximated by the triangular elements. In order to incorporate complicated element shapes into FE meshes *isoparametric elements* are used. These provide a local mapping of each element onto a reference element and we can define the shape functions in terms of polynomials on the reference element. A closer approximation of the curved boundary in figure 2.3 could be obtained by such a technique.

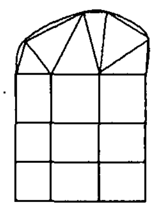


Figure 2.3: Discretisation of a plane body

2.3.2 Evaluation of element matrices

In this section we shall describe how element matrices can be calculated for problems involving linear two-dimensional partial differential equations. We consider problems having the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + c(x,y)u = f(x,y)$$

where $u(x,y)$ is given on the boundary (see section 2.2.3).

To evaluate the unknown u at specific points in the $x - y$ plane we need a two-dimensional discretisation of the domain. We can consequently use elements with rectangular, triangular or other shapes.

Consider the case of a rectangular element whose element matrix we wish to evaluate. In order to model the variation of u across the element we can associate with each node, i, a *shape function*, N_i (see section 2.2.3). This serves as a measure of the contribution of the nodal value u_i towards the value of u at any point within the element. The rectangular element is illustrated in figure 2.4.

At any point (x,y) in the element, we can write

$$u(x,y) = N_i u_i + N_j u_j + N_k u_k + N_l u_l.$$

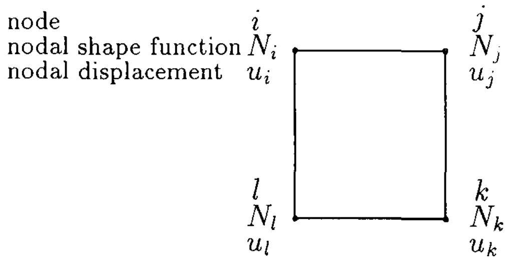


Figure 2.4: Shape functions and nodal unknowns for a rectangular element

The shape functions have the following form

$$N_i = g(x, y)$$

and a set of these forms a basis for a suitable subspace in which the solution is to be approximated such as a piecewise polynomial with respect to the mesh (see below). These functions differ for the various element shapes.

The evaluation of the element matrices involves a numerical integration (see below). It is more convenient to carry out this integration using a *natural* coordinate system. This is a local system that permits the specification of a point within the element by a dimensionless number whose absolute magnitude never exceeds unity. Using such a system with natural coordinates ξ and η, the rectangular element can take the form illustrated in figure 2.5.

As an example, we can express the four bilinear shape functions (one per node) in a quadrilateral element in terms of the new coordinates as (see [72])

$$\hat{N}_i = \frac{1}{4}(1-\xi)(1-\eta)$$

$$\hat{N}_j = \frac{1}{4}(1+\xi)(1-\eta)$$

$$\hat{N}_k = \frac{1}{4}(1+\xi)(1+\eta)$$

$$\hat{N}_l = \frac{1}{4}(1-\xi)(1+\eta)$$

$$where \quad -1 \leq \xi, \eta \leq 1.$$

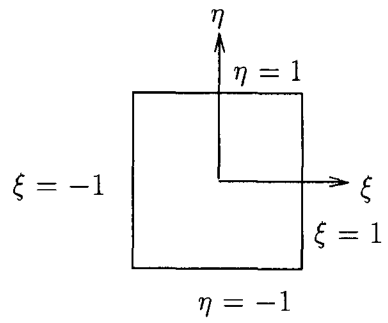


Figure 2.5: A natural coordinate system

For the triangular element in figure 2.6 the linear shape functions have the form

$$\hat{N}_i = 1 - \xi - \eta$$
$$\hat{N}_j = \xi$$
$$\hat{N}_k = \eta$$
$$where \;\; 0 \leq \xi, \eta \leq 1 \;, \xi + \eta \leq 1.$$

When a node is shared by a combination of element types, the shape function used during the numerical integration is that corresponding to the element type under consideration.

The entries in each element matrix can be the coefficients of the set of equations obtained by minimising (2.2) or evaluating (2.7) (see section 2.2.3). Each entry a_{ij} in an element matrix can be evaluated by integrating the appropriate shape functions and their derivatives over the element space, A, in the following way:

$$a_{ij} = \iint_A N_{ix}(x,y)N_{jx}(x,y) + N_{iy}(x,y)N_{jy}(x,y) + c(x,y)(N_i(x,y)N_j(x,y))dx\,dy \;,$$

$$i, j \in A.$$

$c(x, y)$ is the coefficient function in the original differential equation and

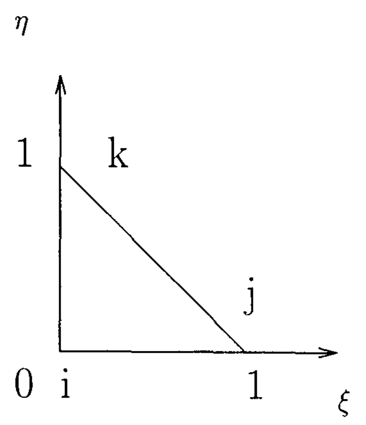


Figure 2.6: A triangular element

$$N_{ix} = \frac{\partial}{\partial x}(N_i(x,y)),$$

$$N_{iy} = \frac{\partial}{\partial y}(N_i(x,y)).$$

The nodal coordinates are transformed to those in the (ξ, η) system. The integration must be evaluated numerically assuming the problem data (eg. c(x,y)) are not smooth (see section 2.2.3). This can be done, for example, by a double application of Gauss-Legendre quadrature with n points. This gives

$$a_{ij} = \sum_{r=1}^{n}\sum_{s=1}^{n} W_r W_s \mid det(J) \mid (\hat{N}_{i\xi}(\xi_r, \eta_s)\hat{N}_{j\xi}(\xi_r, \eta_s) + \hat{N}_{i\eta}(\xi_r, \eta_s)\hat{N}_{j\eta}(\xi_r, \eta_s)$$
$$+ c(\xi_r, \eta_s)(\hat{N}_i(\xi_r, \eta_s)\hat{N}_j(\xi_r, \eta_s)))$$

where r and s are points and W_r and W_s are weights of the quadrature formula for a particular value of n. J is the Jacobian matrix given by

$$[J] = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

evaluated at (ξ_r,η_s). The whole element matrix consists of the evaluation of the above integral for every combination of nodal points in (i,j,k,l).

The evaluation of each element's force vector can be done using the same numerical integration technique as that used for the element matrices. This can have the form

$$f_i = \sum_{r=1}^{n}\sum_{s=1}^{n} W_r W_s \mid det(J) \mid f(\xi_r,\eta_s)\hat{N}_i(\xi_r,\eta_s)$$

where f_i is the component of the force vector corresponding to node i.

For the specified boundary conditions, the element force vector entries are set to be the prefixed values. The element force vectors are accumulated during assembly to form the RHS vector. This can then be used as the right-hand side for the solution of the overall system of equations.

The nodal displacements are required to have continuity across the element boundaries ie. U must be continuous in all cases. The continuity of derivatives depends on the form of the interpolating polynomial. This means that the interelement continuity of derivatives is determined by the number of interpolation points used, the shape functions and the type of elements in the mesh. Hence, with a suitable choice of interpolation we can obtain first and possibly higher order derivative continuity across the element boundaries.

2.3.3 Assembly Phase

The assembly phase consists of accumulating the element matrices into the overall stiffness matrix. This involves the summation of entries for corresponding nodes from different element matrices into the same positions in the overall matrix.

Consider the two-dimensional row-ordered mesh consisting of 16 elements and 25 nodes shown in figure 2.7. Each element matrix has rows and columns corresponding to nodes in the element. For example, elements 6 and 7 have the following element matrix structures:

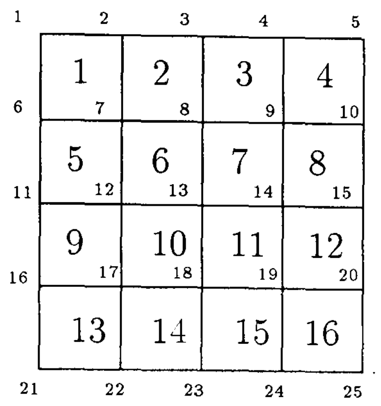


Figure 2.7: A two-dimensional row-ordered mesh

node numbers

$$\begin{array}{c|cccc} & 7 & 8 & 13 & 12 \\ \hline 7 & * & * & * & * \\ 8 & * & * & * & * \\ 13 & * & * & * & * \\ 12 & * & * & * & * \end{array} \qquad \begin{array}{c|cccc} & 8 & 9 & 14 & 13 \\ \hline 8 & + & + & + & + \\ 9 & + & + & + & + \\ 14 & + & + & + & + \\ 13 & + & + & + & + \end{array}$$

Element 6 Element 7

The two elements have a common boundary consisting of the line joining nodes 8 and 13. This means that there are certain entries in the overall stiffness matrix which consist of contributions from distinct entries in the two element matrices. These entries are at positions (8,8),(8,13),(13,8) and (13,13). The element matrices are placed in the overall stiffness matrix as shown in figure 2.8. The common entries are denoted by c's.

The whole assembly process consequently consists of the insertion of the element matrices into the overall stiffness matrix and summing entries at common nodes. We then obtain from various elements' characteristics expressed in terms of their respective element matrices a representation of the

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	. . .
1															
2															
3															
4															
5															
6															
7							*	*				*	*		
8							*	c	+			*	c	+	
9								+	+				+	+	
10															
11															
12							*	*				*	*		
13							*	c	+			*	c	+	
14								+	+				+	+	
. . .															

Figure 2.8: Assembly of element matrices

behaviour of the whole structure.

We can also see how the ordering of the nodes affects the structure of the stiffness matrix. As can be seen from the above representation, we can expect a row-ordered mesh on a rectangular region to give rise to a band matrix (see section 7.3.1). A less regular geometry would give more scattered stiffness matrix entries. Various node ordering schemes can be used to produce desired characteristics in the assembled matrix. Examples of such orderings are minimum degree and red-black orderings [16]. Minimum degree ordering gives a scattered sparsity pattern and red-black matrices consist of one main and several off-diagonal bands (see section 7.2).

With the exception of frontal methods [45] which combine assembly and solution, the next step of finite element analysis is the solution of the nodal equations.

2.3.4 Solution Phase

Once the stiffness matrix is formed we can evaluate the nodal unknowns by solving the system of equations

$$Kd = f$$

for d (see section 2.3.1). The force vector, f, is initially supplied as part of the problem specification and is computed according to the elements' properties. The solution can be obtained by direct or iterative methods.

Direct methods yield the solution after a known finite number of operations. The main problem with such methods is due to fill-in ie. a previously zero matrix entry becoming non-zero as a result of an elimination operation. Direct methods used are usually Gaussian elimination or one of its variants such as LU decomposition or Cholesky factorization for symmetric positive definite K. It is possible to take advantage of the structure of the stiffness matrix to employ strategies such as blocking [33] in the factorization.

For iterative methods, we do not know the number of operations which need to be performed to obtain the solution in advance but there is no extra storage required ie. there is no fill-in (see below).

We shall now discuss iterative methods in some detail.

Iterative Methods

Iterative methods start with an initial estimate to the solution and improve this by obtaining a series of new approximations which converge to the exact solution. Preconditioning is often used to improve the rate of convergence. We return to a study of possible preconditioning matrices M later. The choice of an iterative method for a finite element application depends on the type of problem to be solved.

One popular class of iterative methods for solving large sparse systems are *Krylov subspace* methods [68]. In these, the solution is sought in a particular subspace spanned by multiples of the residual and a finite number of different powers of the coefficient matrix ie. $r, Kr, K^2r, \ldots$. Examples of these methods are the Preconditioned Conjugate Gradient method (PCG) [68], Generalized Minimum Residual (GMRES) [70] [13], Conjugate Gradient Squared (CGS) [75], General Conjugate Residual (GCR) [20], Modified

Conjugate Residual (MCR) [8], Generalized Conjugate Gradients [4], Bi-Conjugate Gradients (Bi-CG) [32], ORTHOMIN [81] and ORTHODIR [84]. Let us now discuss and compare some of these methods.

In the PCG method, we start with an initial guess for the solution and choose successive search directions which are K-conjugate to all previous search directions (see below). The GCS algorithm is a variant of the PCG algorithm and is suitable for nonsymmetric linear systems giving a fast rate of convergence. ORTHOMIN, ORTHODIR and GCR are based on the minimisation of the Euclidean norm of the residual along a particular search direction.

The MCR method is a stabilised version of the GCR method for solving large sparse systems of linear equations. This method has special significance when the system is not positive definite when methods like PCG are inapplicable.

GMRES is used for the solution of nonsymmetric non-singular systems of linear equations, $Kd = f$, when K is not positive real (ie. the symmetric part of K, $\frac{1}{2}(K + K^T)$, is not positive definite). CGR and ORTHODIR often fail to solve such systems. The GMRES algorithm starts by choosing an initial guess to the solution and a dimension of the Krylov subspaces. The Arnoldi process is used to compute an orthonormal basis of the Krylov subspace at each step.

The PCG method is the most popular Krylov subspace method for solving symmetric positive definite linear systems giving rise to efficient implementations for these [68]. Due to the particular suitability of PCG-type methods to the solution of large sparse systems (see below) and the fact that our test matrices in the model program described in chapter 5 are symmetric and positive definite for both node ordering schemes used (ie. row and red-black orderings), we have chosen the PCG method for the purposes of this work.

Let us now make a comparison between direct and iterative methods such as PCG for the purposes of sparse system solution before giving an outline of the PCG algorithm.

Comparison Of Direct And Iterative Methods

One important advantage in using PCG-type methods for sparse matrices is related to storage requirements. For these methods, apart from a few vectors, no additional storage is required. Most importantly, there is very little or no fill-in. For example, incomplete factorization preconditioners either have no fill-in or only low level fill-in (see below and section 3.2.2).

The amount of fill-in due to direct methods can be large which increases both the amount of storage required and the computational cost. This means that one can start with a very sparse coefficient matrix and by the end of a direct solution scheme have generated a very dense matrix (with the exception of band matrices). As the size of the coefficient matrix increases, the extra storage and computational costs associated with fill-in become more substantial.

In brief, PCG-type methods have the advantages of lower operations count and storage requirements over direct methods for large sparse system solution ie. the type of matrices which arise in practical finite element applications.

We shall now describe the PCG method in some detail since it has been used in our model program. This is followed by a discussion of preconditioning and the various types of preconditioners available.

The PCG Method

The PCG algorithm for the solution of the system $Ax = b$ with preconditioner M is given below. We can think of the algorithm below as the conjugate gradient method (CG) [42] applied to the linear system $M^{-1}Ax = M^{-1}b$. Alternatively, the PCG method corresponds to the standard method with the scalar product chosen as $(x, y) = x^T My$. Using the new inner product with the CG method has the property of computing an approximate solution whose preconditioned residual vector $M^{-1}(b - Ax_i)$ is M-orthogonal to all the previous preconditioned residual vectors [68].

```
Algorithm: Preconditioned CG to solve $Ax = b$
1. Preprocess: Compute preconditioner $M$
2. Start: $r_0 \leftarrow b - Ax_0$,
          $p_0 \leftarrow z_0 \leftarrow M^{-1}r_0$
3. Iterate: Until convergence do
      (a) $w \leftarrow Ap_i$
      (b) $\alpha_i \leftarrow \frac{(r_i,z_i)}{(w,p_i)}$
      (c) $x_{i+1} \leftarrow x_i + \alpha_i p_i$
      (d) $r_{i+1} \leftarrow r_i - \alpha_i w$
      (e) $z_{i+1} \leftarrow M^{-1}r_{i+1}$
      (f) $\beta_i \leftarrow \frac{(r_{i+1},z_{i+1})}{(r_i,z_i)}$
      (g) $p_{i+1} \leftarrow z_{i+1} + \beta_i p_i$
```

Preconditioning

Preconditioning aims to improve the condition of the system $Ax = b$ by pre-multiplying the coefficient matrix A by its approximate inverse and hence improving the rate of convergence. The closer this approximation is to A^{-1}, the better the rate of convergence is. The most extreme case of preconditioning is the formation of $M^{-1} = A^{-1}$ which would lead to immediate convergence. This is obviously too costly and we consequently seek a preconditioner for which the extra costs associated with preconditioning are compensated for by the reduction in the number of iterations. By the extra costs we mean the formation of the preconditioner and the preconditioning operations (step 3-e above) at each step.

Several preconditioning schemes are available which differ in the way they form the preconditioning matrix and perform the preconditioning operations. Some popular classes of preconditioners are described below.

Incomplete Factorizations (ILU) ILU preconditioning [1] [73] consists of approximating A as the product of a lower and an upper triangular matrix which are approximate factors of A ie. as $A \approx LU$. The preprocessing phase of the algorithm above would consequently involve computing $M = LU$ where $A = LU + E$ for some *small* E. The preconditioning operation (step 3-e above) consists of a pair of triangular solves involving L and U.

If we allow no fill-in during the factorization the scheme is called ILU(0) (see section 7.3.1). If fill-in due to entries in the original matrix (first-level fill-ins) are allowed then the factorization is ILU(1). In general, ILU(k) refers to a factorization in which the largest level of fill-in allowed is k [14]. As the amount of fill-in allowed increases, the preconditioner becomes more effective ie. the rate of convergence improves (see section 7.4.4). ILU preconditioners are suitable for parallel systems with a small number of processors (see sections 3.2.2 and 7.2).

Polynomial Preconditioning Polynomial preconditioning [68] [46] consists of choosing a polynomial s and replacing the original linear system by $s(A)Ax = s(A)b$. The preconditioned matrix should be as close as possible to the identity matrix in some sense eg. in terms of its set of eigenvalues. Polynomial preconditioning is suited to parallel systems with large numbers of processors (see section 3.2.2).

Domain Decomposition Based Methods The idea in domain decomposition methods (see [7] and section 3.2.1) is to divide the domain of the problem into a number of subdomains which only interact at their common interfaces. If the unknowns are properly sequenced, the resulting coefficient matrices have a block structure consisting of several independent blocks each corresponding to a subdomain together with some interface blocks corresponding to the connection between subdomains.

A domain decomposition preconditioner [74] is one consisting of the factorized blocks of a matrix with the structure mentioned above. Since the subdomain problems are decoupled, their corresponding submatrices can be formed (and factored) independently. The interface blocks must be factored in a different way. Due to the independence of the subdomain blocks, these methods (and hence preconditioners) provide significant opportunities for parallel processing [50].

Hierarchical Basis Functions If the usual nodal basis functions in a finite element mesh are replaced by *hierarchical* basis functions [39], it can be shown that the resulting matrix (A) has improved condition. Hierarchical basis functions consist of the usual nodal basis function(s) on a coarser grid, together with the nodal basis functions for a finer grid corresponding to nodes that are not present in the coarsest grid, together with the nodal basis functions for a still finer grid corresponding to nodes that are not present in any of the coarser levels, etc., up to the finest grid on which it is desired to solve the problem.

If S is the linear transformation which maps $\grave{A}$ onto A such that we can say that $\grave{A} = S^T A S$, then we can define a hierarchical basis preconditioner as $M^{-1} = SS^T$. Such preconditioners are shown to be effective on both serial and parallel architectures [39].

Another type of preconditioner which can be mentioned arises from simple diagonal scaling schemes (see section 7.3.1 and [39]).

Let us now discuss the effects of the ordering of the unknowns on ILU preconditioners since this is our chosen preconditioning scheme (see section 7.2) and discuss some available sparse LU factorization software.

Effects Of Ordering

The ordering of the unknowns affects the amount of fill-in during any factorization (ie. incomplete or full). For example, in the ILU(0) case we allow no fill-in but some orderings would require more fill-in to be ignored than others. More precisely, the incomplete factorization obtained by the ILU(0) scheme can be very close (or identical) to the full LU factors or be very different from the latter depending on the choice of ordering. The quality of the preconditioner in terms of its acceleration property depends upon its closeness to the full factors. Hence, the choice of ordering scheme affects the quality of the preconditioner significantly.

The issue of ordering has also been discussed in detail in section 3.2.2 where a review of some recent work on this subject and the convergence of CG methods in general has been presented. One interesting observation is that the number of CG and GMRES iterations is related to the norm of the residual [18] [19]. In the latter work this property has been used as the basis of a criterion for the assessment of the suitability of ordering algorithms to the problem to be solved. More precisely, the ratio of the Frobenius norms of the residual to that of the coefficient matrix is suggested to be a good criterion for choosing an ordering scheme. Some cases which confirm the usefulness of the mentioned criterion have been presented for the preconditioned GMRES algorithm. For example, reverse Cuthill-McKee ordering [18] used with an ILU(0) preconditioner gives a low residual ratio and rapid convergence.

As mentioned above, the ordering scheme used affects the amount of fill-in for full LU decompositions as well as incomplete decompositions. This means that there may be cases where a full factorization might involve the addition of very few (or no) fill-ins. We may choose to perform an LU factorization with high level fill-ins in these cases to take advantage of the extra information provided so inexpensively. In all cases, the amount of fill-in affects the storage and computational costs significantly and must consequently be taken into account carefully prior to the choice of an ordering scheme for the unknowns.

Sparse LU factorization Software

It is possible to perform sparse LU factorization by means of available software packages such as the NAG library routines [59] and SPARSPACK [36]. These packages provide routines which can be called from a user's program with a specific parameter list.

The NAG routine F01BRF factorizes a real sparse matrix. The routine either forms the LU factorization of a permutation of the entire matrix, or, optionally, first permutes the matrix to block lower triangular form and then only factorizes the diagonal blocks. The factorization is intended to be used by F04AXF [60] which solves sparse systems of the form $Ax = b$ or $A^T x = b$ by block forward and backward substitution.

If several matrices of the same sparsity pattern are to be factorized, F01BSF should be used for the second and subsequent matrices. The latter routine factorizes a real sparse matrix using the pivotal sequence previously obtained by F01BRF when a matrix of the same sparsity pattern was factorized.

2.4 Some Applications of the Finite Element Method

In this section we present some practical applications of the finite element method. These have been provided by the British Gas Engineering Research Station (ERS) at Killingworth and are described here with their kind permission.

2.4.1 GASP4 - A Program for Pipework Stress Analysis

This program is used for stress analysis of gas pipelines. We have studied the program in some detail with a view to its parallelisation (see chapter 4). The program is used interactively as a design tool and is consequently run many times with different sets of parameters in selecting remedial measures. This is why it is desirable to reduce the processing time for this program.

GASP4 is based on the frontal solution technique [45]. Although more general applications are possible, the frontal method can be considered as a particular technique for assembling finite element stiffnesses and nodal forces into a global structural matrix and load vector and solving this system for displacement unknowns by means of Gaussian elimination and backward substitution or some similar direct method. It is designed to minimise the core storage requirements, the arithmetic operations and the use of peripheral equipment.

The main idea of the frontal solution is to perform assembly and elimination of variables at the same time: as soon as the coefficients of an equation are completely summed, the corresponding variable can be eliminated. The complete structure stiffness matrix is never formed as such, it is immediately sent to the back-up storage in reduced form.

The Application When the ground underneath gas pipelines undergoes changes due to, say, coal mining, the pipes experience certain stresses. This could cause problems if measures are not taken in advance to allow for these stresses. A slab of coal removed from below a section of pipeline causes the material above it, including the pipe, to subside. Transverse and axial forces are built up in the pipe section due to this movement.

The actual magnitude of these forces depends on the distance from the pipe to the removed slab. The greater this distance is, the smaller the effect on the pipe would be. When there is a bend of pipe above the slab, the pipe may experience high stress levels at the bend. The major problem that arises, however, is due to excessive axial forces in straight sections of the pipe. These can cause the pipe to buckle if preventive measures are not taken.

Two kinds of remedial action may be taken to overcome the problem: uncovering a section of the pipe or fitting bellows units. When the pipe is uncovered, it experiences substantially reduced stresses. Bellows units can accommodate larger movements than the pipe. Both these actions are taken at problem areas ie. areas where the stresses are likely to be high enough to cause buckling.

The actual mining is carried out in stages and each stage corresponds to a slab of coal being removed. The removal of each slab only affects pipework up to a certain distance from it. Data regarding planned mining strategies and the resulting predicted ground movements are provided by British Coal so that measures can be taken in advance to avoid buckling.

The Program GASP4 is a Fortran program for performing stress analysis due to ground movement. It is primarily used by the ERS for predicting stress levels in sections of pipe due to coal mining and determining the effects of remedial actions. Each run of the program corresponds to one stage of ground movement ie. the removal of one slab of coal. This affects a certain length of pipeline only.

Three-dimensional ground movement data are provided by British Coal. The sections of pipeline which would be affected by mining are divided up into finite element meshes and the ground movement data are interpolated so that they correspond to the nodes in the mesh. Apart from the ground movement data, other inputs to the program are: physical properties of the pipe, internal pressure of the pipe, soil stiffness and soil friction.

The program works out a 12 $\times$ 12 stiffness matrix for each element and calculates the RHS of each element for thermal expansion and pressure forces. The frontal solution technique is then used to determine the stress profile at each node and consequently identify problem points.

Once the problem areas have been identified, the best remedial action is found by trial and error. The choices are uncovering the pipe and fitting bellows units. The best position to take remedial action at is decided upon by certain rules of thumb. Several runs of the program with modified inputs due to remedial actions are made to obtain a satisfactory solution.

For a 1km length of pipeline with few bends, the number of nodes used is around 250 and the run time of the program is approximately 20 minutes on a VAX 8300. The greater the number of bends in the pipeline section, the greater would be the number of nodes (more nodes are needed near bends) and the run time of the program. The program may be run overnight for large problems. The introduction of friction into the analysis causes nonlinearities which increase the computational cost.

2.4.2 CRISP - Critical State Program for Geotechnical Applications

CRISP has been developed by Cambridge University Soil Mechanics Group and is used by British Gas for the analysis of geotechnical problems such as estimating ground disturbance due to construction forces or trenching and modelling soil/pipe interactions. The program determines displacement fields and stresses in such applications using the FEM and the frontal technique.

A series of load steps can be applied to the finite element model. A typical problem may involve about 500 steps. Elements can be removed from the finite element mesh to simulate excavations. Other possibilities are addition of elements, gravity input, additional accelerations (eg. earthquakes) and modelling time-dependent fluid flow.

When tackling nonlinear three-dimensional analysis, the processing times become very large (one hour per load step). An attempt is always made to simplify the problem to two-dimensions if possible.

There are two Fortran programs involved :

- The Geometry Program - this is interactive, performs simple but essential operations and is not costly
- The main program - this uses the output of the Geometry Program as input.

Because of the large amount of memory required for the program, disc I/O has to be performed if this is not available in main memory. The main program contains a subroutine (FRONTZ) which calculates stiffness matrices and solves equations. 70-80% of the processing time is spent in this routine. The program has been analysed by Edinburgh University for implementation on a transputer array. The conclusion reached at is that the program is not suited to such an architecture due to the amounts of I/O and main memory requirements (45-50 Mega-bytes).

2.4.3 DYNA3D - Impact Analysis

This program is used by the ERS for the analysis of the dynamic behaviour of offshore structures in applications such as impact analysis and assessment for blast in pipes, vessels and firewalls. The use of the program for such applications involves finite element analysis at each time step. This can be very costly in terms of computational effort for large problems involving many small time steps.

DYNA3D (Nonlinear Dynamic Analysis of Structures in Three Dimensions) is an explicit three-dimensional finite element code for analysing the large deformation dynamic response of inelastic solids and structures. A contact-impact algorithm permits gaps and sliding along material interfaces with friction. Spatial discretisation is achieved by the use of 8-node solid elements, 2-node beam elements, 4-node shell elements, 8-node solid shell elements and rigid bodies. The equations of motion are integrated in time by the central difference method. A large number of material models are available.

The aim of using this program is to determine pressure/time and pressure/displacement behaviour. Typical calculations have between 1000 to

200000 elements with tens of thousands of time steps of size in the order of hundreds of microseconds. Typical processing times are between 15 minutes to 2/3 hours on a CRAY Y-MP with four processors.

2.5 Scope for Parallelism

In this section we shall discuss some ways in which the various stages of the finite element method can be parallelised. This involves the identification of both the inherently sequential parts of the process and the parts whose parallelisation requires the control of inter-process contention.

The element matrices can be computed independently with no risk of contention. The assembly of the stiffness matrix, however, is not free from such considerations. As explained in section 2.3.3, different element matrices could have entries which contribute to the same position in the overall matrix. The contention issue must be addressed if the possibility of incorrect updates to shared memory is to be avoided. The contention problem applies to both the full and sparse representations of the stiffness matrix. In either case we need to provide mutually exclusive updating of entries in the matrix.

Chapter 6 contains a detailed study of the contention problems associated with parallel assembly for sparse representations of the overall stiffness matrix. We have also presented in that chapter algorithms which successfully overcome these contention problems and shown how to perform efficient parallel assembly of the sparse representation.

The parallelisation of the solution phase basically consists of parallelising any linear equation solver which is suited to finite element analysis. Direct methods can be parallelised by the parallel application of a Gauss step to several rows or columns of the stiffness matrix. The amount of synchronisation required to overcome contention depends on the parallelisation scheme used.

Iterative methods such as the preconditioned conjugate gradient method can be parallelised within the main iteration loop. The main problem here is the parallelisation of the triangular solves at each step. These are costly operations which are not straightforward to parallelise. We have discussed these problems in chapter 7 and presented ways of parallelising the preconditioned conjugate gradient method efficiently.

Chapter 3

Literature Review

3.1 Overview of Chapter

In this chapter we shall survey recent work related to the thesis. The main interest is in results concerning parallel finite element analysis on shared memory machines but some related topics and alternative architectures are also covered. The issues addressed aim to clarify what has been achieved so far in terms of identifying and resolving parallelisation problems and bottlenecks. The issues covered are: parallel assembly, parallel solution (mainly PCG-type methods), domain decomposition, ordering, fill-in, synchronisation costs and granularity effects. Section 3.2.1 contains a summary of miscellaneous research works on parallel assembly and solution methods. The next section (3.2.2) is concerned particularly with preconditioning and parallel PCG.

3.2 The Review

Parallel finite element analysis has been the subject of extensive research for many years. The emphasis in most works on this subject is on the solution phase since it tends to dominate the processing time. Many parallel solution schemes have been suggested for various parallel architectures. The most important part of the whole process as far as parallelisation is concerned is the solution of a set of simultaneous linear equations either to evaluate the nodal unknowns or to improve the convergence rate of an iterative scheme ie. in preconditioning. The relative efficiencies of direct and indirect methods are widely investigated.

We shall now summarise some previous work on parallelising all stages of finite element analysis.

3.2.1 Miscellaneous Topics

One method commonly used in finite element computations is *substructuring*. This involves the division of a finite element mesh into a number of subdomains which only interact at their common interfaces. Figure 3.1 illustrates this idea for the division of a domain into four substructures (S_1 to S_4).

This technique is very attractive for parallel processing because it involves significant amounts of independent operation (ie. processing the internal

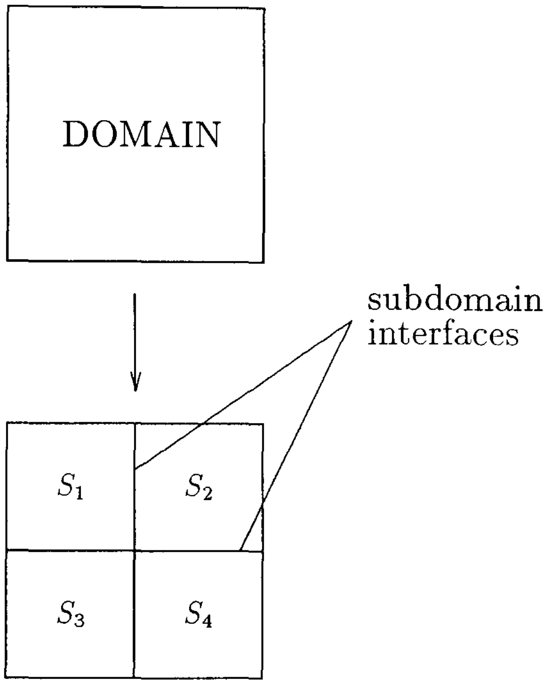


Figure 3.1: Domain decomposition in substructuring

nodes of the substructures). Equations for the nodes at the common interfaces can be formed and solved once the contributions from the substructures are known. *Colouring* schemes provide an approach to domain decomposition involving the division of the structure into lists of disjoint elements. This makes parallel processing possible within these lists.

The work in [7] reviews the inherent and induced parallelism that occurs in finite element analysis. The idea of *subdomain splitting* is presented (see section 2.3.4). This involves subdividing the domain of the problem into a number of overlapping regions and decomposing the problem into one that involves the solution of boundary value problems on the subdomains. These problems can be solved approximately using finite element techniques. Substructuring is then discussed at length as a method related to subdomain splitting. The method was introduced in the 1960's to solve large-scale structural analysis problems to avoid heavy dependence on slow out-of-core solution algorithms.

The entire structure is considered as being made up of a set of substructures. As in the usual finite element method, the entire structure is discretised but each substructure is treated conceptually as a separate domain and can be considered in parallel with other substructures. The element contributions can be calculated and assembled in the usual manner to form each substructure stiffness matrix independently, and using static condensation in which the internal nodes are eliminated the substructure stiffness matrix can be reduced in size, retaining only a few degrees of freedom of interest in the interior as well as the degrees of freedom on the interfaces between neighbouring substructures.

Since many of the internal degrees of freedom of the substructure have been pre-eliminated in the substructure calculations, the resulting final merged system involves only the retained degrees of freedom and is of much smaller size. The reason for retaining some interior degrees of freedom to each substructure may be that the evaluation of the substructure internal unknowns becomes less costly if some of the unknowns are already evaluated at this stage.

In the frontal method (see [45] and section 2.4.1), as soon as the element contributions to a nodal displacement unknown are completed, that degree of freedom is eliminated and its corresponding parts of the overall coefficient matrix are written to secondary storage. The front propagates through the domain interleaving assembly and elimination. The frontal method can consequently be thought of as an element by element strategy because of this approach.

There have been some recent developments conceptually similar to the subdomain strategies and substructure ideas for designing parallel frontal solutions. In these, elements on the left and right extremes of the domain are read to two independent processors which carry out independent frontal elimination on the system until the two fronts are later adjacent, at which time synchronisation is required and a final reduced stiffness equation at the interface is solved. The approach can be extended to multiple fronts.

In [24] a scheme for the automatic creation of substructures is suggested which leads to an overall stiffness matrix in block-arrowed form. The method is described in terms of its implementation on a distributed memory message passing architecture. Each of a set of processors is assigned initially to one substructure. Once a processor completes the processing of its corresponding substructure, it is assigned to a subset of the rows of the common interface matrix. The formation and reduction of the matrix for each subdomain requires no interprocessor communication. Each processor must, however, communicate its condensation terms to the interface block. Once the displacements are found, the determination of the stresses in each subdomain can be carried out concurrently. For a system with 2000 equations, an efficiency of 80 percent of the maximum theoretical value has been obtained using 16 processors on an Intel iPSC.

The domain subdivision algorithm in [24] is also described in [25]. A profile equation solver is used to reduce each subdomain. The subdomain interface equations are solved by a special parallel equation solver which features a concurrent LDL^T factorization as well as parallel forward and backward substitutions. The concurrent factorization involves forming and assembling separate blocks of the stiffness matrix in parallel. These blocks can then be factorized independently by different processors. After this, each processor evaluates its corresponding displacement subvector by forward and backward substitution. A trivial concurrent stress evaluation free from any interprocessor communication terminates the algorithm.

For very large three-dimensional finite element systems with, say, over 10000 equations and large bandwidth, iterative solution techniques compete with direct schemes from the computational time aspect. When these systems arise, their sparsity, and hence their low storage requirements, provide additional motivation for iterative solutions. This paper proposes an alternative to the multicolouring technique (see below) that allows almost parallel SOR iterations, without any constraint on the geometrical domain to be analysed and without any restriction on the pattern of discretisation.

Algorithms for concurrent dynamic analysis of nonlinear problems are given. The performance of the program for large linear static/dynamic problems on a 32-processor hypercube connected Intel iPSC are reported as up to 90 percent of the maximum theoretical value. The overall system has been shown to be suitable for multiprocessors with shared memory, such as the Cray X-MP series [23].

Colouring schemes are described in detail in [27]. The purpose of using colouring is to eliminate the *critical regions* from the code as far as possible. By having large amounts of independent work done by each task and minimising the amount of synchronisation, we can ensure efficient processing. Colouring divides the structure into lists of disjoint elements. Each list corresponds to a particular colour. The lists are processed sequentially but within each list groups of elements can be processed concurrently. Synchronisation only needs to be done at the list level. A minimum number of colours is desirable because it maximises the amount of asynchronous parallel work. In order to achieve this, the FE mesh must be such that large numbers of elements do not share common nodes.

The pattern of the stiffness matrix produced using colouring schemes is such that for each colour there is one diagonal and one off-diagonal block. These blocks are disjoint and consequently provide the possibility of parallel processing. The efficiency of parallel direct solution algorithms based on colouring schemes on a Cray 2 is reported to be between 95 - 99 percent (2 - 4 processors).

Concurrent iterative and direct methods are described for system solution. The iterative method is preconditioned conjugate gradient and the direct method is similar to the method used in [24] for processing the block-arrowed matrix. The software for the algorithms above has been implemented on the Encore Multimax in [27] and other shared memory machines using the *Force* [47] macros.

The Force provides a Fortran style parallel programming language utilising an extensive set of parallel constructs. It is useful because it handles process management automatically and produces portable code. The direct solver is particularly suited to the analysis of very flexible space structures which are inherently ill-conditioned. The speed ups achieved confirm the suitability of such finite element schemes for shared memory machines.

The work in [6] presents an algorithm which involves substructuring and the frontal method. The substructuring phase is very similar to the method in [24] and the algorithm has been coded for the Alliant FX/80, a shared memory machine with vector processors. The vectorised version of the frontal method has been used both for the incomplete factorization of substructure matrices (performed on different processors), and for the factorization of the matrix relevant to the global interface variables. This second step is performed sequentially. Speed ups of around 75 percent of the maximum possible value are obtained.

The problem of inefficiency in parallel code due to large I/O synchronisation overheads is addressed and it is suggested that this could be overcome by a better definition of input and output data organisation on files. The best results have been obtained when the processors were allocated well balanced large grain parallel work as would be expected.

In [26] parallel implementation of a direct factorization of a matrix using profile storage is discussed. The algorithm is based on Doolittle reduction and is sometimes called the *active column* equation solver. The U matrix is usually evaluated column-wise in sequential implementations of this factorization. In the parallel algorithm discussed in this paper, U is computed row-wise, within a column-oriented data structure. Two levels of parallelism exist: concurrency at the outer loop and pipelining at the innermost loops.

The algorithm has been tested on Intel's iPSC and the Encore Multimax. In the latter implementation, the coefficient matrix is stored as two linked lists in shared memory. To avoid memory conflicts, at each step of the factorization each processor copies into its local memory (cache) what is required from the previous step. Synchronisation is only done once, just before this data is updated. Each processor can then work on its set of columns independently, using its private values. The memory contentions due to parallel copying of this value cause only very small delays. Caching on the Multimax minimises memory contention and makes the use of private variables unnecessary.

In the forward substitution phase, the processors deal with a row-oriented skyline data structure consisting of the columns of U. During back substitution, the processors are synchronised at the beginning of the outer loop and evaluate blocks assigned to them independently in between synchronisations. A Fortran implementation of the above algorithms is given using the Force. The code can be easily extended for out-of-core implementation.

As mentioned before, tests have been carried out on the iPSC and the Encore Multimax. The iPSC has a hypercube topology. A 32-processor configuration consisting of Intel 80286 and 80287 processors has been used for the tests. The Multimax configuration used consisted of eight National Semiconductor NS32032 processors but the tests have only been carried out on up to six processors running concurrently.

Results on the iPSC show that for a fixed number of processors, higher rates of efficiency (over 80 percent) are obtained for larger size problems (order over 500). Also, for a fixed problem size, higher rates of efficiency are obtained for coarser grain configurations (ie. smaller number of processors). The Multimax shows very high efficiency rates in general. One reason for this is its coarse granularity.

For an equal number of processors, both machines show the same efficiency, but the Multimax has a much higher MFLOPS rate. For a fixed number of processors, the efficiency rate on the Multimax falls beyond a certain critical value of problem size. This is probably due to the effect of virtual memory on the execution of large jobs.

The above observations were based on tests on dense matrices. It is also reported that tests carried out on a sparse system show that the performance in the sparse case is similar to the performance in a dense system with a size similar to the frontwidth (or bandwidth) of the sparse one. In this case, both the Multimax and the iPSC yield similar execution times but different efficiency rates. The performance of the algorithm depends upon the sparsity of the system and it is optimal for nearly dense problems. The method is best applied to the in-core solution steps of a large finite element problem, where these involve a dense matrix.

The work in [79] describes a finite element method based on the fact that the displacement field calculated using displacement-formulated finite elements converges much more rapidly than the stress field. This means that a relatively coarse mesh may yield a reasonably accurate solution in displacement. This displacement solution can then be used as a displacement boundary condition for a local region in which the stress field is of interest. The local region is analysed with a refined mesh. The algorithm consequently consists of the following steps:

- A global analysis of the displacement field in the whole structure using a coarse mesh
- Stress analysis of local regions with refined meshes using the global displacements as boundary conditions for the local regions.

The local analysis for each region can be performed independently and the local regions can consequently be solved concurrently on separate processors. Parallel computation can be invoked to solve the systems of equations for each local region, thus further increasing the time saving. Parallel programming can therefore be employed in the global analysis and in each local analysis to further improve efficiency. No interprocessor communication is needed in this algorithm.

The algorithm has been tested on the Sequent Balance 21000 with 12 processors and shared memory. In this global-local analysis, one processor is used to perform global analysis with a coarse mesh with the resulting nodal displacements stored in the common memory. Subsequently, a number of processors depending on the number of local regions are used to perform the local analyses concurrently. In each process, conventional Gaussian elimination is used to solve the finite element system of equations.

Two applications have been considered: stress analysis of a thick laminate and calculating the stress field near a crack tip in a centre-cracked panel. In the cracked panel test, the global-local algorithm is shown to be about 30 times faster than a fine-mesh analysis (1000 elements) when two local regions are used. Since no interprocess communication is needed in the analysis if the finite element meshes of the selected local regions are identical, the number of local regions does not alter the computing time. Furthermore, the accuracy of the global-local procedure is affected by the choice of local regions. It is important to avoid using global nodes of questionable accuracies as boundary nodes for the local region.

In [9] parallel methods for the formation of the stiffness matrix for nonlinear large truss structures and the subsequent solution of equations are given. The finite element analysis of such nonlinear systems involves iterative formation and solution. In each iteration, a new nonlinear global stiffness matrix is updated and solved (see section 2.2.3). The algorithms are tested on the Sequent Balance 21000 with 12 processors and shared memory.

During the first step element stiffnesses can be calculated independently and placed at the appropriate locations in the global matrix. The memory contention problem can be resolved by special node numbering (substructuring) or by means of synchronisation locks in shared memory. The solution method involves parallel Gaussian elimination. The rows of the stiffness matrix are independently processed by different processors.

The results of the tests show that the use of locks may cause some loss in efficiency as the number of processors increases. Parallel Gaussian elimination is not efficient for small half-bandwidth (7) on more than six processors but produces good results for a half-bandwidth of 50 with 12 processors. The speed ups are more apparent for systems with large bandwidth as the number of processors used increases. The algorithms are particularly suited to nonlinear system solution which involves formation and solution at many incremental load steps.

The parallel assembly algorithm in [9] is similar to one discussed in a later section (Method 1, section 6.3). The speed ups are similar to those for Method 1 but are lower than those for our most efficient implementation (Method 3, section 6.5). This is because a shared memory lock is used for each entry in the stiffness matrix in [9] whereas we have used one lock per row of the stiffness matrix in Method 3. The use of fewer locks enables us to increase the granularity and hence higher parallel efficiencies are obtained. For a detailed discussion of these issues see chapter 6.

In [51] the architecture of the Cedar machine [34] is described and a block algorithm for solving banded positive definite systems is given. The Cedar 1024L is designed with a two-level memory organisation. It has 1024 processors (1 to 2 MFLOP) sharing a single common memory through an asynchronous switching network. The processors are further organised into clusters of 8 or 16 processors with additional shared memory for each cluster. This additional structure has not been used in [51]. Each processor also has its own private (local) memory. The Cedar can consequently be thought of as a shared memory machine in which each processor is also a shared memory machine.

Efficient use of such a machine involves the decoupling of an algorithm into smaller jobs, each of which uses a subset of the data. The stiffness matrix is put into a block-tridiagonal form and LDL^T factorizations are performed concurrently on independent blocks of data. The final stage of the algorithm is parallel back substitution. Tests carried out on the Cedar show that the block algorithm becomes more efficient as the size and bandwidth of the system increases. For example, for a problem of order 2^{16} and half-bandwidth 2, the speed up is 8. If the problem size is increased to 2^{20}, the speed up increases to 95. The respective speed ups for the two problem sizes with a half-bandwidth of 16 are 36 and 215.

The work in [58] describes an alternative organisation of the frontal method which is suitable for parallel processing. The aim is to reduce the book keeping and data manipulation operations associated with frontal schemes in an algorithm which offers the simplicity of band matrix solvers. The method involves interleaving the assembly and solution stages of finite element analysis as in frontal schemes. The use of the reverse Cuthill-McKee algorithm is advocated for the ordering of the nodes. One of the node numbering algorithms described produces matrices with similar profile, bandwidth and anticipated fill-in to the matrix for the corresponding problem generated by reverse Cuthill-McKee. A simple interface between the assembly and factorization phases is provided and complex preprocessing is avoided.

A disadvantage of the node-driven assembly of the stiffness matrix is that it requires more work than the usual element-driven assembly. More especially, in an element-driven assembly, the basis functions, their derivatives and the numerical integrations associated with an element (see section 2.3.2) are computed only once during the assembly of the element, while in a node-driven assembly, these have to be recalculated each time a node which is in the element is assembled.

The problems associated with the node-driven assembly may be overcome if a window-oriented assembly is used to provide the rows of the stiffness matrix to a multifrontal solver. Two windows (factorization and assembly) travel across the band matrix and each have associated with them a group of processors. The objective should be to distribute the processors among the two groups such that the assembly and factorization proceed at about the same speed. The band matrix can be stored in shared memory and minimal processor synchronisation is required.

The work in [28] considers the I/O aspect of finite element analysis. Relative CPU-I/O times quoted show that I/O manipulations can easily dominate the execution time of a finite element code. Due to the vast amounts of I/O which are present in some finite element programs, out-of-core techniques are often used. However, I/O traffic between the disk and the processor main memory slows down the computations significantly and increases even more significantly the overall cost of the analysis. Reducing the amount of time spent in data transfer is therefore at least as important as parallelising the computational phases in a finite element program.

At present only a few systems offer parallel I/O capabilities. Parallel disk access is possible on the NCUBE and the CRAY-2 offers limited multitasking I/O ie. different tasks can perform I/O simultaneously on different files. Two approaches for parallel I/O are described in this paper: one for local memory machines and one for shared memory machines. The local memory method is based on a substructuring technique. The shared memory method is purely data oriented and involves copying parts of main memory on separate disks. The results of tests performed on a CRAY-2 system with four CPU's confirm the potential for parallel processing in I/O manipulations.

In [15] we are presented with a survey and classification of currently competing algorithms for dense linear algebra. The important factors in the design of algorithms are operations counts, vectorisability, parallelisability, communication costs and scalability.

Block algorithms offer more scope for parallelism and improvements in speed. Some refinements to such algorithms are possible such as adaptive blocking ie. switching from blocked to unblocked form and variable block sizes. Blocked LU factorization on a CRAY Y-MP (block size of 64) gives good speed up for orders around 500. For sparse factorization, if the bandwidth is small (20 or so) then there is no speed up but good results are obtained for larger bandwidths (100 or so).

In [17] some techniques for parallel solution of sparse systems are described. For a general sparse system we can perform some steps of Gaussian elimination on a frontal matrix at each node of the assembly tree. Work corresponding to leaf nodes can proceed immediately and independently. When work on all sons is completed, the father node can be eliminated. The speed ups due to exploiting the tree structure only are as follows:

Number of processors	3	4	6
IBM 3090E	1.9		
CRAY Y-MP	1.8	1.9	2.3

The speed ups due to using the tree and Level 3 Basic Linear Algebra Subroutines (BLAS) are as follows:

Number of processors	3	4	6
IBM 3090E	2.4		
CRAY Y-MP	2.7	3.3	4.1

Semi-direct methods are also described. The matrix is partitioned and the subsystems are solved by a direct method (Gaussian elimination). The overall problem is then solved by an iterative scheme. Results for a block tridiagonal system on an ALLIANT FX/80 (shared memory, 8 processors) are as follows:

order	20700
entries	511050
sparsity	99.9 % approximately
no. of iterations	18 - 33
time	50 seconds
speed up	4 - 6

The speed ups can be increased by using more sophisticated iterative methods with better preconditioning and the incorporation of a better direct solver.

The work in [80] is in favour of using iterative schemes rather than direct methods because of the large computational cost and large amounts of memory needed for direct methods (see section 2.3.4). The iterative schemes described are the conjugate gradient method, GMRES and BiCG, all three of which require preconditioning. The main problem in a parallel implementation of such schemes is the preconditioning step.

3.2.2 PCG Related Works

A parallel implementation of the incomplete LU preconditioning using *level scheduling* (see section 7.3.2) is described in [63]. This involves the identification of the independent unknowns and the subsequent solution for these in parallel. The identification process is not expensive and the whole process is equivalent to reordering the rows and columns of the matrix. The maximum global speed up obtained from runs on a CRAY Y-MP (shared memory) using 4 processors is 2.65. The matrix orders range from 500 to 4500 and the best speed ups were obtained for larger systems.

In [68] an extensive survey of work on parallel conjugate gradient-type methods is given (see section 2.3.4). These methods have proved to be very useful on traditional scalar computers, and their popularity is likely to increase as three-dimensional models gain importance since these problems will involve larger systems of equations. Parallelisation at iteration loop level is reported to suffer from numerical instability (see section 7.2).

The main source of difficulty in the incomplete factorization preconditionings is in the solution of the triangular systems at each step (see section 7.3). A few approaches for implementing efficient parallel forward and backward triangular solutions are described. Among these is level scheduling which gives speed ups in the range 2-5 on an Alliant FX-8. These are similar to the speed ups for our implementations of level scheduling (see chapter 7).

The parallelisation of the dot products in the CG algorithm also constitute a bottleneck on many parallel and vector machines [68]. This is because when all the vectors in the algorithm are split equally among the processors, the dot products require global communication. However, this need not be a problem unless the number of processors becomes large.

There have been several works on the use of polynomial preconditioners (see section 2.3.4) motivated mostly by their potential on vector computers [3] [46] [69] [48] [71] [83]. However, there are doubts surrounding the usefulness of the method on parallel computers. The main attraction of polynomial preconditioning is that the only operations involving the matrix are products with vectors. We also need fewer dot products than with the non-preconditioned CG method to solve a linear system. As mentioned above, the dot products can be bottlenecks for large numbers of processors but may not cause any difficulty otherwise. Thus, polynomial preconditioning is only likely to be efficient when the number of processors is very large.

Polynomial preconditioning gives poor performance on sequential machines or parallel machines with a small number of processors. This is because this type of preconditioning can only be more efficient than the standard CG method when the cost of matrix by vector multiplication is less than half the cost of the operations in a CG step [68]. In order to satisfy this requirement, a very large number of processors must be used so that the dot products dominate the cost of a CG step. On machines with a small number of processors incomplete factorization preconditionings (ILU) produce efficient results (see section 2.3.4, [1], [85] and [40]).

The most efficient fill-in level (see section 2.3.4) for ILU preconditioning is 1 or 2, according to the work in [14]. This is in agreement with the experiments in [68] which conclude that higher level fill-ins cause the cost of the factorization to dominate the computing time. High fill-in levels are hence rarely competitive with the simpler ILU(0) or ILU(1) preconditioners [1] [73]. The ILU(0) preconditioner and those with low level fill ie.1 or 2 are computed inexpensively due to the large number of discarded entries (see section 7.4.1).

In [14] the Minimum Discared Fill (MDF) ordering technique is mentioned. This is effective in finding good ILU preconditioners especially for problems arising from unstructured finite element grids. The algorithm can identify anisotropy in complicated physical structures and orders the unknowns in an appropriate direction.

The MDF scheme is expensive for high level ILU preconditioners. Several less expensive variants of this technique are explored in [14] to produce cost-effective ILU preconditioners. These include the Threshold MDF ordering which combines MDF ideas with drop tolerance techniques to identify the sparsity pattern in the ILU preconditioners. Drop tolerance techniques involve ignoring fill-in entries during factorization which are small compared to the ratio of their corresponding diagonal entries in the original matrix (see section 7.3.3 and [57]).

Another technique introduced is the Minimum Update Matrix (MUM) ordering which is a simplification of the MDF ordering and is an analogue of the minimum degree algorithm. The MUM ordering method is especially effective for large matrices arising from Navier-Stokes problems. Numerical results in [14] show that a high level threshold MDF(l) ordering combined with a drop tolerance produces excellent results for partial differential equation problems having a relatively small molecule. This is because most of the high-level fill is small and can be ignored, but there are a few high-level fill entries that improve the quality of the preconditioner.

The level scheduling method is described in [68] and used in [85] with ILU preconditioning. The experimental results show that the ILU(0) preconditioner performs better than any other ILU(k) preconditioner with k greater than zero. The best speed up obtained on a CRAY Y-MP with four processors is 2.5. The work in [41] also involves level scheduling on a CRAY Y-MP but with eight processors. The speed ups are in the range 3-7 for different matrices. The ILU(0) preconditioner is again reported to be superior to any other ILU(k).

In [31] a parallel ILU preconditioner is described which partitions the matrix in overlapping blocks and performs local incomplete factorizations. The speed ups are up to 3.3 on 4 processors. In [54] a new preconditioner with a fast convergence is introduced. The preconditioner is an approximate inverse for symmetric matrices ie. of the form LL^T. Very efficient parallel implementations of the CG method on an Alliant FX-8 are reported and the preconditioning efficiency is also surprisingly high (improvement ratios in the rate of convergence in the range 7 - 35 depending on system size). Block preconditioners are used in [56] for parallel solution of block tridiagonal systems on a CRAY X-MP. The speed ups are reported to be near optimal. An earlier work on the same computer [71] reports a 30 percent loss of overall speed up due to the cost of barrier synchronisations.

In [67] the triangular solves of the PCG algorithm are parallelised using level scheduling on an SGI 4D/340 shared memory multiprocessor. This machine has a deep memory hierarchy (ie. one consisting of several levels) and the authors suggest that on such machines previously proposed parallelisation approaches result in little or no speed up. This is attributed to the large amounts of memory system traffic in such parallel implementations. Significant improvements in speed ups are reported by using techniques for limiting data traffic. These include data re-mappings and new processor synchronisation techniques to decrease the use of auxiliary data structures. Data re-mapping consists of the restoration of the spatial locality that was present in the sequential code but has been lost in the parallel approaches.

A detailed analysis of the effect of ordering on the performance of the PCG method is given in [18]. It is shown empirically that there can be a significant difference in the number of iterations required by the CG method depending on the original ordering of the unknowns (see section 2.3.4). Incomplete factorization preconditioners are considered the most useful in practice in terms of the acceleration they yield and being easy to generate and use (see section 7.3.1).

The orderings which give the best results are reported to be those which are *local* in the sense that neighbouring nodes in the underlying mesh (ie. unknowns in the original system) have numbers that are not too far apart. This is the case for orderings such as row ordering and Cuthill-McKee. There also seems to be in general an incompatibility between parallelism and good orderings for incomplete factorization preconditioning. This means that many of the orderings well suited to parallel processing such as the dissection methods do not give very good results. This is because they lack the locality property which gives good convergence rates. On the other hand, the non-local structure of orderings such as nested dissection is important for the decoupling property required for efficient parallel processing.

Chapter 4

Parallelisation of GASP4

4.1 Overview of Chapter

In this chapter we report on our experiences in parallelising a large Fortran program for stress analysis using the finite element method on a shared memory multiprocessor (the Encore Multimax with 14 processors). The program (GASP4) has been developed at the British Gas Engineering Research Station (ERS) at Killingworth. This program has been discussed in detail in section 2.4.1).

The next section contains a brief description of the program. In section 4.3, the run time behaviour of the program is analysed in order to identify the more computationally expensive parts. Section 4.4.1 describes the Encore Parallel Fortran (*epf*) parallelising compiler. The parallel version of GASP4 generated by the *epf* compiler is also described in this section and some timing results are presented. These results are analysed in section 4.5 where we discuss the reasons for the inefficiency of the parallel code and describe some possible techniques for obtaining better speed up.

4.2 Program Description

GASP4 is a Fortran program for performing stress analysis due to ground movement. It is primarily used by the ERS for predicting stress levels in sections of pipe due to coal mining. The structure is modelled using the finite element method and the frontal solution technique [45] forms the basis of the program. The main subroutines in the program are as follows:

DATINP Reading in the geometric and physical properties of the discretised pipe section and its surroundings.

ELTMTX A 12 $\times$ 12 stiffness matrix is calculated for each linear element and the corresponding right hand side vector is determined for thermal expansion and pressure forces. The degrees of freedom consist of variables which relate to the forces and displacements (in three dimensions) at the elements' extremities.

FRONT1 The frontal solution technique is used to determine the displacements at each node. The method interleaves the assembly of the element matrices into the overall stiffness matrix and the elimination of the variables.

PRINT1 Postprocessing of displacement data is performed to produce output files containing extensive information on predicted stresses. Problem points are consequently identified.

For a 1 km length of pipeline with few bends, the number of nodes is around 250 and the run time of the program is approximately 20 minutes. A typical realistic configuration can consist of several kilometers of pipeline divided into hundreds of elements. The greater the number of bends in the pipeline section, the greater is the number of nodes (more nodes are needed near bends) and the run time of the program. The program may be run overnight for larger analyses.

4.3 Execution Profile

In order to achieve efficient parallelisation we need to determine the sections of GASP4 where most of the execution time is spent. Although this depends to a certain extent on the kind of problem at hand and problem size, the profiles of most problems are generally quite similar. The profiling has been done using the UNIX *gprof* profiler which provides graph profile data.

The program has been profiled for a problem with 101 nodes. Even though the problem size here is small, the run time behaviour of the program is a good guide to that of larger problems. The information given below is in the form of percentages of a total sequential execution time of 14 seconds spent in each subroutine. Subroutines with insignificant contributions have been omitted.

```
* DATINP     15.7

     ELDAT    5.1
     COORDS   3.6
     DISP     2.0
     EXLOD    2.0
     LIST     1.8
```

```
* ELTMTX      9.2

    ELT12     7.6
    ELT3      1.6

* FRONT1     33.1

    ELIMIN   20.5 ------ LMPYF  5.8
                         ATBTOL 4.3

    DPRINT    7.0
    STRESS    3.1
    BAXSUB    1.4
    RPRINT    1.1

* PRINT1     41.9

    TD12A    21.8 ------ SHKDWN 18.7
    OUTPUT   13.5
    DFACT     6.4
```

The profile shows that a large amount of I/O is performed by the program. Most of the I/O is done in DATINP and PRINT1 which makes these subroutines almost completely non-parallelisable. The sequence of operations in PRINT1 cannot be reordered in a straightforward manner to make parallelisation possible since the postprocessing calculations and printing are interleaved. The main interest is in parallelising the formation of the stiffness matrix and the solution for the displacement unknowns. This involves parallelising ELTMTX (calculation of the element matrices) and FRONT1 (assembly of the stiffness matrix and elimination).

4.4 Parallelisation Using the *epf* Compiler

4.4.1 The Compiler

The *epf* compiler [21] parallelises sequential Fortran code by making an analysis of data dependency information it extracts from the program. It is one of the simpler parallel compilers but there are no alternative compilers for the Encore Multimax.

During the execution of a parallel program, tasks are only created once and subsequently idled and restarted as necessary (see section 1.4). The number of tasks is specified by the user by setting the appropriate environment variable to a desired value.

Various parallel constructs are available for insertion into the program such as DOALL loops spreading the iterations over the available parallel tasks and other constructs for handling synchronisation such as LOCK's and EVENT's. CRITICAL SECTION's provide explicit mutual exclusion. The compiler parallelises DO loops and inserts necessary synchronisations into the loops to control memory contention by parallel tasks. Loops containing READ and WRITE statements are not parallelised.

The use of some of the synchronisation constructs provided by *epf* is given below. These can be invoked in users' programs as necessary. They are also inserted by the compiler into these programs when the autoparallel switch is on.

- **The PARALLEL construct** - The code between the PARALLEL and END PARALLEL statements is duplicated for each active task. This is the basic facility for the initiation and running of parallel threads of execution within a program. Any variables declared inside a PARALLEL block are local ie. each task has a separate variable of that name and type. All other variables are shared. The tasks may continue beyond the block when all tasks have reached the END PARALLEL statement.

- **DOALL** - This construct partitions iterations of a DO loop among members of the active task set. The iterations can be synchronised by the explicit use of intertask communication. The loop index is spread randomly. The block is bound by the DOALL and END DOALL statements.

- **LOCK's** - These are variables stored in shared memory and can be set to .LOCKED. and .UNLOCKED. A WAIT LOCK statement is only allowed to proceed when its corresponding LOCK variable has not been set to .LOCKED. by another task ie. when it is .UNLOCKED. If the LOCK is taken then the task is halted at that point and can only proceed when the task holding the LOCK releases this by a SEND LOCK operation.

- **EVENT's** - These shared variables can be set to .WAIT. and .GO. WAIT SYNC stops a task from executing until the EVENT(s) indicated by the event variable(s) are completed. SEND SYNC completes the EVENT named by the indicated EVENT variable by setting this to .GO. and then unblocking any task waiting on the EVENT.

- **CRITICAL SECTION's** - This construct ensures that only a single task is allowed inside the bounded code segment at a time and that every active task will execute the code. The protected segment is terminated by END CRITICAL SECTION.

- **BARRIER's** - These provide a synchronisation mechanism used to prevent any tasks from continuing beyond a BARRIER statement until all tasks have arrived at that point.

Loops containing I/O statements and others which involve complex data dependencies are not parallelised. Consequently all loops with subroutine calls are untouched. Parallelisation of loops varies from simple array initialisations which require no synchronisation to parallel loops which contain extensive and complex synchronisations (see section 4.4.2). Also, the code structure within or outside the loop is sometimes altered and new variables are introduced in order to improve efficiency and ensure correctness. Parallelisation is sometimes made possible by the reordering of nested loops.

4.4.2 The Parallel Program

In this section we shall discuss the autoparellelised program. Some general examples are analysed together with certain specific GASP4 code sections. Since DATINP, PRINT1 and the subroutines associated with them deal mainly with I/O, they are not parallelised by *epf* except for simple array initialisations and some postprocessing in PRINT1. Subroutines ELTMTX and FRONT1 together with other subroutines called by these contain many DO loops which *epf* considers parallelisable.

There are many straightforward conversions of DO loops to DOALL loops without further synchronisation for operations such as array initialisations. One important feature of a DOALL loop is the random allocation of loop indices to parallel tasks. This means that DOALL ensures that each index is used once (and only once), but not necessarily in sequential order. This makes any loops with index I containing assignment statements such as

$$A(I) = A(I-1) + 1$$

non-parallelisable since the correctness of such loops depends upon an ordered execution of loop indices. In order to ensure correct parallel execution of such loops, a great deal of synchronisation is required which makes parallelisation not worthwhile.

The above limitation does not mean that any loop containing calculations of array elements which involve other elements of the same array is not parallelised by the compiler. For example, consider the loop:

```
DO I=1,N
    A(2*I) = I
    A(2*I+1) = A(2*I) + 1
END DO
```

The values of array elements with odd indices are dependent on the previous even indexed elements and the latter are dependent only on the loop index. The correct execution of such a loop does not require an ordered sequence of iterations. Such loops are consequently converted to DOALL loops with no further synchronisation by *epf*.

One of the major tasks of the compiler is to control memory contention due to parallel code (see section 1.4). Memory contention occurs when two or more parallel tasks attempt to update the same location in shared memory simultaneously. The result of such inadvertent access could be the loss of updates to shared variables. There are two main cases in which contention can occur in DOALL loops:

```
(i) DOALL (I=1:N)

        $A = A + function\ (I)$

    END DOALL
```

and

```
(ii) DOALL (I=1:N)

        $J = function\ (I)$

        $increment\ A(J)$

    END DOALL
```

There is an obvious risk of contention in (i). For the loop in (ii), contention is a problem if two or more values of I give the same value for J. In both cases the shared variable A must be updated under mutual exclusion. An example of such updates is in the accumulation of the element matrices into the overall stiffness matrix (see section 2.3.3). The elements of the array that are incremented in each iteration are determined by the element number (loop index). Also, if incrementing $A(J)$ involves another array element (for example during the matrix factorization), there may also be an ordering problem.

In the autoparallelised version of GASP4, mutual exclusion is achieved by the use of EVENT's. The appropriate synchronisation primitives (ie. WAIT SYNC and SEND SYNC) are inserted into the DOALL loop so that the necessary synchronisations are made to ensure mutual exclusion during updates such as (i) and (ii) above.

An example of the use of EVENT's by *epf* is given below. The code between the PARALLEL and END PARALLEL statements is duplicated according to the number of tasks that are to be used. The PRIVATE variables are local to each task. The EVENT's are stored in shared memory. TASKID provides an integer between zero to the number of tasks-1 to identify each task.

All tasks are suspended at the BARRIER statement and one is allowed to proceed once they all reach this point. After this, each task waits on its respective EVENT (II4(1) for task 0, II4(2) for task 1 and II4(3) for task 2). Since II4(1) is set to .GO., task 0 is the only task that is initially allowed to set its EVENT to .WAIT. and execute the protected statements. All other tasks are now waiting on their EVENT's.

```
EVENT II4(20)
PARALLEL
INTEGER II2,III1,I
PRIVATE II2,III1,I
II1 = TASKID + 1                  ⟸  This task's EVENT
II2 = MOD(II1,NTASKS()) + 1       ⟸  The EVENT of the
II4(II1) = .WAIT.                     task to be released
BARRIER BEGIN                         by the current task
II4(1) = .GO.
END BARRIER
DOALL (I=1:N)
perform computations which need NOT be
done under mutual exclusion
WAIT SYNC (II4(II1))
II4(II1) = .WAIT.
perform computations which MUST be
done under mutual exclusion
SEND SYNC (II4(II2))
END DOALL
END PARALLEL
```

An Example of the use of EVENT's

Once task 0 completes its mutually exclusive operations, it releases task 1 by a SEND SYNC statement. We hence have in effect some form of pipelining procedure which passes on the thread of active execution from one task to another in order. It must be remembered that each task ensures its own future suspension within the DOALL construct until it is released at the next cycle of the pipeline. Also, the statements within the DOALL loop are executed with a different value of the loop index for each task. The scheme is illustrated in figure 4.1 for the case of three parallel tasks.

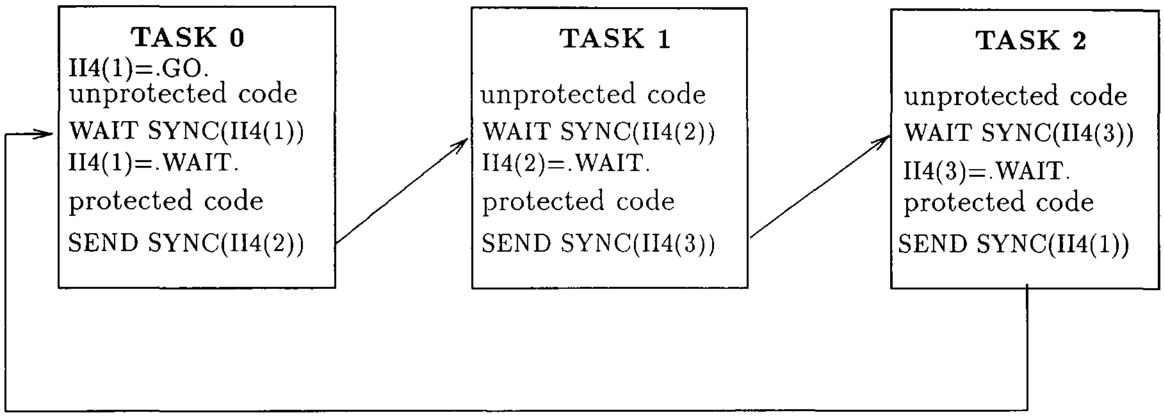


Figure 4.1: An example of synchronisation using EVENT's

The efficiency of this scheme is determined by the amount of work which is done in parallel compared to the cost of synchronisation. This means that we require sufficiently large amounts of processing in the unprotected segments to compensate for the cost of implementing the protected zones. This issue is discussed in detail in section 4.5.1.

4.4.3 Timing Results

The results obtained from timing the parallel version of GASP4 generated by the *epf* compiler are presented in table 4.1. Two different mesh sizes have been used. N_p is the number of processors. All times are in seconds.

We should not expect any significant improvement in the results for very large problems since the program would still be using small grain sizes (see section 1.6) which prove to be inefficient. This is the subject of discussion in the next section.

Table 4.1: Performance of Autoparallelised GASP4

N_p	101 nodes	441 nodes
Sequential	17.85	39.86
1	19.83	46.22
2	27.68	60.80
4	27.75	61.17
6	28.91	65.55
8	30.16	66.12

4.5 Ideas for Efficient Parallelisation

4.5.1 Reasons for Inefficiency

The timing results in section 4.4.3 show that we are not benefiting from the parallelisation inserted into GASP4 by the compiler. Moreover, there is an increase in the processing time as the number of processors increases. This is due to the increasing amount of overhead associated with setting up parallel tasks and controlling synchronisation. Since we do not gain sufficiently from parallelism to compensate for this overhead, there is an overall increase in execution time. The sudden increase in execution time when two processors are used is probably due to the synchronisation overhead associated with accessing shared data.

The main reason for the inefficiency of this code is the level at which parallelism is implemented. In order to make more benefit from parallelism we need to allocate to each parallel task approximately equal amounts of work and make sure that the work done in between synchronisations is sufficiently large to compensate for the overhead.

Since the data dependency analysis is too complex at the higher levels, *epf* makes use of a great deal of small grain parallelisation with a large number of synchronisations. This is very costly and in many cases would not be necessary if certain modifications were made to the sequential code. Also, there are cases where the compiler inserts synchronisation into the code without realizing that there is no need for protection.

4.5.2 Ideas for Hand Parallelisation

In order to implement parallelism at levels higher than those done by *epf* we need information on data dependency from the designers of GASP4. We have been provided with some such information and have used this to provide examples of how the program can be hand parallelised. The actual hand parallelisation of the whole program is a major exercise with little research value. We have consequently concentrated on providing general guidelines rather than attempting to write a parallel version of the program.

Many large loops are not parallelised by *epf* because they contain I/O statements. It is possible, however, to parallelise some of these loops bearing in mind the following facts:

- WRITE statements can be left in DOALL loops as long as they can be done in a random order. If their order is important and depends upon the loop index then we should try to take these out of the DOALL loop if possible and execute them sequentially after the loop.
- READ statements can also be left in DOALL loops and the same considerations should be made as for WRITE statements.

An example of such loops is in subroutine ELTMTX in which ELT12 or ELT3 is called for each element. It is possible to have parallel calls to these subroutines which means that the element loop can be spread. The only difficulty is that both of these subroutines write to the same file and the order in which the writing is done with respect to the element number is important. The only way to overcome this is to store the data to be written such that the WRITE statements can be taken out of the DOALL loop. The rest of the processing can be done in parallel.

Parallel READ's are useful in cases such as reading in element and nodal data for finite element meshes if the order in which the data is stored is not important. In practice, parallel READ's tend to cause run time errors in many cases. This is caused by the simultaneous access to the file pointer by parallel tasks which can inadvertently place incorrect values into program data sections.

In many loops there are statements for updates of variables. If these loops are to be parallelised, the updates must be done under mutual exclusion. An example of the contention problem of type (i) of section 4.4.2 is in subroutine

COORDS. The variable IERSUM is incremented by one in each iteration. In the parallelised loop, this must be done in a CRITICAL SECTION.

We can also employ a certain strategy to make the parallel loop more efficient. Instead of having the main update in the DOALL loop, we can accumulate variables local to each parallel task in each iteration and then have each task perform a mutually exclusive update of the shared variable (IERSUM) after its allocated iterations. In this way the overhead associated with the execution of the CRITICAL SECTION construct is minimised since it is executed only once by each parallel task rather than once for every loop index. The code is as follows, where TMPIER is a local variable to each parallel task:

```
PARALLEL
   INTEGER TMPIER
   DOALL (I=1:NNP)
      ⋮
      TMPIER = TMPIER + 1
      ⋮
   END DOALL
   CRITICAL SECTION
      IERSUM = IERSUM + TMPIER
   END CRITICAL SECTION
END PARALLEL
```

One could also experiment with implementing the update of the global variable (IERSUM) in parallel steps so as to minimise the number of addition operations necessary and hence minimise the cost of synchronisations using CRITICAL SECTION's. This will only be beneficial if the cost of the synchronisations required for the parallel additions are low enough to make their implementation beneficial.

One further strategy for improving the efficiency of the parallel code involves the elimination of the cost of the DOALL construct. This can be achieved by a small amount of preprocessing to allocate to each parallel task a block of consecutive loop indices and setting the tasks to process their respective blocks in parallel. In order to achieve good load balancing, the

Table 4.2: Performance of Hand Parallelised GASP4

N_p	Time	
	Hand	Auto
Sequential	4.21	4.21
1	4.42	4.71
2	3.17	5.14
4	2.79	5.62
6	2.58	6.02
8	3.05	6.36

blocks must be of equal or nearly equal size. This scheme has proved to increase the efficiency significantly for large loops.

An example of updates of type (ii) of section 4.4.2 is in subroutine IELREF. There is a loop containing an update of the array element NREF(INOD). If each loop index does not necessarily give a unique value of INOD, there is a contention problem when this update is made. In order to test the above ideas for parallelisation, we have used them in subroutine ELTMTX (see section 4.2). The timing results for the 441 node mesh (see section 4.4.3) are presented in table 4.2 as processing times for ELTMTX (in seconds) together with the corresponding times for ELTMTX using autoparallelisation.

We can see that there has been some benefit in using the mentioned parallelisation strategies even though this has not been very large. Better speed ups can be expected for larger meshes. We must also bear in mind the extra cost associated with reordering the code to make it parallelisable. Certain sections of the program require extensive reordering to improve their condition. This is not a good way to go about designing parallel code since by doing so at this stage we may make the underlying sequential algorithm less efficient.

The processing times for ELTMTX using autoparallelisation show that there is no benefit due to parallel processing. In fact the processing time increases as N_p is increased. For example, when using 8 processors autoparallelisation takes more than twice as long as hand parallelisation when processing ELTMTX. The reasons for the lack of efficiency of the autoparalellised code were discussed in section 4.5.1.

4.5.3 Some Suggestions for Approaching Parallelisation

Parallelising GASP4

Given the limitations of *epf*, significant benefits from parallelisation can only be obtained by starting with the sequential code and using the information provided by its designers to make suitable modifications which would allow us to parallelise at higher levels. The task of simple parallelisations such as spreading independent iterations over available tasks can then be left to the compiler.

In order to benefit from parallelisation to the largest extent we need to parallelise DO loops containing calls to large subroutines. We consequently need to know whether it is possible to run parallel copies of these subroutines. If shared data are updated in the DO loop or its associated subroutines, they must be protected during updates by constructs such as CRITICAL SECTION's. The use of such constructs should be kept as low as possible to avoid large overheads. Strategies such as accumulating local sums and blocking the loop indices discussed in the previous section can be used for such purposes. In some cases, parallelisation of loops can be made possible by the reordering of nested loops or rearranging the loop structure such that some parts of the loop are executed sequentially in separate loops.

Whenever possible, I/O statements should be taken out of loops before parallelisation. If they cannot be taken out then the order in which they would be executed should be noted. DO loops which do small amounts of work must not be parallelised.

Writing Parallelisable Code

The best strategy for writing sequential code that can be parallelised easily and efficiently is to bear in mind the following points:

- The code must not contain complex data dependencies which are difficult to analyse. These usually involve the dependence of array subscripts on calculations within or outside the loop.
- Subroutines in DO loops must be written such that parallel copies of them could be run with small amounts of simple synchronisations. It

is consequently better if most of the processing in subroutines involves local variables, keeping access to global (shared) variables to a minimum. The use of parameters for subroutines helps in writing such code and is to be preferred to using COMMON block data.

- I/O statements should be performed independently of computation and in separate DO loops whenever possible.

In general, sequential code must be written such that it performs large amounts of independent work with simple and clear data dependencies. The use of GO TO and RETURN statements should be avoided whenever possible since they add to the complexity of the code.

Chapter 5

The Model Program

5.1 Overview of Chapter

The aim of this chapter is to describe the finite element model program we have developed for the purpose of experiments on parallelising the method. The model forms the basis of several Fortran programs used for testing various aspects of parallel finite element analysis.

The basic model structure is first described together with an explanation of the sparse storage scheme used for the stiffness matrix. An example run is presented to illustrate the program user interface.

5.2 The Model

The model described here is a Fortran program which can be used for the numerical solution of linear two-dimensional problems of the type discussed in chapter 2. The example problem in section 5.4 involves a stress field and the program is used for two-dimensional stress analysis. The boundary conditions are specified as fixed displacements at particular nodes. Physical and geometrical properties and information on applied forces are input to the program. The output is in the form of displacements at various points in the problem domain. The displacements can then be used to compute stresses at these points using stress/strain relations involving the characteristics of the elements in which they occur.

One important feature of the model program is that it allows experimentation with different meshes, assembly strategies, solution schemes and various numerical techniques for each phase of finite element analysis. This can be done by inserting alternative program modules in relevant sections of the code while preserving the overall program structure.

A separate program has been written which generates suitable regular two-dimensional meshes and provides for the main program data concerning the connectivity and positions of nodes. We have used both linear triangular and bilinear quadrilateral elements on a square mesh. Other types of elements can, however, be easily accommodated for analysis. The node orderings used are row and red-black orderings.

The program can be used for the solution of a wide range of finite element problems by supplying the appropriate element matrix routines. The solution phase of the program can be modified for the solution of nonlinear problems (see section 2.2.3). The assembly and solution procedures are otherwise independent of problem type.

The model program is structured as follows:

Data Input Reading in the geometrical properties of the structure in the form of nodal connectivity information and nodal coordinates; reading in the physical properties of the structure; reading in applied forces.

Stiffness Matrix Formation Element matrices are calculated and immediately assembled. The mathematical formulation of the element matrices is explained in section 2.3.2. To calculate an element matrix, its nodal coordinates are converted to those of a natural coordinate system ie. one which permits the specification of a point within the element by a dimensionless number whose absolute magnitude never exceeds unity. The numerical integration of the shape functions evaluated at the nodes is done by the Gauss-Legendre quadrature using four sampling points for each element. The result is an element matrix representing the behaviour of the element under stress.

The element matrix is then added on to appropriate positions in the overall stiffness matrix. The contribution of the right-hand side is also calculated at this stage and added on to the force vector (see section 2.3.2). The stiffness matrix formation can be achieved by using any sequential or parallel assembly strategy. Some possible implementations are discussed in the next chapter.

Displacement Evaluation Nodal displacements are evaluated by solving the resulting system of simultaneous equations. This is done by the preconditioned conjugate gradient method. An incomplete LU preconditioner is calculated and used in the subsequent steps of solution. For the details of the formation of the preconditioner and the implementation of the preconditioned conjugate gradient method see chapter 7. The reasons for choosing the PCG method with ILU preconditioning in our model are given in sections 2.3.4 and 7.2. It must also be said that it is possible to use methods other than PCG for this stage. Examples of these are direct methods such as Gaussian elimination.

5.3 Storing The Stiffness Matrix

The overall stiffness matrix is very sparse and its sparsity increases substantially as matrix order increases. The sparsity can be exploited to avoid the large amounts of storage required when the size of the matrix (ie. the number of degrees of freedom) is large. In practice, most real problems give rise to large stiffness matrices. In order to benefit from parallelism to the fullest extent we need to consider such large systems which require a large amount of storage.

To overcome the storage problem we have designed our program so that the stiffness matrix is represented in one of the standard Fortran representations of such matrices [16], as a row-linked list using four one-dimensional arrays. These arrays can be abstracted to a table consisting of the following entries:

- IROWST - position of the first entry for each row in the table (integer array)
- JCN - column number of entry (integer array)
- VAL - value of entry (real array)
- LINK - pointer to the next table entry for a row (integer array).

IROWST must be of size equal to the order of the stiffness matrix whilst the other three arrays must be as large as the number of entries in the stiffness matrix. Figure 5.1 illustrates how the storage scheme can be used to store the sparse matrix A.

In the table, a zero LINK denotes the end of a row. In order to locate an entry in a row we must follow the LINK's through the table starting from the position indicated by IROWST. Let us illustrate this with an example. If we were to look for the fourth entry in row 2, we would start by looking for the start of row 2 ie. IROWST(2). This has the value 1, meaning that row 2 starts at position 1 in the table. JCN(1)=1 and VAL(1)=6 which shows that A(2,1)=6. LINK(1)=4 and in this position of the table we find that A(2,4)=4. We have found the required matrix entry. LINK(4)=0 indicating that there are no further entries in row 2.

$$\begin{pmatrix} 4&0&2&0&0 \\ 6&0&0&4&0 \\ 0&0&2&5&0 \\ 1&0&0&0&7 \\ 0&3&2&4&0 \end{pmatrix}$$

A

subscript	1	2	3	4	5	6	7	8	9	10	11
IROWST	3	1	8	5	2						
JCN	1	3	3	4	5	1	2	3	4	1	4
VAL	6	2	2	4	7	4	3	2	4	1	5
LINK	4	7	6	0	10	0	9	11	0	0	0

Figure 5.1: Sparse Representation in Tabular Format

This sparse representation provides a very efficient means of storage. As an example, one of our test problems with 8281 degrees of freedom is represented fully by approximately 70 million (8281 $\times$ 8281) reals whereas the sparse representation only requires about 200000 integers and 100000 reals. There is consequently a substantial saving in storage when the sparse data structure is used and this allows us to process larger systems and test these for speed up.

The entries in the table have not been ordered. For example, one could place the entries for a particular row of the stiffness matrix in column order. This ordering has associated with it the cost of moving existing entries during assembly when necessary. One advantage of such an ordering would be in the assembly phase. When searching through the entries in a particular row (see algorithms for Methods 1 and 2, chapter 6) to determine whether a certain entry exists or not, we would not necessarily need to go through the whole row. This is because if we go past the column number for the entry to be assembled, we would know that the entry has not appeared before. We then need to perform an insert rather than an update operation. There can consequently be some saving in processing time due to the ordering in this way.

Other benefits due to column ordering could arise during the factorization of the stiffness matrix (see chapter 7). It could be useful to use the information provided by the ordering when we are trying to obtain factors with particular characteristics (see section 7.3). Due to the mentioned costs associated with ordering we have assembled the stiffness matrix unordered.

It is unlikely that the savings would be larger than the costs due to ordering unless some efficient means of ordered assembly is designed.

The factors obtained by the incomplete LU decomposition of the stiffness matrix are stored in the same tabular format, linking entries in the same column. An analysis of the scope for the parallel assembly of the stiffness matrix is made in chapter 6. We also present there algorithms for achieving efficient parallel sparse assembly. The solution of the system of equations represented by the stiffness matrix is done using the preconditioned conjugate gradient method. Chapter 7 contains a description of the method and suggests ways of implementing it in parallel efficiently. The solution algorithms are tested with row and red-black orderings of the nodes in the underlying mesh.

5.4 An example run

Consider the row-ordered mesh in figure 5.2 which consists of quadrilateral and triangular elements. The element numbers are in bold in the diagram. The number of elements is 31 and the number of nodes is 36. The corresponding input and output data format for the program are summarised below.

Input Format

```
31  36                            (number of elements, number of nodes)
1  0   3  1  2  7                 (element number, element type, element nodes)
2  0   3  7  2  8
3  0   3  2  3  8
4  0   3  8  3  9
5  1   4  3 4 10 9
6  1   4  4 5 11 10
.
.
etc.
1  0.  0.  0.0                 1      (node number, x, y, z coordinates,
2   1.00000  0.  0.0           1       nodal degrees of freedom)
3   2.00000  0.  0.0           1
```

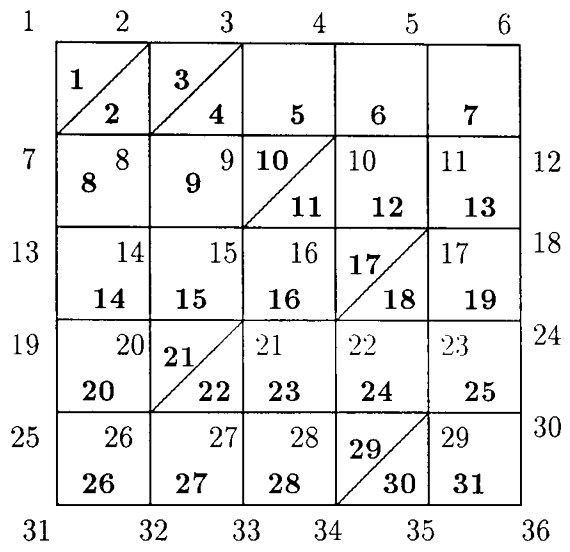


Figure 5.2: An example two-dimensional mesh for the model program

```
  4   3.00000  0.  0.0              1
  5   4.00000  0.  0.0              1
  6   5.00000  0.  0.0              1
  .
  .
  etc
  1  100                 (node number, applied force)
  2  100
  3  100
  4  100
  5  200
  6  200
  .
  .
  etc
12         (number of boundary conditions)
1   0.0    (node number, fixed displacement)
```

```
6   0.0
7   0.0
12  0.0
13  0.0
18  0.0
  .
  .
  etc
```

Output Format

```
NODAL DISPLACEMENTS
1       0.                      (node number, nodal displacement)
2         282.14198006880
3         447.78501315771
4         460.56186475065
5         346.10553005216
6       0.
.
.
etc
```

It can be seen that the program uses the geometrical and physical properties of a particular problem to evaluate the relevant nodal unknowns. In the case of a problem in the field of structural analysis, a discretised body which is subjected to external forces is analysed to provide a displacement profile. As mentioned before, the assembly and solution procedures can be used for *any* type of problem. They only need to be supplied with the element matrices which specify the problem characteristics.

Chapter 6

Parallel Assembly Methods

6.1 Overview of Chapter

In this chapter we present methods for the assembly of several element matrices in parallel into the same sparse representation of the overall stiffness matrix (see section 5.3). As noted in section 2.5, if the stiffness matrix is to be updated in shared memory we require certain synchronisation schemes to ensure the correctness of the resulting matrix. We also need to make these schemes as efficient as possible to achieve good speed ups.

Some algorithms for parallel sparse assembly are presented and tested for different mesh sizes. The performance of these algorithms is shown in terms of their run times and their efficiency is discussed.

6.2 Sources of Contention

The element matrices can be evaluated in parallel without any need for synchronisation. The main source of contention in the parallel element by element assembly of the stiffness matrix is the sharing of nodes by adjacent elements in the mesh. This causes different element matrices to contribute to the same positions in the overall stiffness matrix (see section 2.3.3).

The effect of the node sharing is that if we are to perform parallel assembly of different element matrices into our sparse representation, we would need to resolve the problem of simultaneous access to the same table entry by parallel tasks. We discuss this issue in detail below and suggest ways of controlling this contention problem. The only other source of contention in a parallel assembly scheme for the sparse representation of the stiffness matrix is access to the table subscript during read and update (see sections 6.3 to 6.5).

The entries in the sparse representation of the stiffness matrix are stored as a number of rows whose entries form a linked list (see section 5.3). We hence have two main options when deciding on the design of a parallel element by element assembly scheme: allow simultaneous assembly of the same row by parallel tasks or allow only the assembly of different rows.

The first option involves a great deal of potential memory contention which can only be overcome by complicated and costly synchronisations. Possible contention problems are:

- (i) Simultaneous *rowstart* - Parallel tasks contending for the same *rowstart* array position. For example, two tasks may both be initiating a new row and attempting to assign the same array position to the start of their respective rows.
- (ii) Simultaneous update of the same entry - Parallel tasks attempting to update the same existing table entry.
- (iii) Simultaneous creation of the same entry - Parallel tasks attempting to create the same table entry.

If we only allow the assembly of different rows simultaneously, however, we only need to provide enough synchronisation to avoid inadvertent access to the table subscript indicating the next free table position. All other sources of contention which involve simultaneous access to the same entry in the stiffness matrix are eliminated since we are always dealing with distinct entries at any one time. We have implemented both these approaches as explained below. The results of tests for the efficiencies of the implementations are given in section 6.6.

6.3 Method 1 : Parallel assembly of the same rows

The implementation of this scheme involves contention problems (i) to (iii) in section 6.2. We have overcome (i) and (iii) by using a single LOCK variable for the whole table (SUBSC) which is used to protect the shared data during searching and insertion. Simultaneous update of an existing entry is protected by an explicit critical section mechanism.

The use of the shared memory LOCK and its associated WAIT LOCK and SEND LOCK operations (see section 4.4.1) ensures mutually exclusive access to the table entries during operations such as the insertion of a new row, searching through existing entries and the insertion of new entries. Also, the table subscript is always read and updated under such exclusion. Since it is possible to update existing entries while other operations are taking place, a CRITICAL SECTION primitive is all that is required to protect the entry. The following pseudo-code section outlines the algorithm:

```
DO in parallel (I=1,number of elements)
   FOR each row (J) of element matrix DO
      IF row (J) has not appeared before
         in the overall matrix THEN
         WAIT LOCK (SUBSC)
            create new row —(A)
         SEND LOCK (SUBSC)
      ELSE
         FOR each entry in row (J) DO
               IF entry has appeared before THEN
                  CRITICAL SECTION
                     update it —(B)
                  END CRITICAL SECTION
               ELSE
                  WAIT LOCK (SUBSC)
                     insert it —(C)
                  SEND LOCK (SUBSC)
               END IF
            END FOR
      END IF
   END FOR
END DO
```

Algorithm for Method 1

The locking mechanism ensures that when a task is in regions (A) or (C) no other tasks are allowed in that region until the LOCK is released by a SEND LOCK operation. The CRITICAL SECTION construct ensures that while a task is in region (B) no other tasks are allowed into that region until this task has completed the execution of (B). The difference between the two circumstances is that a CRITICAL SECTION only protects one section of code whereas any number of sections may be protected by the same LOCK. We can consequently see that LOCK's provide us with the ability to implement *named* protected regions consisting of several code segments.

6.4 Method 2 : Parallel assembly of different rows

Our scheme for ensuring parallel assembly of different rows only involves the use of two types of LOCK's. We associate with each row I of the stiffness matrix a shared memory synchronisation variable called ROWLOCK(I). All these are initialised to be available ie. unlocked.

Before attempting to assemble a row of an element matrix, each task must first check the corresponding ROWLOCK and can proceed with the assembly if the LOCK is not taken. The check is done using the WAIT LOCK construct which will allow a task to proceed and lock the row if the LOCK is not already taken. Otherwise, the task is suspended at that point until it is released by a SEND LOCK from another task indicating the completion of the assembly of the row it was waiting on by that task.

All the table operations such as the creation of a row, creation of an entry and the update of an existing entry may proceed concurrently by parallel tasks as long as the shared table subscript (ie. the next free entry in the table) is read and updated under mutual exclusion. We implement this by using a single LOCK called SUBSC. Each task must WAIT LOCK on SUBSC before reading the subscript for its local purpose and updating it. The lock is then released and can be taken by a suspended task. The task's local job of allocation and setting up of new entries can proceed after the release using the local value of the subscript. The outline of the algorithm is as follows:

```
DO in parallel (I=1,number of elements)
    FOR each row (J) of element matrix DO
        WAIT LOCK (ROWLOCK(J)) ⇐ Point (A)
            IF row (J) has not appeared before
                in the overall matrix THEN
                create row (J)
                insert entries in row (J)
            ELSE
                FOR each entry in row (J) DO
                    IF entry has appeared before THEN
                        update it
                    ELSE
                        insert it
                    END IF
                END FOR
            END IF
        SEND LOCK (ROWLOCK(J))
    END FOR
END DO
```

Algorithm for Method 2

The table subscript is read and updated during the insert operations in the above algorithm as follows:

WAIT LOCK (SUBSC)

- read the subscript and store it locally
- update shared subscript

SEND LOCK (SUBSC)

Once a task completes assembling a row it releases its corresponding ROWLOCK, thereby allowing any other tasks suspended on this LOCK to proceed. These are tasks which are attempting to assemble neighbouring elements (which share common nodes and hence ROWLOCK's). The time spent by tasks waiting at point (A) above can consequently be reduced by minimising the assembly of adjacent elements at the same time. This idea is the motive behind an improved version of Method 2 which is discussed below.

6.5 Method 3 : Parallel assembly of different rows - improved version

A modified version of the parallel assembly scheme presented in the previous section is now given. The aim is to minimise the synchronisation overhead associated with the locking of the rows.

Tasks assembling adjacent elements in the mesh simultaneously will be suspended if they attempt to assemble a locked row. The cost of this waiting can be minimised by assembling disjoint elements concurrently as far as possible. This can be achieved by a colouring scheme in which all elements belonging to a particular colour are disjoint.

Consider a mesh with row-ordered elements divided into four colours as shown in figure 6.1. Elements of the same colour can be grouped together in array COLOURS such that the following four groups are stored one after another:

1 3 5 11 13 15 21 23 25
6 8 10 16 18 20
2 4 12 14 22 24
7 9 17 19

By mapping the row-ordered mesh onto this array we can reduce the number of adjacent elements assembled in parallel. Method 3 can consequently be implemented by the following modification to the algorithm for Method 2:

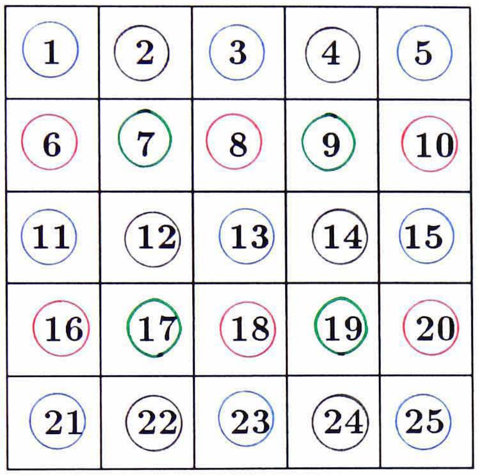


Figure 6.1: Division of a mesh into several colours

replace

```
DO in parallel (I=1,number of elements)
    assemble element(I)
END DO
```

by

```
DO in parallel (I=1,number of elements) in ascending order
    assemble element(COLOURS(I))
END DO
```

The coloured array can be formed by simple functions traversing the mesh and adding on their corresponding elements. These functions can pick combinations like (odd row,odd column), (even row,odd column) and so on.

Four-colour and Cuthill-McKee orderings [18] provide more general types of meshes which can be assembled with minimal contention. The cost of obtaining the colouring is small and it provides an efficient way to reduce waiting time on locked rows. Also, we can see that for the assembly phase it is the ordering of the elements that is of importance rather than the ordering of the nodes.

6.6 Results

Table 6.1 shows how the parallel assembly schemes presented in this chapter perform for a regular rectangular row-ordered finite element mesh with 40000 elements (40401 nodes). Table 6.2 compares the speed ups of the improved version for three different problem sizes. N_p is the number of processors. Sp is the speed up. All times are in seconds.

Table 6.1: Performance of the three methods of parallel assembly

	Method 1		Method 2		Method 3	
N_p	Assembly Time	Sp	Assembly Time	Sp	Assembly Time	Sp
Seq.	111.36	1.00	111.36	1.00	111.36	1.00
1	125.12	0.89	117.12	0.95	119.35	0.93
2	78.42	1.42	64.74	1.72	59.55	1.87
4	44.02	2.53	33.04	3.37	29.54	3.77
6	32.12	3.47	24.75	4.50	20.78	5.36
8	23.79	4.68	18.84	5.91	15.51	7.18
10	21.65	5.14	16.81	6.62	13.04	8.54
12	20.19	5.52	15.75	7.07	11.12	10.01

The results are illustrated graphically in graphs 6.1 and 6.2. The former represents the variation of speed up with number of processors for the largest problem and the latter shows how speed up is related to problem size. The results are discussed in the next section.

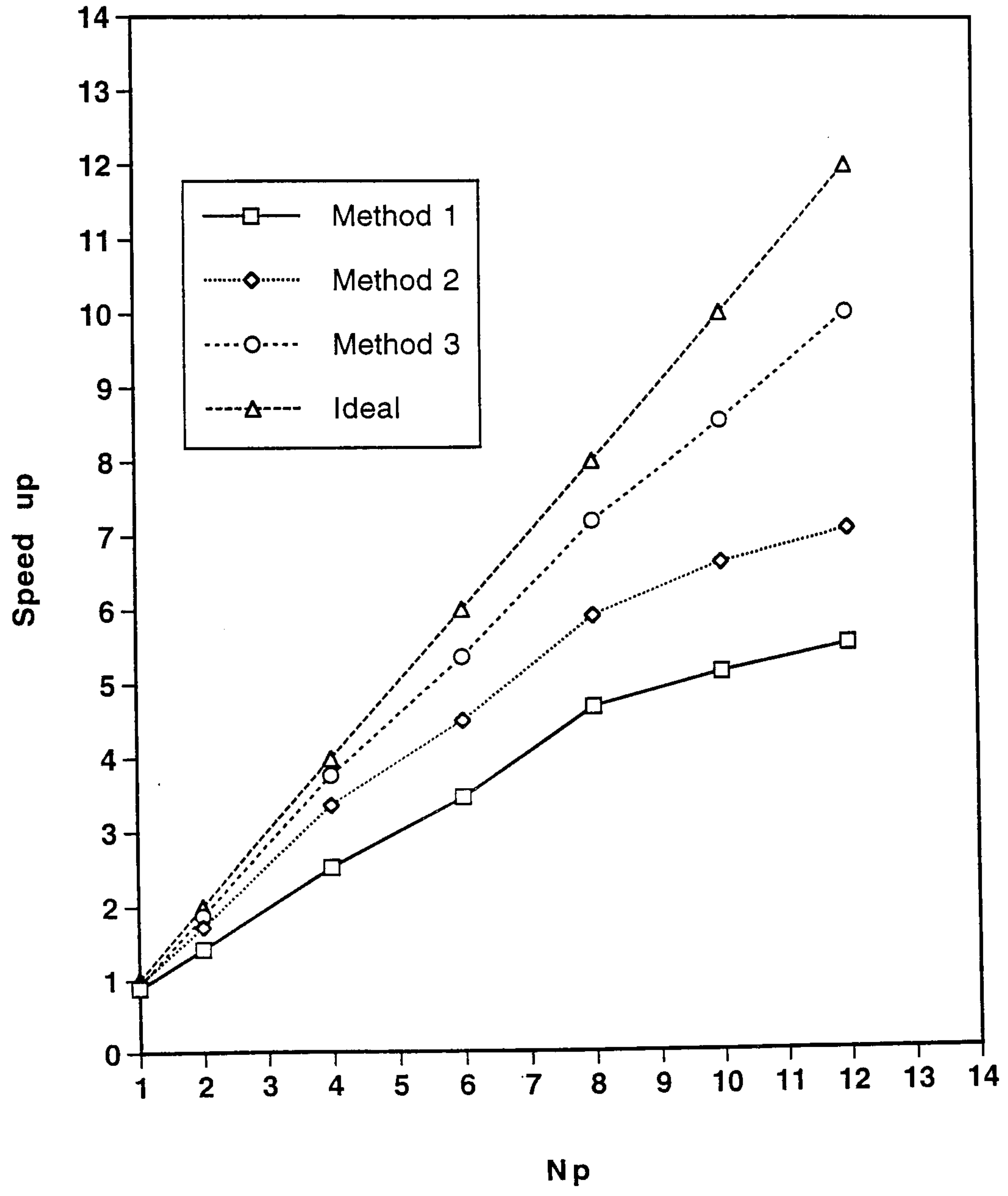
Table 6.2: Relation of Speed up to Problem Size for Method 3

N_p	400 elements	2500 elements	40000 elements
Sequential	1.00	1.00	1.00
1	0.97	0.96	0.93
2	1.55	1.61	1.87
4	2.64	2.92	3.77
6	4.07	4.35	5.36
8	3.78	5.93	7.18
10	3.16	6.29	8.54
12	3.01	7.78	10.01

Graph 6.1

Performance of the Assembly Methods

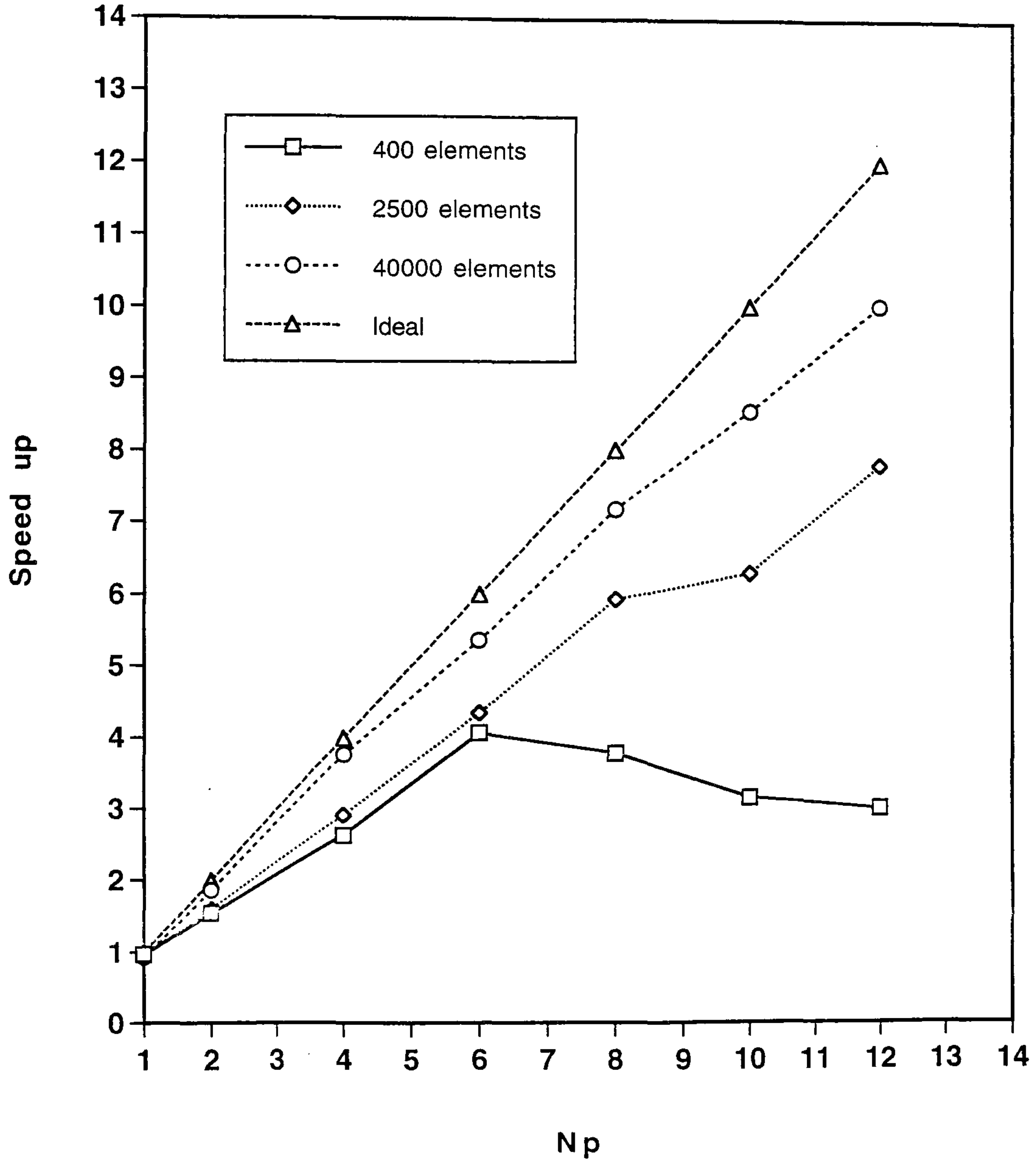
40000 elements



Graph 6.2

Variation of Speed up with Problem Size

Method 3



6.7 Discussion of Results

For all three methods we observe a reduction in parallel efficiency as N_p increases. This is inevitable and is due to the increasing overheads associated with setting up and managing parallel tasks. We have aimed to minimise this reduction, however, by providing good load balancing and high computation to communication ratios.

We observe from table 6.1 that when we use one processor, all three parallel implementations are slower than sequential assembly. This is because when $N_p = 1$ we are paying certain overheads due to the extra costs associated with parallelisation without performing any parallel processing. We consequently have some extra work to do without any benefit due to parallelisation. For larger numbers of processors this extra work is compensated for by a gain from parallel processing.

The mentioned loss at $N_p = 1$ is more pronounced for algorithms with a larger amount of synchronisation or extra processing for parallelisation. Hence, Method 1 has the largest processing time when $N_p = 1$, followed by Method 3 which has a few more computations to perform than Method 2.

Table 6.2 reveals another point related to the performance of the algorithms at $N_p = 1$. We observe that for a particular parallel implementation, as the problem size increases, so does the extra cost associated with parallel processing. This means that when using one processor, the parallel efficiency of the algorithm falls as the problem size increases. The benefit due to parallel processing when using two or more processors causes the opposite effect such that the parallel efficiency of Method 3 rises with problem size in those cases.

As shown by graph 6.1, parallel assembly using the *same rows* scheme is not very efficient. This is to be expected due to the large amounts of synchronisation required by this algorithm. This implementation uses low grain parallelism which does not bring about large enough speed ups to compensate for the synchronisation overhead sufficiently. The poor performance for large numbers of processors emphasises this fact.

The use of ROWLOCK's in the second approach increases the granularity substantially and hence this algorithm performs better than the previous one. We are now doing sufficiently large amounts of work between the synchronisation points to compensate for the cost of their implementation satisfactorily. Some waiting is done on locked rows which obviously degrades the performance to a certain extent. Good parallel efficiency is obtained with small N_p for Method 2.

The colouring algorithm eliminates some of the waiting on ROWLOCK and hence provides further speed up. We now have a suitable grain size and assemble the elements such that they are less likely to contend for the same row update. The parallel efficiency for Method 3 is consequently very high with any N_p.

As before, the improvements compared to the uncoloured algorithm are more significant when using large numbers of processors. For example, when $N_p = 2$, the speed up ratio between methods 2 and 3 is 1.09. For $N_p = 6$ and $N_p = 10$, the ratios are 1.20 and 1.29 respectively. The reason for this increase is that when N_p is large, an inefficient algorithm would suffer from its shortcomings more significantly due to increased overhead. By the same token, any improvements in an algorithm would produce larger benefits when N_p is large.

Let us now consider the effect of problem size on speed up for Method 3. We have chosen sufficiently different mesh sizes to illustrate this point. As can be seen from graph 6.2, the largest meshes can be assembled concurrently with near linear speed up. Increased overhead degrades performance for large numbers of processors but the granularity is large enough for this problem size to produce good speed ups for any value of N_p. For the 2500 element mesh, however, the granularity is much lower and the synchronisation overhead degrades the performance rather more significantly. The efficiency is not very far from ideal for small numbers of processors but when N_p is large, the speed ups obtained for the larger problem are much better than those for this mesh.

For the smallest mesh we observe fairly good speed ups for small N_p (up to six processors) which are comparable with those for the larger meshes. When more than six processors are used, however, the speed up reduces with increasing N_p and for the largest numbers of processors the values are much lower than the corresponding ones for the larger mesh sizes. The reason for the poor performance with such small mesh sizes is that the granularity is too low to provide a good computation to communication ratio. We are not doing enough work between synchronisations to obtain satisfactory parallel efficiencies.

The algorithm therefore performs far more efficiently when large meshes are assembled. This is a quite typical characteristic of parallel schemes involving non-trivial synchronisation strategies. For further discussion of the results see section 8.3.

Chapter 7

Parallel Preconditioned Conjugate Gradients

7.1 Overview of Chapter

This chapter is concerned with parallelising the preconditioned conjugate gradient method. The method has been described in section 2.3.4 from an algorithmic viewpoint. This included an explanation of how preconditioning works. The potential of the method with respect to parallelisation is addressed in section 7.2.

We then describe the construction of effective preconditioners suited to the parallel schemes used for the triangular solves at each iteration. This is followed by a description of our parallelisation strategies and the results of their implementations. These results are then discussed and compared with theoretical values.

7.2 Scope For Parallelism

The PCG algorithm has been discussed in chapter 2. The algorithm with ILU preconditioning (see section 2.3.4) is as follows:

```
Algorithm: Preconditioned CG to solve $Ax = b$
1. Preprocess: Compute preconditioner
        $M = LU$ where $A = LU + E$
2. Start: $r_0 \leftarrow b - Ax_0$,
        $p_0 \leftarrow z_0 \leftarrow M^{-1}r_0$
3. Iterate: Until convergence do
        (a) $w \leftarrow Ap_i$
        (b) $\alpha_i \leftarrow \frac{(r_i,z_i)}{(w,p_i)}$
        (c) $x_{i+1} \leftarrow x_i + \alpha_i p_i$
        (d) $r_{i+1} \leftarrow r_i - \alpha_i w$
        (e) $z_{i+1} \leftarrow M^{-1}r_{i+1}$
        (f) $\beta_i \leftarrow \frac{(r_{i+1},z_{i+1})}{(r_i,z_i)}$
        (g) $p_{i+1} \leftarrow z_{i+1} + \beta_i p_i$
```

The main operations in the above algorithm are as follows:

- Setting up the preconditioner (1)
- Matrix vector multiplications (3-a)
- Vector Updates (3-c, 3-d and 3-g)
- Dot products (3-b and 3-f)
- Preconditioning operations (2 and 3-e).

In the above list the potential bottlenecks are in setting up the preconditioner and in the solution of the linear systems with M ie. operations 2 and 3-e. The other steps are quite straightforward to parallelise efficiently (see section 7.4.2).

For polynomial preconditioning, the only operations involving the matrix are products with vectors. We also need fewer dot products than with the non-preconditioned method. The dot products can be bottlenecks for large numbers of processors [68]. Thus polynomial preconditioning is efficient for use on machines with very large numbers of processors and when the dot products dominate the cost of a CG step.

We have chosen ILU preconditioning since it is more suited to the parallel architecture we use ie. one with a small number of processors (see [68] and 3.2.2) and it is easy to form and use. Also, we have found this to be a very efficient preconditioner since it can be formed with low cost (see section 7.4.1) and provides satisfactory reductions in the number of iterations. The formation and performance of an ILU(0) preconditioner is discussed in section 7.3.1. Furthermore, we chose to use ILU and not incomplete Cholesky (IC) preconditioners despite the symmetry of our test matrices since the former allow us to draw conclusions covering a wider range of problems.

Due to the low cost of obtaining the ILU(0) preconditioner we have concentrated on the parallelisation of the other PCG steps. The parallelisation of the PCG method at iteration loop level involves the parallel processing of two or more iterations. This requires synchronisations which ensure that the sequential nature of the iterative process is preserved. It must therefore be ensured that only values which are completely computed in a particular iteration are used in subsequent iterations. The synchronisations required to achieve this would be quite complex and the scheme is reported to suffer from numerical instability [68]. We have consequently investigated parallelisation within the main iteration loop.

As mentioned above, operation 3-e is a potential bottleneck. In the case of ILU preconditioning this involves a pair of triangular solves at each iteration. Such operations are not straightforward to parallelise and account for about 40 percent of the total sequential solution time (see section 7.4.1).

The parallelisation of the triangular solves requires an analysis of L and U in order to identify the possibilities for parallel processing based on dependency information. We consequently need to design parallel schemes based on such information or form L and U such that they would suit our schemes. This includes the reordering of these factors. Two schemes are described in the next section which implement parallel row-oriented (section 7.3.2) and parallel column-oriented (see section 7.3.3) forward and backward substitutions.

We have chosen red-black ordering as an alternative since it gives rise to stiffness matrices which consist of diagonally structured entries (see figure 7.1, page 101 and [18]) which are similar to the type of sparsity patterns required for the independent columns scheme (see section 7.3.3) and are hence likely to require the dropping of fewer entries. Also, the results in [10] and [11] relate to red-black ordering and can be used for comparison with the results presented in this work.

7.3 Parallel Preconditioning

In this section we shall discuss the formation and the effectiveness of ILU preconditioners suited to parallel processing. The effects of ordering and level of fill-in are analysed and we describe two schemes for parallel triangular solution.

7.3.1 Formation And Effectiveness Of The ILU(0) Preconditioner

The preconditioner used in our model is obtained by an incomplete factorization of the stiffness matrix based on Doolittle's algorithm where the diagonal entries of L are set to be 1. The entry $a(i,j)$ in the stiffness matrix can be written as

$$a_{ij} = \sum_{p=1}^{min(i,j)} l_{ip}u_{pj}, \quad i,j = 1,2,\ldots,n$$

and the entries in L and U are given by

$$l_{ij} = (a_{ij} - \textstyle\sum_{p=1}^{j-1} l_{ip}u_{pj})/u_{jj}, \quad i > j$$

$$u_{ij} = a_{ij} - \textstyle\sum_{p=1}^{j-1} l_{ip}u_{pj}, \quad i \leq j.$$

We can obtain a factorization with no fill-in (ILU(0)) by only computing entries in L and U where there is a corresponding entry in the stiffness matrix ie. if $a_{ij} = 0$ then $l_{ij} = 0$ and $u_{ij} = 0$. This can be conveniently implemented by going through the linked list of entries in rows and columns of the stiffness matrix and factoring these entries. The factorization proceeds by alternating between the formation of a row of U and a column of L.

Figure 7.2 shows the structure of the stiffness matrix for a 16 node row-ordered mesh and its corresponding L and U factors. We have found the cost of obtaining such a factorization to be only a small percentage of the overall solution time (see section 7.4.1). The reduction in the number of iterations due to the ILU(0) preconditioner is quite significant as illustrated in tables 7.1 and 7.2. *niter* is the number of iterations. *precond* is the time for the computation of the preconditioner. *total* is the overall solution time. All times are in seconds.

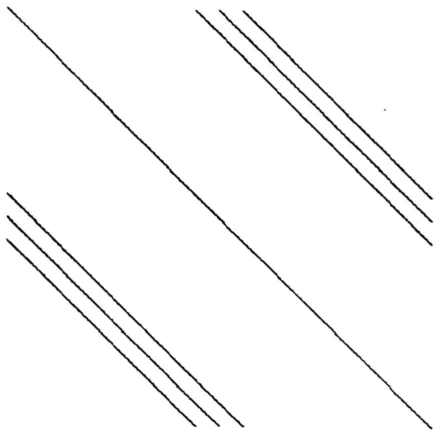


Figure 7.1: The sparsity pattern of a typical red-black matrix

```
**  **
*** ***
 *** ***
  **  **
**  **  **
*** *** ***  A
 *** *** ***
  **  **  **
    **  **  **
    *** *** ***
     *** *** ***
      **  **  **
        **  **
        *** ***
         *** ***
          **  **
```

```
*
**
 **
  **
**  *
*** **  L
 *** **
  **  **
    **  *
    *** **
     *** **
      **  **
        **  *
        *** **
         *** **
          **  **
```

```
**  **
 ** ***
  ** ***
   * **
    **  **
     ** ***  U
      ** ***
       *  **
        **  **
         ** ***
          ** ***
           *  **
            **
             **
              **
               *
```

Figure 7.2: The stiffness matrix and its factors for the row-ordered mesh

Table 7.1: Performance of the ILU(0) Preconditioner (row ordering)

Size	niter		time per iter		precond	total	
(nodes)	CG	ILU(0)	CG	ILU(0)		CG	ILU(0)
441	23	14	0.21	0.27	0.56	4.91	4.28
2601	63	31	0.34	0.67	1.04	21.69	20.79
10201	130	60	1.51	2.73	4.10	196.53	163.93
40401	265	118	8.14	11.87	32.36	2157.17	1400.15

Table 7.2: Performance of the ILU(0) Preconditioner (red-black ordering)

Size	niter		time per iter		precond	total	
(nodes)	CG	ILU(0)	CG	ILU(0)		CG	ILU(0)
441	19	12	0.20	0.23	0.41	3.75	3.12
2601	57	28	0.31	0.59	0.98	17.67	17.50
10201	112	55	1.29	2.54	3.86	144.48	143.56
40401	231	105	7.79	10.11	28.67	1799.49	1090.22

We can see that the preconditioner becomes more effective as the mesh size (number of nodes) increases. Since the ILU(0) preconditioner can be obtained at low cost and is sufficiently efficient, we have used this factorization in the parallel schemes for triangular solves described later in this chapter.

Diagonal scaling involves the division of the coefficients in every row of a system of equations (including the RHS) by the corresponding diagonal entry. The aim is to preserve the numerical accuracy during operations such as factorization. We studied the effect of diagonal scaling on the performance of the CG method in order to compare this with the performance of our preconditioning schemes.

Our experiences with diagonal scaling show that we do not benefit from the scaling of the equations with the diagonal of the stiffness matrix or its upper triangular factor (U). The number of iterations is unaffected by the former and actually increases with the latter. Also, if the forward and backward substitutions are carried out with the mentioned diagonals as the preconditioner, the rate of convergence becomes slower. This is different to diagonal scaling in the sense that a preconditioning operation (division by the diagonals) is performed at each iteration.

7.3.2 Level Scheduling

This scheme has been described in [85] and can be thought of as a reordering scheme whose objective is to obtain a block triangular system such that the unknowns in each block can be computed in parallel. Consider the lower triangular matrix in figure 7.3 which shows the block partitioning of $Lx = b$. If the L_i's are diagonal matrices, we can compute all the entries in each

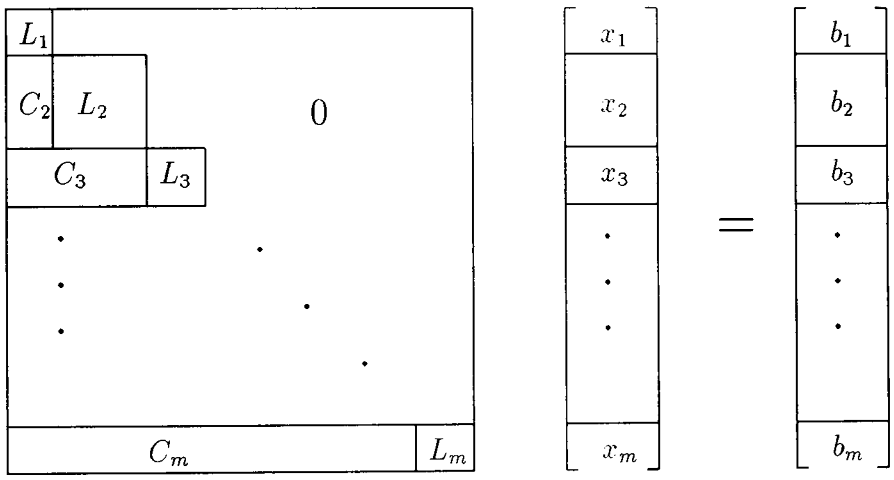


Figure 7.3: Block partitioning for Lx = b using level scheduling

subvector x_i concurrently because a diagonal L_i ensures the *independence* of the unknowns corresponding to the entries of L_i. We only need to synchronise at block level so that at step i of the forward substitution all x_{i-1}'s are already determined. Figure 7.4 shows the structure of an example matrix reordered by levels.

We can see that the basic requirement of this technique is the identification of independent unknowns and grouping their corresponding rows into blocks which must be processed in sequence. This can be done by associating

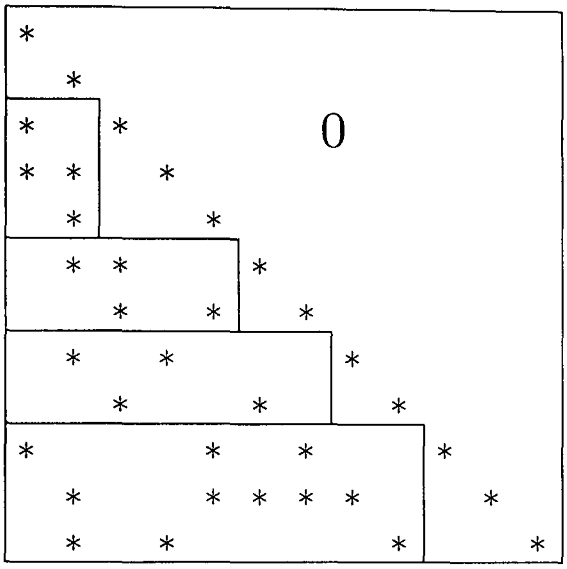


Figure 7.4: Matrix reordered by levels

with each row of L a *depth* computed as

$$depth(i) = \begin{cases} 1 & \text{if } l_{ij} = 0 \ \forall \ j < i \\ 1 + max_{j<i}\{depth(j) : l_{ij} \neq 0\} & \text{otherwise} \end{cases}$$

A *level* of L can be defined as the set of nodes with the same depth. The rows of L with only a diagonal entry will be at level 1. The next level consists of rows dependent only on subvector x_1 and so on. This scheme is made attractive by the low cost of the preprocessing to determine the levels.

The algorithm can be implemented without physically reordering the matrix by solving the row equations in increasing order of the depth of their nodes, distributing the nodes at each level across the processors. In order to explain this further let us introduce the following:

NLEV - The number of levels in L

IORDER - An integer array consisting of the ordering of the rows of L by increasing depth

ILEVEL - An integer array consisting of the index to the start of each level in IORDER.

The two arrays can be set up quite easily and with low cost once the depths have been determined. The forward elimination can be implemented as follows:

```
DO $k$ = 1,NLEV
    DOALL ($j$ = ILEVEL(k):ILEVEL(k+1)-1)
        $i = IORDER(j)$
        $x_i = \frac{1}{l_{ii}} \left(b_i - \sum_{\{j<i:l_{ij}\neq 0\}} l_{ij}x_j\right)$
    END DOALL
END DO
```

The allocation of rows within a block to processors is such that the processors start from the first rows in the block and process the next free rows as they complete their current ones. The level scheduled back substitution can be implemented in the same way. A detailed analysis of the number and length of levels for different row-ordered mesh sizes is given in section 7.4.3. Section 7.4.4 contains the results of our implementations of this technique. Section 7.5 contains a detailed theoretical analysis of parallelism for level scheduling and compares theoretical and actual speed ups.

7.3.3 Independent Columns

The level scheduling scheme described in the previous section involves the identification and subsequent parallel processing of independent rows of L. The only requirements for this type of parallel scheme are the diagonal blocks (L_i's) at each level. In this section we shall describe an alternative scheme involving the parallel processing of columns rather than rows.

In the *independent columns* scheme we compute part of the x vector at each stage and update the subvector of b corresponding to these unknowns. The idea is to update the entries in b concurrently using distinct columns of L. This requires that at each step the group of columns used for this update have only one entry per row so that no two columns would contribute to the same entry in b (ie. to avoid any contention in updating b). The structure of L should therefore be as shown in figure 7.5 where the C_i's are sparse rectangular blocks with only one entry per row in each block of columns. The algorithm can be written as follows:

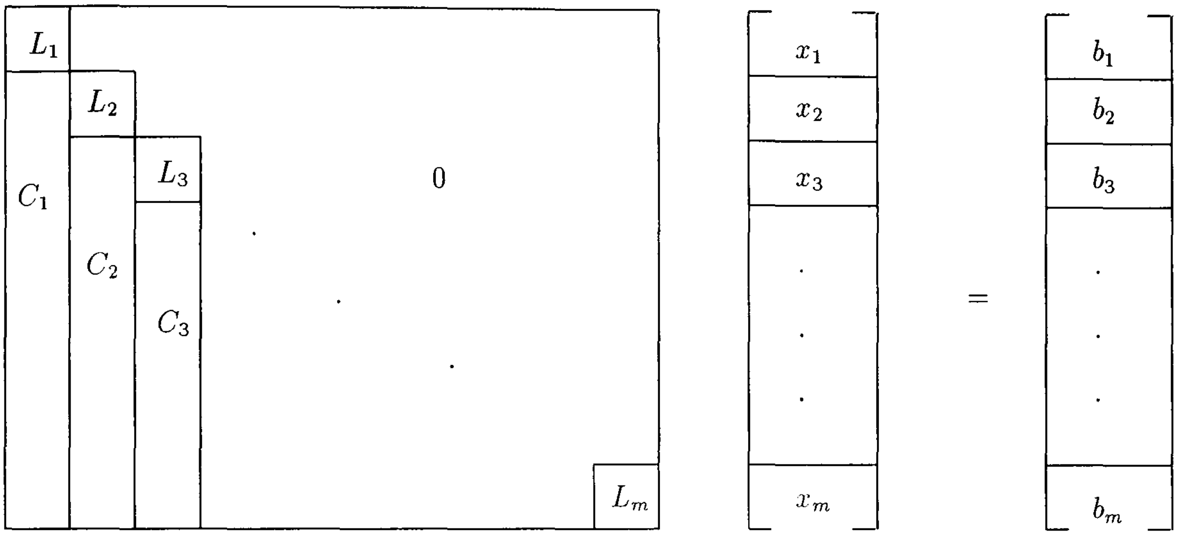


Figure 7.5: Matrix with blocks of independent columns

```
DO $k = 1, m$
    $x_k = L_k^{-1} b_k$
    Update $b_{k+1}, \ldots, b_m$ using $x_k$ and $C_k$   $(if\ \ k < m)$
END DO
```

The blocks do not have to be of equal size. The first step of the above algorithm involves a triangular solve unless the L_i's are diagonal in which case simple division yields the solution subvector. At each step one block of unknowns is computed. The update operation does not need to be performed at the final step. The number of steps depends on the structure of L ie. the number of blocks of independent columns. We may wish to design L such that at each step a number of columns equal to the number of available processors are used for updating the RHS. In that case $m = \lceil N/N_p \rceil$ where N is the order of L and N_p is the number of processors.

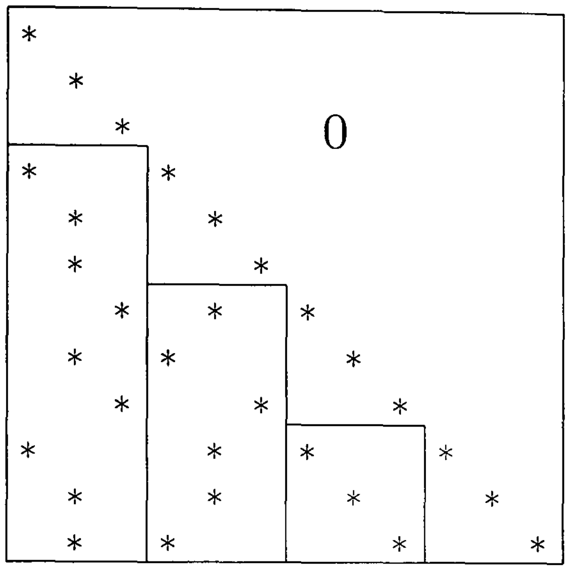


Figure 7.6: An example matrix with independent columns

Let us now expand on the parallel aspects of the above algorithm. If the L_i's are diagonal then the components of x_i can be computed independently at each step. The second stage of each step is an update operation which involves using each column in C_k with its corresponding entry in x_k. We update all entries in b for which there is a corresponding entry in that column. This means that when processing column j, if we come across an entry in row i (l_{ij}) we subtract $l_{ij}x_j$ from entry b_i. L is stored by columns in order to provide more efficient access to the entries in each column. The update at step k can consequently be written as:

```
DO for all columns in $C_k$ ⟨ in parallel ⟩
    find each entry $l_{ij}$
    $b_i = b_i - (l_{ij}x_j)$
END DO
```

This parallel loop can only be correct if there is no inadvertent access to the same entry b_i by two or more processors. The only way to ensure this without locking the b_i's is to allow only one entry per row in C_k. The results of our implementations of this method are given in section 7.4.4.

Preconditioner Design

Let us now discuss the design of suitable preconditioners for the independent columns method. One possible approach is the formation of L and U factors in the usual way (see section 7.3.1) and to drop any entries which will disturb the independence properties required by the method during the factorization. The resulting matrices are *incomplete* ILU(0) factors of the stiffness matrix.

For the purposes of our test implementations we have performed the dropping of the appropriate entries in the factorization as the rows and columns of the factors are being formed. It is also possible to perform the factorization after the appropriate elements of A are dropped or forming an incomplete factorization first and then dropping the entries. The effect of the dropping scheme on the rate of convergence is insignificant [10].

In our implementations we have designed L and U such that all the C blocks (see figure 7.5) are of the same size and equal to N_p ie. one column is assigned to each processor. Also, all L_i's (and U_i's) are diagonal blocks allowing parallel computation of the unknowns in each x_i. An example of the structure of such test matrices with $N_p = 3$ is given in figure 7.6.

The dropping of entries in L and U must involve some form of check on the state of the matrices as the factorization proceeds. More accurately, before allowing any entry into L or U we must make sure that no other entry exists in the same row as this entry in the block of columns it appears in (see figure 7.5). In this way the independence of the unknowns corresponding to the columns in each block is ensured. The checking method can be implemented by keeping a count of the entries in each row of the block.

A further measure can be taken in order to try to improve the quality of the preconditioner. One could aim to preserve the largest entries in each row of each block. This can be done by using a drop tolerance technique involving the dropping of entries which are small compared to their corresponding diagonal entries [57].

Another possibility is to keep a record of all candidates for insertion in the section of each row and choose the largest as the successful entry. We have implemented the latter scheme but little or no improvement has been obtained in the quality of the preconditioners (see table 7.3). The improve-

Table 7.3: The effect of preserving largest entries on the rate of convergence

	number of iterations					
	row			red-black		
Mesh size	2601	10201	40401	2601	10201	40401
before	31	60	118	28	55	105
after	31	59	116	27	53	101

ment is marginally better in the case of red-black ordering. Greater reductions in the number of iterations are necessary to make the implementation cost-effective.

The reason for the lack of larger improvements in the quality of the preconditioners may be that the structure of our row and red-black ordered test matrices is such that they do not benefit from this measure to a significant extent. Other orderings (eg. minimum degree) could show more pronounced improvements in the quality of the resulting preconditioners by preserving the largest entries. A detailed discussion of the design of preconditioners for the independent columns scheme is given in section 7.5.2 and [10].

7.4 Parallelising The Main Iteration Loop

In this section we shall provide information on the execution profile of the solution phase of our model. This will show how the costs of the preconditioning and other steps of the PCG implementation compare. We then go on to explain how these steps are parallelised and present the results of testing our parallel implementations. These results are analysed in section 7.5.

7.4.1 Profile Of The Method

A typical profile of the PCG method implemented in our model is given below as approximate percentages of sequential solution time for each step (see section 7.2). The profile varies very little for different problem sizes and node orderings.

Table 7.4: Relative Cost of the ILU(0) Preconditioner

Problem Size	Relative Cost of ILU(0)	
(nodes)	row	red-black
2601	5.0%	5.6%
10201	2.5%	2.7%
40401	2.3%	2.6%

- step 3-a - 35% : matrix vector multiplication
- step 3-b - 5% : dot product
- step 3-c - 5% : vector update
- step 3-d - 5% : vector update
- step 3-e - 40% : preconditioning operation
- step 3-f - 5% : dot product
- step 3-g - 5% : vector update

We can see that the matrix vector multiplication (step (a)) and the triangular solves (step (e)) dominate the solution time.

The cost of obtaining the ILU(0) preconditioner is given as approximate percentages of the total sequential solution time for different problem sizes using row and red-black orderings in table 7.4. Also, for all three problem sizes, the cost of obtaining the preconditioner is about four times the time spent in one sequential iteration. These results indicate that the cost of obtaining an ILU(0) preconditioner is very low and becomes insignificant compared to the cost of the other steps for larger meshes. This is why we have aimed to parallelise these steps rather than the formation of the preconditioner.

7.4.2 Steps Other Than The Preconditioning Operations

We shall consider the three different types of operations in this category separately (see section 7.2). For each step, further efficiency can be provided by eliminating the cost associated with the DOALL mechanism (see section 4.3.2). Each task can perform a little preprocessing to determine a unique block of the loop indices it will execute. This is a block of consecutive indices and is executed sequentially by each task. We therefore process several parallel blocks of the loop index. The loop indices are divided in such a way that the blocks are of equal or nearly equal sizes in order to obtain good load balancing.

Let us now consider the parallelisation of the steps.

Matrix vector multiplications (3-a) These can be parallelised by computing distinct entries in the product vector independently. Each processor multiplies a different row of A by p at any one time and there are consequently no synchronisation considerations. The DO loop covering the rows of A can be replaced by the blocking strategy described above. This means that N_p parallel blocks of rows of A are processed.

Vector updates (3-c, 3-d and 3-g) The entries in the resulting vector can be computed independently at these steps. We consequently only need to break up the loop covering these entries into blocks and process these concurrently. Steps 3-c and 3-d are independent and are consequently merged to increase granularity.

Dot products (3-b and 3-f) These operations involve the accumulation of a single variable and must be parallelised bearing this in mind. This means that even though the loop index can be spread among the processors, we need to update the global sum under mutual exclusion. We consequently use a combination of the blocking strategy and the local sums strategy (see section 4.5.2) to implement the dot products in parallel.

Each task accumulates its local sum according to its block of indices and adds this on to the global sum under mutual exclusion once all the block indices have been executed. In this way, the overhead due to DOALL

Table 7.5: Average Level Lengths for Level Scheduling - Row ordering

Mesh Size	Number of Levels	Average Length of Levels
121	26	3
441	56	6
2601	146	16
10201	296	33
40401	596	66

is removed and we only need to pay the cost of the synchronisation primitives implementing the mutually exclusive update of the global sum (ie. the CRITICAL SECTION's).

7.4.3 The Triangular Solves

This step (3-e) is the main bottleneck in the PCG method. Matrix vector multiplication (3-a) is also costly but can be parallelised quite efficiently. We have implemented parallel triangular solve schemes based on level scheduling (see section 7.3.2) and the independent columns method (see section 7.3.3) for row and red-black orderings of the nodes. These implementations are discussed separately below.

Level Scheduling

The same leveling strategy is used for the forward and backward substitutions. The number of levels and the average number of rows per level (level length) for different mesh sizes are given below for row ordering. There are many rows at level 1, all of which correspond to rows with a single entry. These have been excluded in the calculation of the average level lengths since we obtain a fairer picture of the distributions in this way. Due to the symmetry of the stiffness matrix, L and U always have the same number of levels and level lengths.

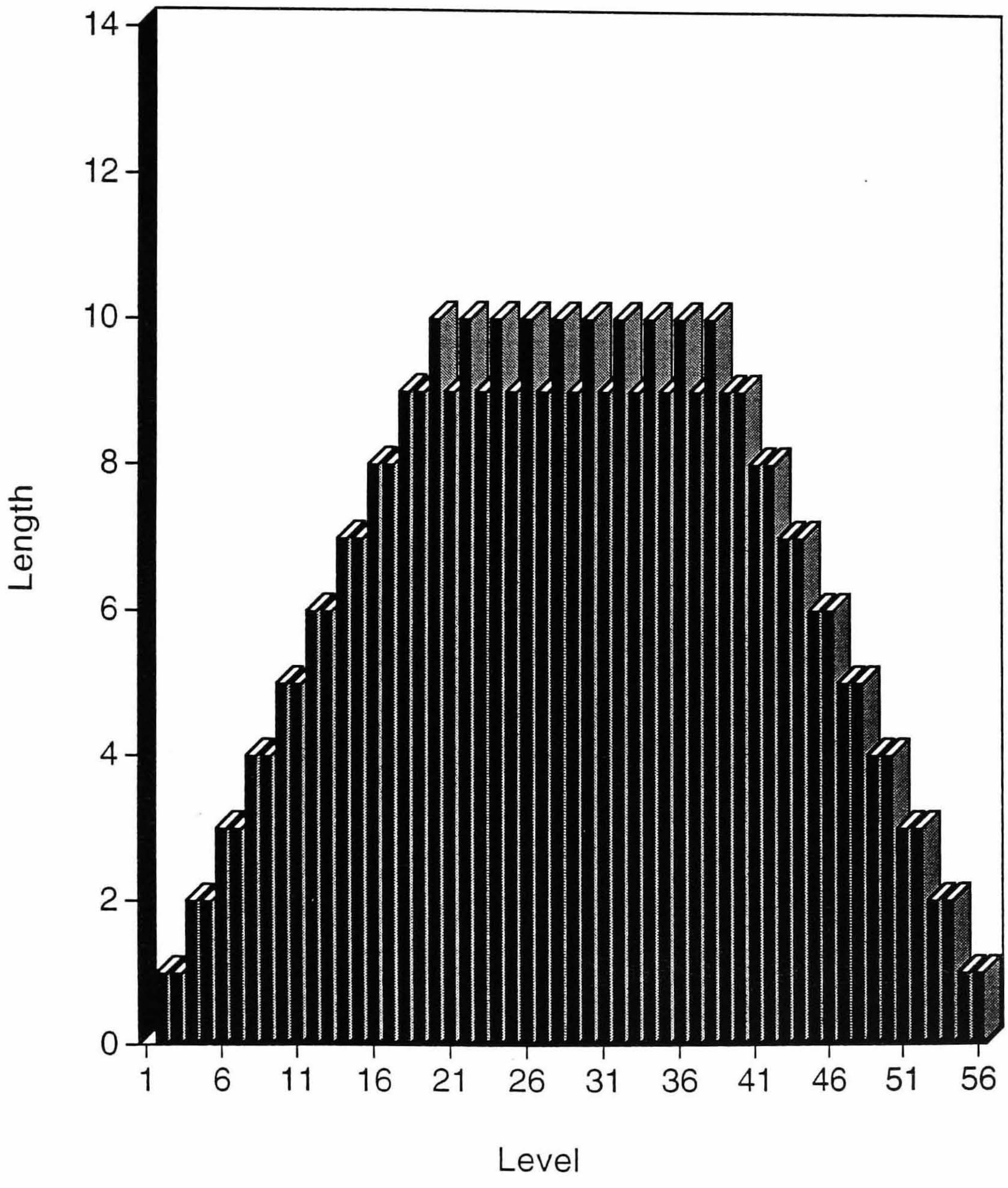
We can see from table 7.5 that the number of levels does not increase in proportion with the mesh size and this is why we can expect to benefit substantially from level scheduling for large meshes. We would obviously like the number of levels to be minimal so that we need to synchronise at fewer points. The lengths of the levels for larger meshes are more likely to provide the large granularity required for efficient parallel processing.

The distribution of level lengths for two mesh sizes (row ordering) are presented in graphs 7.1 to 7.4. Graphs 7.1 and 7.3 show the lengths for all levels excluding level 1. Graphs 7.2 and 7.4 show what percentage of all levels (excluding level 1) belong to a particular level length. These graphs are discussed in detail in section 7.5.2.

Graph 7.1

Level Length Distribution

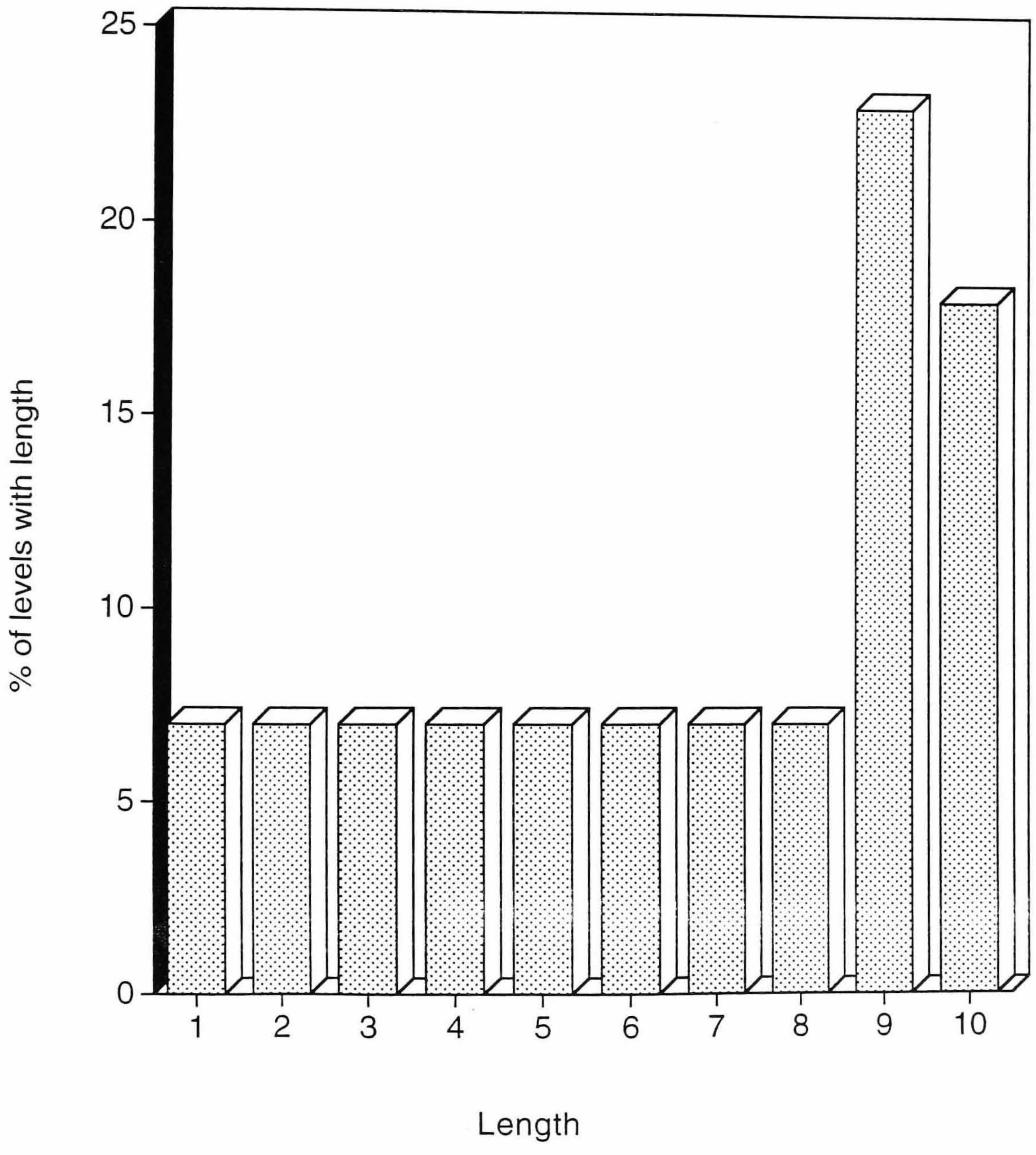
441 nodes



Graph 7.2

Percentage of levels with the same length

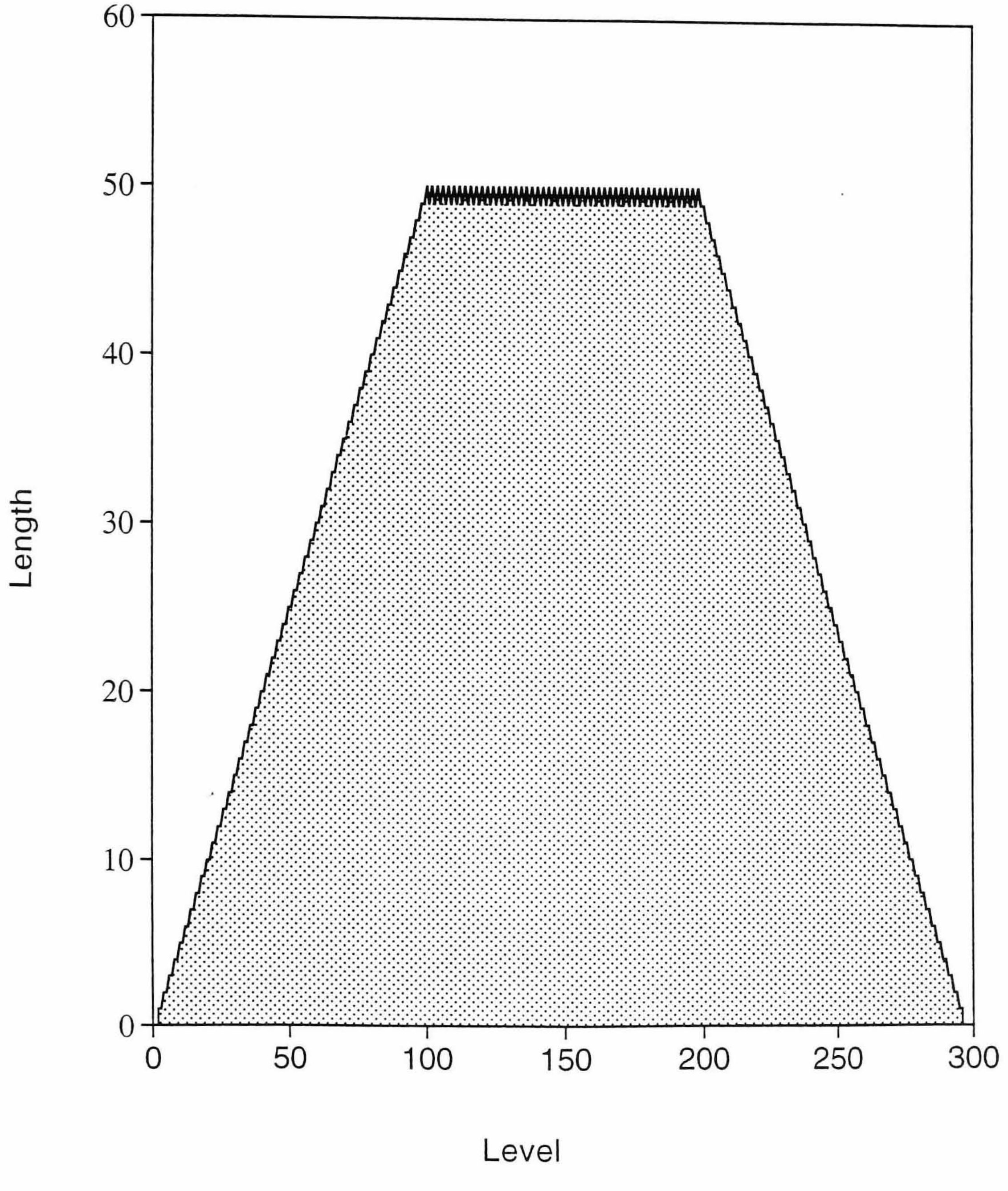
441 nodes



Graph 7.3

Level Length Distribution

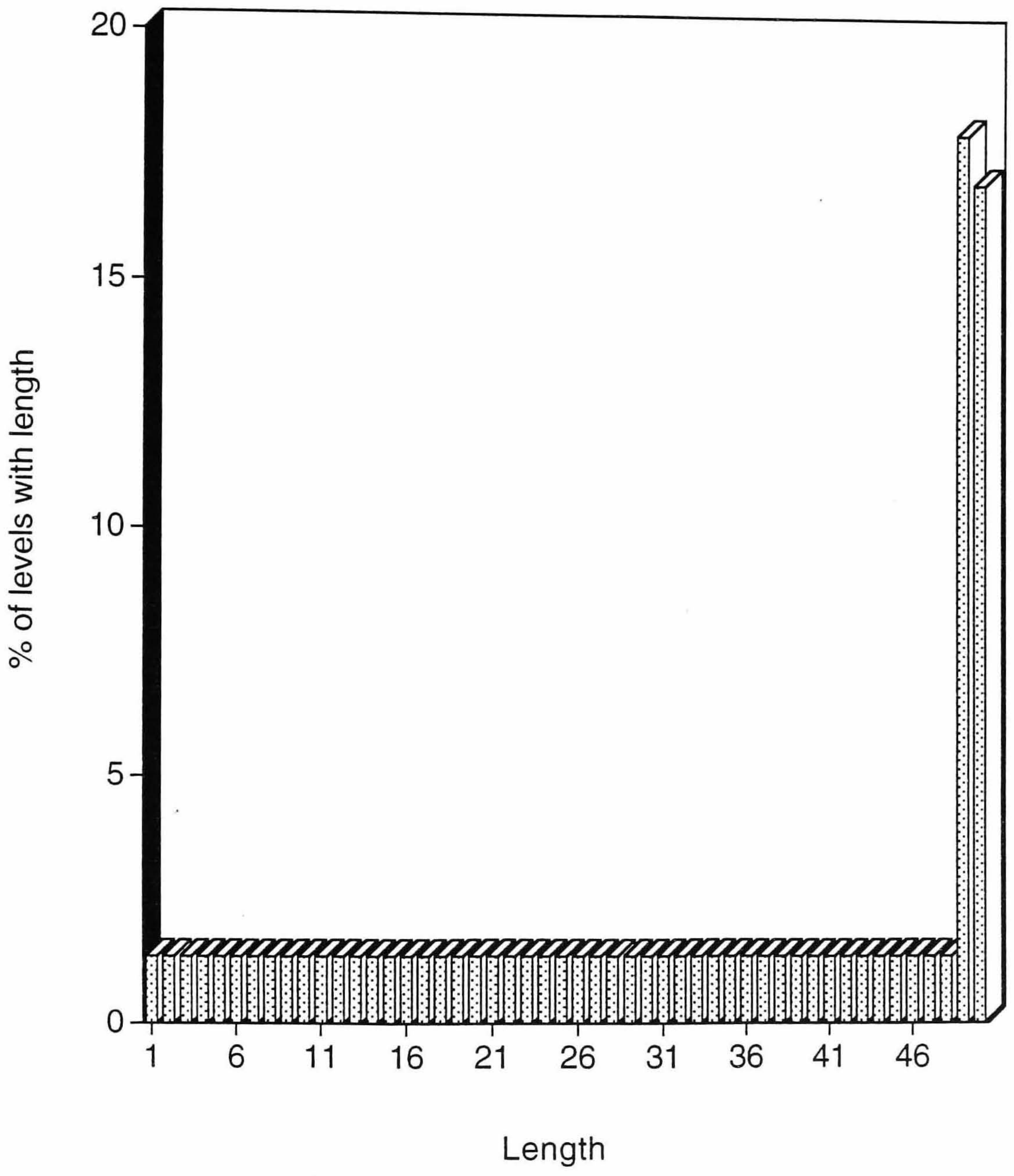
10201 nodes



Graph 7.4

Percentage of levels with the same length

10201 nodes



Independent Columns

The performance of preconditioners suitable for the independent columns method for row-ordered and red-black ordered meshes is given in tables 7.6 and 7.7 respectively. The CG column corresponds to the unpreconditioned method. The results indicate that for both orderings, as the number of entries dropped during the factorization increases, the rate of convergence becomes slower. This means that as we increase the number of processors (ie. the block size) to which the factors must be suited, the effectiveness of the resulting preconditioner falls.

We have increased the block size up to values which make the preconditioners cause the same rates of convergence as the unpreconditioned method ie. CG (see [10] and [11]). The largest block sizes are, of course, not realistic numbers of processors for shared memory architectures. They are only used to determine the pattern of convergence rates as the block size is increased.

At one extreme we have a *full* ILU(0) preconditioner for which all of the factors of the original matrix are included. At the other extreme we have diagonal factors for which all entries apart from those on the diagonal are dropped. The best performance in terms of the reduction in the number of iterations is due to the former and the latter performs identically with the unpreconditioned implementations (see table 7.1 and 7.2).

In between the two extreme cases mentioned above we have a range of performance figures. We can see from the results that the degradation in the performance is particularly large when going from one to two processors. This is because the *full* ILU(0) structure which is identical to that of the stiffness matrix is disturbed. After two processors, the increase in the number of iterations is more steady. The actual rate of this increase falls as we go on to larger block sizes. This is because the increase in block size at these stages involves the dropping of fewer entries than before. The resulting factors consequently differ only slightly and show smaller differences in their performance.

Also, a study of the effect of problem size on the performance of the preconditioners shows that the rate of change of the number of iterations is very similar for different problem sizes. Similar results are obtained in [10] and [11] where the above issues are also discussed.

7.4.4 Parallel Implementation : Experimental Results

In this section we have presented the results of tests for our parallel implementations. Tables 7.6 and 7.7 relate to the performance of independent columns preconditioners as mentioned in the previous section. In the other tables presented in this section, letters A-G refer to PCG steps (see section 7.2), *row* and *rb* refer to row and red-black node orderings respectively and the other headings are defined as in section 7.3.1. All times are in seconds.

Tables 7.8 to 7.11 present the performance of level scheduling for different mesh sizes. The performance of the independent columns method is presented in tables 7.12 to 7.15. For the level scheduling tables, the number of iterations are fixed because the preconditioner does not vary with N_p whereas for the independent columns tables each block size is associated with a different preconditioner. For both methods we have shown the variation of the time taken for one iteration, the total solution time and the speed up with increasing N_p.

The speed ups are computed as the ratio between the sequential solution times with an ILU(0) preconditioner (see tables 7.1 and 7.2) and the parallel solution time *including* the time spent forming the appropriate preconditioner. The latter varies with block size for the independent columns method and is hence specified in the relevant tables together with other information relating to a particular block size. Tables 7.11 and 7.15 show how the two parallelisation schemes speed up the preconditioning step. A breakdown of the speed ups for the non-preconditioning steps (row ordering) is also presented (table 7.16) to show how efficiently they are parallelised.

The actual sequential time taken to solve the equations for a 40401 node row ordered mesh is about 1400 seconds. This is over twelve times more than the time taken to assemble the stiffness matrix (see section 6.6) showing that, as expected, the solution phase dominates the processing time for finite element analysis. The cost of obtaining the level scheduling information adds about 30 percent on to the cost of obtaining the preconditioner (see section 7.4.1). The cost of obtaining suitable preconditioners for the independent columns method is only slightly larger than the cost of computing the standard ILU(0) factors.

As mentioned above, all preprocessing costs have been taken into account in the computation of the speed ups for both methods. For the independent columns method this consists of the cost of computing the preconditioner.

Table 7.6: Performance of The Independent Columns Preconditioner - Row ordering

Size	Number of Iterations												
	CG	*full*	Block size (N_p)										
(nodes)		ILU(0)	2	4	6	8	10	12	24	48	96	192	384
2601	63	31	42	48	53	57	59	60	61	62	63	63	63
10201	130	60	82	91	102	110	117	122	127	127	128	130	130
40401	265	118	152	164	189	201	219	234	248	257	262	263	265

Table 7.7: Performance of the Independent Columns Preconditioner - Red-black ordering

Size	Number of Iterations													
	CG	*full*	Block size (N_p)											
(nodes)		ILU(0)	2	4	6	8	10	12	24	48	96	192	384	768
2601	57	28	34	38	41	44	47	49	51	54	57	57	57	57
10201	112	55	68	77	85	91	94	96	103	106	109	111	112	112
40401	231	105	121	132	140	146	150	152	174	191	201	212	229	231

For level scheduling, the preprocessing costs consist of the cost of computing the preconditioner plus the cost of obtaining the level scheduling information. The results are discussed in section 7.5.2.

Table 7.8: Performance of Level Scheduling - 2601 nodes - niter=31(row),28(rb),precond=1.34s(row),1.30s(rb)

	time per iter		total		Sp	
N_p	row	rb	row	rb	row	rb
1	0.66	0.58	21.88	17.68	0.95	0.99
2	0.34	0.30	11.88	9.72	1.75	1.85
4	0.18	0.15	6.92	5.45	3.00	3.21
6	0.13	0.09	5.36	3.87	3.88	4.52
8	0.11	0.08	4.90	3.41	4.24	5.13
10	0.13	0.07	5.24	3.23	3.97	5.42
12	0.13	0.06	5.39	3.10	3.86	5.65

Table 7.9: Performance of Level Scheduling - 10201 nodes - niter=60(row),55(rb),precond=5.61s(row),5.29s(rb)

	time per iter		total		Sp	
N_p	row	rb	row	rb	row	rb
1	2.81	2.54	174.21	145.01	0.94	0.99
2	1.46	1.30	93.14	76.79	1.76	1.86
4	0.75	0.68	50.61	42.73	3.24	3.36
6	0.52	0.46	36.67	30.54	4.47	4.70
8	0.44	0.35	31.89	24.37	5.14	5.89
10	0.51	0.30	36.03	21.82	4.55	6.58
12	0.53	0.27	37.34	20.28	4.39	7.08

Table 7.10: Performance of Level Scheduling - 40401 nodes - niter=118(row),105(rb),precond=40.12s(row),37.25s(rb)

	time per iter		total		Sp	
N_p	row	rb	row	rb	row	rb
1	12.56	10.24	1522.20	1112.47	0.92	0.98
2	6.31	5.13	784.70	576.84	1.78	1.89
4	3.17	2.68	414.18	318.78	3.38	3.42
6	2.25	1.75	305.04	221.14	4.59	4.93
8	1.80	1.33	252.73	176.41	5.54	6.18
11	1.97	1.16	272.40	158.69	5.14	6.87
12	2.12	1.07	290.49	149.55	4.82	7.29

Table 7.11: Performance of Level Scheduling - Step E only

	Speed Up					
	2601 nodes		10201 nodes		40401 nodes	
N_p	row	rb	row	rb	row	rb
1	0.96	0.99	0.96	0.99	0.95	0.98
2	1.63	1.71	1.65	1.73	1.75	1.79
4	2.78	2.95	3.20	3.28	3.27	3.33
6	2.85	3.61	3.80	3.89	3.86	4.19
8	2.98	4.32	4.05	4.69	4.59	5.22
10	2.62	4.93	3.16	5.12	3.80	5.82
12	2.41	5.01	3.01	5.81	3.34	6.10

Table 7.12: Performance of Independent Columns - 2601 nodes

Block	niter		time per iter		precond		total		Sp	
size (N_p)	row	rb	row	rb	row	rb	row	rb	row	rb
1	31	28	0.66	0.61	1.20	1.09	21.66	18.04	0.96	0.97
2	42	34	0.28	0.28	1.22	1.10	13.08	10.80	1.59	1.62
4	48	38	0.19	0.13	1.24	1.12	10.34	6.23	2.01	2.81
6	53	41	0.13	0.08	1.27	1.14	8.12	4.46	2.56	3.92
8	57	44	0.10	0.06	1.30	1.17	6.86	3.83	3.03	4.57
10	59	47	0.10	0.05	1.34	1.20	7.05	3.61	2.95	4.85
12	60	49	0.10	0.05	1.41	1.23	7.37	3.48	2.82	5.03

Table 7.13: Performance of Independent Columns - 10201 nodes

Block	niter		time per iter		precond		total		Sp	
size (N_p)	row	rb	row	rb	row	rb	row	rb	row	rb
1	60	55	2.86	2.70	4.51	4.23	176.27	152.72	0.93	0.94
2	82	68	1.10	1.08	4.62	4.39	94.76	77.60	1.73	1.85
4	91	77	0.62	0.54	4.79	4.51	60.94	46.01	2.69	3.12
6	102	85	0.37	0.36	4.86	4.63	42.91	37.58	3.82	4.36
8	110	91	0.30	0.23	4.91	4.78	37.69	25.54	4.35	5.62
10	117	94	0.15	0.13	5.02	4.90	39.89	22.86	4.11	6.28
12	122	96	0.13	0.11	5.09	5.01	41.71	20.99	3.93	6.84

Table 7.14: Performance of Independent Columns - 40401 nodes

Block	niter		time per iter		precond		total		Sp	
size (N_p)	row	rb	row	rb	row	rb	row	rb	row	rb
1	118	105	13.04	11.24	34.11	30.68	1573.20	1211.36	0.89	0.90
2	152	121	4.91	4.55	35.36	32.02	782.21	583.01	1.79	1.87
4	164	132	2.62	2.18	36.12	32.96	465.17	320.65	3.01	3.40
6	189	140	1.70	1.35	37.01	33.89	358.09	222.95	3.91	4.89
8	201	146	1.25	0.98	37.92	34.78	289.89	178.14	4.83	6.12
10	219	150	1.22	0.80	38.81	35.81	306.38	156.19	4.57	6.98
12	234	152	1.23	0.77	39.95	36.79	329.45	153.37	4.25	7.11

Table 7.15: Performance of Independent Columns - Step E only

	Speed Up					
	2601 nodes		10201 nodes		40401 nodes	
N_p	row	rb	row	rb	row	rb
1	0.97	0.98	0.94	0.96	0.91	0.93
2	1.42	1.49	1.61	1.72	1.65	1.82
4	1.87	2.24	2.31	2.69	2.40	2.89
6	2.01	3.09	3.16	3.39	3.22	3.95
8	2.42	3.59	3.57	4.26	3.64	5.19
10	2.11	3.94	3.28	4.97	3.40	5.89
12	2.06	4.14	3.09	5.25	3.12	6.12

Table 7.16: Speed ups for all non-preconditioning PCG steps - 40401 nodes - row ordering

	PCG Step				
Np	A	B	C+D	F	G
2	1.94	1.92	1.85	1.84	1.87
4	3.62	3.59	3.71	3.67	3.59
6	4.94	4.82	4.79	4.77	4.85
8	6.01	5.92	6.21	6.34	6.15
10	8.19	7.81	7.75	7.81	8.02
12	10.01	9.53	9.89	9.14	9.96

7.5 Analysis Of Parallelism

In this section we shall construct a model for the estimation of the theoretical parallelism of the level scheduling algorithm. A detailed discussion of the experimental results for level scheduling and independent columns is then presented. This includes a comparison between actual and theoretical speed ups for level scheduling.

7.5.1 Theoretical Analysis

The theoretical model used in this section is similar to that in [2]. Our aim here is to analyse the average parallelism for the level scheduling algorithm. We shall make the assumption that the time taken to solve one row equation in the triangular system is constant. This is a reasonable assumption for the type of test matrices we have used since these have an almost constant number of non-zero entries per row (ie. A, L and U).

Let us now define a time step as the time required to solve one row equation. According to this definition, solving four equations on two processors and solving three equations on two processors both require two time steps. We have consequently provided some means of accounting for the efficiency lost when the work at each level is not evenly balanced among the processors.

We can now define the parallelism for a particular problem as

$$average\ parallelism = \frac{number\ of\ tasks\ completed}{number\ of\ steps\ required}$$

where the number of tasks completed is the number of rows and the number of steps is given by

$$number\ of\ steps = \sum_{I=1}^{NLEV} \left\lceil \frac{ILEVEL(I+1) - ILEVEL(I)}{N_p} \right\rceil .$$

Using the above definitions, we have calculated the average parallelism for the three mesh sizes in section 7.4.4 using both row and red-black orderings. These are presented in tables 7.17 and 7.18.

Table 7.17: Theoretical vs Actual Speed ups for Level Scheduling - Row ordering

	Speed Up					
	2601 nodes		10201 nodes		40401 nodes	
N_p	Theo.	Act.	Theo.	Act.	Theo.	Act.
2	1.95	1.63	1.97	1.65	1.99	1.75
4	3.70	2.78	3.80	3.20	3.93	3.27
6	5.27	2.85	5.49	3.80	5.78	3.86
8	6.70	2.98	7.07	4.05	7.57	4.59
10	7.79	2.62	9.16	3.16	9.55	3.80
12	9.91	2.41	10.12	3.01	10.52	3.34

Table 7.18: Theoretical vs Actual Speed ups for Level Scheduling - Red-black ordering

	Speed Up					
	2601 nodes		10201 nodes		40401 nodes	
N_p	Theo.	Act.	Theo.	Act.	Theo.	Act.
2	1.96	1.71	1.98	1.73	1.99	1.79
4	3.73	2.95	3.82	3.28	3.94	3.33
6	5.32	3.61	5.56	3.89	5.80	4.19
8	6.76	4.32	7.12	4.69	7.64	5.22
10	7.87	4.93	9.24	5.12	9.60	5.82
12	9.96	5.01	10.31	5.81	10.65	6.10

7.5.2 Discussion Of Results

As expected for both methods the overall solution process and the preconditioning step parallelise better for larger meshes due to increased granularity (see tables 7.8 to 7.11). The best overall speed up obtained is 7.29 on 12 processors using level scheduling. This indicates that we are paying fairly large synchronisation costs for large N_p which reduces the computation to communication ratio significantly even for the 40401 node mesh. This is very much influenced by the length and number of levels in the preconditioning step and the effect of these on the load balancing (see below).

The overall speed ups are greater than those for the preconditioning step which indicates that the other PCG steps are parallelised more efficiently than this step. The non-preconditioning steps parallelise similarly in terms of the speed ups obtained. The parallel efficiencies are very high and in some cases near ideal (see tables 7.11 and 7.15). As expected, the best efficiencies are yielded with smaller numbers of processors.

For level scheduling, even though the average level length is quite satisfactory for large meshes (see section 7.4.3), the degradation due to the short levels is not insignificant. Therefore we would expect better performance if there were very few short levels and many with length near or more than the average length. The short levels bring about inefficiency in two ways. Firstly, if there are many of these we need to synchronise with small granularity too often. Secondly, for large numbers of processors, there could be several idle tasks at each level since we only assign one row per task.

The theoretical average speed ups are very near ideal for both orderings (see tables 7.17 and 7.18). The actual results are comparable to theoretical values for small N_p but for large numbers of processors we observe much poorer performance than theoretically possible. The reasons given above account for this difference together with the overheads associated with setting up and managing parallel tasks. The difference between the two sets of speed ups is also significantly smaller for red-black ordering, especially for the larger meshes. This can be explained by the improved parallel performance of the red-black ordered preconditioners (see below).

Most of the actual level scheduling speed ups for the preconditioning step are in the range 2-5 (see table 7.11). The speed ups reported in a similar implementation on an Alliant FX/8 [85] are in the same range. Better efficiency can be obtained by processing meshes with long levels and many entries per level. The maximum theoretical speed up for any mesh size can not exceed $TOTNOD/NLEV$. This is independent of the number of processors used.

The speed ups for the independent columns scheme are influenced by the distribution of entries in each block of columns. We should expect to obtain maximum efficiency when the columns are almost equally full such that minimal time is spent by idle tasks waiting for other columns to be processed. The granularity for this scheme is determined by the number of entries in each column. We should consequently expect higher parallel efficiency for larger problems since the average number of entries per column is higher. This is confirmed by the results in tables 7.12 to 7.15. We could also benefit from better load balancing by designing the preconditioner such that the columns in the same block have equal or nearly equal numbers of entries. Better parallel efficiencies are consequently obtained for red-black ordering.

The speed ups for the independent columns method are also dependent on the entry distribution in each block. This is determined by the ordering scheme used. The actual speed ups are quite similar to those for level scheduling which indicates that the scheme is efficient for preconditioning operations. The best overall speed up obtained using independent columns is 7.11 on 12 processors which is quite close to that for level scheduling (7.29 on 12 processors).

We can see that both methods provide us with convenient and inexpensive means of parallelising the triangular solves associated with preconditioning operations. Good speed ups can be obtained if the preconditioning matrices have reasonable structures.

The performance of preconditioners for a row-ordered mesh suited to the independent columns scheme is satisfactory (see table 7.6). In order to obtain more efficient preconditioners it is useful to reorder the stiffness matrix before factorization such that the subsequent dropping of entries preserves the original structure to a greater extent. One could also use a more suitable ordering to produce the same effect. The performance of our independent columns preconditioners for red-black ordering are examples of such a case.

Experiments in [10] compare speedometer and red-black orderings. The results obtained show that both orderings yield effective preconditioners. Orderings such as minimum degree which do not cluster the entries near the diagonal are also likely to be suited to this scheme. Due to the scattered sparsity pattern associated with such orderings we are likely to have to drop fewer entries to obtain suitable preconditioners. We need to take such measures to reduce the number of extra iterations when increasing the block size.

One more consideration concerning the formation of good preconditioners relates to the level of fill-in allowed. It is likely that most of the important entries will be level 1 or 2 (see sections 2.3.4 and 3.2.2 and [14]). We could consequently aim to include such low level fill-ins in the factors rather than those in higher levels. This has been discussed in more detail in section 3.2.2.

The above modifications to the basic scheme for the formation of suitable preconditioners for the independent columns method aim to improve the quality of the factors (ie. the preconditioner) by obtaining a more accurate picture of the stiffness matrix. The larger entries are more important for achieving this and hence their preservation is aimed for in the above schemes. The use of the mentioned modifications can be the subject of experiments involving the design of other suitable preconditioners.

An implementation preserving the largest entries in each block of columns has not improved the quality of row and red-black ordered preconditioners significantly (see section 7.3.3). However, other orderings (eg. minimum degree) might benefit from this measure. We have also discussed the performance of the independent columns method on a transputer array in [10] and [11].

We shall now make a detailed comparison between the quality of preconditioners for the independent columns scheme produced by row and red-black orderings of the underlying mesh. Our aim is to examine the effectiveness of the preconditioners for both serial and parallel processing.

First of all let us discuss tables 7.6 and 7.7 ie. the effect of ordering on the rate of convergence of *incomplete* ILU(0) factors. We can see that the rate of convergence is in general faster for red-black ordering. As we increase the block size and drop more entries to meet the necessary requirements for the independent columns scheme, the rate of convergence becomes slower. This is, of course, because the factors resemble the stiffness matrix to a lesser extent as the block size increases. What is interesting, however, is that this degradation in performance is smaller in the case of red-black ordering.

The last statement shows that as we originally expected, fewer entries need to be dropped with red-black ordering and we consequently manage to preserve the authenticity of the factors with respect to the stiffness matrix to a greater extent. This was, indeed, our motive for experimenting with this alternative ordering. By this we mean that red-black was thought to be a good choice of ordering for the independent columns method because it results in sparsity patterns which conform with the required structure to a certain extent (see section 7.2).

Even though we may expect other orderings (possibly minimum degree) to exhibit even further improvements, we can say that mesh ordering can have a significant effect on the parallel performance of the preconditioners and that a red-black ordering appears well suited to this method. The results in [10] also show this ordering to perform better than speedometer ordering for the independent columns method.

Let us discuss the factors determining the achievement of real gain from preconditioning before going on to examine the meaning of the results in the tables for parallel implementations using independent columns (ie. tables 7.12 to 7.15). The costs associated with preconditioning are those due to the computation of the preconditioner and the extra steps required in each iteration. In the case of ILU preconditioning these steps correspond to the triangular solves at each iteration. If we are to make an overall gain from the use of preconditioning we need a preconditioner which:

- is not expensive to compute
- reduces the number of iterations significantly compared to the CG method
- does not increase the cost of each iteration unreasonably
- can be parallelised effectively.

Tables 7.1 and 7.2 show how we make an overall gain from preconditioning. We can see from these tables that even though the cost of each iteration increases when we use preconditioning, the number of iterations drops sufficiently to cause an overall gain. This is, of course, also related to the fact that the cost of obtaining our preconditioners is low.

We shall now relate the above mentioned issues to the results in tables 7.12 to 7.14, focusing on the second and fourth items ie. number of iterations and efficient parallelisation. Bearing in mind what was said in the previous paragraph we can see why the tables have been presented in their particular form. For each block size (ie. N_p) our ultimate aim is to determine a speed up which reflects the real performance of the preconditioner in a parallel environment. In order to achieve this we need to remember the factors mentioned above during the computation of the speed up. This means that the speed ups presented in these tables take into account the cost of computing the preconditioner and are calculated by comparing the performance with N_p processors to that of the fastest sequential algorithm doing the same job ie. the results in tables 7.1 and 7.2.

The speed ups for level scheduling have been computed in the same manner. We can consequently say that for both methods the speed up figures indicate the net gain from using preconditioning with parallelisation. The results show that both methods produce satisfactory speed ups for large enough meshes with a suitable ordering.

We are now in a position to examine the core issue of parallel performance. Our speed up figures for different block sizes indicate that the independent columns method performs significantly better with red-black ordering. This means that not only are the serial performances of the preconditioners better, but also that they parallelise more efficiently. This is no longer an issue concerning the quality of the preconditioner but one relating to its structure.

More precisely, the deciding factors for parallel efficiency are granularity and load balancing. The results indicate that our red-black preconditioners bring with them increased granularity. But why should the load balancing be any better? This is accounted for by the favourable sparsity pattern in red-black matrices. For the case of row ordering, we drop entries from the band in such a way that many columns end up with more entries than others. Since red-black matrices are already to a certain extent in the form we require, the dropping preserves the diagonal structure and the final matrices have many columns which are exactly or nearly equally full. This improves the load balancing significantly.

As far as the actual speed up values are concerned we can see that, as expected, parallel performance improves with mesh size with a maximum of 7.11 on 12 processors for red-black ordering. For row ordering, the best performance is a speed up of 4.83 on 8 processors. We must also note that whereas the speed ups continue to rise for red-black ordering beyond 10 processors, there is a fall in the speed ups for row ordering after a peak at 8 processors.

Let us now return to level scheduling and discuss the distribution of the level lengths for row ordering. There are a large number of rows at level 1. These have not been included in the distribution graphs 7.1-7.4. After level 1, the level length steadily rises and eventually falls in the same way (see graphs 7.1 and 7.3). The symmetry observed is due to the regularity of the underlying row-ordered mesh. There is a flat area in these distributions which corresponds to the longest levels. Graphs 7.2 and 7.4 tell us that approximately one third of the levels have the longest lengths.

We can also see from graphs 7.2 and 7.4 that the same percentage of the levels are associated with most level lengths. The actual number of levels for these is 4. This means that there are 4 of most block sizes. As far as the variation of the distribution with problem size is concerned, we observe that there is a reduction in the relative size of the flat area as problem size increases (see graphs 7.1 and 7.3). This means that fewer levels are of the longest lengths for larger meshes. The actual percentage change is from 41% (441 nodes) to 35% (10201 nodes) which means that the pattern is preserved to a large extent with increased problem size.

Another important relevant issue is the relation between the level length distribution and parallel efficiency. We ideally want the distribution to have the shape of a tall and narrow rectangle so that the number of levels is small and these are all long relative to the number of processors. Furthermore, if N_p divides exactly into the lengths of the levels then we have no idle processors. Otherwise, the remainder rows in each level have to be processed using fewer than N_p processors leaving some processors idle. Also, the broader the distribution, the greater are the number of synchronisation points. This is why we desire narrow rectangles. If there are many short levels we have the overhead due to idle processors when N_p is greater than the length of many of these levels. Consequently for good parallel efficiency we require:

- A high proportion of long levels (large granularity and minimum idle processors for large N_p)
- Few levels (minimum synchronisation)
- Many level lengths divisible by N_p (minimum idle processors).

Let us now relate the above issues to our current distributions (see graphs 7.1 and 7.3). If we use a small number of processors (eg. $N_p = 2$), we shall obtain high parallel efficiency since we have very little loss due to idle processors. If we use a large number of processors, we reduce the overhead associated with the allocation of tasks to rows at long levels but have to pay a large overhead due to idle processors at short levels. The speed up does increase as N_p increases but the parallel efficiency drops due to the idle processors. This means that this type of distribution does not scale very well.

It is important that N_p is a factor of the level length of highest frequency and divides exactly into many others. One possibility is to choose the number of processors on the basis of the level lengths. As the problem size increases, it becomes safer to use large numbers of processors since the proportion of the remainder rows is small compared to the level lengths.

The factors discussed above explain why the speed ups obtained for level scheduling are so much lower than the theoretical values (see tables 7.17 and 7.18). Our distributions for the row-ordered mesh show that if we use a large number of processors compared to the short level lengths there is some loss in efficiency due to idle processors. Also, there are many levels in the distribution and this means many synchronisation points. The granularity for short levels is far from ideal when N_p is large. Finally, many long level lengths are not multiples of the numbers of processors. There is a certain degree of improvement in some of these sources of inefficiency as the problem size increases. However, a more fundamental change in the distributions is necessary if there are to be significant improvements in the speed ups.

Since the distributions are determined by the ordering of the underlying mesh, we need to seek optimal distributions by experimenting with different orderings. The best orderings should be those for which there are large groups of independent rows in the stiffness matrix such as red-black or minimum degree [18].

This brings us to the discussion of how row and red-black orderings compare in the case of level scheduling according to our experiments (see tables 7.8 to 7.11). As expected, the speed ups improve with problem size for both orderings. The speed ups for red-black ordering are higher than those for row ordering but the improvement is smaller than the case of independent columns. This is due to the fact that the improvements in granularity and load balancing are more pronounced when we switch orderings for the independent columns method. The best speed up for level scheduling is 7.29 on 12 processors for red-black ordering and 5.54 on 8 processors for row ordering. The improvement in speed ups stops at $N_p = 8$ for row ordering whereas it continues to rise beyond $N_p = 10$ for red-black ordering. This suggests that we obtain better level length distributions with red-black ordering which in turn improve the granularity and load balancing.

Finally let us make some comments referring to how level scheduling and the independent columns method compare in terms of parallel performance. Level scheduling performs better than independent columns in general as far as our results show. The ordering scheme used alters the performance of both methods and further experiments could lead to other interesting conclusions. Tables 7.11 and 7.15 show how effectively the two methods parallelise the forward and backward substitutions only. We can see from these tables that from this point of view the performance of independent columns is even closer to that of level scheduling. Level scheduling uses a *full* ILU(0) preconditioner, whereas the independent columns method uses an (increasingly) incomplete ILU(0) preconditioner. This accounts for some of the difference in the performances of the two methods.

Chapter 8

Conclusions

8.1 Overview of Chapter

This chapter contains our conclusions regarding all aspects of finite element analysis. The topics are itemised and discussed separately. Some implementation details have been included to clarify the conclusions. The final section explains how one might build upon the work presented in the thesis.

Our objectives have been the analysis of the finite element method with the aim of exploring its potential for parallelisation. We have autoparallelised a large finite element program (GASP4) and tested its efficiency (see chapter 4). A study of the structure of the autoparallelised program has been made. We have made recommendations concerning the design of programs suitable for parallelising compilers.

The FEM consists of the assembly of a stiffness matrix and the subsequent solution of a set of equations with the stiffness matrix as the coefficients. We have implemented three algorithms for the parallel assembly of the stiffness matrix stored in a sparse matrix format (see chapter 6). Two of these algorithms have proved to be very efficient giving speed ups that are near ideal.

For the solution phase, we have used the PCG method. This method is suited to the solution of large sparse systems due to its low storage requirements and low operations count (see section 2.3.4). The main difficulty with parallelising this method is the preconditioning step which consists of a pair of triangular solves for our chosen preconditioner which is an incomplete factorization of the stiffness matrix. The ILU(0) preconditioner has been found to be an effective means of reducing the number of CG iterations. This preconditioner can be computed at a low cost.

We have implemented two algorithms for parallel triangular solution. One is level scheduling which is a row-oriented blocking strategy (see section 7.3.2). This scheme has proved to be quite efficient (see section 7.4.4). The other parallel triangular solution scheme is a novel column-oriented approach (independent columns) based on the parallel update of RHS entries using distinct columns of the coefficient matrix at each stage (see section 7.3.3). The speed ups obtained using this method are similar to those for level scheduling (see section 7.4.4). We have designed methods for obtaining efficient preconditioners which have the necessary structure for the implementation. The two methods of parallel solution have been tested for row and red-black orderings of the unknowns.

8.2 Autoparallelisation

In order to benefit from the use of autoparallelisation the code must be written with simple and clear data dependencies. This is because if the dependencies are too complex then existing parallelising compilers are incapable of analysing these effectively enough.

If the code is hard to analyse at higher levels, the compiler will implement parallelism at lower levels where it can be sure of correctness in parallel processing. This gives rise to low grain parallelism which involves large synchronisation overheads and is unlikely to be beneficial.

We recommend that code to be autoparallelised is written such that the possibility of large grain parallelisation can be easily detected by the compiler. For example, it would be quite efficient to run parallel copies of large or medium size subroutines. Subroutines must consequently be written such that this kind of parallel processing can be affected without the need for synchronisation. This requires a conscious effort on the part of the designers of the code who should be made aware of the prospects and limitations of the compiler with regards to parallelisation. A well structured program with simple and clear data dependencies is a necessity for efficient autoparallelisation.

The autoparallelisation of GASP4 using the *epf* compiler proved to be inefficient (see section 4.4). The processing time actually increased as the number of processors was increased. The reason for this is the large amount of synchronisation overhead paid when the number of processors increases. This overhead is not sufficiently compensated for by a reduction in the processing time due to parallel processing and we consequently have a net increase in processing time.

The level at which parallelism is implemented by *epf* in GASP4 is too low to be beneficial. Since the data dependencies in the program are too complex at higher levels, the compiler only parallelises what it is certain to work correctly. This is often very low grain and hardly ever substantial. There is no point having several parallel tasks performing operations such as initialising small arrays. The overhead associated with setting up and managing these tasks is too high compared to the gain due to parallel execution. Furthermore, this type of inefficiency increases as we use more processors. It is very important to allocate enough work to tasks to compensate sufficiently for the cost of their implementation.

Another source of inefficiency in the autoparallelised version of GASP4 is the presence of a great deal of low level synchronisation (see section 4.4.2). This means that parallel tasks are required to communicate a large number of times during the execution of a few lines of code. This is again due to the lack of ability of the compiler to detect possibilities for parallelisation at higher levels. The overheads associated with the communications necessary for implementing low level synchronisation can become very large. This makes it very difficult to obtain any real gain from such parallel implementations.

In summary, we should say that we can only expect to benefit from the use of autoparallelising compilers if we have one or both of the below items:

- We supply them with well structured code with simple and clear data dependencies. The code must also be written such that large grain parallelism is made possible.

- The compilers are made sophisticated enough to be able to detect parallelism even when the code is difficult to analyse. One further desirable feature would be the ability of the compiler to restructure the code with the aim of making efficient parallelisation possible.

While we are not in possession of the second item we need to write *autoparallelisable* programs bearing in mind the points discussed above.

We have obtained some speed up by hand parallelising parts of GASP4 (see section 4.5.2). Strategies such as accumulating local sums to reduce the overhead due to critical sections and processing parallel blocks of DO loop indices to avoid the cost of DOALL constructs have been used to provide efficient parallel processing. In order to parallelise more of the program we would require a detailed understanding of the data dependencies so that the code can be restructured and written as an efficient parallel program with the appropriate granularity.

8.3 Parallel Assembly

We have identified the sources of contention for the parallel assembly of different elements into a sparse representation of the stiffness matrix (see section 6.2). These are due to the sharing of nodes by different elements which means that the same entry in the overall matrix can be dependent on several entries in different element matrices. We also need to control inadvertent access to the table subscript by parallel tasks. Three different methods of parallel sparse assembly have been designed.

It is not efficient to allow parallel assembly into the same row of the overall stiffness matrix by parallel tasks assembling their respective element matrices (see section 6.3). Each of these will be assembling different parts of the same row with the possibility of a shared overall entry. The synchronisation required to control this type of contention has to be implemented at low levels giving rise to small granularity and large overheads. A synchronisation LOCK provides mutually exclusive access to the table entries by parallel tasks during row creation and entry insertion. A CRITICAL SECTION primitive is used to protect the update of existing table entries.

As mentioned above, the low grain parallelisation used is inefficient. This source of inefficiency degrades the parallel performance and the speed ups obtained are far from ideal. The loss in efficiency is more pronounced for large numbers of processors (ie. $N_p \geq 6$). With 4 processors, the speed up is 2.53 and when using 12 processors this is only increased to 5.52 (see section 6.6).

The idea in performing parallel assembly of different rows only (see section 6.4) is to remove the contention problem associated with the sharing of nodes by different elements. If we can be sure that at any one time one particular row of the overall stiffness matrix is created or updated by only a single task then we only need to protect the table subscript during the parallel assembly of different rows.

Each row of the stiffness matrix has associated with it a unique LOCK variable. This ROWLOCK is checked by each task attempting to create or update a row. If the ROWLOCK is free the task proceeds. Otherwise it is suspended at that point awaiting a signal by another task currently processing the row (ie. holding the ROWLOCK). This is an efficient way of performing parallel sparse assembly (see section 6.6). The only substantial overhead is due to the tasks waiting on a ROWLOCK. The speed ups are now nearer ideal at 3.37 for $N_p = 4$ and 7.07 for $N_p = 12$.

Further improvement in the efficiency of parallel assembly of different rows is possible by the reduction in the waiting time on ROWLOCK's. This can be achieved by a reordering of the assembly sequence such that disjoint elements are assembled simultaneously as far as possible (see section 6.5). A colouring strategy is used to group disjoint elements such that the mesh is divided into several colours. Instead of the parallel assembly of different elements at random we now assemble adjacent elements first by processing the colours in sequence. The waiting on ROWLOCK's only occurs at the boundary of the colours. This brings about a significant reduction in waiting time with a low preprocessing cost. The speed ups for this improved version of parallel assembly of different rows have increased to 3.77 for $N_p = 4$ and 10.01 for $N_p = 12$ (see section 6.6).

Finally, we should comment on the general effect of problem size and increasing numbers of processors on parallel efficiency. As can be seen from table 6.2, the speed ups improve as problem size increases. This is because for larger problems the granularity is increased and the synchronisation overheads are better compensated for by the reduction in processing time due to parallel execution. This means that each task is now doing more work before it needs to communicate and the computation to communication ratio is higher. This increases the parallel efficiency.

For a given problem size, as we increase the number of processors we observe a reduction in the parallel efficiency (see table 6.1). This is because as N_p increases we have to pay an increasing amount of overhead associated with the setting up and management of parallel tasks. If the granularity happens to be low then the usage of a large number of processors causes a large amount of inefficiency due to a small computation to communication ratio. For large N_p there is also a greater chance of having idle processors. All these factors account for a drop in the parallel efficiency as N_p increases. We have aimed to minimise this effect by bearing in mind the above factors at the design stage.

8.4 Parallel Solution

We have implemented the PCG method in parallel. The profile of the method is such that most of the time is spent on performing matrix multiplications and the triangular solves (see section 7.4.1). The ILU(0) preconditioner is found to be an efficient means of reducing the number of iterations. The formation of this preconditioner involves a low cost (see section 7.4.1).

All the PCG steps are straightforward to parallelise apart from the preconditioning operations (triangular solves for ILU preconditioning). The speed ups for the non-preconditioning steps are near ideal (see table 7.16).

We have implemented parallel triangular solves using level scheduling and a new method involving parallel update with *independent columns.* Both methods have been tested with row and red-black orderings of the unknowns.

Level scheduling is based on a blocking abstraction in which independent rows of the coefficient matrix are identified and grouped together in *levels* (see section 7.3.2). These levels must then be processed in sequence while within each level the rows can be processed concurrently. The independent columns scheme is a column-oriented approach to parallel triangular solution (see section 7.3.3). At each stage of this method a number of unknowns are computed and used in the simultaneous update of distinct RHS entries (using distinct columns). For this parallel update to be possible, the L and U factors must have a certain structure.

Level scheduling has proved to be quite efficient giving rise to speed ups which are mostly in the range 2 - 5 with a maximum of 6.10 on 12 processors for red-black ordering (see table 7.11). These are comparable to theoretical speed ups for small N_p but for large numbers of processors the theoretical values are much higher (see tables 7.17 and 7.18). The reasons for this difference are non-optimal level length distributions (see below and section 7.5.2) and large overheads when using many processors. The speed ups increase with problem size due to increased granularity.

The independent columns scheme gives similar speed ups to level scheduling (see tables 7.12 to 7.15). The parallel efficiency of this scheme is determined by the distribution of the entries in each block of columns and the average number of entries per column in the block. An even distribution of entries minimises the overhead due to idle processors waiting for dense columns. As the problem size increases, the average number of entries per column becomes larger and this increase in granularity increases the parallel efficiency. We could also improve the load balancing by aiming to have equal or nearly equal numbers of entries in columns of the same block. The speed ups obtained using independent columns indicate that the entry distributions for row ordering are not ideal for minimising idle processor time.

The above considerations are necessary for improving the parallel performance of this scheme. This has been the motive behind testing the independent columns scheme with an alternative ordering ie. red-black. We have obtained improved performance with the latter ordering (see tables 7.12 to 7.15). The reasons for this improvement are discussed below (also see section 7.5.2).

In order to obtain suitable preconditioners for the independent columns scheme we have performed an *LU* factorization during which entries which disturb the independence requirements are dropped (see section 7.3.3). The performance of these preconditioners is satisfactory for both orderings (see tables 7.6 and 7.7). We observe an expected increase in the number of iterations as the block size increases. This is due to the fact that we need to drop an increasing number of entries during factorization to satisfy the independence requirements and the factors become less effective. There is, however, an overall reduction in the solution time due to using the method (see tables 7.12 to 7.14). This means that even though we sacrifice a few more iterations each time we increase the block size, this is compensated for sufficiently by the reduction in processing time due to the possibility of parallel processing.

The best speed up obtained due to the independent columns scheme is 7.11 on 12 processors (see table 7.14). The mentioned speed ups are owed to the design of efficient preconditioners (see tables 7.6 and 7.7). The improvement due to the use of red-black ordering is due to the fact that the performance of red-black preconditioners is not degraded to a large extent as block size is increased. We have consequently obtained a more suitable ordering of the underlying mesh. Other possible candidates are orderings such as minimum degree which are also likely to require the dropping of small numbers of entries. This issue has also been discussed in section 7.5.2.

Our implementation aiming to preserve the largest entries in each block of columns in order to improve the quality of the preconditioner has not resulted in significantly faster rates of convergence (see section 7.3.3). Other orderings (eg. minimum degree) might benefit from such a scheme to a greater extent.

One further point must be made regarding the structure of the preconditioners for the independent columns scheme. If we allow non-diagonal entries into the L_i blocks (see figure 7.5, page 106), the effectiveness of the preconditioner will be enhanced due to the extra entries. There will, however, be an increased amount of work to be done during the first step of the algorithm ie. the determination of the solution subvector (see section 7.3.3). This must be done using a triangular solve since the parallel division possible for the case of a diagonal L_i is no longer possible. A study of the effect of this trade-off on the efficiency of the algorithm can give rise to interesting results.

Let us now discuss further the reasons for the improved convergence rates obtained using red-black *incomplete* ILU(0) preconditioners (see tables 7.6 and 7.7). For both orderings, as the block size increases the rate of convergence becomes slower. In the case of red-black ordering, however, this degradation in performance due to an increasing number of dropped entries is smaller. This is because fewer entries need to be dropped in the red-black case for each block size since the entries are already scattered in the form required by the independent columns method to a certain extent (see section 7.2).

We can see from tables 7.12 to 7.15 that the speed ups for the independent columns method are affected by the choice of ordering to a great extent. The speed ups obtained using red-black ordering are significantly higher than those for row ordering. This is due to better load balancing associated with red-black ordering which minimises idle processor time (see section 7.5.2). Also, the granularity due to the processing of larger meshes accounts for the fact that for both orderings the best parallel efficiencies are obtained for the largest problem sizes.

In order to benefit from level scheduling we need many levels with large lengths relative to the number of processors. This will ensure large granularity which is essential for good parallel efficiency. The distribution of the level lengths for row ordering is such that around one third of the levels have the largest level length (see graphs 7.1 to 7.4). This means that we can expect better performance from orderings which produce a more uniform distribution with a larger number of long levels. The processing of short levels is inefficient, especially for large numbers of processors, since we not only have many synchronisation points but also some idle processors.

This calls for a comparison between row and red-black orderings with regards to their parallel performance using level scheduling. The speed ups for red-black ordering are higher than those for row ordering in general (see tables 7.8 to 7.11). This is because we obtain better level length distributions with red-black ordering which improve the granularity and load balancing (see section 7.5.2).

We shall now address the issue of an optimal number of processors for level scheduling. By looking at the level length distribution graphs we can see that for row ordering the least amount of overhead would be paid by using a number of processors approximately equal to the average level length. This is because even though the speed up increases with N_p, we pay an increasing amount of overhead due to idle processors during the processing of short levels. For maximum parallel efficiency we need to seek an optimal trade-off such that we use as many processors as possible without paying a large cost due to idle processors. As mentioned above, this occurs at around the average level length.

The most efficient level length distributions must have the shape of tall and narrow rectangles ensuring a small number of levels all with large level lengths (ie. large granularity). It is also important that N_p divides exactly into many large level lengths. We can then be sure that when processing these levels there is no significant loss due to idle processors. This is an issue because the cost of idle processors during the processing of *left over* rows at each level can become large. The improved parallel performance associated with red-black ordering suggests that the sparsity pattern for this ordering is more suited to the achievement of minimum idle processor time.

8.5 Suggestions for Future Work

The *independent columns* scheme seems to have a great deal of potential for performing efficient triangular solves. It would be interesting to investigate further the formation of other suitable preconditioners for this scheme. This should be done in terms of experimenting on different orderings of the underlying mesh. We need preconditioners (L and U factors) which conform to the desired structure closely such that the dropping of entries during factorization does not degrade the rate of convergence adversely.

The parallel efficiency of the independent columns scheme can also be improved by the use of orderings which produce more uniform entry distributions in each block of columns. Some such orderings may also benefit from schemes such as preserving the largest entries in each block of columns.

The level scheduling scheme can also benefit from more suitable orderings. It would be useful to experiment on the effect of ordering on level length distributions since it is the latter that ultimately dictate the efficiency of the level scheduled scheme. The aim should be to determine which orderings produce distributions with fewer and longer levels.

The parallel assembly code has been ported and run successfully on a British Gas Cray computer. The code for the parallel solution schemes will be ported next.

Bibliography

[1] Anderson,E.C.,*Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations*,Technical Report 794,University of Illinois,CSRD,Urbana,IL,1988.

[2] Anderson, E.C., and Saad, Y, *Solving Sparse Linear Systems On Parallel Computers*, International Journal of High Speed Computing, Vol.1, No.1, pp. 73-95, 1989.

[3] Ashby,S.F., *Polynomial preconditioning for conjugate gradient methods*, Ph.D. thesis, Computer Science Dept., University of Illinois, Urbana, IL, 1987.

[4] Axelsson,O.,*A Generalized Conjugate Direction Method and its Application on a Singular Perturbation Problem*, Lecture Notes in Mathematics 773,Springer-Verlag,Berlin,Heidelberg, New York,1980,pp. 1-11.

[5] Becker,E.B.,Carey,G.F.,and Oden,J.T.,*Finite Elements - An Introduction*, Volume 1,Prentice-Hall,1981.

[6] Brusa,L.,and Riccio,F.,*Substructure technique for parallel solution of linear systems in finite element analyses*,In "Parallel Computing. Methods, Algorithms and Applications", Proceedings of the International Meeting on Parallel Computing, Verona, Italy, 28-30 Sept. 1988.

[7] Carey,G.F.,*Parallelism in Finite Element Modelling*, Communications in Applied Numerical Methods,Vol.2,1986.

[8] Chandra,R.,Eisenstat,S.C.,and Schultz,M.H.,*A modified conjugate residual method for partial differential equations*, In "Advances in Computer Methods for Partial Differential Equations - 2" (Vichnevetsky,R.,ed.),Publ.IMACS(AICA),1977.

[9] Chien,L.S.,and Sun,C.T.,*Parallel processing techniques for finite element analysis of large truss structures*, Computers and Structures,Vol.31,No.5,1989.

[10] Cook,R.,Pakzad.,M.,and Phillips,C.,*Parallel Implementation of Conjugate Gradient-Type Methods*,presented at The Sixth International Conference on Scientific Computing,University of Benin,Nigeria, 24-28 Jan.1994.

[11] Cook,R.,Pakzad.,M.,and Phillips,C.,*Parallel Preconditioners for the Conjugate Gradient Method*,Tech.Rept.No.467,Dept.Comput.Sci., Univ.Newcastle upon Tyne,1994.

[12] Cuthill,E.,and McKee,J.,*Reducing the bandwidth of sparse symmetric matrices*, Proc. of the 24th National Conference of the ACM,ACM Publications,1969.

[13] Da Cunha,R.D.,and Hopkins,T.,*A Parallel Implementation of the Restarted GMRES Iterative Method for Nonsymmetric Systems of Linear Equations*, Technical Report No. 7/93,Computing Laboratory,Univ. of Kent at Canterbury, May 1993.

[14] D'Azevedo,E.F.,Forsyth,P.A.,and Tang W.P.,*Towards a cost-effective preconditioner with high level fill*, BIT,Vol.32,pp. 442-463,1992.

[15] Du Croz,J.,*Evolution of Parallel Algorithms in Dense Linear Algebra*, In "Parallel Computation",Proceedings of the IMA Conference on Parallel Computation,18-20 Sept. 1991, Oxford University,IMA,1994.

[16] Duff,I.S.,Erisman,A.M.,and Reid,J.K.,*Direct Methods for Sparse Matrices*, In "Parallel Computation",Proceedings of the IMA Conference on Parallel Computation,18-20 Sept. 1991, Oxford University,IMA,1994.

[17] Duff,I.S.,*Exploitation of Parallelism in Direct and Semi-Direct Solution of Large Sparse Systems*, In "Parallel Computation",Proceedings of the IMA Conference on Parallel Computation,18-20 Sept. 1991, Oxford University,IMA,1994.

[18] Duff,I.S. and Meurant,G.A.,*The effect of ordering on preconditioned conjugate gradients*, BIT,Vol.29,pp.635-657,1989.

[19] Dutto,L.C.,*The effect of ordering on preconditioned GMRES algorithm for solving the compressible Navier-Stokes equations*, International Journal for Numerical Methods in Engineering,Vol.36, pp. 457-497,1993.

[20] Eisenstat,S.C.,Elman,H.C.,and Schultz,M.H.,*Variational iterative methods for nonsymmetric systems of linear equations*, SIAM Journal of Numerical Analysis,Vol.20,pp. 345-357,1983.

[21] Encore. *Encore Parallel Fortran*, Ref.No. 724-06785,Encore Computer Corporation,Fort Lauderdale,Florida, 1988.

[22] Farhat,C.,*A simple and efficient finite element method domain decomposer*,Computers and Structures,Vol.28,No.5,1988.

[23] Farhat,C.,*Multiprocessors in computational mechanics*, Ph.D. Dissertation, University of California,Berkeley,1986.

[24] Farhat,C.,Wilson,E.,and Powell,G.,*Solution of Finite Element Systems on Concurrent Processing Computers*, Engineering with Computers,Vol.2,1987.

[25] Farhat,C.,and Wilson,E.,*A New Finite Element Concurrent Computer Program Architecture*, International Journal for Numerical Methods in Engineering,Vol.24,1987.

[26] Farhat,C.,and Wilson,E.,*A parallel active column equation solver*, Computers and Structures,Vol.28,No.2,1988.

[27] Farhat,C.,and Crivelli,L.,*A general approach to nonlinear FE computations on shared-memory multiprocessors*, Computer Methods in Applied Mechanics and Engineering,Vol.72,1989.

[28] Farhat,C.,Pramono,E.,and Felippa,C.,*Towards parallel I/O in finite element simulations*, International Journal for Numerical Methods in Engineering,Vol.28,1989.

[29] Farhat,C.,and Wilson,E.,*Concurrent iterative solution of large finite element systems*, Communications in Applied Numerical Methods,Vol.3,1987.

[30] Farhat,C.,and Wilson,E.,*Modal superposition dynamic analysis on concurrent multiprocessors*, Eng. Computations,1987.

[31] Fillipone,S.,Marrone,M.,and Radicati di Brozolo,G.,*Parallel preconditioned conjugate-gradient type algorithms for general sparsity structures* Intern. J. Computer Math.,Vol.40,pp. 159-167,1992.

[32] Fletcher,R.,*Conjugate Gradient Methods for Indefinite Systems*, Lecture Notes in Mathematics 506,Springer-Verlag,Berlin,Heidelberg, New York,1976,pp. 73-89.

[33] Freeman,T.L.,and Phillips,C.,*Parallel Numerical Algorithms*, Prentice Hall Int.,1992.

[34] Gajski,D.,Kuck,D.,Lawrie,D.,and Sameh,A.,*Plan for the construction of a large scale multiprocessor*,83-1123,Feb.1983,Dept. of Computer Science, University of Illinois at Urbana-Champaign,(Cedar Doc. No. 5).

[35] George,A.,*Computer implementation of the finite element method*, Tech. Rept. STAN-CS-208,Stanford University,1971.

[36] George,A.,and Liu,J.W.,*Computer solution of large sparse positive definite systems*,Prentice Hall,1981.

[37] Golub,G.H. and van Loan,C.F.,*Matrix Computations*, Second Edition,The Johns Hopkins University Press,1989.

[38] Gladwell,I.,and Wait,R.,*A Survey of Numerical Methods for Partial Differential Equations*, Oxford University Press,1979.

[39] Greenbaum,A.,Li,C., and Chao,H.Z.,*Comparison of Linear System Solvers Applied to Diffusion-Type Finite Element Equations*, Numer.Math.,Vol.56,pp.547-589,1989.

[40] Hammond,S.W. and Schreiber,R.,*Efficient ICCG on a shared memory multiprocessor*, Int. J. High Speed Comput.,Vol.4,No.1,pp. 1-21,1992.

[41] Heroux,M.A.,Vu,P.,and Yang,C.,*A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP*, Applied Numerical Mathematics,Vol.8,pp. 93-115,1991.

[42] Hestenes,M.R.,and Stiefel,E.,*Methods of conjugate gradients for solving linear systems*,J.Res.Nat.Bureau Standards,Vol.49,pp.409-436,1952.

[43] Hockney,R.W.,and Jesshope,C.R.,*Parallel Computers 2*, Adam Hilger(IOP),1988.

[44] Hwang,K.,and Briggs,F.A.,*Computer Architecture and Parallel Processing*,McGraw-Hill,1987.

[45] Irons,B.M.,*A Frontal Solution Program for Finite Element Analysis*, International Journal for Numerical Methods in Engineering,Vol.2,1970.

[46] Johnson,O.G.,Micchelli,C.A., and Paul,G.,*Polynomial preconditionings for conjugate gradient calculations*, SIAM J.Numer.Anal.,Vol.20,pp. 362-376,1983.

[47] Jordan,H.,Benten,M. and Arenstorf,N.,*Force user's manual*, Department of Electrical and Computer Engineering,Uni. of Colorado, Boulder,1987.

[48] Jordan,T.L.,*Conjugate gradient preconditioners for vector and parallel processors*, In "Elliptic Problem Solvers 2, Proceedings of the Elliptic Problem Solvers Conference",Monterey,CA, Jan.10-12 1983, (Birkhoff,G.N.,and Schoenstadt,A.,eds.),pp. 127-139,Academic Press,1983.

[49] Kantorovich,L.V.,and Krylov,V.I.,*Approximate Methods of Higher Analysis*, P.Noordhoff,Ltd.,The Netherlands,1958.

[50] Keyes,D.E.,and Gropp,W.D.,*A comparison of domain decomposition techniques for partial differential equations and their parallel implementation*, SIAM J.Sci.Stat.Comput.,Vol.8,166-202.

[51] Lawrie,D.H.,and Sameh,A.H.,*Applications of structural mechanics on large-scale multiprocessor computers*,In "Impact of new computing systems on computational mechanics",(A.Noor,ed.),The ASME,1983

[52] Lee,P.A.,*Parallel Processing on the Multimax Computer System*, Parallel Processing Memorandum (PPM/001),Computing Laboratory, University of Newcatle upon Tyne,1987.

[53] Leuze,M.R.,*Parallel triangularisation of substructured finite element problems*, Linear Algebra and its Applications,Vol.77,1986.

[54] Luo,J.C.,*An incomplete inverse as a preconditioner for the conjugate gradient method*, Computers Math. Applic.,Vol.25,No.2,pp. 73-79,1993.

[55] Mitchell,A.R.,and Wait,R.,*The Finite Element Method in Partial Differential Equations*, John Wiley and Sons,1977.

[56] Meurant,G.,*Multitasking the conjugate gradient method on the CRAY X-MP/48*, Parallel Computing,Vol.5,pp. 267-280,1987.

[57] Munksgaard,N.,*Solving Sparse Symmetric Sets of Linear Equations by Preconditioned Conjugate Gradients*, ACM Transactions on Mathematical Software,Vol.6,No.2,pp.206-219,1980.

[58] Melhem,R.G.,*A Modified Frontal Technique Suitable For Parallel Systems*,SIAM J.Sci.Stat.Comput.,Vol.9,No.2,1988.

[59] The Numerical Algorithms Group Ltd.,*The NAG Fortran Library Manual*, Vol.5,Mark 16,1993.

[60] The Numerical Algorithms Group Ltd.,*The NAG Fortran Library Manual*, Vol.6,Mark 16,1993.

[61] Nath,B.,*Fundamentals of Finite Elements for Engineers*, Athlone Press,1974.

[62] Nour-Omid,B.,Raefsky,A.,and Lyzenga,G.,*Solving finite element equations on concurrent computers*, Proc. of the ASME Symposium on Parallel Computations and their Impact on Mechanics,Dec. 13-18,1987.

[63] Pini,G.,Zilli,G.,and Contenaro,E.,*Sparse Systems of Linear Equations and Related Eigenanalysis in a Parallel Environment*, In "Parallel Computation",Proceedings of the IMA Conference on Parallel Computation,18-20 Sept. 1991, Oxford University,IMA,1994.

[64] Prenter,P.M.,*Splines and Variational Methods*, John Wiley and Sons.,1975.

[65] Quinn,M.J.,*Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill,1988.

[66] Reddy,J.N.,*An introduction to the finite element method*, McGraw-Hill,1984.

[67] Rothberg,E. and Gupta,A.,*Parallel ICCG on a hierarchical memory multiprocessor - Addressing the triangular solve bottleneck*, Parallel Computing,Vol.18,pp. 719-741,1992.

[68] Saad,Y.,*Krylov Subspace Methods on Supercomputers*, SIAM J.Sci.Stat.Comput.,Vol.10,No.6,pp. 1200-1232,1989.

[69] Saad,Y,*Practical use of polynomial preconditionings for the conjugate gradient method*, SIAM J.Sci.Stat.Comput.,Vol.6,pp. 865-881,1985.

[70] Saad,Y.,and Schultz,M.H.,*GMRES:a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J.Sci.Stat.Comput.,Vol.7,pp. 856-869,1986.

[71] Seager,M.K.,*Parallelising conjugate gradient for the CRAY X-MP*, Technical Report,Lawrence Livermore National Lab.,Livermore,CA,1984.

[72] Segerlind,L.J.,*Applied Finite Element Analysis*, John Wiley and Sons,Inc.,1984.

[73] Simon,H.D.,*Incomplete LU preconditioners for conjugate gradient type iterative methods* In "Proceedings of the SPE 1985 reservoir simulation symposium",pp. 302-306, Society of Petroleum Engineers of AIME,Dallas,TX,Paper number 13533,1988.

[74] Smith,B.F.,*An optimal domain decomposition preconditioner for the finite element solution of linear elasticity problems*, SIAM J.Sci.Stat.Comput.,Vol.13,pp. 364-378,1992.

[75] Sonneveld,P.,*GCS, A Fast Lanszos-Type Solver for Nonsymmetric Linear Systems*, SIAM J.Sci.Stat.Comput.,Vol.10,No.1,pp. 36-52,1989.

[76] Stoker,M.A.,*The exploitation of paralellism on shared memory multiprocessors*,Ph.D. Thesis, Computing Laboratory,University of Newcastle upon Tyne,1990.

[77] Stone,H.S.,*Parallel Computers*,In "Introduction to Computer Architecture",(Stone,ed.),SRA,1980.

[78] Strang,G.,and Fix,J.,*An analysis of the finite element method*, Prentice-Hall,1973.

[79] Sun,C.T.,and Mao,K.M.,*A global-local finite element method suitable for parallel computations*, Computers and Structures,Vol.29,No.2,1988.

[80] Van der Vorst,H.,*Parallel Aspects of Iterative Methods*, In "Parallel Computation",Proceedings of the IMA Conference on Parallel Computation,18-20 Sept. 1991, Oxford University,IMA,1994.

[81] Vinsome,P.W.,*Orthomin,an Iterative Method for Solving Sparse Sets of Simultaneous Linear Equations*, paper SPE 5729,Society of Petroleum Engineers of AIME,1976.

[82] Wilson,E.L.,and Farhat,C.H.,*Linear and Nonlinear Finite Element Analysis on Multiprocessor Computer Systems*, Communications in Applied Numerical Methods,Vol.4,1988.

[83] Wong,Y.S.,*Solving large elliptic difference equations on CYBER 205*, Parallel Comput.,Vol.6,pp.195-207,1988.

[84] Young,D.M.,and Jea,K.C.,*Generalized Conjugate-Gradient acceleration of nonsymmetrizable iterative methods*,In "Linear Algebra and its Applications",Vol.34,pp. 159-194,1980.

[85] Zilli,G.,*Iterative methods for solving sparse linear systems with a parallel preconditioner*, Intern. J. Computer Math.,Vol.44,pp. 111-119,1992.