A COMPARISON OF SPARSE AND ELEMENT-BY-ELEMENT STORAGE SCHEMES ON THE EFFICIENCY OF PARALLEL CONJUGATE GRADIENT ITERATIVE METHODS FOR FINITE ELEMENT ANALYSIS

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

Mechanical Engineering

---

by

Shivaraju B. Gowda

August 2002

Advisor: Dr. Lonny L. Thompson

August 2, 2002

To the Graduate School:

This thesis entitled "A Comparison of Sparse and Element-By-Element Storage Schemes on the Efficiency of Parallel Conjugate Gradient Iterative Methods for Finite Element Analysis" and written by Shivaraju B. Gowda is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Department of Mechanical Engineering.

Dr. Lonny L. Thompson, Advisor

We have reviewed this dissertation
and recommend its acceptance:

Dr. Sherrill B. Biggers

Dr. Richard S. Miller

Accepted for the Graduate School:

DEDICATION

I would like to dedicate this thesis to everyone who lent me a helping hand during my academic endeavors, especially to my parents, whose constant love and encouragement has helped me overcome the problems I encountered during this study.

ABSTRACT

The expanding use of distributed multi-processor supercomputers has made a significant impact on the speed and size of the problems which can be solved by the finite element method. The adoption of a standard Message Passing Interface protocol (MPI) has enabled programmers to write portable and efficient codes across a wide variety of parallel architectures. Finite element analysis is now feasible on many complex domains with large number of degrees of freedom. The key issues in parallel finite element analysis are the efficient decomposition of the domain to balance the load across the processors, solution of the system of equations that arise out of the discretization and optimal storage of the matrices.

This work describes the parallel implementation of the Conjugate Gradient iterative method for the solution of large linear equation systems resulting from the finite element method. Domain decomposition is used to distribute the unknowns in the finite element mesh to different processors to achieve scalable parallel performance. A diagonal Jacobi preconditioner is used with the iterative solver in order to accelerate the convergence. Two different types of matrix storage schemes, Element-By-Element (EBE) and Compressed Sparse Row (CSR) are implemented to compare the relative efficiency, scalability and total solution time. In order to illustrate the performance of the parallel finite element solver, Poisson's equation is solved for two and three dimensional model problems with Dirichlet and Neumann boundary conditions. The scalability analysis of the code comparing EBE and CSR schemes are conducted on two different parallel architectures; Sun UltraSPARCII and SGI MIPS.

Results show that the CSR format reduces computation time by approximately 30% compared to EBE. Good scalability is achieved for SGI machines with a vendor optimized implementation of MPI. On Sun machines, with the publicly available

implementation of MPI (MPICH), communication time dominates the total solution time as the number of processors is increased. Diagonal preconditioning is moderately effective in accelerating the convergence of the solver. Some amount of reduction in communication time is possible by overlapping computation and communication.

ACKNOWLEDGEMENTS

I am thankful to my advisor Dr. Lonny L. Thompson, who amidst his busy schedule, spared time to guide me through the project. Without his support, motivation and useful suggestions, this project would not have taken its present shape.

I would also like to thank Dr. Sherrill B. Biggers and Dr. Richard S. Miller for serving as members of my research committee and for their valuable comments which has brought this work to completion. Special thanks to Dr. Steven J. Stuart for sharing his vast expertise in parallel computing, which has helped me immensely in coding and analyzing the present work.

I am grateful to Cristian Ianculescu, Sethuramalingam Subbarayalu and Naveen Nandula for several enlightening discussions. I also owe my gratitude to Savitha E., Juneedasif Hussain and Uma Havaligi for their friendship, support and encouragement.

Table of Contents

List of Figures

Chapter 1

INTRODUCTION

The Finite Element Method (FEM) is extensively used to understand the failure mechanisms of engineering components [5]. The type and the complexity of the problems investigated by the finite element method are increasing with the increase in processor speed and decrease in memory cost. With the speed of the processor reaching its asymptotic limit, a major research thrust is focused on efficient parallel implementation of the finite element method.

Discretization of partial differential equations using the finite element method leads to large and sparse matrices. A matrix is termed sparse if it has very few nonzero coefficients. Traditionally, direct solution methods, which use some form of Gaussian elimination are used to the solve equations arising from FEM. Today, large three-dimensional models are commonplace. The memory required and the computational time for solving three-dimensional models involving multiple degrees of freedom per node, may seriously challenge the most efficient direct solvers available today because of their $O(n^2)$ floating point operations and memory requirements. The other disadvantage of the direct methods is that they may require backfill, when they operate on a matrix, hence the type of data structure of the matrix should store all coefficients in band or skyline including zeros. The discovery of efficient iterative solvers coupled with preconditioners for solving very large linear systems, have triggered a noticeable and rapid shift towards iterative techniques. Iterative solvers are methods which march towards the true solution using successive approximations. A preconditioner is a matrix which, coupled with the iterative solver, accelerates its convergence. The focus on iterative methods also stems from their applicability to parallel architecture

and scalability. The disadvantage of iterative methods is that they are not as robust and predictable as direct methods. Knowledge of the problem is necessary in order to apply iterative techniques and obtain a good rate of convergence.

As indicated earlier, the FEM involves manipulating large sparse matrices. One interesting and very useful property of iterative methods is that they require only the matrix-vector product of the stiffness matrix with a trial vector. As a result, the sparse matrices may be stored in an efficient way. The natural way to increase the efficiency of the solver is to not store the coefficients which have a value of zero in the matrix. There has been considerable research in this field in the late 80's and early 90's. Banded storage scheme and skyline storage schemes are some of the methods used to store the matrices in an effective way. The Element-By-Element (EBE) data structure, which is another form of storing the values of the stiffness matrix has gained wide popularity for iterative solvers [6, 7, 8]. In this scheme, the element stiffness matrices are stored and never assembled. Whenever a matrix-vector product is required in the iterative solver, the element-node connectivity matrix is used to achieve the matrix-vector product by using an element-by-element process, and vector assembly. Sparse storage schemes are also very prominent when using iterative solvers. Sparse storage schemes only store the nonzero entries of the assembled stiffness matrix. A popular implementation of sparse storage schemes is the Compressed Sparse Row format (CSR) [1].

There has been a considerable amount of research on the parallel implementation of the finite element method, with some recent work focusing on Linux clusters (Beowulf) [8]. Most of the research has focussed on the analysis algorithm, though some of the research also looked at methods to parallelize the Input/Output (I/O) [4]. Efficient methods to obtain good partitions of the finite element domain for distributed memory architectures are discussed in [9, 10]. The type of analysis algorithms studied includes static [8, 4] and transient [7, 11]. A number of researchers present results

for the application of the analysis algorithms on different parallel machines. These studies show that one algorithm might perform better than another algorithm on one machine, but may perform worse on another machine. This is due to the fact that the performance of the processors, memory units and communication network can vary substantially between different parallel machines. A lot of the work done in parallel finite element analysis is on distributed memory machines using message passing such as MPI for communication [12]. High Performance Fortran (HPF) is also used for parallelization [8] . HPF is a set of compiler flags which are used in conjunction with Fotran90 to parallelize loops in the code. This type of fine-grained data processing is applicable to shared memory computers. Another similar type of compiler directives is OpenMP. This type of parallel programming is simpler to implement but more limited in terms of functionality.

In the following sections, a brief overview of key parallel solution methods are given. A literature review of the direct and iterative methods commonly used to solve the linear equation systems arising from the FEM is presented.

1.1 Direct Solvers

Work in this group focuses on solving the fully formed linear system of equations using a direct method. Typically some form of Gaussian elimination is used. Gaussian elimination is performed using a variety of matrix storage schemes. Some popular storage schemes are the banded matrix scheme, the profile storage scheme [13] and a sparse storage scheme [14]. The way in which the matrix elements are assigned to the processors differ in most of the work presented. For example, the profile storage scheme, [13], assigns the columns of the matrix in a column cyclic manner among the processors, and to improve on the performance [13] assigns the columns in a block cyclic manner to the processes, adding zeros to the profile to obtain a blocked profile

scheme. There has also been a number of papers presented by mathematicians in which they look at the parallel solution of linear equations by direct means for band and sparse storage schemes. For banded schemes there are two approaches advocated. For banded matrices with a very small band, a divide and conquer approach is suggested, in which the equations are renumbered to allow the substructuring method to be employed [15]. For banded matrices with a very large band, no renumbering of the equations occurs and the elements of the matrix are assigned in a column cyclic or block column cyclic manner to the processes[15].

1.2 Iterative Solvers

Iterative methods are very popular for parallel computers. Work in this group focuses on solving the fully formed linear system of equations using an iterative method. A number of iterative methods are used. A lot of work is reported on the parallel implementation of the Conjugate Gradient (CG) Iterative Method [7, 16, 4]. In most of the work, a Jacobi preconditioner is applied with the CG method [16]. A popular approach when implementing the conjugate gradient method is the element-by-element approach [17]. In [7] a Jacobi and Hughes-Winget preconditioner is applied along with EBE implementation. Most preconditioners developed for serial and shared-memory parallel solvers require the information of the assembled stiffness matrix. Examples include Incomplete Choleski (IC) factorizations and Symmetry Successive Over Relaxation (SSOR). Current parallel approaches focus on preconditioners better suited for distributed memory computers to achieve data locality and better scalability. These include, for example subdomain-by-subdomain methods [17], which develop preconditioners from sparse matrices assembled over multiple subdomains of the mesh. The implementation of such kinds of preconditioners include block meth-

ods using standard preconditioners (e.g. Jacobi, Gauss-Seidel, incomplete Choleski) and Schur-complement decompositions for each subdomain.

The EBE framework requires the preconditioner to be expressed using only element-level calculations. So the number of preconditioners for EBE framework is limited. The most frequently used preconditioners for EBE based solvers include Jacobi and Hughes-Winget (HW) [7]. The Jacobi (diagonal preconditioner) has good convergence rates for diagonally dominant matrices but it has poor convergence rates for systems of equations with large condition numbers. The HW preconditioner often exhibits better convergence for equations approaching an ill-conditioned state and equations derived from 3-D finite element models. However, implementation of HW preconditioner is significantly more complex.

1.3 Domain Decomposition Methods

In order to enable processing of a finite element mesh on a distributed memory parallel computer, the computational domain is divided into separate subdomains with each subdomains assigned to a separate process. The tasks required of the subdomain are typically being performed on that process. This process is called Domain Decomposition. Domain Decomposition methods are commonly used in parallel finite element analysis. They are divide and conquer methods, which makes the task of assignment particularly easy for the parallel programmer.

Of the domain decomposition methods, the Schur complement method has been the most commonly used ([17],[18]). In the Schur complement method the equations are generated and solved on the subdomain interfaces. These methods require exact subdomain solutions at each iteration. The iterative substructuring method is also frequently used. In this method the complete global system is solved as a partitioned matrix. The finite element tearing and interconnecting (FETI) method [19, 11] de-

composes the domain into subdomains and then applies direct solvers within the subdomain and a modified preconditioned conjugate gradient algorithm to enforce continuity across subdomain boundaries.

1.4 Scope

In this work, an iterative substructuring method is used for parallel implementation of the finite element method. Parallel I/O is also utilized following the formulation in [4]. Domain decomposition is accomplished using METIS, a publicly available graph partitioning software package which is being actively developed by the Department of Computer Science at University of Minnesota [20]. A Preconditioned Conjugate Gradient method is used for the solution of the linear system. Two types of matrix storage schemes, Element-By-Element (EBE) and sparse storage (CSR), are compared. The scalability and performance analysis of the implementation is compared on two multiprocessor machines; (1) Sun HPC and (2) SGI Onyx2. To illustrate the performance of the parallel finite element solver, a Poisson's equation is solved for two and three dimensional model problems with Dirichlet and Neumann boundary conditions.

Objectives of this study are :

1. To implement efficient parallel programming techniques for FEM on distributed memory computers using domain decomposition and MPI.

2. Study the performance of Jacobi (diagonal) preconditioner with the conjugate gradient iterative method.

3. Analyze the performance of Element-By-Element (EBE) and Compressed Sparse Row (CSR) matrix storage schemes on different parallel computer architectures.

4. Study the performance of overlapping computation with communication using nonblocking message sending, MPI_ISend.

1.5 Outline

In Chapter 2, key issues in parallel computing are discussed, including different types of parallel architectures. In Chapter 3, the finite element formulation of elliptic second order partial differential equations are reviewed. The different matrix storage schemes are discussed and the algorithms for the conjugate gradient method and preconditioned conjugate gradient method are explained. In Chapter 4, the parallel version of the preconditioned conjugate gradient algorithm is explained along with the implementation details. Chapter 5 gives the results and discusses the performance of the implementation. Chapter 6 gives conclusions and suggestions for future work. In Appendix A a brief explanation of the code is presented and Appendix B gives the analytical form of the element stiffness matrices used in the implementation.

Chapter 2

INTRODUCTION TO PARALLEL DISTRIBUTED COMPUTING

A parallel computer, as defined by Almasi and Gottlieb(1989) [21], is *a collection of processing elements that cooperate and communicate to solve large problems fast.* Parallel computers can be viewed as a collection of processors and memory units which are connected by an interconnection network. The purpose of parallel computing is to get work done in less time and also to solve problems which cannot fit in a single computer's memory. The general idea behind the term parallel computing is to have more than one processor working on the same job at a particular instant of time with some kind of cooperation between them. Distributed computing refers to processing on computers that are separated from each other with interconnection through a local or wide area networks.

There are two main architectural paradigms associated with parallel computing: Distributed memory and Shared memory. They differ in the way the memory is accessed by each of the processors. Memory access refers to the way in which the working storage, be it "main-memory","cache-memory", or whatever is viewed by the programmer is accessed. Regardless of the way the memory is actually implemented, e.g., if it's actually remotely located but is accessed as if it were local, the access method plays a very large role in determining the conceptualization of the relationship of the program to its data.

In Shared memory architecture parallel computers the same memory is accessible by multiple processors. All processors associated with the program access the same

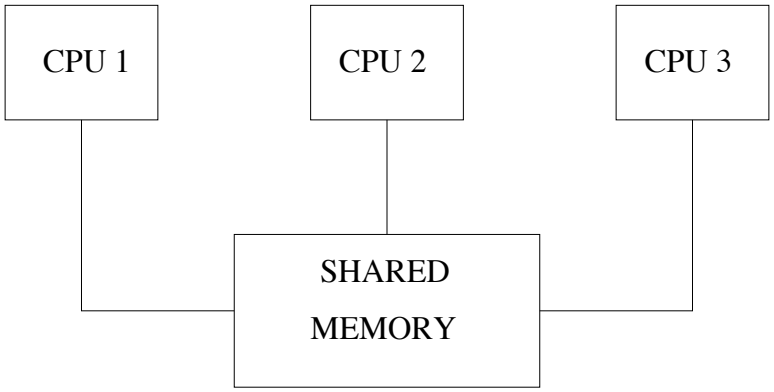


Figure 2.1: Shared Memory Architecture

storage as shown in Figure 2.1. In Distributed Memory architecture parallel computers, memory is physically distributed among processors, each local memory is directly accessible only by its processor. Synchronization is achieved by moving data between processors by a fast interconnection network, see Figure 2.2.

In this work, the focus is on the distributed memory model. The advantage of this kind of architecture is that it is scalable to a large number of commodity processors. A major concern with this scheme is data decomposition; the data being operated should be divided equally to balance the load, and also should be done in an efficient manner in order to minimize communication. The scalability depends on the type of interconnection between the processors. Some examples of Distributed Memory architectures include tightly coupled distributed memory parallel computers such as Cray T3E or the IBM-SP2, loosely coupled Beowulf clusters and also networks of individual workstations which are configured to cooperate in a concurrent manner. The distributed memory programming paradigm is an abstraction of a distributed memory computer. Such programs can also be executed on shared memory architectures, i.e., a distributed memory paradigm can be simulated on shared memory machines.

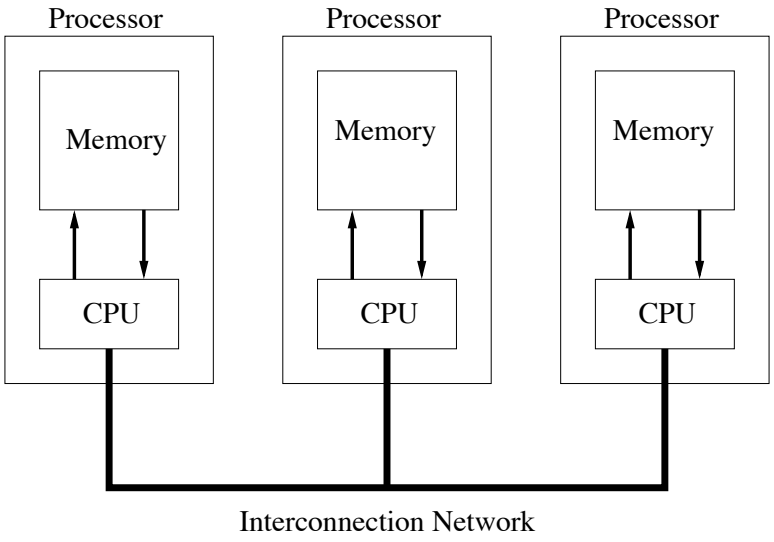


Figure 2.2: Distributed Memory Architecture

2.1 Message Passing Interface (MPI)

When working with a distributed memory computer it is necessary to ensure that each processor has its own copy of each data item that is required for each computation that is performed. Often, some or all of this data needs to be communicated to other processors. The way this is achieved is termed as message-passing. Message Passing Interface (MPI) [22], is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. MPI is a library of a set of standard subroutine calls which allow parallel programs to be written in distributed memory paradigm. The goal of the MPI is to provide a widely used standard for writing message-passing programs. The interface attempts to establish a practical, portable, efficient, and flexible standard for message passing. The MPI does not specify the implementation of the message passing, consequently the efficiency of the parallel program varies with different implementations of MPI.

MPICH [23] is a popular implementation of MPI, which is publicly available. It is being actively developed by the MCS division at Argonne National Laboratory. MPICH runs on a wide variety of architectures. On the two architectures we test our

code, Sun HPC runs on MPICH and SGI has a vendor supplied MPI. It was observed that communication time taken on the SGI machine (with vendor supplied MPI) was far less than on the Sun machine (using MPICH, which might not be optimized for the particular architecture).

2.2 Domain Decomposition

Domain Decomposition is a method of partitioning the domain into a number of subdomains in an efficient way in order to balance the load and minimize the communication (by keeping the coupling between the subdomains to a minimum). It is used frequently in parallel computing in order to assign the jobs to different processors. In the context of the finite element method, domain decomposition may be thought of as distributing the elements and nodes of the mesh to different processors. Domain decomposition is generally done with the aim to minimize communications time by distributing the elements in the domain in some optimal way. Communication time is governed by both latency and bandwidth by the equation,

$$\text{communication} = \text{latency} + \frac{\text{message size}}{\text{communication bandwidth}} \tag{2.1}$$

The number of nodes on the subdomains interface is minimized in order to minimize the bandwidth of communication. This approach is ideal for slow networks (lower communications bandwidth) or whenever many small messages are passed. It neglects the latency in the above expression. In this approach, the total length of inter-domain boundaries is minimized (which is length of all shared data). In other words, it minimizes the shared part of perimeter/surface area of the domain. Popular techniques to implement this approach need the eigenvalues of the connectivity matrix, and thus can be expensive for large problems. However, this approach produces the best decomposition for large problems. A publicly available graph partitioning

software called METIS [20], which is being developed by the Department of Computer Science at University of Minnesota, is available for this task. For structured meshes, domain decomposition can be done without much effort. For the unstructured meshes, the above mentioned domain method is essential. METIS is used for the implementation since the architecture we are aiming for is distributed memory with comparatively slow network communications.

Chapter 3

FINITE ELEMENT FORMULATION

In this Chapter the various issues that arise in the solution of a partial differential equation using the Finite Element Analysis (FEA) are discussed. The different types of matrix storage schemes which can be used to store the matrices arising out of the discretization are reviewed with particular attention to Element-By-Element (EBE) and sparse storage schemes (CSR), which are used in the implementation. The methods used for the solution of the linear system formed by the FEA are discussed with emphasis on the Conjugate Gradient method. A second order elliptic partial differential equation with the primary variable having only one degree of freedom (e.g. a heat-transfer problem) is considered to illustrate the formulation. This results in a linear system of equations, which is symmetric and positive-definite.

3.1 Elliptic Boundary Value Problem

Consider a second order linear elliptic partial differential equation in two dimensions, namely the Poisson equation, governing a single scalar valued function u in the domain Ω [24],

$$-\frac{\partial}{\partial x}\left(a\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a\frac{\partial u}{\partial y}\right)=f \qquad (x,y)\in\Omega \tag{3.1}$$

subject to the boundary conditions,

1. Dirichlet boundary condition :

$$u(x,y) = g(x,y) \qquad \text{for } (x,y) \; on \; \Gamma_g \tag{3.2}$$

2. Neumann boundary condition :

$$a\frac{\partial u}{\partial n} = h(x,y) \qquad \text{for } (x,y) \; on \; \Gamma_h \tag{3.3}$$

where $a(x,y)$ and $f(x,y)$ are given data inside Ω, $g(x,y)$ and $h(x,y)$ are given data on some parts of the boundary Γ.

This equation arises in a large number of applications including heat transfer, fluid mechanics, electrostatics and solid mechanics. For example, in the case of heat transfer, the interpretation of the variables arising in the above equations are as follows:

Primary variable	(Temperature)	$u = T$
Material constant	(Conductivity)	$a = k$
Source variable	(Heat source)	$f = q$
Secondary variables	(Heat flow)	$-a\nabla T = \underline{Q}$
Boundary data	$g(x,y) = T$,	specified temperature on the boundary
Boundary data	$h(x,y) = Q_n$,	heat flow entering the system

The governing differential equation is multiplied by a weighting function $w(x,y)$, such that $w = 0$ on Γ_g, and is integrated over the problem domain in order to get the weak form, which forms the basis for the standard finite-element method of approximation.

$$\int_\Omega w\frac{\partial}{\partial x}\left(a\frac{\partial u}{\partial x}\right)dxdy + \int_\Omega w\frac{\partial}{\partial y}\left(a\frac{\partial u}{\partial y}\right)dxdy = \int_\Omega wf\,dxdy \tag{3.4}$$

Using Green's theorem and integrating by parts,

$$\int_\Omega \left(\frac{\partial w}{\partial x} a \frac{\partial u}{\partial x} + \frac{\partial w}{\partial y} a \frac{\partial u}{\partial y} \right) dxdy = \int_\Omega w f dxdy + \int_\Gamma w a \frac{\partial u}{\partial n} d\Gamma \tag{3.5}$$

Implementing the boundary conditions and simplifying, the weak form reduces to;

$$\int_\Omega \left(\frac{\partial w}{\partial x} a \frac{\partial u}{\partial x} + \frac{\partial w}{\partial y} a \frac{\partial u}{\partial y} \right) dxdy = \int_\Omega w f dxdy + \int_{\Gamma_h} w h d\Gamma \tag{3.6}$$

The Galerkin finite element formulation is obtained by introducing a finite element approximation for the trial solution and weighting function; $u^h \approx u$; $w^h \approx w$. The approximation of u defined over a typical element Ω^e is expressed in terms of interpolation of nodal data,

$$u^h(x,y) = \sum_{i=1}^{n_{en}} N_i^e(x,y) d_i^e, \qquad (x,y) \in \Omega^e \tag{3.7}$$

where n_{en} is the number of nodes in an element and $d_i^e = u^h(x_i, y_i)$ is the value of u^h at the ith node (x_i, y_i) of the element, and N_i^e are interpolation (shape) functions, with the property

$$N_i^e(x_j, y_j) = \delta_{ij} \tag{3.8}$$

δ_{ij}, is Kronecker delta, with a value of 1 if $i = j$ and 0 if $i \neq j$. Similarly, for the weighting function,

$$w^h(x,y) = \sum_{i=1}^{n_{en}} N_i^e(x,y) w_i^e \qquad (x,y) \in \Omega^e \tag{3.9}$$

where $w_i^e = w^h(x_i, y_i)$.

Discretizing the domain Ω into n finite elements Ω^e and substituting the values of the finite element approximations to the weak form gives,

$$\sum_{e=1}^{n} w^{eT} k^e d^e = \sum_{e=1}^{n} w^{eT} f_f^e + \sum_{e=1}^{n} w^{eT} f_h^e \tag{3.10}$$

where,

$$k^e = \int_{\Omega^e} \left[\frac{\partial N^{eT}}{\partial x} a \frac{\partial N^e}{\partial x} + \frac{\partial N^{eT}}{\partial y} a \frac{\partial N^e}{\partial y} \right] dxdy \tag{3.11}$$

$$f_f^e = \int_{\Omega^e} N^{eT} f(x,y) dxdy \tag{3.12}$$

$$f_h^e = \int_{\Gamma_h^e} N^{eT} h(x,y) d\Gamma \tag{3.13}$$

In the above, N^e is a vector of shape functions for element e, k^e is the element stiffness matrix, and f_f^e is the element force vector due to the known source $f(x,y)$ and f_h^e is the element force vector due to the known boundary flux $h(x,y)$.

The local element degrees of freedom (dof) d^e are related to the global dof $\underline{d}$ through a local destination array L, using an assembly procedure, global arrays are formed,

$$w^T(A\underline{d}) = w^T(F_f + F_h) \tag{3.14}$$

The equality holds for any arbitrary weighting function w, so we get the linear system of equations

$$A\underline{d} = \underline{b} \tag{3.15}$$

where, $\underline{b} = F_f + F_h$, is the force vector, A is the system matrix and $\underline{d}$ is the vector of solution unknowns. The global matrix A and the vectors F_f and F_h are assembled using the local destination array L. The linear system is solved to obtain the solution

vector. In this case A is symmetric positive definite. A real square matrix A is said to be symmetric positive definite if it is symmetrical about its diagonal and satisfies

$$(A\underline{u}, \underline{u}) > 0, \qquad \forall \underline{u} \in \Re^n, \ \underline{u} \neq 0. \tag{3.16}$$

where, the notation $(\underline{u}, \underline{v})$ indicates the inner product $\underline{u}^T \underline{v}$.

It follows that the central steps in the finite element computations consist of the element stiffness matrix calculations, assembly to the algebraic system, implementation of the boundary conditions and sparse system solution. Thus, in a typical finite element analysis, the element matrix and vector contributions k_e, f_f^e, f_h^e are first computed from (3.11) - (3.13), for $e = 1, 2, \ldots, n$ elements. These are then accumulated to the global system (3.15), essential boundary conditions are imposed and then the system is solved using a suitable sparse solver. A constant $a(x, y)$ in the domain Ω and linear shape function for the elements (triangular, tetrahedral and rectangular elements) results in an analytical form of the element stiffness matrix k^e given in Appendix B.

Since the element contributions k_e, f_f^e, f_h^e are calculated independently in the main element loop, this phase of the analysis can easily be made in parallel. Moreover, from (3.10) we see that the accumulation process is simply additive over the elements of the discretization.

3.2 Matrix Storage Schemes

The discretization of a partial differential equation by the finite element method typically lead to large sparse matrices. A sparse matrix is defined as a matrix with very few non-zero entries [1]. One of the key issues of solving the finite element equations is to define data structures for the sparse matrices that are well suited for its efficient implementation.

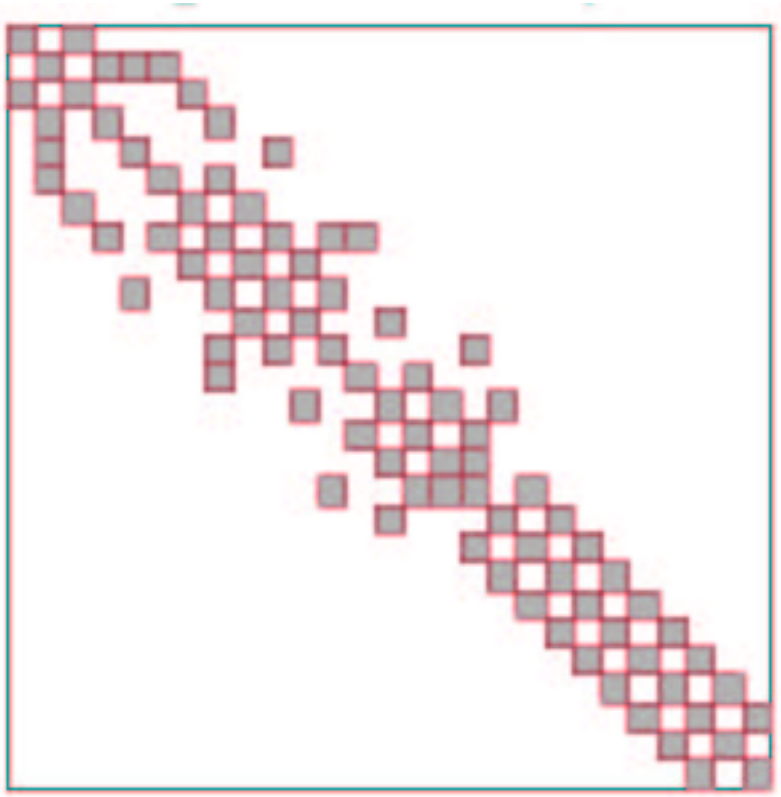


Figure 3.1: Compressed Sparse Row storage format.

Traditionally, finite element programs were based on the assembly techniques described in the previous section. The linear algebraic equations (3.15) were solved by some form of Gaussian elimination. The Gaussian elimination process modifies the matrix by "fill-in" (replacing some zero coefficients with non-zero coefficients). All the coefficients including zeros contained within a "band" or "skyline" must be stored and manipulated. Iterative techniques don't change the matrix during the solution phase. They only require a matrix-by-vector product of the coefficient matrix. Hence there is more flexibility in choosing the data structure of the stiffness matrix. The only restriction on the data structure is that it should provide an efficient matrix-by-vector product. The following subsections give a overview of Element-By-Element and Compressed Sparse Row (CSR) types of matrix storage schemes.

3.2.1 Compressed Sparse Row (CSR)

Compressed sparse row format is a popular sparse matrix storage scheme which stores only the nonzeros row-by-row. This is illustrated in the Figure 3.1. The following example illustrates the CSR format [1].

$$A = \begin{bmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{bmatrix} \tag{3.17}$$

$$\begin{aligned} \text{AA} &= [\ 1.\ 2.\ 3.\ 4.\ 5.\ 6.\ 7.\ 8.\ 9.\ 10.\ 11.\ 12.\ ] \\ \text{JA} &= [\ 1\ \ 4\ \ 1\ \ 2\ \ 4\ \ 1\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\ ] \\ \text{IA} &= [\ 1\ \ 3\ \ 6\ \ 10\ \ 12\ \ 13\ ] \end{aligned}$$

The data structure consists of three arrays :

1. A real array AA, containing all the real (or complex) values a_{ij} of the non zero elements of A from row 1 to n, where n is the size of the matrix. The size of AA is Nz, (the number of non-zero coefficients in a matrix).

2. An integer array JA, containing the column indices of the coefficients a_{ij} as stored in the array AA. The length of JA is Nz.

3. An integer array IA, containing the pointers to the beginning of each row in the arrays AA and JA. The size of IA is $n + 1$ with IA$(n + 1)$ containing the number IA$(1) + Nz$,

The CSR storage format requires Nz floating points and $Nz+n$ integers. Which is considerably less when compared with band or skyline storage formats. The Fortran90 pseudo-code for matrix-by-vector product of a matrix stored in CSR format is given

```
Function MatmulCSR (AA, IA, JA, x, y)
        y = 0
        Do i = 1, n
                p1 = IA(i)
                p2 = IA(i + 1) − 1
                Do j = p1, p2
                    y(i) = y(i) + AA(j) ∗ x(JA(j))
                End Do
        End Do
End Function MatmulCSR
```

Figure 3.2: Algorithm for Matrix-by-vector product for matrix stored in CSR format [1].

in Figure 3.2. In Fortran90, the inner loop may be replaced by a single line using Matlab like array indexing:

$$y(i) = Dot\_Product(AA(ptr1 : ptr2), x(JA(ptr1 : ptr2))) \tag{3.18}$$

However, our numerical experience showed that the indirect indexing implied in 3.18 was significantly slower than the explicit Dot Product implementation in Figure 3.2.

Figure 3.3 gives the algorithm for assembling the CSR matrix [2]. Loop over the elements in the connectivity array to determine the elements associated with each node. Knowing which elements are associated to a global node, allows us to determine which other nodes are connected to this node. This information is equivalent to the column numbers of the non-zeros entries in the row corresponding to that node. With this information IA and JA of the CSR matrix are determined. Having the IA, JA and the element stiffness matrices, the AA array is then assembled.

3.2.2 Element-By-Element (EBE)

In EBE, only the element contributions, k_e of each element $e = 1, 2, \ldots, E$, are stored. The global stiffness matrix is never assembled. The storage format is illustrated in

1. Form array $elem(n, e)$ containing the elements associated with each global node n.
2. For each node, from the *elem* array and the connectivity array get the other global nodes coupled to it.
3. Assemble IA and JA.
4. Using IA, JA, and the element stiffness matrix arrays, assemble AA.

Figure 3.3: Algorithm for assembling the CSR matrix [2]

.

the Figure 3.4. Whenever the matrix-vector product is needed, the connectivity matrix L, is used along with the element stiffness matrices to obtain the product. Element-by-element methods are motivated by the characteristic of finite element implementations on distributed memory architectures in which the data structure can be stored and operated at a local(element) basis. The primary purpose of the EBE applications was to keep memory requirements to a minimum when compared to band and skyline storage, but the advent of parallel distributed memory computers has given this approach a new interest, especially for large-scale 3-D problems. The storage requirements for EBE grow linearly with the increase in number of elements. The EBE stores more data when compared with CSR and also EBE requires more floating point operations for every matrix-by-vector product when compared to CSR. However EBE is simple to implement and uses less indirect memory addressing than CSR. The other advantage of EBE is that element stiffness matrices can be calculated as and when required without storing them, in this way the implementation doesn't require any storage of stiffness matrix, however this type of implementation might be very slow.

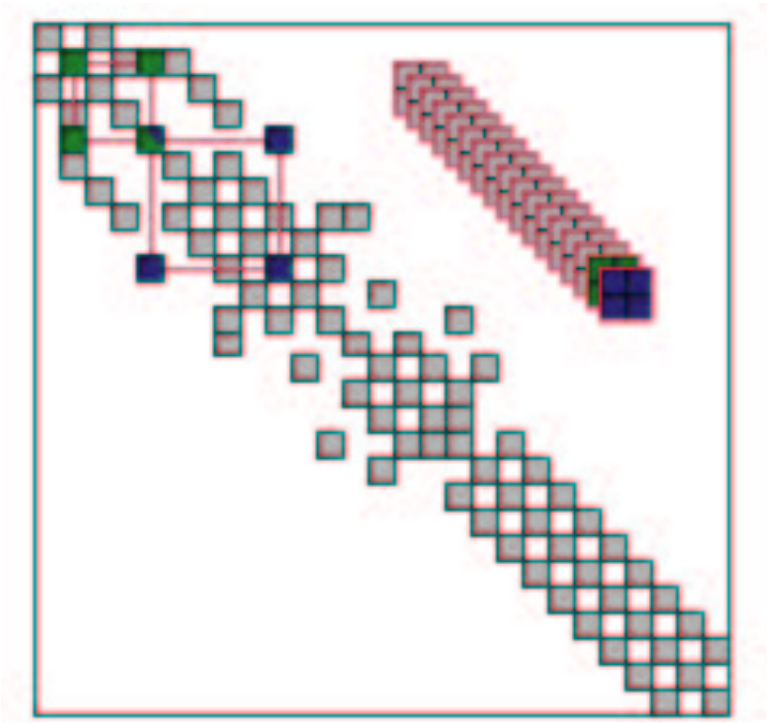


Figure 3.4: Element-By-Element storage format

3.3 Sparse Matrix Solvers

A finite element code typically spends the greatest amount of time in solving the linear system of equations. The two basic types of solution methods are direct methods and iterative methods. Direct methods are based on Gaussian elimination. These methods compute the solution after a finite number of arithmetic operations. Iterative methods generate a sequence of approximate solutions that (hopefully) will converge to the true solution. For positive definite systems, which arise out of the Poisson's equation, the Conjugate Gradient iterative method may be used.

3.3.1 Direct Methods

Sparse direct solvers can provide within numerical errors an exact solution to the stiffness matrix arising out of finite element method in a fixed number of operations. These methods compute reliable solutions to any nonsingular matrix. The disadvan-

tage though is that the operation count and memory size usually increase rapidly with the problem size. Parallel implementations of sparse direct solvers usually employ thread code for execution on a shared memory.

Most direct methods for sparse linear systems perform an LU factorization of the matrix and try to reduce cost by minimizing the fill-ins (i.e., nonzero elements introduced during the elimination process in positions which were initially zeros). The data structures employed are rather complicated. The early codes relied heavily on linked lists which are convenient for inserting new nonzero elements. Linked-list data structures were dropped in favor of other more dynamic schemes that leave some initial "elbow" room in each row for insertion and then adjust the structure as more fill-ins are introduced.

A typical sparse direct solution solver for positive definite matrices consists of four phases[1]. First, preordering is applied to the matrix to minimize fill-in. Two popular methods are used: minimal degree ordering and nested-dissection ordering. Second, a symbolic factorization is performed. This means that the factorization is processed only symbolically. Finally, the forward and backward triangular sweeps are executed for each different right-hand side.

3.3.2 Iterative Methods

Iterative methods refer to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system at each step. The convergence of the iterative method is dependent on the condition number of the matrix. The condition number of a matrix is the ratio of the largest eigenvalue to the smallest eigenvalue of the matrix. For ill-conditioned problems, with large condition numbers, the iterative methods might not converge. Preconditioning is usually employed in order to improve the conditioning of the matrix solved. The Conjugate

Gradient iterative method, which is extremely effective for symmetric positive definite systems, is reviewed in the next section.

3.4 Conjugate Gradient Algorithm

A large number of iterative methods for solving linear systems can be thought of as minimization methods [3]. Consider a quadratic function,

$$\phi(\underline{x}) = \frac{1}{2}\underline{x}^T A\underline{x} - \underline{x}^T\underline{b} + c \tag{3.19}$$

where $\underline{b} \in \Re^n$ and $A \in \Re^{n\times n}$ is symmetric positive definite. The minimum value of ϕ is achieved by setting $\underline{x} = A^{-1}\underline{b}$. So solving $Ax = b$ is equivalent to minimizing the function ϕ.

Many minimization methods have a search direction $\underline{p}_k$ and proceed with,

$$\underline{x}_{k+1} = \underline{x}_k - \alpha_k\underline{p}_k \qquad k = 0, 1, \ldots. \tag{3.20}$$

where α_k is chosen so as to minimize ϕ along $x_k - \alpha p_k$, so that

$$\phi(\underline{x}_k - \alpha_k\underline{p}_k) = \min_{\alpha>0}\phi(\underline{x}_k - \alpha\underline{p}_k) \tag{3.21}$$

For fixed $\underline{x}_k$ and $\underline{p}_k$, $\phi(\underline{x}_k - \alpha_k\underline{p}_k)$ is a quadratic function of α and may be minimized explicitly to give

$$\alpha_k = -(\underline{p}_k, \underline{r}_k)/(\underline{p}_k, A\underline{p}_k) \tag{3.22}$$

where $\underline{r}_k = \underline{b} - A\underline{x}_k$. For α_k, given by (3.22) $\underline{p}_k$ can be chosen in many ways. At the current position $\underline{x}_c$ if the direction vector $\underline{p}_k$ is equal to the negative gradient of

ϕ, i.e.,

$$\underline{p}_k = -\nabla\phi(\underline{x}_c) = \underline{b} - A\underline{x}_c \tag{3.23}$$

then this method is called the steepest descent method. The disadvantage of this method is that the speed of convergence may be prohibitively slow if the condition number of the matrix A is large. In this case, the level curves of ϕ are very elongated hyper ellipsoids and minimization corresponds to finding the lowest point on a relatively flat, steep-sided valley. In the steepest descent method, we are forced to traverse back and forth across the valley rather than down the valley. The gradient directions $\underline{p}_k$ that arise during the iterations are too similar thus slowing progress towards the minimum value.

If the n non-zero direction vectors $p_0, \ldots, p_{n-1}$, are chosen such that

$$\underline{p}_i^T A\underline{p}_j = 0, \qquad i \neq j. \tag{3.24}$$

Then the vectors are orthogonal to the inner product$(\underline{x}, \underline{y})_A \equiv \underline{x}^T A\underline{y}$ defined by A, and are thus called $A-$*orthogonal*; they are also called *conjugate* with respect to A. In this case it can be shown that, [3], for any initial value $\underline{x}_0$ the iterates (3.20), where α_k is chosen by (3.22) converge to the solution of $A\underline{x} = b$ in no more that n steps, where n is the size of the matrix A. This method is termed as the conjugate directions method.

The Conjugate Gradient Method is an efficient implementation of the conjugate directions method in which the search directions are constructed by conjugation of the residuals. The algorithm is given in the Figure 3.5. In the description of the algorithm, k defines the iteration count, $\underline{r}$ specifies the linear residual, $\underline{x}$ represents the solution vector, $\underline{p}$ defines the step direction, α specifies the step length, and β denotes the correction factor. In the implementation the L2 error norm of the residual,

1. Choose $\underline{x}^0$, initial guess
2. $\underline{r}^0 = \underline{b} - A\underline{x}^0$
3. $\underline{p}^0 = \underline{r}^0$
4. $\gamma^0 =$ Inner Product $(\underline{r}^0, \underline{r}^0)$

   Do $k = 0, 1, 2, \ldots k_{max}$

5. $\quad \underline{q}^k =$ Matrix Multiply $(A, \underline{p}^k)$
6. $\quad \tau^k =$ Inner Product $(\underline{p}^k, \underline{q}^k)$
7. $\quad \alpha^k = \gamma^k / \tau^k$
8. $\quad \underline{x}^{k+1} = \underline{x}^k + \alpha^k \underline{p}^k$
9. $\quad \underline{r}^{k+1} = \underline{r}^k - \alpha^k \underline{q}^k$
10. $\quad \gamma^{k+1} =$ Inner Product $(\underline{r}^{k+1}, \underline{r}^{k+1})$
11. $\quad$ If $(\sqrt{\gamma^{k+1}} \leq$ Tolerance) Exit
12. $\quad \beta^k = \gamma^{k+1} / \gamma^k$
13. $\quad \underline{p}^{k+1} = \underline{r}^{k+1} + \beta^k \underline{p}^k$

   End Do

Figure 3.5: Conjugate Gradient Algorithm [3],[4]

γ is used for the convergence check. The advantage of this is that it requires no extra work since γ is needed on the next step if the convergence has not occurred.

From the algorithm, it can be seen that every iteration of the Conjugate Gradient method requires one matrix-vector product, two inner products and three vector updates (vector = vector + scalar $\times$ vector). The matrix-vector product is the most costly operation in the routine. The result of the matrix-vector product can be computed independent of the matrix storage structure used. Thus it is an advantage for sparse matrices, which arise in FEM, because we don't need to store the large number of zeros.

3.5 Preconditioning

The rate of convergence of an iterative method depends greatly on the spectrum of the coefficient matrix. In finite element analysis the condition number of A grows like $O(h^{-2})$, as $h \longrightarrow 0$, where h is the element size of the mesh [25]. Hence, iterative methods usually involve a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called a preconditioner. A good preconditioner improves the convergence sufficiently to overcome the extra cost of constructing and applying the preconditioner.

The rate of convergence of matrix A can be increased by preconditioning A with the congruence transformation [3],

$$S(A\underline{x} = \underline{b}), \tag{3.25}$$

where S is a nonsingular preconditioning matrix

$$SA\underline{x} = S\underline{b} \tag{3.26}$$

$$SAS^T S^{-T}\underline{x} = S\underline{b} \tag{3.27}$$

The system to be solved is then in principle,

$$\hat{A}\hat{x} = \hat{b}, \tag{3.28}$$

where $\hat{A} = SAS^T$, $\hat{x} = S^{-T}x$ and $\hat{b} = Sb$. S is chosen so that cond$(\hat{A}) <$ cond (A).

The conjugate gradient algorithm of Figure 3.5 is applied to system 3.28 above to obtain iterates $\hat{x}_k$. By simplifying and defining $x_k = S^T\hat{x}_k$, the Conjugate Gradient Algorithm shown in Figure 3.6 is obtained [26], where matrix M is defined by,

$$M = (S^T S)^{-1}. \tag{3.29}$$

If M is the Identity matrix , the preconditioned algorithm in Figure 3.6 reduces to the regular conjugate gradient algorithm. In general, we do not define S and then form M. Rather we choose M directly, and S is never explicitly used. M should be symmetric positive definite since $S^T S$ is symmetric positive definite. The preconditioned matrix $\hat{A}$ should have a smaller condition number than A. The aim is to reduce the condition number of $\hat{A}$ as much as possible while at the same time making the auxiliary equation $M\underline{z} = \underline{r}$, relatively easy to solve. Clearly, the two requirements are in conflict with each other, so we need to strike a balance in choosing M.

3.5.1 Diagonal Preconditioner

If we choose $M = D$, the main diagonal of A, then this is called "diagonal scaling" or Jacobi preconditioning. Diagonal preconditioning is easy to assemble and invert for serial as well as parallel computers and performs well for diagonally dominant systems. Other preconditioners such as, Incomplete LU factorization (ILU) and Symmetric Successive Over Relaxation (SSOR), give better performance, but are difficult to implement on parallel computers. In this work we use diagonal scaling for the preconditioner.

1. Choose $\underline{x}^0$, initial guess
2. $\underline{r}^0 = \underline{b} - A\underline{x}^0$
3. Solve $M\underline{z}^0 = \underline{r}^0$
4. $\underline{p}^0 = \underline{z}^0$
5. $\gamma^0 =$ Inner Product $(\underline{z}^0, \underline{r}^0)$

   Do $k = 0, 1, 2, \ldots k_{max}$

6. $\quad \underline{q}^k =$ Matrix Multiply $(A, \underline{p}^k)$
7. $\quad \tau^k =$ Inner Product $(\underline{p}^k, \underline{q}^k)$
8. $\quad \alpha^k = \gamma^k / \tau^k$
9. $\quad \underline{x}^{k+1} = \underline{x}^k + \alpha^k \underline{p}^k$
10. $\quad \underline{r}^{k+1} = \underline{r}^k - \alpha^k \underline{q}^k$
11. $\quad$ Solve $M\underline{z}^{k+1} = \underline{r}^{k+1}$
12. $\quad \gamma^{k+1} =$ Inner Product $(\underline{z}^{k+1}, \underline{r}^{k+1})$
13. $\quad$ If $(\sqrt{\gamma^{k+1}} \leq$ Tolerance) Exit
14. $\quad \beta^k = \gamma^{k+1} / \gamma^k$
15. $\quad \underline{p}^{k+1} = \underline{z}^{k+1} + \beta^k \underline{p}^k$

   End Do

Figure 3.6: Preconditioned Conjugate Gradient Algorithm [3],[4].

Chapter 4

PARALLEL IMPLEMENTATION

In distributed parallel finite element analysis, the domain considered is first decomposed into a number of subdomains. Then in general, one processor is assigned for each subdomain, though other variations are also possible. Computations are then performed on each processor on the local subdomain and communication takes place with the other processors whenever needed. The domain might be divided into a set of overlapping subdomains in which case Overlapping Schwartz methods are used for the solution of the system [27]. If the domain is partitioned into a set of non-overlapping subdomains then the methods used are called iterative substructuring methods. The two basic types of non-overlapping domain decomposition methods used in the finite element method are the subdomain-by-subdomain (SBS) and the Schur complement method [17, 18]. The first approach is based on multi-element group partitioning of the entire domain. The global stiffness matrix is stored as a partitioned matrix and the dominant matrix-vector operations of the stiffness matrix and the preconditioner solutions in the iterative algorithm are performed on the SBS basis. In the second approach the iterative algorithm is applied to the interface problem after first eliminating the internal degrees of freedom.

The matrix-by-vector products, preconditioner solutions, dot-products and vector updates are the basic building blocks of all the iterative methods. The matrix-by-vector product is the most computationally expensive operation among all the others. Preconditioner solutions might overtake matrix-by-vector product in some cases. In the parallel version, it is desirable to distribute data in an efficient manner in order to reduce the time taken by these operations. Ideally, we need to make sure that

each processor gets an equal amount of work (this could be achieved by distributing an equal number of elements to each processor) and the number of interface nodes between the subdomains should be kept to a minimum in order to reduce communication between the processors. In our implementation, the domain decomposition is achieved using METIS [20].

In this chapter, the implementation of the SBS method is discussed. Each subdomain is assigned to a separate processor. The conjugate gradient method is used for the solution of the distributed matrices. The implementation of the diagonal preconditioner is discussed. The distributed matrices are stored in Compressed Sparse Row (CSR) format and Element-By-Element (EBE) format, the algorithm for the matrix-by-vector product in parallel for these types of formats is also given.

4.1 Domain Decomposition

Consider a finite element domain Ω partitioned into s nonoverlapping subdomains (substructures), $(\Omega_j, j = 1, \ldots, s)$.

$$\Omega = \bigcup_{j=1}^{s} \Omega_s,$$

The mesh nodes which are common to the subdomain interfaces define a global interface noted by N_I. Numbering first the nodal point unknowns within Ω_j not belonging to N_I and last the interface nodes N_I, results in an arrow pattern for the global stiffness matrix. Thus the equations of equilibrium $A\underline{x} = \underline{b}$ for a linear system in a block

arrowhead matrix structure are given by,

$$\begin{bmatrix} A_1 & & & & C_1 \\ & A_2 & & & C_2 \\ & & \ddots & & \vdots \\ & & & A_s & C_s \\ C_1^T & C_2^T & \cdots & C_s^T & A_I \end{bmatrix} \begin{Bmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \vdots \\ \underline{x}_s \\ \underline{x}_I \end{Bmatrix} = \begin{Bmatrix} \underline{b}_1 \\ \underline{b}_2 \\ \vdots \\ \underline{b}_s \\ \underline{b}_I \end{Bmatrix} \tag{4.1}$$

with

$$A_I = \mathop{\mathbf{A}}_{j=1}^{s} A_{I_j} \qquad \text{and} \qquad b_I = \mathop{\mathbf{A}}_{j=1}^{s} b_{I_j} \tag{4.2}$$

where $\mathop{\mathbf{A}}_{j=1}^{s}$ sign in the equation (4.2) represents assembly of equations or vectors for subdomain $j = 1, \ldots, s$. In the above, each $\underline{x}_j, j = 1, \ldots, s$ represents the subvector of unknowns that are interior to subdomain Ω_j, and $\underline{x}_I$ represents the vector of all interface unknowns. $A_1, \ldots, A_s$ are square matrices associated with internal unknowns for each subdomain with dimension $q_j \times q_j$, where q_j is the number of unknowns in Ω_j, and A_I is a square matrix with dimension $N_I \times N_I$. The total number of interface unknowns for the linear system, $\underline{x}_I$ is given by, $\underline{x}_I = \mathop{\mathbf{A}}_{j=1}^{s} \underline{x}_{I_j}$, where x_{I_j} is the total number of unknowns on the interfaces in subdomain Ω_j. The C_j are $q_j \times N_I$ matrices representing the subdomain to global interface coupling seen from the subdomains, while C_j^T are $N_I \times q_j$ matrices representing the global interface to subdomain coupling seen from the interface nodes. Each of these matrices has been assembled from finite element matrices. The total number of equations is equal to $n = \sum_{j=1}^{s} q_j + N_I$. In general, the number of interface unknowns N_I will be considerably less than the number of internal unknowns $q = \sum_{j=1}^{s} q_j$, i.e., $N_I \ll q$. In (4.1) the block arrowhead structure of the matrix A stems from the local support of the finite element basis functions.

Each of the blocks A_j, C_j (and therefore C_j^T) and $\underline{b}_j$ may be computed entirely by processor j (which works only with submesh Ω_j, for $j = 1, \ldots, s$). The blocks A_I and $\underline{b}_I$ both have contributions from elements in all of the subdomains, however it is possible for each processor to assemble the contributions to each of these only from those elements on its subdomain. On processor j these contributions are referred to as A_{I_j} and $\underline{b}_{I_j}$, respectively.

So, for each processor j which operates on its own local subdomain we have:

$$\begin{bmatrix} A_j & B_j \\ B_j^T & A_{I_j} \end{bmatrix} \begin{Bmatrix} \underline{x}_j \\ \underline{x}_{I_j} \end{Bmatrix} = \begin{Bmatrix} \underline{b}_j \\ \underline{b}_{I_j} \end{Bmatrix} \tag{4.3}$$

where,

A_j - Assembled local interior node matrix

A_{I_j} - Local interface nodes matrix

B_j - Coupling of the local interior nodes to local interface nodes

B_j^T - Coupling of the local interface nodes to local interior nodes

$\underline{x}_j$ - Local interior solution subvector

$\underline{x}_{I_j}$ - Local interface solution subvector

$\underline{b}_j$ - Local interior force subvector

$\underline{b}_{I_j}$ - Local interface force subvector

The coupling matrix of local interior nodes to global interface nodes is given by $B_j = PC_j$, where P is a permutation matrix. B_j doesn't include the contributions of the interface nodes which are not in subdomain j, since the contribution of the interface nodes which are not on the Ω_j to the interior nodes of Ω_j are zeros. In a similar fashion, $B_j^T = PC_j^T$. Every processor will concurrently assemble each of the blocks $A_j, B_j, B_j^T, \underline{b}_j, A_{I_j}$ and $\underline{b}_{I_i}$, thus the system (4.1) is stored in a distributed manner.

Summarizing, domain decomposition of both element and nodal data is used. Processors own all relevant data for elements in their domain and also the data for internal nodes. Vectors such as the force vector, solution vector or trial vector, are duplicated between the processors for nodes on the interfaces. For example, some of the data in $\underline{x}_{I_j}$ is duplicated between shared nodes between subdomains. Along with the above data every processor has the following information about its neighbors .

Shared(i) = number of processors which share the node on the subdomain boundary which has the local number i.

Common(k) = number on nodes shared between this processor and processor k (this is set to zero when k = the rank of this process).

Neighbor(j, i) = local number of the node on the partition boundary which is the i^{th} node shared with processor j.

The array of the nodes in Neighbor which contain information of the nodes shared between two processors are kept in the same order in both the processors. Once every processor has the above information the system is ready to be solved. In the coming section, Matrix-by-vector product, Inner Product and Preconditioner solutions which are used in Conjugate Gradient method are explained with reference to this distributed matrix.

Matrix-by-Vector Product

Consider a global vector $\underline{p}$ (stored in distributed manner across the processors) which needs to be multiplied by the global matrix A to give the product $\underline{q}$. This can be done in parallel in the following manner:

$$\begin{Bmatrix} \underline{q}_j \\ \underline{q}_{I_j} \end{Bmatrix} = \begin{bmatrix} A_j & B_j \\ B_j^T & A_{I_j} \end{bmatrix} \begin{Bmatrix} \underline{p}_j \\ \underline{p}_{I_j} \end{Bmatrix} \qquad j = 1, 2, \ldots, np \tag{4.4}$$

```
Subroutine Update (a)

——— Send ———————
      Do j = 1, np
         If Common(j) > 0
            Do i = 1, Common(i)
               Buf(i) = a(Neighbor(j, i))
            End do
            MPI_Send (Buf, Common(j), j)
         End if
      End do
——— Receive ———————
      Do j = 1, np
         If Common(j) > 0
            MPI_Receive (Buf, Common(j), j)
            Do i = 1, Common(i)
               a(Neighbor(j, i)) = a(Neighbor(j, i)) + Buf(i)
            End Do
         End If
      End do

End Subroutine Update
```

Figure 4.1: Algorithm for the subroutine Update

Algorithm for generic matrix-by-vector product in parallel :

For each processor j:

1. $\underline{q}_j = A_j \underline{p}_j + B_j \underline{p}_{I_j}$
2. $\underline{q}_{I_j} = B_j^T \underline{p}_j + A_{I_j} \underline{p}_{I_j}$
3. Update $(\underline{q}_{I_j})$

The above algorithm gives the steps involved for matrix-by-vector product in parallel, the particular implementations for EBE and CSR will be given in the coming sections. The first step calculates $\underline{q}_j$, which doesn't require communication. The second step calculates the contribution of the processor to $\underline{q}_I$, which correspond to the interface nodes. The update routine is given in Figure 4.1. The variable Buf in the algorithm refers to a temporary buffer and np is the number of processors. The function of this routine is to update the vector of interface nodes by sending in the nodal contributions of the local processor to the processors which share the node, and at the same time receive the contributions of those processors and add them to the corresponding values. The routine makes use of point to point communication from each processor to each of its neighbors. Two processors are called neighbors if their corresponding meshes share at least one common node. At the end of the matrix-by-vector product routine each processor will have a copy of its part of vector $\underline{q}$.

Matrix-by-Vector Product for CSR

The parallel matrix-by-vector product for the CSR format is given in the Figure 4.2. In this case all of the associated matrices are stored in CSR format. The algorithm is similar to the generic case, the "sends" and the "receives" are performed as in Update

routine. The algorithm in Figure 3.2 is used to multiply individual matrices with the corresponding vectors.

Overlapping Computation and Communication

In the above matrix-by-vector product the communication and computation can be overlapped by using "nonblocking send" [28]. In this way there might be some reduction in total solution time for architectures with high latency. This algorithm is illustrated in the Figure 4.3. The interface matrix-by-vector product is computed first and then its communication is initiated using non-blocking MPI_ISend. Then the interior part of the matrix-by-vector product is computed for which no communication is necessary. In the end the the contributions of the Neighbors for the interface nodes are received and updated.

Matrix-by-Vector Product for EBE

Figure 4.4 shows the algorithm for multiplying the matrix stored in EBE format by a vector. In EBE the matrix-by-vector product is obtained by looping over the elements and assembling the contributions to the value of the vector corresponding to the nodes of the elements. Since the information of the interface nodes is not available before hand, overlapping communication and computation is not possible. If the code has the information on the elements lying on the interface, then the interface part of the vector can be assembled first and the computation can be overlapped with communication.

Inner Product

Given two global vectors $\underline{a}$ and $\underline{b}$ (stored in distributed manner across the processors), the Inner product can be obtained by the Function shown in Figure 4.5. The func-

Subroutine CSRmatmul $(Acsr, \underline{p}_j, \underline{p}_{I_j}, \underline{q}_j, \underline{q}_{I_j})$
$\underline{q}_j = MatmulCSR(A_j, \underline{p}_j) + MatmulCSR(B_j, \underline{p}_{I_j})$
$\underline{q}_{I_j} = MatmulCSR(B_j^T, \underline{p}_j) + MatmulCSR(A_{I_j}, \underline{p}_{I_j})$
Send to the Neighbors the corresponding values of $\underline{q}_{I_j}$ using MPI_Send
Receive from Neighbors the corresponding values of $\underline{q}_{I_j}$ using MPI_Recv
Add the contributions of the Neighbors to $\underline{q}_{I_j}$
End Subroutine CSRmatmul

Figure 4.2: Matrix-by-vector product in parallel for a matrix stored in CSR format

Subroutine CSRMatmul
$\underline{q}_{I_j} = MatmulCSR(B_j^T \underline{p}_j) + MatmulCSR(A_{I_j} \underline{p}_{I_j})$
Send to the Neighbors the corresponding values of $\underline{q}_{I_j}$ using MPI_ISend
$\underline{q}_j = MatmulCSR(A_j \underline{p}_j) + MatmulCSR(B_j \underline{p}_{I_j})$
Receive from Neighbors the corresponding values of $\underline{q}_{I_j}$ using MPI_Recv
Add the contributions of the Neighbors to $\underline{q}_{I_j}$
Subroutine CSRMatmul

Figure 4.3: Algorithm for overlapping communication and computation

Subroutine EBEmatmul $(Ke, \underline{p}_j, \underline{p}_{I_j}, \underline{q}_j, \underline{q}_{I_j})$
Loop over elements $e = 1, el$
Loop over the nodes in the element, $i = 1, nen$
If the node is interior node assemble the contributions to $\underline{q}_j$
Else if the node is interface node assemble the contributions to $\underline{q}_{I_j}$
End do
End do
Update$(\underline{q}_{I_j})$
End Subroutine EBEmatmul

Figure 4.4: Matrix-by-vector product in parallel for a matrix stored in EBE format

```
Function InnerProduct (a_j, a_{I_j}; b_j, b_{I_j})
        sum = Dot_Product (a_j, b_j)
        n = size(b_{I_j})
        sum = sum + Σ_{i=1}^{n} [a_{I_j}(i) × b_{I_j}(i)]/Shared(i)
       MPI_AllReduce( sum, InnerProduct, MPI_SUM)
End Function InnerProduct
```

Figure 4.5: Algorithm for the function InnerProduct

tion involves one global communication, which is a global reduction operation with a "sum", the operation calculates the sum of the contributions of all the processors and provides each processor with a copy of the sum. The contributions of the interface nodes to the inner product are scaled by the number of processors which share the node. This is because the calculation is duplicated on all the processors for shared interface nodes. This represents a parallel overhead which could be avoided by assigning each of the shared nodes to a particular process. However, in order to assign the node to a particular processor, communication is required in the setup routines and also this requires additional change in the data structure. Since the number of shared interface nodes is small when compared to the interior nodes, this overhead can be neglected.

Preconditioning

The diagonal preconditioner is the diagonal of the stiffness matrix. It can be setup by choosing the diagonal elements of A_j and A_{I_j} of the assembled matrix. Even without the assembly of the stiffness matrix, parallel computation of the preconditioner is simple. Each processor loops over its elements to determine the local contribution to the diagonal of A_j and A_{I_j}. The following algorithms illustrate the way the diagonal preconditioner is setup for the distributed matrices and the operations performed when a preconditioner solution is required.

For all processor j ;

1. Loop over the elements and assemble the contributions of the diagonal of A_j to $\underline{di}_j$ and the contributions of the diagonal of A_{I_j} to $\underline{d}_{I_j}$
2. Update $(\underline{d}_{I_j})$
3. Invert the diagonal vectors $\underline{d}_j = 1/\underline{d}_j; \qquad \underline{d}_{I_j} = 1/\underline{d}_{I_j},$

Figure 4.6: Algorithm to set up the diagonal preconditioner

Subroutine Solve $(\underline{z}_j, \underline{z}_{I_j}; \underline{r}_j, \underline{r}_{I_j})$
$\underline{z}_j = \underline{d}_j * \underline{r}_j$
$\underline{z}_{I_j} = \underline{d}_{I_j} * \underline{r}_{I_j}$
End Subroutine Solve

Figure 4.7: Subroutine for diagonal preconditioner solution

In the algorithm below the diagonal preconditioner is setup by looping over the elements, the preconditioner is stored as a vector. It is also inverted. Inverting the diagonal matrix is trivial and the inverted diagonal matrix is equal the reciprocal of all the diagonal elements. Once the diagonal elements are inverted the values corresponding to the interface nodes are updated with the "Update" routine to save on the communication during the PCG iteration phase.

Once the diagonal preconditioner is setup, whenever the conjugate gradient algorithm requires a preconditioner solution, the preconditioner solution is achieved by multiplying each of the elements of the given vector by the element of the inverted diagonal vector. Suppose the preconditioner solution is given by $z = M^{-1}r$, the 4.7 illustrates the algorithm to implement it.

For each processor;

1. $\underline{b}_{I_j} =$ Update $(\underline{b}_{I_j})$
2. $\underline{x}_j^0 = \underline{0}; \qquad \underline{x}_{I_j}^0 = \underline{0}$
3. $\underline{r}_j^0 = \underline{b}_j; \qquad \underline{r}_{I_j}^0 = \underline{b}_{I_j}$
4. Solve $M(\underline{z}_j^0, \underline{z}_{I_j}^0) = (\underline{r}_j^0, \underline{r}_{I_j}^0)$
5. $\underline{p}_j^0 = \underline{z}_j^0; \qquad \underline{p}_{I_j}^0 = \underline{z}_{I_j}^0$
6. $\gamma^0 =$ Inner Product $(\underline{z}_j^0, \underline{z}_{I_j}^0; \underline{r}_j^0, \underline{r}_{I_j}^0)$

   Do $k = 0, 1, 2, \ldots, k_{max}$

7. $(\underline{q}_j^k, \underline{q}_{I_j}^k) =$ Matrix Multiply $(A, (\underline{p}_j^k, \underline{p}_{I_j}^k))$
8. $\tau^k =$ Inner Product $(\underline{p}_j^0, \underline{p}_{I_j}^0; \underline{q}_j^0, \underline{q}_{I_j}^0)$
9. $\alpha^k = \gamma^k / \tau^k$
10. $\underline{x}_j^{k+1} = \underline{r}_j^k + \alpha^k \underline{p}_j^k; \qquad \underline{x}_{I_j}^{k+1} = \underline{x}_{I_j}^k + \alpha^k \underline{p}_{I_j}^k$
11. $\underline{r}_j^{k+1} = \underline{r}_j^k - \alpha^k \underline{q}_j^k; \qquad \underline{r}_{I_j}^{k+1} = \underline{r}_{I_j}^k - \alpha^k \underline{q}_{I_j}^k$
12. Solve $M(\underline{z}_j^{k+1}, \underline{z}_{I_j}^{k+1}) = (\underline{r}_j^{k+1}, \underline{r}_{I_j}^{k+1})$
13. $\gamma^{k+1} =$ Inner Product $(\underline{z}_j^{k+1}, \underline{z}_{I_j}^{k+1}; \underline{r}_j^{k+1}, \underline{r}_{I_j}^{k+1})$
14. If $(\sqrt{\gamma^{k+1}} \leq$ Tolerance $)$ End
15. $\beta^k = \gamma^{k+1} / \gamma^k$
16. $\underline{p}_j^{k+1} = \underline{z}_j^{k+1} + \beta^k \underline{p}_j^k; \qquad \underline{p}_{I_j}^{k+1} = \underline{z}_{I_j}^{k+1} + \beta^k \underline{p}_{I_j}^k$

   End Do

Figure 4.8: Modified Preconditioned Conjugate Gradient Algorithm for Parallel [4]

4.2 Parallel Conjugate Gradient Algorithm

Steps in the parallel version of the Preconditioned Conjugate Gradient Algorithm are shown in the Figure 4.8. In the description, k defines the iteration count, r specifies the linear residual, x represents the solution vector, M denotes the preconditioner matrix, z represents the pseudo-residual (the residual after the application of the preconditioner), p defines the step direction, α specifies the step length, β denotes the correction factor, γ denotes the square of the L2 norm of the residual and τ is a temporary variable.

Chapter 5

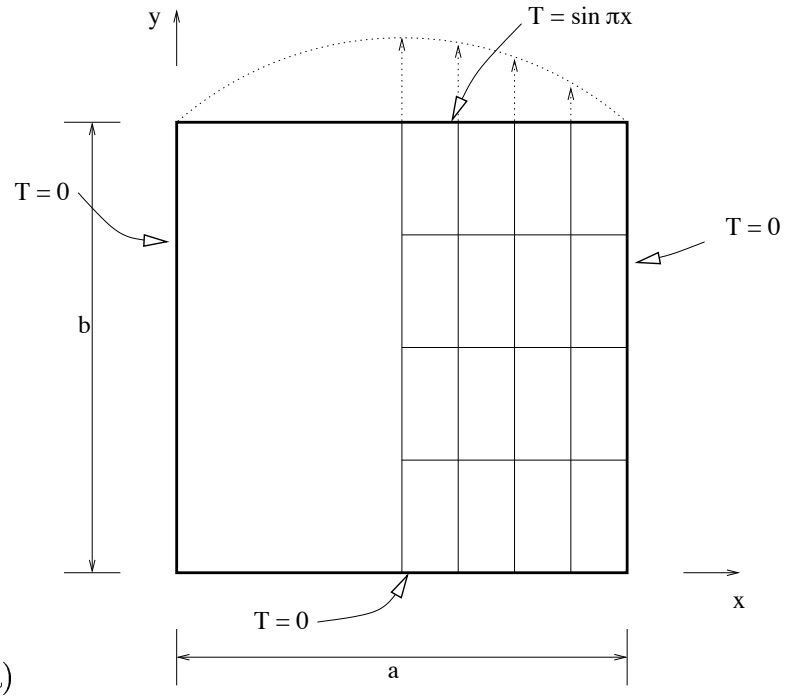
RESULTS

To demonstrate the performance and the applicability of the code to different types of elements and boundary conditions, the code was applied to four example problems. Poisson's equation is solved in all the cases. Applying the code to other types of equations can be accomplished by changing the element stiffness calculating routines. In the first example a rectangular domain problem is considered with four node rectangular elements. In the second example a square domain problem with mixed boundary conditions is solved. In the third example an unstructured triangular mesh generated using I-DEAS is used as an input to the program. In the fourth example a 3D domain is meshed with I-DEAS using unstructured tetrahedral elements. Scalability analysis is conducted for two different parallel architectures: 8 cpu SGI Onyx2 infinite reality system [29] and 14 cpu Sun HPC. Analysis was conducted on dedicated nodes. The code incorporates the METIS routines [20] to generate domain decomposition for all the example problems as discussed in Chapter 4. The scalability analysis is compared for both Element-By-Element and Compressed Sparse Row storage format.

5.1 Poisson's Equation with Dirichlet BC

The first example considered is a Poisson's equation on a rectangular domain. The governing differential equation and the boundary conditions are :

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0, \qquad 0 < x < 1,\ 0 < y < 1 \tag{5.1}$$



(a)

Figure 5.1: Heat Transfer Problem with Dirichlet B.C.

subject to Dirichlet boundary conditions

$$\begin{aligned} T(x=0) &= 0, \\ T(x=1) &= 0, \\ T(y=0) &= 0, \\ T(y=1) &= \sin(\pi x) \end{aligned} \tag{5.2}$$

The exact solution is given by,

$$T(x,y) = \frac{\sin(\pi x) \cdot \sinh(\pi y)}{\sinh(\pi)} \tag{5.3}$$

This problem models isotropic steady-state heat conduction with temperature field $T(x,y)$. The simulation domain and boundary conditions are depicted in Figure 5.1. Due to symmetry only half of the domain is meshed and solved. The element stiffness matrix for 4-node rectangular element has an analytical form given in Appendix B. The domain is discretized with 200 x 400 rectangular elements having a total 79,800

degrees of freedom. The partitioned mesh and the contours of the numerical solution are given in Figure 5.3.

Figure 5.4 shows the convergence of the CG iterative solution with and without diagonal preconditioning. The L2 norm of the residual is used as the check for convergence with 600×600 mesh. The CG method takes 1640 iterations while the diagonal preconditioned PCG takes 1219 iterations for a tolerance of 1.0e-06. The graph shows that both algorithms proceed in a similar fashion up to 1150 iterations; after that, the diagonal PCG converges faster when compared with CG. This behavior may be because of the structured mesh.

5.2 Scalability Analysis

In this section the parallel performance of the implementation is analyzed. The algorithm was implemented with MPI and Fortran 90 on two different multiprocessor machines, using appropriate Fortran 90 compilers. A 14 cpu Sun UltrasparcII with 336MHz cpu, 4MB cache and 4GB shared memory and a 8 cpu SGI MIPS R12000 with 400 MHz cpu, 8MB cache and 8GB shared memory. Though both computers are shared memory machines, distributed memory is simulated using MPI. The freely available, yet non optimized MPI version MPICH [23], from Argonne National laboratory is used on Sun machines, while the optimized vendor supplied version of MPI is used on the SGI machine. A separate serial version of the program was coded to calculate the scalability and efficiency of the parallel version. The coding and the data structure of the serial version was consistent with the parallel version. The compiler flags used on the Sun machine was -fast, and on SGI, -O3, was used. These compiler flags were very effective and reduced the total solution time by almost 50%.

For the scalability analysis of Example 1, the computational domain is discretized into 600 $\times$ 600 elements, giving a total of 361,201 nodes and 359,400 degrees of

freedom. The scalability analysis is done only for the diagonal preconditioned PCG solver. Parallel performance is described for elapsed wall clock time for the solution of the equations. This includes computation time and communication time. Speedup and the efficiency are defined by.

$$\text{Speedup} = \frac{\text{Time taken by one processor}}{\text{Time taken by N processors}} \tag{5.4}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Total number of processors}} \tag{5.5}$$

The iterative solution is stopped when a tolerance value of 1.0e-06 is reached. For any number of processors the number of iterations to converge is the same(1219) on both the Sun and SGI machines.

Figure 5.5 shows the of communication, computation and the total solution time for EBE and CSR format on the Sun machine. Similarly Figure 5.6 shows the timings on the SGI machine. Figure 5.7 gives the total solution time versus the number of processors for Element-By-Element (EBE) and Compressed Sparse Row format(CSR) for the Sun and SGI machines. Figure 5.8 shows the speedup of the implementation on Sun and SGI machines. Figure 5.9 plots the efficiency achieved on different processors for the Sun and SGI machines. Figure 5.10 shows the communication time for the CSR format with messages are sent with standard blocking calls to MPI. This is compared to nonblocking sends where computation is overlapped with communication.

Comparison of EBE and CSR

Figure 5.7 clearly shows that CSR takes less time for the solution when compared to EBE in all the cases and for both the machines. The CSR format takes 21 - 33 % less time than EBE on the Sun machine with an average of 27% and on the SGI machines this value ranges between 22 - 38% with an average of 30%. It was observed

that the matrix-vector multiplication was the most computationally intensive part in every iteration of the conjugate gradient method. The advantage of EBE method is that data is accessed in a sequential fashion, when there is a loop over the number of elements. So we can expect smaller number of cache misses. The disadvantage when compared to CSR format is that the computation is duplicated for the coefficients of the global matrix which have contributions from more than one element. CSR format has predominantly indirect memory access patterns, so there will be larger number of cache misses. The advantage of the CSR format is the lower number of floating point operations required. From the timings we can conclude that the number of duplicated operations in EBE are significantly high, so they negate the advantage EBE has over CSR in cache memory usage. In case of CSR, there is some amount of extra setup time to assemble the matrices, but in our implementation this was not significant when compared to the total solution time.

Communication

Figure 5.5 and Figure 5.6 give the split of communication time and computation time. For Sun machines the communication time increases with an increase in the number of processors. For 12 and 14 processors the communication time is very significant when compared to the computation time. This significantly reduces the efficiency of the implementation. For SGI machine, which uses a optimized vendor supplied MPI version, the time is relatively small, even for large number of processors.

As explained in Chapter 4, the communication and computation can be overlapped by using non-blocking Send and blocking Receive with computation during the communication. This is applicable only for CSR format since the matrices are stored. In EBE format, whenever matrix multiplication is done, the the present implementation loops over the elements and then multiplies the contributions of the element to the trial vector and assigns it to the result, so values of the interface nodes cannot be

calculated in advance. As a result, overlapping computation and communication is not possible in the present implementation of the EBE format. Figure 5.10 shows the advantage of using overlapping computation and communication on the Sun machine, the analysis was not conducted for SGI since the communication time taken by the implementation was small. The results show that, in general, some amount of time (around 10 seconds) is saved by overlapping communication and computation. Since the Sun HPC is also a shared memory machine, the communication might be fast when compared to other distributed memory machines like Beowulf clusters in which the communication takes place over a fast ethernet. Overlapping communications and computations might significantly increase the speedup for such kinds of machines in which communication is slow.

		EBE			CSR		
procs	machine	Comp	Comm	Total	Comp	Comm	Total
4	Sun	279.33	10.71 (3.69%)	290.04	199.39	15.67(7.29%)	215.06
	SGI	148.21	2.63(1.74%)	150.84	114.02	2.58(2.22%)	116.61
6	Sun	191.63	17.83(8.51%)	209.46	138.30	15.27(9.95%)	153.58
	SGI	73.12	9.99(12.02%)	83.11	49.60	5.59(10.13%)	55.20
8	Sun	144.84	19.74(12.0%)	164.59	98.68	27.93(22.06%)	126.61
	SGI	49.34	5.42(9.90%)	54.76	31.64	6.18(16.34%)	37.83

Table 1: Shows timings for EBE and CSR on two machines for 4, 6 and 8 processors. procs = number of processors, comp = computation time, comm = communication time and total = total time

Efficiency and Speedup

Figure 5.8 and Figure 5.9 shows the speedup and efficiency of the implementation respectively. The calculation of speedup and efficiency are very much dependent on

the timings of the single processor serial code ($N = 1$) case. For the Sun HPC, with the problem size considered, the program scales well up to 10 processors. From Figure 5.5 it can be observed that the communication time becomes prominent after 10 processors, greatly affecting the scalability of the system. The plots show that EBE scales better than CSR. At 10 processors, using EBE we get a speedup of 6 , for CSR we get a speedup of 5. Efficiency plots on the Sun machine indicate that the efficiency continues decreasing as we increase the number of processors, with an average efficiency of 0.75 for EBE and 0.6 for CSR.

For the SGI machine, the results indicate super-linear speedup for the CSR case. The SGI machine performs well because the time taken for communication is small and also the SGI machines have more cache memory than the Sun machine. The size of the matrices distributed to each processor memory will decrease as the number of processors increases. As a result the matrices might be able to fit in the cache memory, consequently reducing the cache-miss rate. This might be the explanation for super-linear speedup. Both the matrix storage schemes have about the same speedup up to 5 processors; after that CSR format performs better than EBE format.

5.3 Validation for other models

5.3.1 Poisson's Equation with Mixed BC

The next example is used to illustrate the application of the code for mixed boundary conditions. The governing differential equation and the boundary conditions are:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0, \qquad 0 < x < 1,\ 0 < y < 1 \tag{5.6}$$

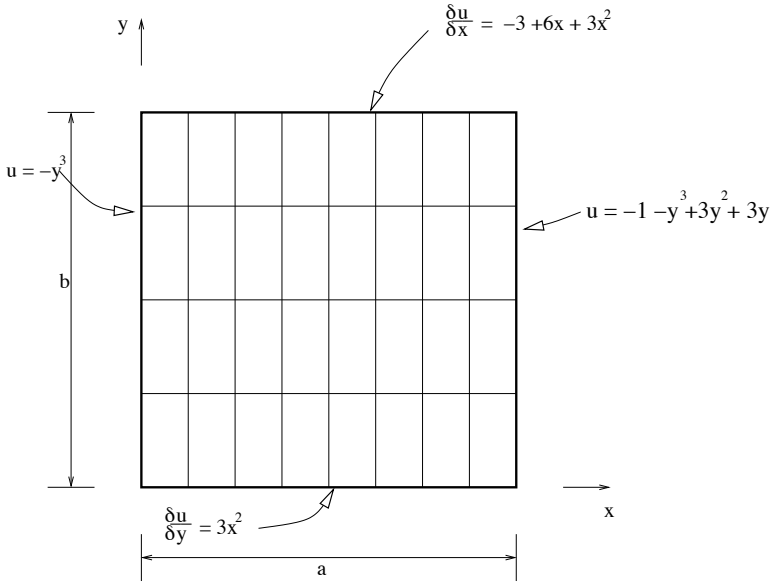


Figure 5.2: Example2: Mixed Boundary Condition Problem

subject to boundary conditions

$$\begin{aligned} T(x=0) &= -y^3, \\ T(x=1) &= -1 - y^3 + 3y^2 + 3y, \\ \frac{\partial T}{\partial y}(y=0) &= 3x^2 \\ \frac{\partial T}{\partial y}(y=1) &= -3 + 6x + 3x^2 \end{aligned} \tag{5.7}$$

The exact solution is given by,

$$T(x,y) = -x^3 - y^3 + 3xy^2 + 3x^2y \tag{5.8}$$

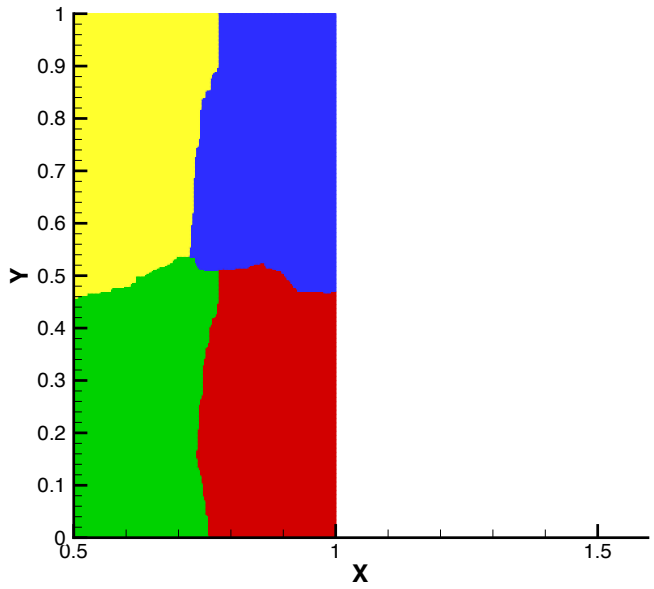
In this case the Neumann boundary represent applied heat flow conditions. The simulation domain and boundary conditions are depicted in Figure 5.2. The domain is meshed with 200 x 200 elements with a total of 39800 degrees of freedom. Figure 5.11 a indicates the partition of the domain into 4 subdomains using METIS. A contour plot of the solution is given in Figure 5.11 (b). The contours for the numerical solution match closely with those from the exact solution.

5.3.2 Unstructured mesh problem in 2D

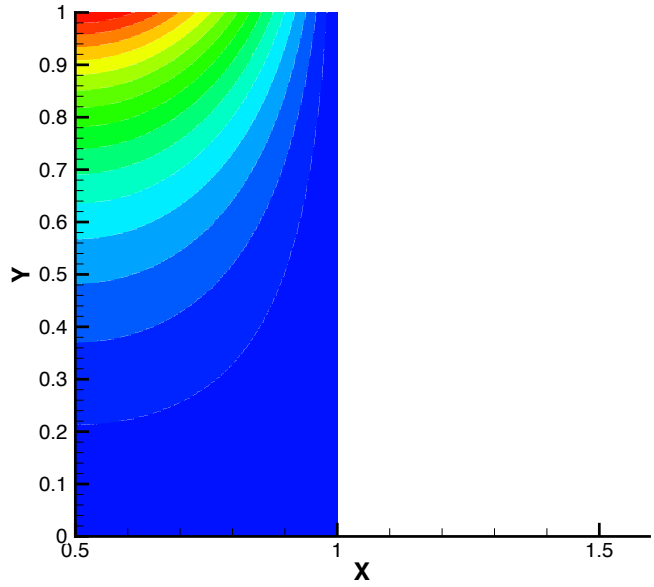
An unstructured triangular mesh is considered for the third example. A mesh file generated using the commercial I-DEAS software, with 104 nodes and 163 elements is used as an input to the code. A preprocessing routine called *CONMESH* was written to convert the mesh file to the one readable by the code. Figure 5.12 (a) illustrates the mesh and Figure 5.12 (b) shows the mesh partitioned into 4 subdomains using METIS. The boundary conditions are illustrated along with the mesh. The temperature is set to T = 0 on the left edge. Inside the circular area the temperature is set to T=1. All other boundaries of the model are insulated. The Figure 5.13 (a) illustrates the contour plot of the solution generated by the parallel code. Figure 5.13 (b) shows the contour plot of the solution generated using the I-DEAS software. The results validate the correctness of the parallel code. Figure 5.14 (a) shows the convergence rate for CG and PCG algorithms for this problem.

5.3.3 Unstructured mesh problem in 3D

A 3D problem is created by extruding the 2D surface in the Z-direction. The resulting volume is meshed with 4-node tetrahedral elements with a total of 437 nodes and 1456 elements. A tetrahedron meshed domain is used to apply the code for 3-D problems. The Figure 5.15 (a) illustrates the domain meshed with tetrahedral elements. The Figure 5.15 (b) illustrates the partitioned mesh into 4 parts using METIS. Figure 5.16 (a) and (b) show the solution contours for the parallel solution compared with that obtained with I-DEAS. Figure 5.14 (b) shows the convergence rate for CG and PCG algorithms for this problem.



(a)



(b)

Figure 5.3: Heat Transfer problem with Dirichlet B.C. : (a) Domain discretized into rectangular mesh partitioned into 4 sub-domains using METIS. (b) Solution contour plot

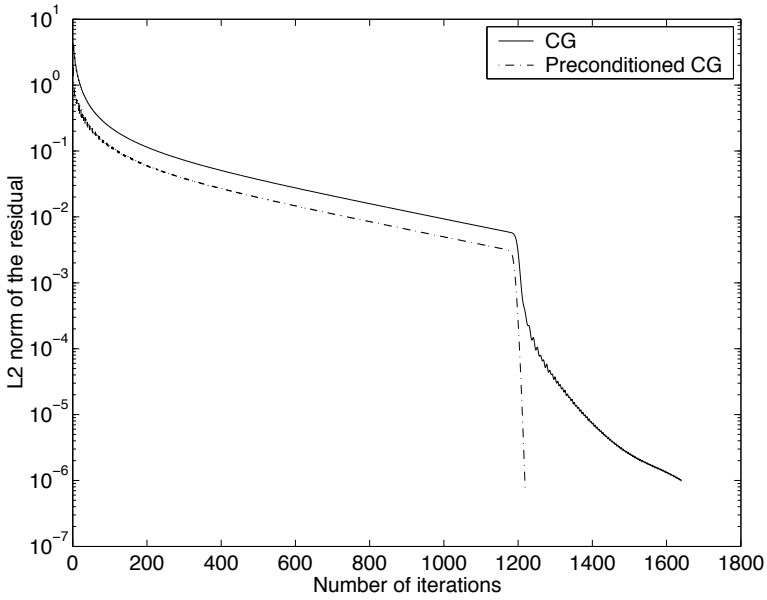


Figure 5.4: Comparison of convergence as a function of iteration count for Conjugate Gradient (CG) algorithm and Diagonal Preconditioned Conjugate Gradient algorithm (PCG).

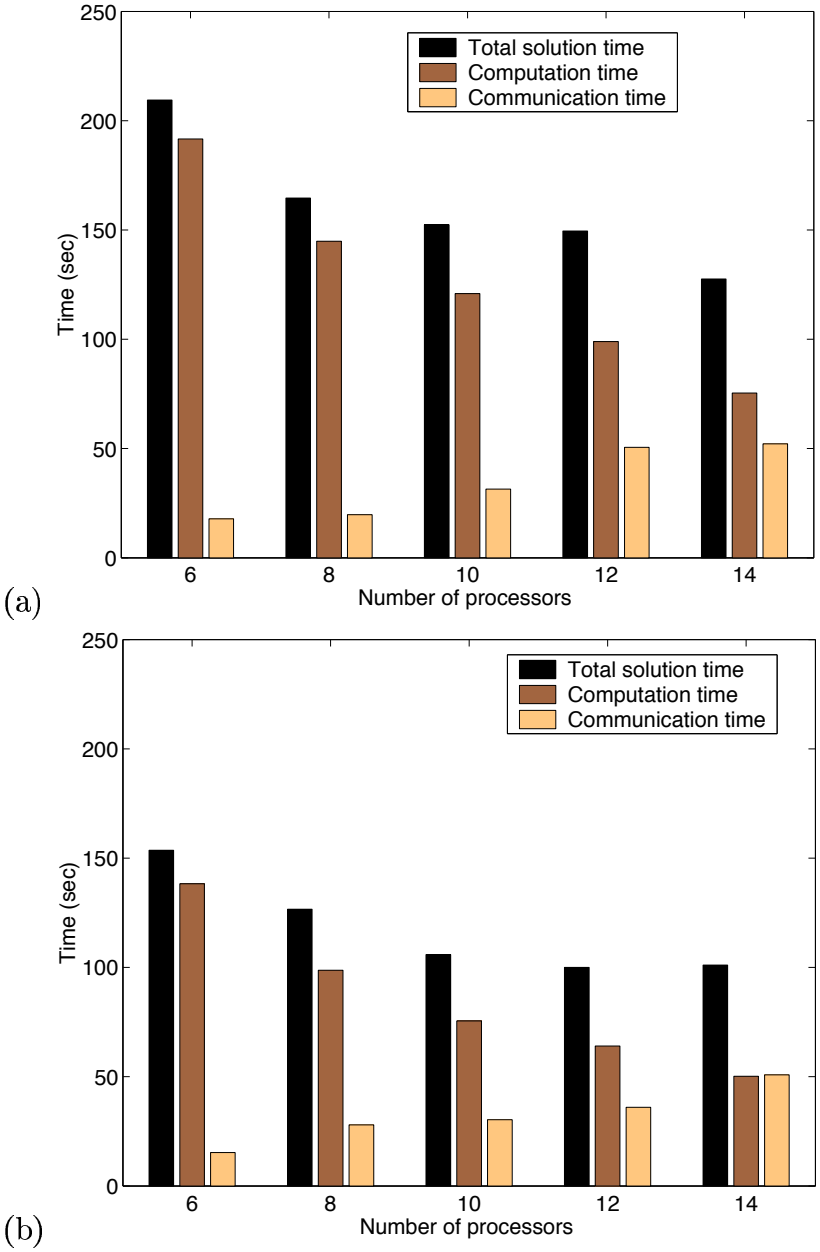
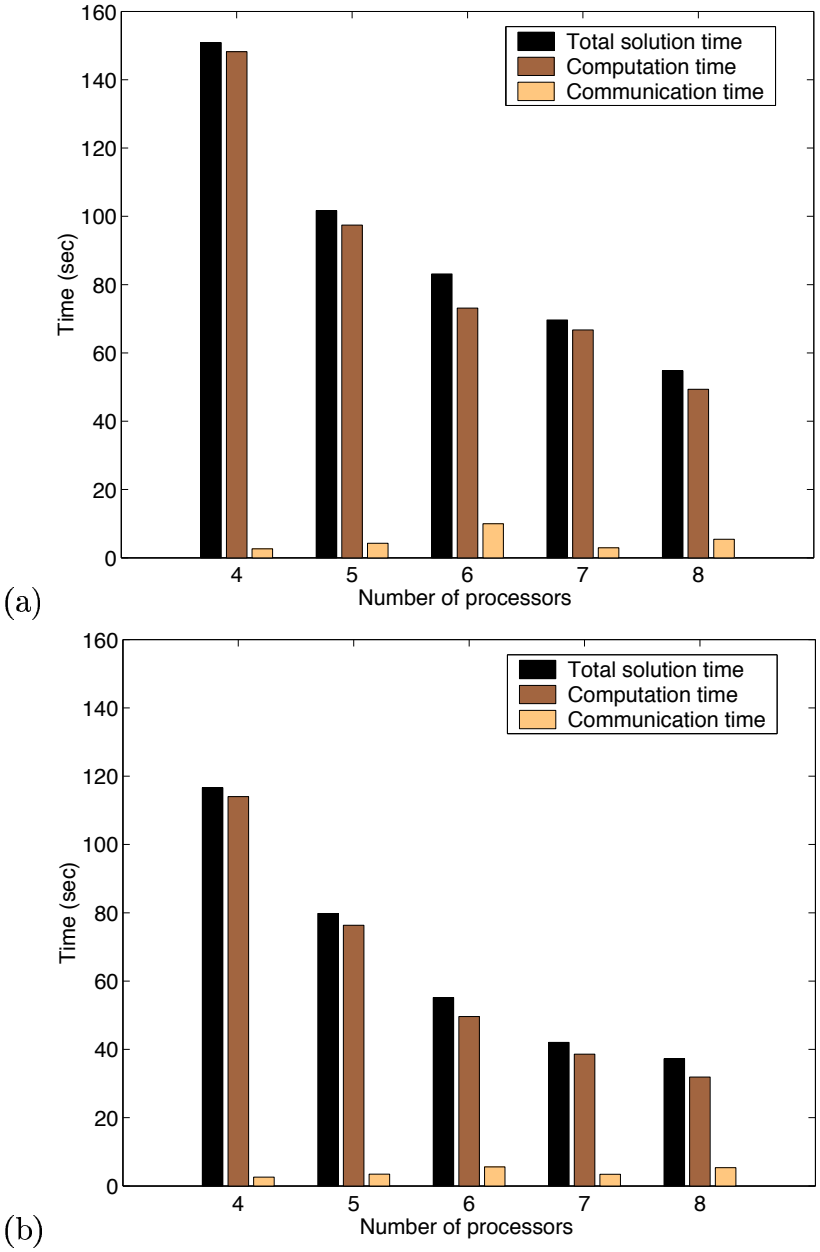


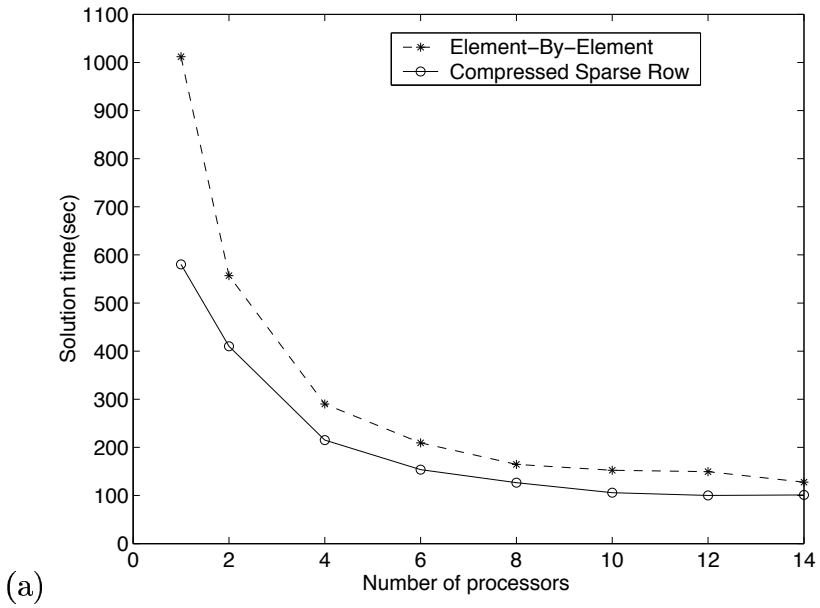
Figure 5.5: Split-up of Computation and Communication time on Sun HPC Machine: (a) EBE (b) CSR.



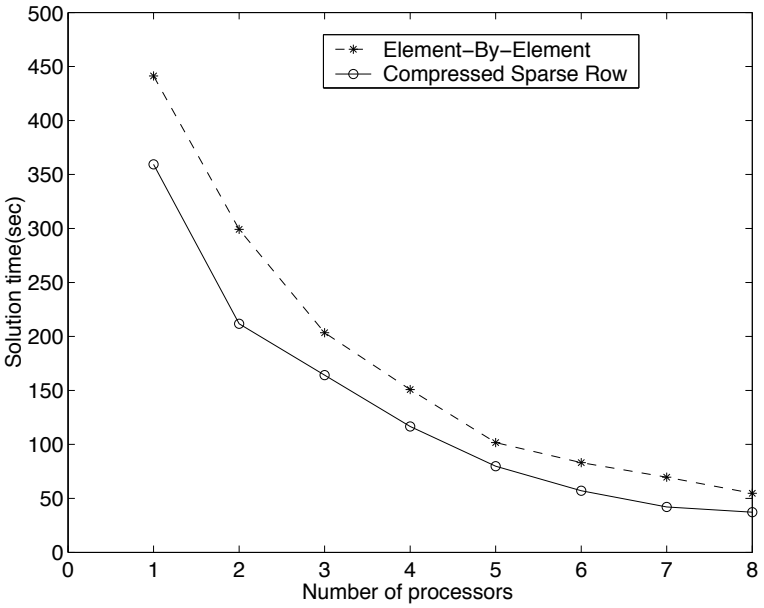
(a)

(b)

Figure 5.6: Split-up of Computation and Communication time on SGI Machine: (a) EBE (b) CSR.



(a)



(b)

Figure 5.7: Total solution time versus number of processors : (a) Sun HPC (b) SGI .

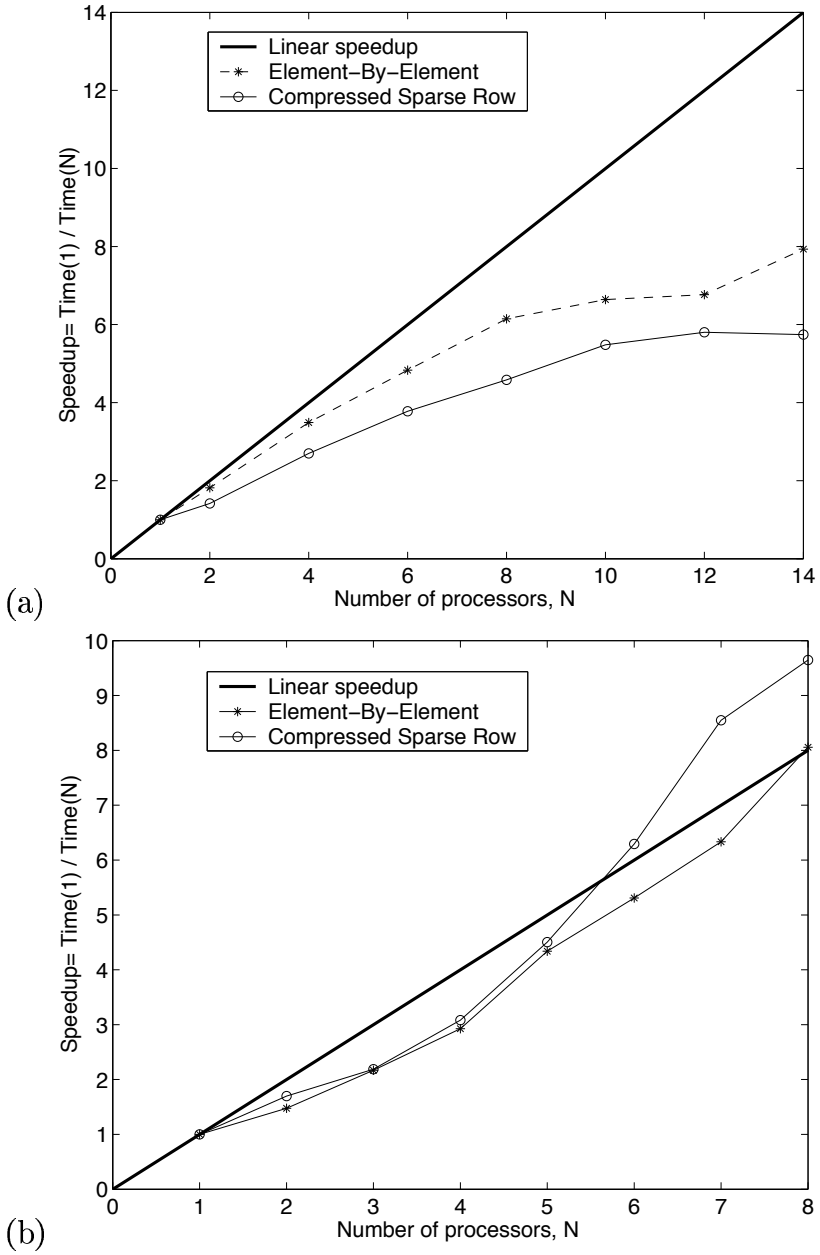
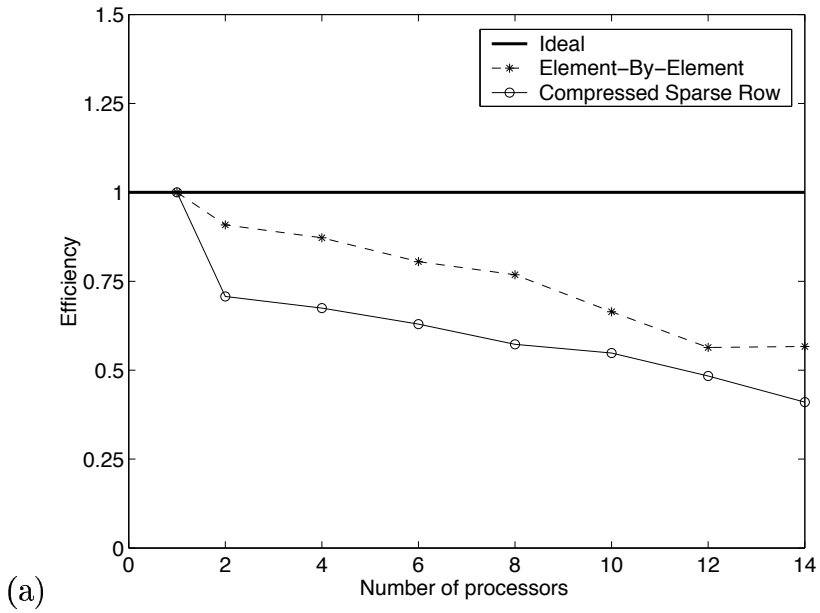
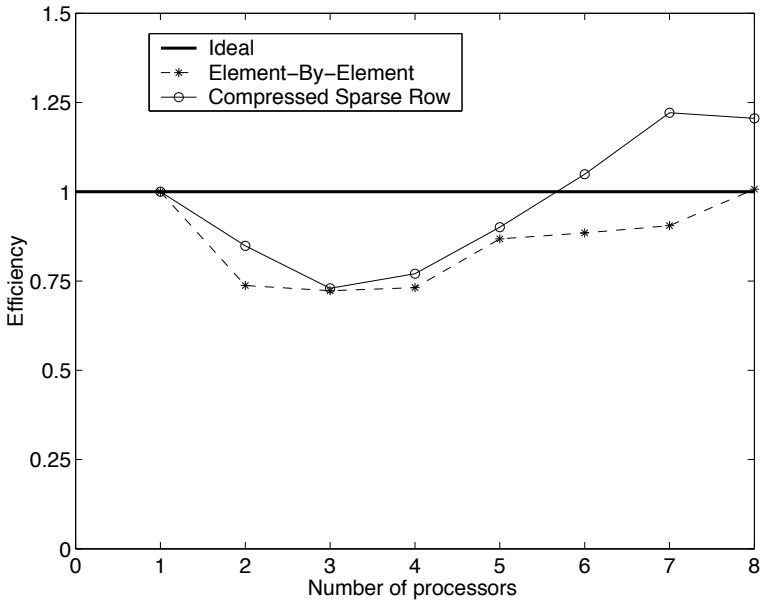


Figure 5.8: Speedups for computational tasks : (a) Sun HPC (b) SGI .



(a)



(b)

Figure 5.9: Efficiency : (a) Sun HPC (b) SGI .

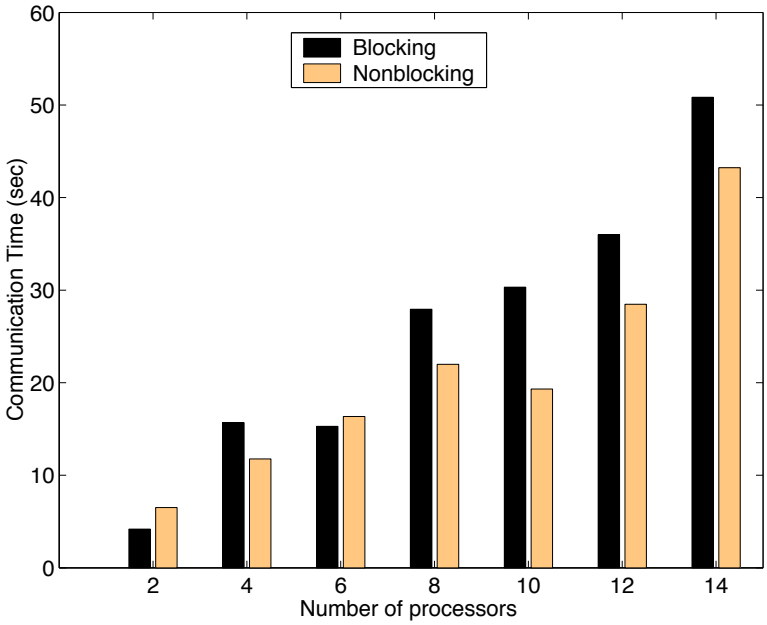
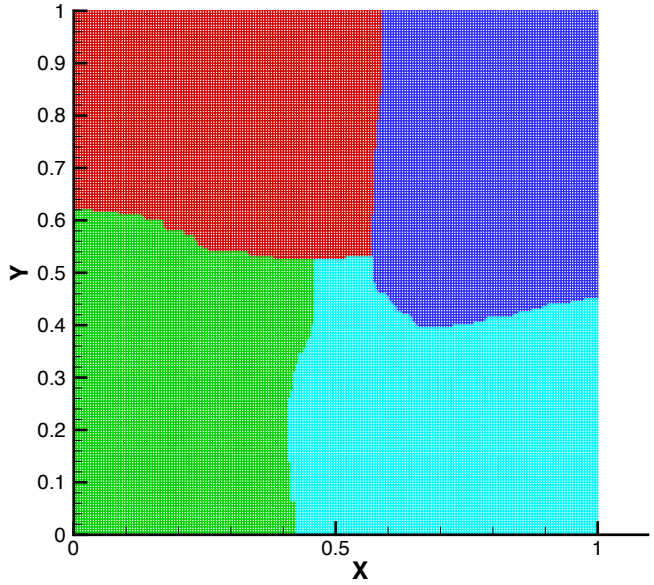
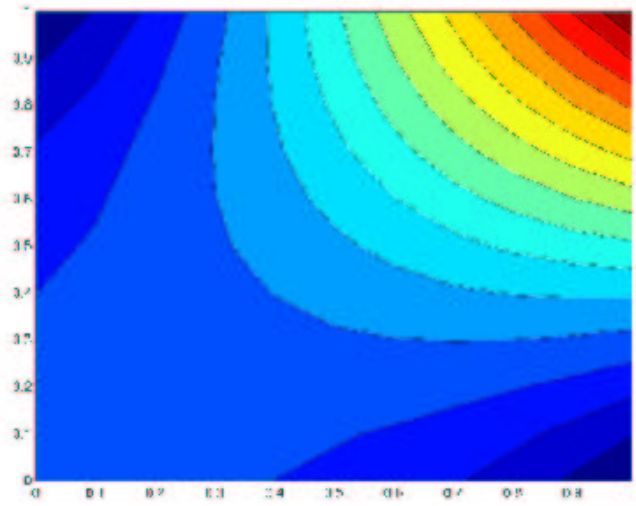


Figure 5.10: Comparison of the communication time for blocking vs. nonblocking communication.

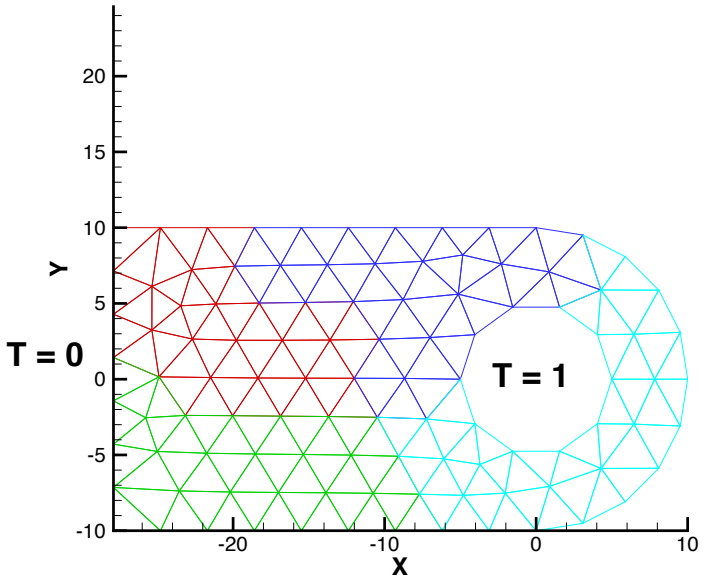


(a)

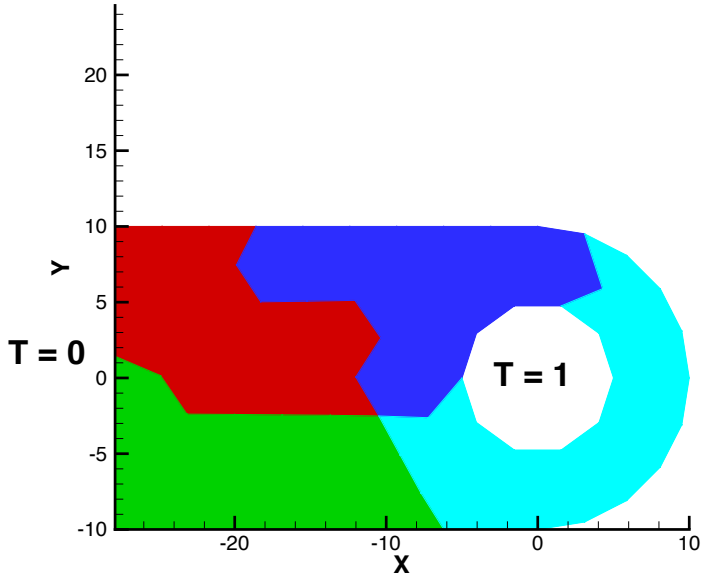


(b)

Figure 5.11: Heat Transfer problem with mixed B.C. : (a) Domain discretized into rectangular mesh partitioned into 4 sub-domains using METIS . (b) Solution to the mixed BC problem

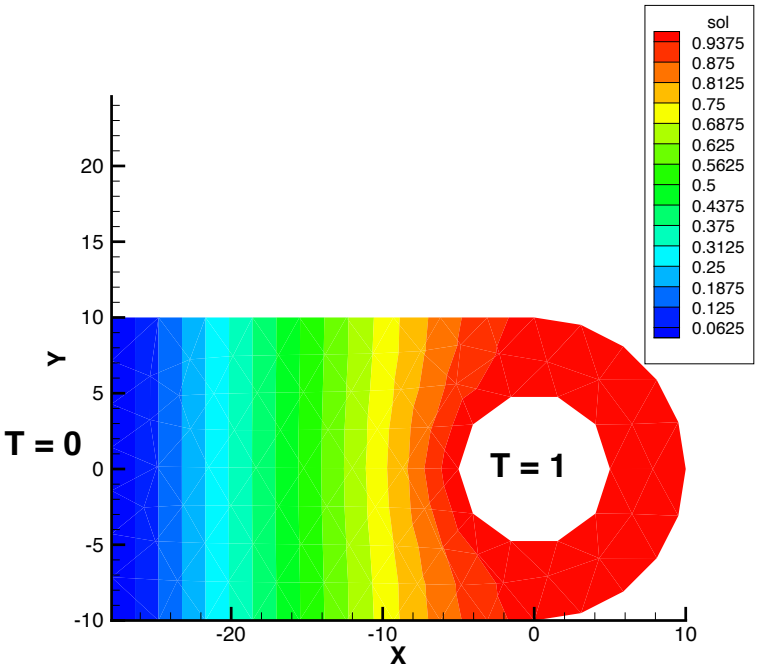


(a)

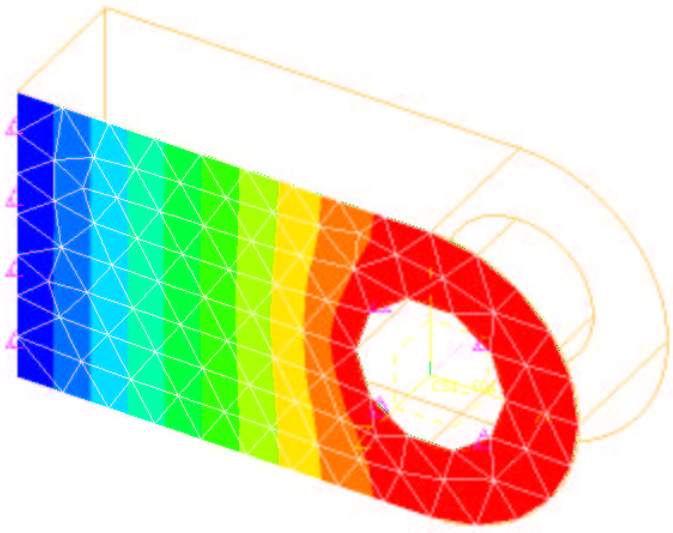


(b)

Figure 5.12: Analysis of 2D domain discretized using unstructured triangular mesh: (a) Domain discretized into triangular mesh (b) Domain partitioned into 4 subdomains using METIS.

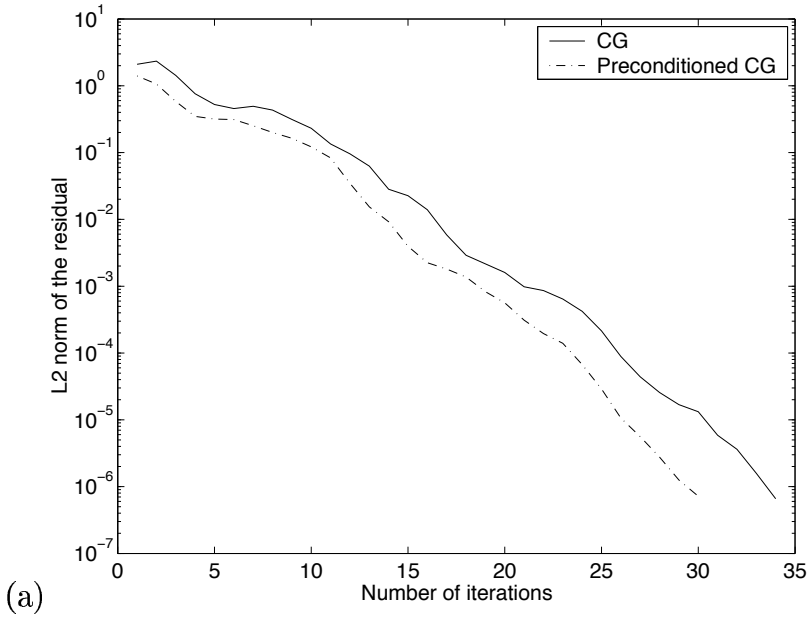


(c)

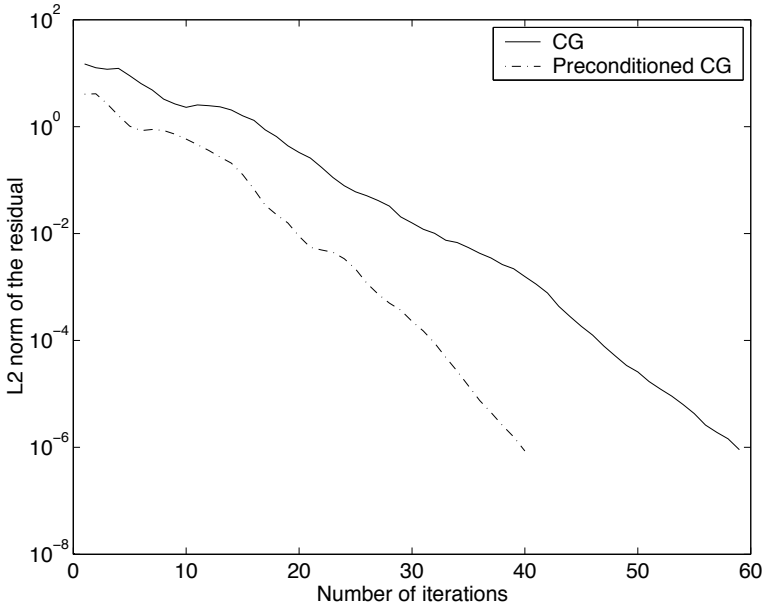


(d)

Figure 5.13: 2D problem, contour plots: (a) Solution generated by the parallel program (b) Solution Generated by the I-DEAS software

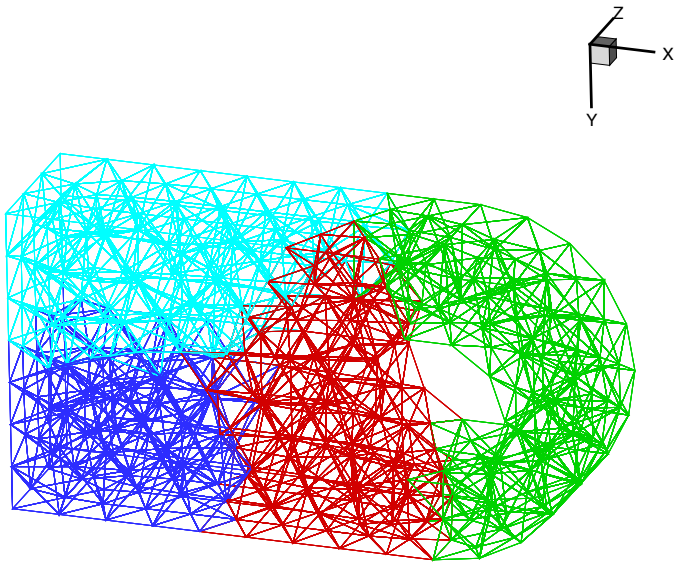


(a)

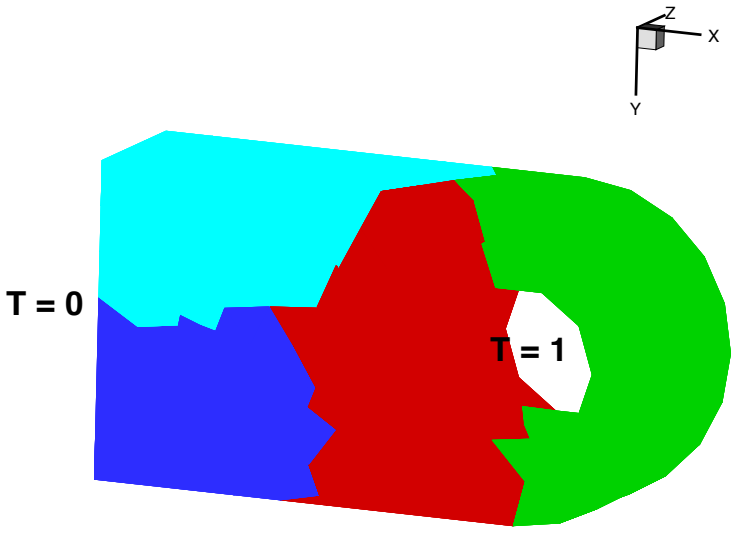


(b)

Figure 5.14: Comparison of convergence as a function of iteration count for Conjugate Gradient (CG) and Diagonal Preconditioned Conjugate Gradient algorithm (PCG) : (a) 2D problem (b) 3D problem.

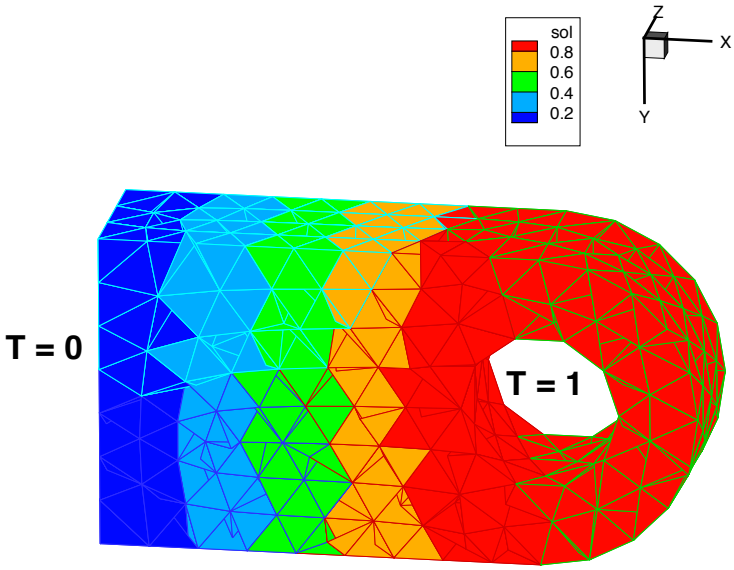


(a)

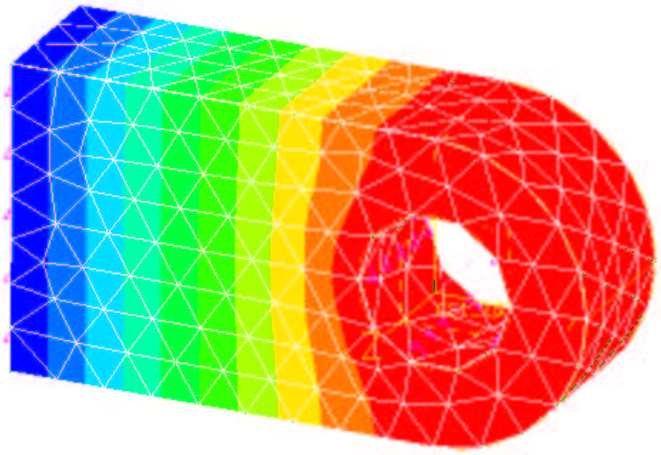


(b)

Figure 5.15: Analysis of 3D problem with tetrahedral elements: (a) Domain discretized into tetrahedral elements; (b) Domain partitioned into 4 sub-domains using METIS.



(c)



(d)

Figure 5.16: 3D problem Contour plots : (a) Solution generated by the parallel program; (b) Solution Generated by the I-DEAS software.

Chapter 6

CONCLUSIONS

The two important steps in finite element analysis, element stiffness matrix calculation and the solution of the system of equations, can be parallelized. In this implementation, domain decomposition is used for the parallel implementation of the finite element method. A conjugate gradient solver is used for the solution of the linear system arising out of the discretization. Two types of matrix storage schemes, EBE and CSR, are compared. Scalability analysis is conducted for both the matrix storage schemes on two parallel computers; Sun UltraSparcII and SGI MIPS. A Jacobi preconditioner is used in order to accelerate the convergence of the solver. The code was validated by applying it for problems with Dirichlet and Neumann BC and also for problems with a 3D unstructured mesh.

Specific conclusions include:

1. CSR format takes 30% less cpu time when compared to EBE. (This comparison is for "blocking send" implementation of matrix-by-vector products).

2. With optimized vendor supplied MPI on SGI the code shows super-linear speed up by taking advantage of the cache memory, for the problem tested.

3. Because of the communication overhead on Sun machine, for the particular problem the code scales well up to 10 processors. Beyond 10 processors loss of efficiency is shown.

4. By overlapping communication and computation some amount (approximately 10 seconds) of total communication time, can be saved.

5. The diagonal preconditioner moderately accelerates the convergence of the CG solver.

6.1 Future work

1. Perform the scalability analysis for large 3D problems.

2. Perform the scalability analysis on Beowulf clusters which are (physically) distributed memory machines and look into the behavior of communication time.

3. Implement the Schur complement method ([17],[18]).

4. Implement other types of preconditioners like Incomplete LU factorization (ILU) and Symmetric Successive Overrelaxation (SSOR) [27].

5. Generalize to non symmetric and non positive definite systems. Requires implementation of other Krylov subspace iterative methods such as Bi-Conjugate Gradient Stabilized (Bi-CGS), Quasi Minimum Residual (QMR), Generalized Minimum Residual(GMRES), see [30].

Appendices

Appendix A

PROGRAM STRUCTURE

The outline of the implementation of the code is shown in the Figure A.1. A Perl script *runtest* encompasses the whole operation. Once the mesh is generated the perl script can be called specifying the number of processors. The perl script ensures that the file is partitioned by METIS, preprocessed and solved using *pfem* solver.

The mesh for the problems is generated using an in house program *meshgen* or it might be generated from I-DEAS software and converted to the file readable by the program using the program *conmesh*. The mesh generated using I-DEAS software is exported in ABAQUS format and with minor modification to the text file it is input to another program *conmesh* which parses it and writes into two separate files *mesh.tecplot* and *mesh.metis*. *mesh.metis* is used as input to the METIS program which writes the output as a single array to a file indicating which processor the corresponding element belongs to. The Preprocessor is a routine *preproc* written in C, it is used to read in the mesh and the output of domain decomposition algorithm (METIS) and to create a different file for each processor with local node and element numbers. These files also have information of the subdomain boundary nodes and the processors the local processor is connected to. These files are read by the solver and after the solution has converged, the solver writes the solution to N different files where N is the number of processors used by the program. The solution is analyzed using Tecplot.

The main data structures in the code are Node, Element and CSRmatrix. The node data type contains the position of the node the local node number, it's boundary condition and its type (interior, subdomain boundary, or global boundary). The

element data structure contains the node number each element is connected to. The CSRmatrix contains data structure for storing the CSR format matrix.

Under the main pfem directory are the source files and the subdirectories *data* and *scripts* . A brief description of each of the source files and the subdirectories follows.

main.f90: This file contains the main module. It calls the routines to read the data and then applies the boundary conditions and calls the solver to solve the equations. Finally it calls routines to write the solution.

global_data.f90 : This has all the global data structures needed by the program

inout.f90 : This file contains module which incorporates routines to read in the files and write out the solution.

stiff.f90 : This file contains module which incorporates routines to calculate element stiffness matrix for triangular, rectangular and tetrahedral elements and also routines to assemble the global stiffness matrix in CSR format.

precon.f90 : Contains module which has routines to assemble the diagonal preconditioner and a routine used for preconditioner solutions.

vectorop.f90 : Contains module which has routines to do matrix-by-vector multiplication and dot-product for EBE and CSR matrix formats.

algor.f90 : Contains routines for the implementation of conjugate gradient and preconditioned conjugate gradient algorithm.

data : This is a subdirectory which contains *meshgen* , *conmesh* and *preproc* routines. This directory also stores the input and the output files.

scripts : This is a subdirectory which contains the *runtest* routine. The *pfem.cfg* file is used to input the parameters to the *runtest* script. The output of the *runtest* is written to the *run.out* file, once the test has been successfully completed.

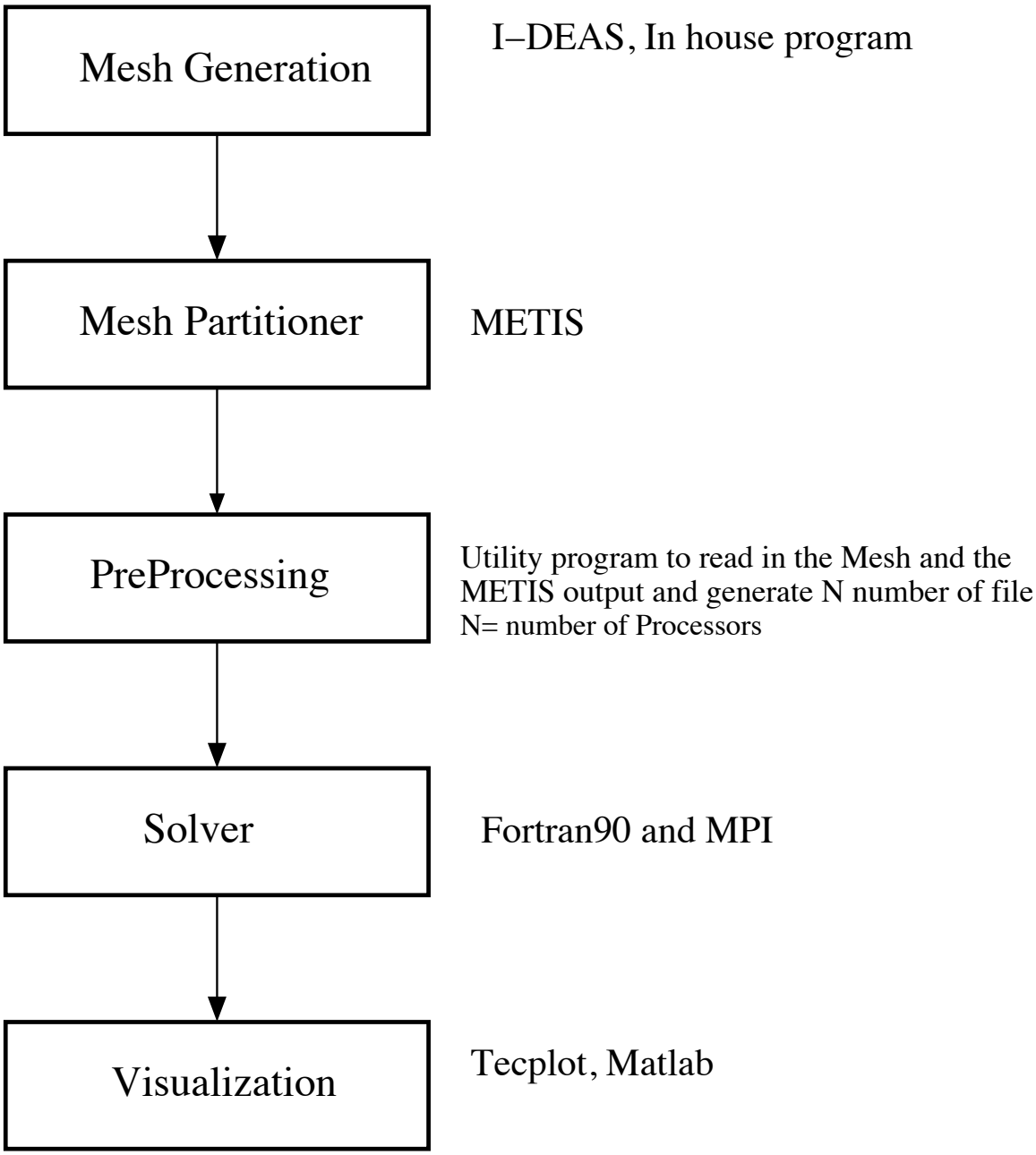


Figure A.1: Outline of the Implementation of the code

Appendix B

ANALYTICAL FORMS OF ELEMENT STIFFNESS MATRICES

This appendix gives the analytical form of element stiffness matrix for triangular, rectangular and tetrahedral elements. For a detailed description, see [5].

B.1 3-Node Triangular Element

Given three nodal coordinates (x_i, y_i), of the unstructured isotropic triangular element and the thermal conductivity inside the element k_e, the element stiffness matrix is calculated in the following way.

$a_1 = x_3 - x_2 \qquad b_1 = y_2 - y_3$

$a_2 = x_1 - x_3 \qquad b_2 = y_3 - y_1$

$a_3 = x_2 - x_1 \qquad b_3 = y_1 - y_2$

$$2A_e = a_2 b_1 - a_1 b_2 \tag{B.1}$$

$$K_{ij}^e = \frac{k_e}{4A_e}(b_i b_j + a_i a_j) \qquad for\ i, j = 1, 2, 3 \tag{B.2}$$

B.2 4-Node Rectangular Element

For a rectangular isotropic element, given h_x, the width of the element, h_y, the height of the element, the element stiffness matrix is calculated in the following way.

$\alpha = \frac{h_y}{h_x}, \qquad \beta = \frac{h_x}{h_y}$

$$K^e = \frac{k_e\alpha}{6}\begin{bmatrix} 2 & -2 & -1 & 1 \\ -2 & 2 & 1 & -1 \\ -1 & 1 & 2 & -2 \\ 1 & -1 & -2 & 2 \end{bmatrix} + \frac{k_e\beta}{6}\begin{bmatrix} 2 & 1 & -1 & 2 \\ 1 & 2 & -2 & -1 \\ -1 & -2 & 2 & 1 \\ -2 & -1 & 1 & 2 \end{bmatrix} \tag{B.3}$$

B.3 4-Node Tetrahedral Element

Consider the tetrahedral element shown in Figure B.1 with corners 1-4. This element is a four-noded solid. The nodes of the element must be numbered such that when viewed from the last node, the first three nodes are numbered in a counterclockwise manner. Given four nodal coordinates (x_i, y_i), of the unstructured isotropic tetrahedron element and the constant thermal conductivity inside the element k_e, the element stiffness matrix is calculated in the following way.

The volume of the element is obtained by,

$$6V = \begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} \tag{B.4}$$

The coefficients are calculated by,

$$\beta_1 = -\begin{vmatrix} 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \\ 1 & y_4 & z_4 \end{vmatrix} \qquad \gamma_1 = \begin{vmatrix} 1 & x_2 & z_2 \\ 1 & x_3 & z_3 \\ 1 & x_4 & z_4 \end{vmatrix} \qquad \delta_1 = -\begin{vmatrix} 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \\ 1 & x_4 & y_4 \end{vmatrix} \tag{B.5}$$

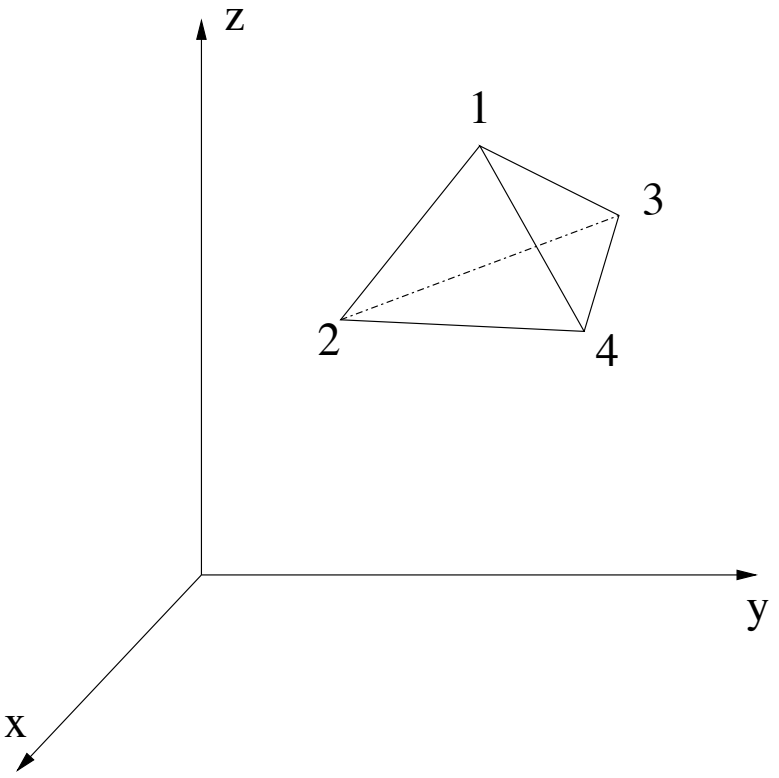


Figure B.1: Tetrahedral solid element.

$$\beta_2 = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_3 & z_3 \\ 1 & y_4 & z_4 \end{vmatrix} \qquad \gamma_2 = -\begin{vmatrix} 1 & x_1 & z_1 \\ 1 & x_3 & z_3 \\ 1 & x_4 & z_4 \end{vmatrix} \qquad \delta_2 = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_3 & y_3 \\ 1 & x_4 & y_4 \end{vmatrix} \tag{B.6}$$

$$\beta_3 = -\begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_4 & z_4 \end{vmatrix} \qquad \gamma_3 = \begin{vmatrix} 1 & x_1 & z_1 \\ 1 & x_2 & z_2 \\ 1 & x_4 & z_4 \end{vmatrix} \qquad \delta_3 = -\begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_4 & y_4 \end{vmatrix} \tag{B.7}$$

$$\beta_4 = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \qquad \gamma_4 = -\begin{vmatrix} 1 & x_1 & z_1 \\ 1 & x_2 & z_2 \\ 1 & x_3 & z_3 \end{vmatrix} \qquad \delta_4 = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} \tag{B.8}$$

The B matrix is given by,

$$B = \begin{bmatrix} \beta_1 & \gamma_1 & \delta_1 \\ \beta_2 & \gamma_2 & \delta_2 \\ \beta_3 & \gamma_3 & \delta_3 \\ \beta_4 & \gamma_4 & \delta_4 \end{bmatrix} \tag{B.9}$$

Then for constant k_e, the element stiffness matrix is given by:

$$[K^e] = k_e[B]^T[B]V \tag{B.10}$$

Bibliography

[1] Saad, Y., *Iterative Methods for Sparse Linear Systems*, PWS publishing company., Boston, 1996.

[2] Ianculescu, C. and Thompson, L., Private Communication.

[3] Golub, G. and Ortega, J., *Scientific Computing An Introduction with Parallel Computing*, Academic Press Inc., San Diego, 1997.

[4] Jimack, P. and Touheed, N., Developing Parallel Finite Element Software Using MPI.

[5] Logan., D. L., *A First Course in the Finite Element Method, 3rd Edition*, BROOKS/COLE., Pacific Groove, CA, 2000.

[6] Barragy, E. and Carey, G. F., A Parallel Element-By-Element Solution Scheme, (1987).

[7] Gullerud, A. and Dodds, R. J., MPI-based implementation of a PCG solver using an EBE architecture and preconditioner for implicit, 3-D finite element analysis, *Computers & Structures*, **79**:553–575 (2001).

[8] Thiagarajan, G. and Aravamuthan, V., Parallelization Strategies for Element-by-Element Preconditioned Conjugate Gradient Solver Using High-Performance Fortran for Unstructured Finite-Element Applications on Linux Clusters, *J. of Computing in Civil Engineering*, **16**(1) (2002).

[9] Farhat, C., Maman, N., and Brown, G., Mesh Partitioning for Implicit Computations via Domain Decomposition., *Internat. J. Numer. Methods Engng.*, **38**:989–1000 (1995).

[10] Hodgson, D. and Jimack, P., Efficient Mesh Partitioning for Parallel P.D.E. Solvers on Distributed Memory Machines, *Proceedings of the sixth SIAM conference on Parallel Processing for Scientific Computing*, (1993).

[11] Farhat, C. and Crivelli, L., A transient FETI methodology for large-scale parallel implicit computations in structural mechanics., *Internat. J. Numer. Methods Engng.*, **37**:1945–75 (1994).

[12] Gropp, W. and Lusk, E., The Message Passing Interface (MPI) standard, Technical report, http://www-unix.mcs.anl.gov/mpi/.

[13] Farhat, C. and Wilson, E., A Parallel Active Column Equation Solver., *Computers and Structures.*, **28**:289–304 (1988).

[14] Law, K. and D.R., M., A Parallel Row-Oriented Sparse Solution Method for Finite Element Structural Analysis, *Internat. J. for Numer. Methods in Engrg.*, **36**:2895–2919 (1993).

[15] Dongarra, J. and Johnsson, L., Solving Banded Systems on a Parallel Processor, *Parallel Computing*, **5**:219–246 (1987).

[16] Khan, A. and Topping, B., Parallel finite element analysis using Jacobi-conditioned conjugate gradient algorithm, *Advances in Enginnering Software*, **25**:309–319 (1996).

[17] Papadrakakis, M. and Bitzarakis, S., Domain decomposition PCG methods for serial and parallel processing, *Advances in Engineering Software.*, **25**:291–307 (1996).

[18] Thompson, L., Zhang, L., and Ingel, P., Domain Decomposition Methods with Frequency Band Interpolation for Computational Acoustics, *Proceedings of the 2001 International Mechanical Engineering Congress & Exposition, New York, USA*, (2001).

[19] Farhat, C., A method of finite element tearing and interconnecting and its parallel solution algorithm, *Int J Numer Meth Engng.*, **32**:1205–27 (1991).

[20] Karypis, G. and Kumar, V., METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Technical report, Department of Computer Science/Army HPC Research Center, University of Minnesota, Available at http://www-users.cs.umn.edu/karypis/metis, 1998.

[21] Almasi, G. and A., G., *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA, 1989.

[22] Gropp, W., Lusk, E., and Skjellum, A., *Using MPI:Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, Massachusetts, London, England, 1996.

[23] Gropp, W. and Lusk, E., User's Guide for mpich, a Portable Implementation of MPI, Technical report, Argonne National Laboratory, 1996.

[24] Reddy, J., *An Introduction to the Finite Element Method, 2nd Edition*, McGraw-Hil, Inc., New York, 1993.

[25] Johnson, C., *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Cambridge University Press, 1987.

[26] Golub, G. and Van Loan, C., *Matrix Computations*, The Johns Hopkins University Press., Baltimore, Maryland, 1991.

[27] Jimack, P. and Hodgson, D., Parallel Preconditioners Based Upon Domain Decomposition, *School of Computer Studies, University of Leeds.*

[28] Ianculescu, C. and Thompson, L., Efficient Parallel Iterative Methods For Acoustic Scattering With Exact Nonreflecting Boundaries, *Proceedings of The Forum Acusticum Sevilla, Spain, Sept. 2002.*

[29] Silicon Graphics Inc., *http://www.sgi.com/homepage.html.*

[30] Barrett, R., Berry, M., and et. al., Templates for the solution of Linear Systems: Building Blocks for Iterative Methods.