CRANFIELD UNIVERSITY

Dulcenéia Becker

Parallel Unstructured Solvers for Linear Partial Differential Equations

School of Engineering
Applied Mathematics and Computing Group

PhD Thesis

CRANFIELD UNIVERSITY

School of Engineering, Department of Process and Systems Engineering

Applied Mathematics and Computing Group

PhD Thesis

Academic Year 2005 - 2006

Dulcenéia Becker

Parallel Unstructured Solvers for Linear Partial Differential Equations

Professor C. P. Thompson

Supervisor

May 2006

This thesis is submitted in partial fulfilment

of the requiremetns for the degree

of Doctor of Philosophy

ALL MISSING
PAGES ARE
BLANK
IN
ORIGINAL

To my parents, Nelson and Marlise,
to whom I owe so much.

Abstract

This thesis presents the development of a parallel algorithm to solve symmetric systems of linear equations and the computational implementation of a parallel partial differential equations solver for unstructured meshes. The proposed method, called distributive conjugate gradient – DCG, is based on a single-level domain decomposition method and the conjugate gradient method to obtain a highly scalable parallel algorithm.

An overview on methods for the discretization of domains and partial differential equations is given. The partition and refinement of meshes is discussed and the formulation of the weighted residual method for two- and three-dimensions presented. Some of the methods to solve systems of linear equations are introduced, highlighting the conjugate gradient method and domain decomposition methods. A parallel unstructured PDE solver is proposed and its actual implementation presented. Emphasis is given to the data partition adopted and the scheme used for communication among adjacent subdomains is explained. A series of experiments in processor scalability is also reported.

The derivation and parallelization of DCG are presented and the method validated throughout numerical experiments. The method capabilities and limitations were investigated by the solution of the Poisson equation with various source terms. The experimental results obtained using the parallel solver developed as part of this work show that the algorithm presented is accurate and highly scalable, achieving roughly linear parallel speed-up in many of the cases tested.

Acknowledgements

I would like to thank my supervisor Professor Chris Thompson, Dr. Phil Rubini and Dr. Julian Turnbull for the guidance throughout this work. My sincere thanks go to Professor Rudnei Dias da Cunha who was a source of guidance and support both before and during the course of this work. I would also like to thank CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - for funding this project and the Cambridge-Cranfield High Performance Computing Facility for allowing me the use of its computing facilities.

I wish to express my thankfulness to my sister Daiana for her invaluable help. I also would like to thank my friends and colleagues from UFRGS and Cranfield for their encouragement and support. I especially would like to thank Aaron, who kindly spared his time to proofread some of my documents. Finally, I wish to express my gratitude to every person that in one way or another has supported me during the course of this work.

Contents

List of Figures

List of Tables

Notation

$A, \ldots, Z$	matrices
$a, \ldots, z$	vectors
$\alpha, \beta, \ldots, \omega$	scalars
A^T	matrix transpose
A^{-1}	matrix inverse
$A_{i,j}$	matrix element
$A_{:,j}$	j-th matrix column
A_i	matrix block
x_i	vector block
$(\cdot,\cdot)$	vector dot product (Euclidean inner product)
$(\cdot,\cdot)_M$	M-inner product
$diag(A)$	diagonal of matrix A
$\mathcal{K}(\cdot,\cdot)$	Krylov subspace
$\|x\|_2$	2-norm
λ_{max}, λ_{min}	maximun and minimum eigenvalues of a matrix
RGMRES(c)	RGMRES restart parameter

Chapter 1

Introduction

Since the late 1940s, the introduction of high-speed computing machines has made the field of numerical solutions of partial differential equations (PDEs) grow very rapidly. The progress in computer technology and numerical algorithms has allowed the solution of many complex problems. Nevertheless, there are still many problems that cannot be adequately solved using today's most powerful computers. Examples include computational aerodynamics and hydrodynamics, weather forecasting, plasma simulation, structural analysis and turbulence modelling. How to solve them accurately and efficiently is one of the most challenging goals. In order to get high throughput performance, algorithms and computer architectures must exploit parallelism.

The numerical solution of PDEs has been of major importance to the development of many technologies and has been the target of much of the development of parallel computer hardware and software. Parallel computing offers the promise of greatly increased performance and the routine calculation of previously intractable problems. To sustain high performance on these machines many fundamental issues need to be addressed, including the development of scalable algorithms.

The accuracy of the numerical solution of a PDE depends mainly on the discretization scheme used to transform the PDE into a discrete problem and on

the method used to solve the discrete problem. The discrete problem is usually a large, sparse system of linear algebraic equations for which the number of unknowns may vary from a few hundred to a few million. Greater accuracy is usually achieved by refining the mesh which increases the number of unknowns and hence the need for computational resources. Solving systems of linear equations (SLEs) is in many cases the most time consuming part of the whole approach. Therefore, the efficiency of the numerical scheme relies strongly on the methods used to solve SLEs.

The most commonly used methods to discretize PDEs are mesh-based which means that a mesh or grid[1] must be constructed before the PDE can be discretized. The mesh generation process, also called domain discretization, is an important and, in many cases, time consuming step of the solution. There are two fundamental classes of meshes in multidimensional regions, namely, structured and unstructured meshes. Structured grids are characterized by the organization of its cells, which is defined by a general rule. Unstructured meshes consist of a nearly absolute absence of any restrictions on grid cells, organization or structure, providing the most flexible tool for the discrete description of a geometry.

Another two areas related to meshing are mesh partitioning and mesh refinement. Mesh partitioning consists basically of splitting a mesh into parts so that the number of nodes of each part is nearly the same and the number of nodes on the interfaces is minimized. Mesh refinement can be performed globally or locally. The current research in the field is mainly concerned with local or adaptive refinement. These methods aim to refine the mesh only in regions where the approximation is not sufficiently accurate. Parallelization of adaptive refinement methods involves adaptation, repartitioning and rebalancing. In order to maintain the load balance

[1]Often the literature refers to structured grids and unstructured meshes. In the context of this work, the definition of grid and mesh is interchangeable.

among processors, the mesh must be dynamically repartitioned after it is refined.

The process of discretizing a PDE can be divided into spatial and temporal discretization. In practice, time derivatives are discretized almost exclusively using finite difference methods. Spatial derivatives are discretized by a variety of methods including finite difference, finite element, finite volume, spectral methods and SPH (Smoothed Particle Hydrodynamics) gridless methods. Each of these methods has its advantages and disadvantages. Parallelization of these methods is usually straightforward. Most of them can be implemented in such a way that communication is not needed. In this work, the finite element method on unstructured meshes is the method of choice because it supports unstructured meshes and hence allows easier study of complex geometries and also because it can have certain optimality properties.

The methods to solve systems of linear equations can be divided into two main classes, namely, direct and iterative methods. For systems of moderate[2] size, the use of direct methods is often advantageous. However, direct methods are not well suited to parallel implementations due to the inherent sequential nature of the forward and backward substitutions needed. Iterative methods are used primarily for solving large and complex problems for which, because of storage and arithmetic requirements, it would neither be feasible nor more efficient to solve by a direct method.

Only in the decade just passed did iterative methods start to gain considerable acceptance in real-life industrial applications. The inclusion of iterative solvers into such applications has been a slow process which is ongoing. How to choose an iterative method from the many available is still an open question since any

[2]In 2D, the crossing point between direct and iterative methods is approximately 20,000 equations, but direct methods may perform well up to 100,000 equations (especially if they are parallel). In 3D, however, the situation is considerably different.

method may solve a particular system in very few iterations while diverging on another.

Although there is a vast amount of research activity concerned with the solution of systems of linear equations, it is still one of the most challenging areas of scientific computing. Beyond the challenge of developing fast, efficient, accurate methods for serial computers, is the challenge of achieving high parallelism. Direct methods are inherently serial and hence extremely difficult to parallelize. Many techniques to induce parallelism on direct methods have been developed. However, such techniques must compromise in order to increase parallelism. Iterative methods have their parallelism mostly based on two basic linear algebraic operations, namely, matrix-vector products and inner products, and on the amount of information that needs to be exchanged among processors.

The parallel dot product operation is, in many cases, responsible for the loss on parallel performance as the number of processors increases. A parallel dot product operation requires a global reduce operation which means that the data of all processors need to be clustered and all processors need to own the clustered data. This has to be done in a given order and hence some processors might need to wait for data before carrying on the calculations.

Domain decomposition methods offer the possibility of avoiding parallel dot product operations. In the simplest case, a sub-system is solved at each processor separately and therefore parallel linear algebraic operations are not needed. The efficiency of such a method depends on how each sub-system is solved and the means used to communicate information among processors. The algorithm developed in this thesis aims to achieve high parallelism by approximating each sub-system using a fast serial method.

A brief overview of the contents of the thesis is as follows. Chapter 2 presents a

literature review that includes domain discretization methods, partial differential equations discretization techniques and methods used to solve systems of linear equations. Chapter 3 presents the formulation of the finite element method used in the software developed as part of this work. In Chapter 4, basic definitions used by iterative methods are reported and some of the methods used on the development of the algorithm introduced in this work, namely DCG, are discussed. Chapter 5 presents parallel computing concepts, a set of experiments in processor scalability and an overview of the parallel solver used to obtain the results presented in the following chapters. Chapter 6 reports a preliminary study taken to develop DCG and also presents the derivation and parallelization of DCG. Chapter 7 reports the numerical results obtained. Finally, Chapter 8 presents the conclusions and discusses further possible developments.

Chapter 2

Literature Review

There is a vast amount of research activity concerned with the numerical solution of PDEs, it is still one of the most challenging areas of Scientific Computing due to the versatile and often complicated structure of PDEs and because of the large amount of variables that need to be computed for two or higher dimensional problems.

The numerical solution of PDEs might be generally divided into discretization, both of the domain and the PDEs, and algebraic solver. Most of the methods to discretize PDEs are mesh-based, i.e. the solution is obtained throughout a mesh. The meshing process, that includes generation, partition and refinement of meshes, is an area of research on its own. More recently gridless methods have been proposed since the mesh is still a delicate component of the solution although meshing techniques are well developed.

The discretized problem usually results in a system of algebraic equations. Such systems might be solved by direct or iterative methods. Several methods have been proposed and to date none has been proven to outperform the others in general. Each method may be optimal for one problem and diverge for another problem. Besides, methods that perform well in serial computers do not necessarily are highly parallelizable. For instance, direct methods are not highly parallelizable.

In this chapter, first techniques to discretize and partition a domain are presented. Additionally, methods of global and local mesh refinement are discussed. A brief overview of the methods to discretize PDEs, highlighting the finite element method, is introduced in Section 2.2. The next section presents methods to solve systems of linear equations, with emphasis to the conjugate gradient method and domain decomposition methods. This section is followed by a summary of the methods and techniques discussed.

2.1 Domain Discretization

Numerical methods based on the spatial subdivision of a domain into polyhedra or elements (mesh-based methods) imply the need for generating a grid or mesh[1]. There are two fundamental classes of grids in multidimensional regions, namely, structured and unstructured meshes. These classes differ in the way the mesh points are locally organized. If the local organization of the grid points and the form of the grid cells are defined by a general rule, the mesh is called structured, otherwise, unstructured [71]. A third class, called multiblock or block-structured, is derived from the two fundamental classes.

Structured grids have an implicit connectivity: each grid point, cell and face may be specified uniquely by its computational coordinates. The data structures lend themselves readily to vectorization. However, structured grids are very hard to produce for general complex geometries and often require the user to carry dense grids all the way into the farfield because there is no way to reduce the number of cells in a given logical coordinate plane [19].

[1]Often the literature refers to structured grids and unstructured meshes. In the context of this work the definition of grid and mesh is interchangeable.

Multiblock or block-structured grids were introduced to avoid the geometric modelling difficulties associated with structured grids in complex geometries. Basically, the domain is divided, without holes or overlaps, into a few contiguous subdomains and a separate structured grid is generated in each block. Grids of this kind can be considered structured at the level of an individual block and unstructured when viewed as a collection of blocks. Multiblock grids are considerably more flexible than structured grids but their generation is more complicated than unstructured meshes. The topology of each block and the interaction between adjacent grid blocks are crucial to obtain an acceptable solution (see [27], [60] and [61] for details).

Unstructured grids consist, in contrast to structured grids, of a nearly absolute absence of any restrictions on grid cells, organization or structure, providing the most flexible tool for the discrete description of a geometry. They can, in principle, be composed of cells of arbitrary shapes built by connecting a given point to an arbitrary number of other points. They allow a natural approach to local adaptation by either insertion or removal of nodes.

An unstructured mesh inherently possesses more geometric flexibility than a structured grid. Therefore, it may require many fewer cells to model adequately a given geometry, decreasing memory requirements. Unstructured mesh generation is in theory qualitatively the same for complex as well as simple domains. Therefore, more of the grid generation can be automated, speeding the generation process considerably. For applications with complex geometry or requiring rapid turnaround time, the unstructured formulation appears to be the method of choice.

In structured grids, the neighbours of the points and cells are defined directly from the indices. For example, the left (west) and right (east) neighbours of the point (i, j) in a structured grid are defined by the points $(i-1, j)$ and $(i+1, j)$,

respectively. This is lost in unstructured grids and makes coding and optimization harder. Extra memory is required to store information about the connection between cells and nodes of the mesh.

Unstructured grids are usually used with finite element methods and, increasingly, with finite volume methods. Computer codes for unstructured grids are more flexible. They do not need to be changed when the grid is locally refined or when elements or control volumes of different shapes are used [42]. The main advantages of unstructured grids lie in their ability to represent complex geometries and the natural grid adaptation by the insertion or removal of nodes. However, the numerical algorithms based on an unstructured grid topology are more complicated and are the most costly in terms of operations per time step and memory per grid point.

2.1.1 Sequential and Parallel Mesh Generation

Mesh generation is an active area of research by itself. With the availability of more versatile field solvers and powerful computers, analysts have attempted the simulation of ever-increasing geometrical and physical complexity. At some point (probably around 1985), the main bottleneck in the analysis process became the grid generation itself. The late 1980s and 1990s have seen a considerable amount of effort devoted to automatic grid generation, resulting in a number of powerful and, by now, mature techniques [72].

There is extensive literature on both structured and unstructured grid generation (e.g. [72, 71, 73, 116, 68, 60]) and several web pages that attempt to compile some of the current literature and software available, e.g. [89] and [104]. Owen [90] presents a survey of some of the fundamental algorithms in unstructured mesh generation. A discussion and categorization of triangle, tetrahedral, quadrilateral

and hexahedral mesh generation methods in use in academia and industry is included. An informal survey of available mesh generation software is also provided comparing some of their main features.

A relatively new research area is parallel mesh generation where, in general, the original problem is decomposed into smaller subproblems which are solved (i.e. meshed) concurrently using a number of processors. The subproblems can be formulated to be either tightly coupled, partially coupled or even decoupled. The coupling determines the intensity of the communication and the amount and type of synchronization required between the subproblems. Chrisochoides [29] presents a survey of parallel unstructured mesh generation methods. They show that it is possible to develop parallel meshing software using off-the-shelf sequential meshing codes without compromising in the stability of parallel mesh generation methods.

There are several dozens of well-known mesh generation tools which may be stand-alone applications or part of larger pre-processing software suites. Some well-established industry standard tools are: Gridgen [52], ANSYS with ANSYS ICEM CFD [5], Fluent with GAMBIT [46], MSC PATRAN [82], AutoForm with AutoForm-AutoMesher [7] and Altair with HyperMesh [3].

2.1.2 Mesh Partitioning

Partitioning a mesh into a number of pieces is equivalent to partitioning a graph since a mesh can be described by a graph. To balance the computation among processors and minimize communication, the partitioning of a mesh for parallel computation must be done so that the number of nodes assigned to each subdomain (or piece) is almost the same and the number of connected cells (or elements) assigned to different processors is minimized, i.e. the interface is minimal. A large number of efficient partitioning heuristics have been developed during recent years.

Some of the most used packages are METIS [64, 65, 66], JOSTLE [123], CHACO [57, 56] and PARTY [94].

2.1.3 Mesh Refinement

The initial mesh used to solve a specific problem may not be suitable to produce sufficiently accurate or reliable approximations. An improved mesh is obtained through, in a wider sense, refining the structure of the current mesh either uniformly or in specific areas. These are called global and local refinement, respectively. The indiscriminate use of uniform refinement may lead to some inefficiencies [101, pp. 62]. Therefore, a mesh should ideally only be improved in regions where the approximation is not sufficiently accurate, i.e. should be adaptively refined.

The use of adaptive refinement to obtain an improved grid for the discretization of a PDE has been the subject of a vast amount of research in the past few decades (see [77] and references therein). In order to minimize the number of nodes of the grid and obtain an accurate solution, the idea is to automatically construct a mesh which is coarse where the solution is well behaved and fine near singularities, boundary layers, *et cetera.* and has a smooth transition between the coarse and fine parts. A general question in this context is which criterion should be used to determine the elements to refine.

Although both originally structured and unstructured meshes can be locally refined, adaptive refinement is more often applied to unstructured meshes. During local refinement of a structured mesh, the representation in the form of a simple data structure becomes more complex. Computational storage requirements increase in comparison to truly structured grids, since the position, size and shape of the refined regions must be stored [125, pp 16]. In addition, the grid becomes

nonconforming, i.e. hanging nodes are introduced and hence adequate procedures for the hanging node constrains must be adopted. A conforming (or compatible) mesh is a mesh that does not have hanging nodes. Figure 2.1 shows a conforming and a nonconforming grid. A hanging node may be defined as a node that is the vertex of one cell but is not a vertex of the adjacent cell, as illustrated in Figure 2.2.

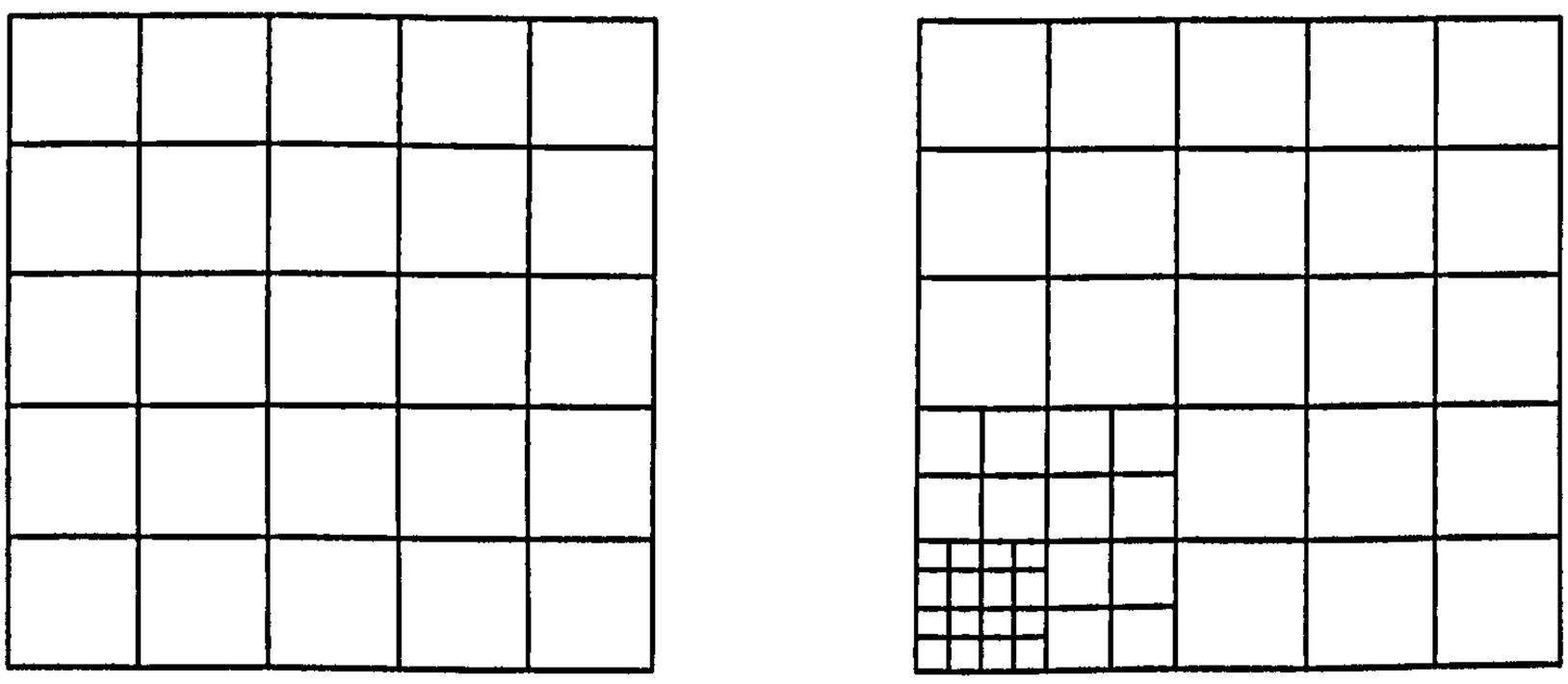


Figure 2.1: Conforming grid (left) and nonconforming grid (right)

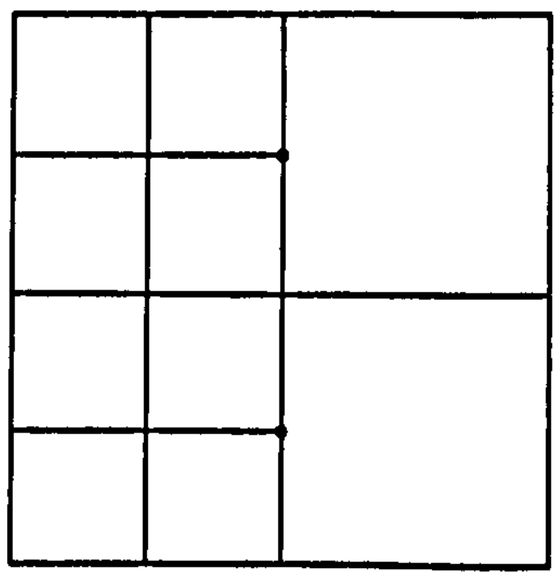


Figure 2.2: Local refinement of an originally conforming structured grid. The nonconformity is due to the hanging nodes (black dots).

Adaptive refinement of unstructured meshes can be performed quite straightforwardly [90, 62]. A number of algorithms has been shown to maintain the main mesh properties, i.e. after the refinement the mesh is still conforming, graded

or smooth (adjacent elements do not differ dramatically in size) and the angles within the elements remain bounded accordingly to the specification of each type of element. Three of the most widely used methods are the regular refinement algorithm of Bank *et al.* [11]; the longest-edge bisection algorithm proposed by Rivara [98] in which a triangle or tetrahedron is bisected by its longest edge; and the newest-node algorithm of Sewell [106] where the edge opposite to the newest-node is bisected. Mitchell [78] briefly describes and presents some properties of these methods. Throughout a series of numerical experiments, it is shown that none of the bisection algorithms is consistently superior. However, the adaptive refinement is superior to using uniform refinement, except on smooth problems.

In a parallel or distributed environment, the complexity of algorithms and software for mesh adaptation is significantly greater than it is for non-adaptive computations. Because a portion of the given mesh and its corresponding equations and unknowns is assigned to each processor, the refinement of an element may cause the refinement of adjacent elements that may be in neighbouring processors. To maintain the load balancing among processors the mesh must be dynamically repartitioned after it is refined. However, dynamical repartitioning of the mesh is usually much faster than the initial partitioning.

Parallelization of adaptive refinement methods involves adaptation, repartitioning and rebalancing. Several algorithms have been developed. The main difficulty is to keep load balance without migrating a large number of elements. Jones and Plassman [63] present a brief review of methods for adaptive refinement and an algorithm to perform parallel adaptive refinement on two-dimensional meshes. Castaños and Savage address these problems in [21, 22, 23] and present PARED [24], an integrated system for the parallel adaptive solution of PDEs showing the advantages of remeshing the grid locally instead of globally. Oliker *et al.* [86] show that the performance of the parallel implementation of the 3D_TAG

unstructured mesh adaptation algorithm deteriorates when the refinement occurs on specific regions due to severe load imbalance. To address this problem they add the PLUM [85] load balancer to 3D_TAG improving dramatically the parallel performance. In other words, the load balance is a very important factor to obtain parallel performance.

The problem of refining and repartitioning meshes has been addressed in several ways. Recently, multilevel approaches that initially solve a small problem on a coarse mesh and use a posteriori error estimator to predict future element densities and partition the mesh has shown to be effective. For example, the method presented by Bank [10] performs the refinement mostly independently on each processor and obtains nearly load-balanced mesh distribution. Another approach, used by Pébay [91], is to make adjacent elements on different processors generate the same refinement. Such approach avoids communication during the refinement step but does not address load balancing issues, which has been shown can deteriorate the performance due to severe imbalance.

2.2 Discretization of Partial Differential Equations

The study of partial differential equations started in the eighteenth century in the work of Euler, d'Alembert, Lagrange and Laplace as a central tool in the description of mechanics of continua and more generally, as the principal mode of analytical study of models in physical science. The analysis of physical models has remained to the present day one of the fundamental concerns of the development of PDEs. Beginning in the middle of the nineteenth century, particularly with the work of Riemann, PDEs also became an essential tool in other branches of

mathematics[2].

Despite the development on the analytical study of PDEs, on a practical level, almost all PDEs are studied by computational means, i.e. continuous problems are transformed into discrete problems and numerical methods are used as solvers. One of the most important and striking phenomena of applications of PDEs in the physical sciences and engineering since the second world war has been the impact of high-speed digital computation. It has drastically changed the structure of practice in applied mathematics and has given rise to new problems and new perspectives.

Various methods of discretization have from time to time been proposed. There are three classical families, namely finite difference[3], finite volume[4] and finite element[5] methods. These methods are mesh-based, i.e. the solution is computed upon a mesh and depends strongly on the mesh itself. Although large progress has been made in the theory and practice of mesh generation, the construction of the mesh is still a very delicate component of the numerical solution of differential equations. For this reason there is an interest in the development of methods that eliminate or reduce the need for a mesh, namely gridless or meshless methods. Meshless methods date back almost thirty years and, in comparison to mesh-based methods, are in their initial stage of development[6].

The discretization of continuous problems has been approached differently by

[2]For an historic overview on the theory of PDEs refer to Brenzi and Browder[18] and references therein.

[3]According to Zienkiewicz *et al.* [131, Chapter 16, pp 446], an excellent summary of the state of the art of finite difference methods may be found in Orkisz [88].

[4]The finite volume method is presented in Bruner [19] and Versteeg [121].

[5]For a comparison between finite element and finite volume methods see [50, 51]. Comprehensive literature on finite element methods may be found in [1, 2, 92, 95, 105, 115, 130].

[6]For an overview of the current state of meshless methods see Atluri *et al.* [6], Babuška *et al.* [9], Weissinger [125] and references therein.

engineers and mathematicians. Engineers often approach the problem more intuitively by creating an analogy between real discrete elements and finite portions of a continuum domain. Mathematicians have developed general techniques applicable directly to differential equations governing the problem, such as finite difference approximations. Since the early 1960s much progress has been made and today the purely mathematical and analog approaches are fully reconciled leading to the finite element method. Table 2.3 shows the process of evolution which led to the present-day concepts of finite element analysis.

The finite element method (FEM) can be analysed mathematically and has shown to have optimal properties for certain types of equations. An important advantage of the finite element method is the ability to deal naturally with arbitrary geometries, due to the use of unstructured grids [42], and with any type of boundary condition. The principal drawback, which is shared by any method that uses unstructured grids, is that the matrices of the linearized equations are not as well structured as those for regular grids, making it more difficult to find efficient solution methods. A key issue is the need to use indirect addressing that inhibits most hardware memory access optimization.

A wide range of problems has been solved by FEM. However, only over the last few years, FEM has become an active area of research on computational fluid dynamics (CFD) [50, 51]. The method has been successfully used in several classes of problems and schemes. Examples are the works by Sherwin and Karniadakis [107, 108], Voyages and Nikitopoulos [122] and Waltz [124]. Despite this development, little work has been done on direct numerical simulation (DNS) for turbulence. Current computations of DNS typically use finite difference schemes or a combination of spectral and finite difference schemes [79]. One of the reasons is that high-order schemes should preferably be used because they are, in principle, more precise when the time step is small enough and the mesh fine enough.

However, Mathieu and Scott [74] suggest that high-order schemes could not involve less computer time for a given accuracy since they are more complicated to implement and require more calculations than low-order methods.

2.3 Systems of Linear Equations

An important aspect of numerical approximation of partial differential equations is the numerical solution of the finite algebraic systems that are generated by the discrete equations. Systems of linear algebraic equations

$$Ax = b, \tag{2.1}$$

where A is an $n \times n$ matrix and x and b are vectors of size n, can be solved by direct or iterative methods. For systems of moderate size, the use of direct methods is often advantageous. However, direct methods are not well suited to parallel implementations mainly due to the inherent sequential nature of the forward and backward substitutions needed. Iterative methods are used primarily for solving large and complex problems for which, because of storage and arithmetic requirements, it would not be feasible or it would be less efficient to solve by a direct method. How to choose an iterative method from the many available is still an open question, since any method may solve a particular system in very few iterations while diverging on another.

Iterative methods for solving linear systems find their origin in the early nineteenth century (work by Gauss). Since the late 1940s, the field has seen an enormous increase in activity mainly due to the introduction of high-speed computing machines. The thesis of David Young [129] (1950) is referred by G. H. Golub and R. Kincaid as *one of the monumental works of modern numerical analysis.* Young's creation, development and analysis of the successive overrelaxation method (SOR)

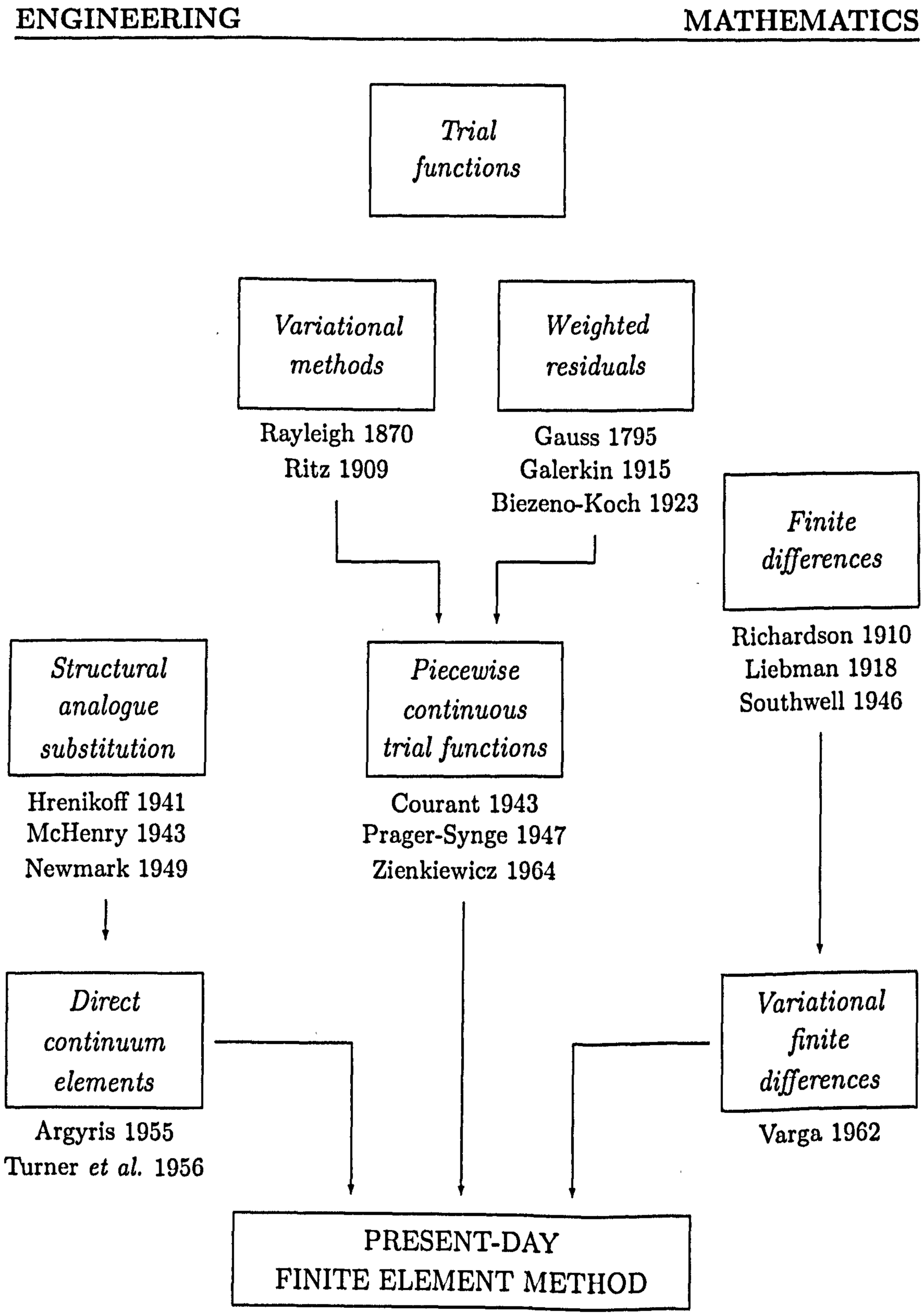


Figure 2.3: Evolution of finite elements analysis (from [131, pp. 3])

is seen as fundamental on the understanding of iterative methods for solving systems of linear equations even though at that time, many methods that are today described as iterative methods were then classified as direct methods. The inclusion of iterative solvers into applications has been a slow process that is ongoing.

An historic review of the main developments in iterative methods over the past century is presented by Saad and van der Vorst [103]. They attempt to relate the contributions to one another, giving the reader a clear picture of what has happened throughout the years. They understand that almost certainly the usage of these methods will increase substantially in application areas, partly due to parallel architectures. Another important observation is that direct and iterative methods used together may result in better, more robust, methods.

An overview of the state-of-the-art of iterative methods, emphasizing preconditioning techniques and the effectiveness of such methods is presented in works by Beauwens [13] and van der Vorst [117]. The latter highlights the importance of Krylov subspace methods and that they have become accepted as powerful tools for solving very large linear systems. In [38], Duff and van der Vorst review some important developments and trends in algorithm design for the solution of linear systems of equations.

The literature about direct and iterative methods is quite vast. This work is mainly concerned about domain decomposition and conjugate gradient methods which will be explored in the following sections. The reader is referred to the publications cited in this section and references therein for a better insight. Also worth mentioning are the books by Varga [120] and Young [128] as classical literature with excellent historical references and the more recent publications by Barret *et al.* [12], Golub and van Loan [49] and Saad [101].

2.3.1 Preconditioning

Preconditioning is used both to improve the rate of convergence and to prevent divergence of the basic iterative method. It is well known that the convergence of iterative methods depends on spectral properties of the coefficient matrix. Hence one may attempt to transform the linear system into one that is equivalent in the sense that is has the same solution, but that has more favourable spectral properties. System (2.1) can be rewritten as

$$M^{-1}Ax = M^{-1}b, \tag{2.2}$$

where M is the *preconditioner*. Equation (2.2) is the case of left preconditioning. Another two options are right and symmetric preconditioning. The choice of M varies from purely black-box algebraic techniques which can be applied to general matrices to problem dependent preconditioners which exploit special features of a particular problem class.

Originally, preconditioners were based on direct solution methods such as the incomplete LU factorization [118]. Preconditioners based on incomplete Cholesky and LU (ILU) factorizations are among the most widely used (see [48]). However, these factorizations are sometimes numerically unstable and, due to the inherently sequential nature of the forward and backward substitutions needed, are not well suited to parallel implementations. Additionally, ILU(0) algorithms (ILU with zero fill-in) breakdown if there is a zero pivot, making it unsuitable for indefinite matrices.

A number of alternative techniques can be developed that are specifically targeted at parallel environments. The simplest approach is to use a Jacobi preconditioner that consists of the diagonal of A. To enhance performance, this preconditioner can itself be accelerated by polynomial iteration, i.e. a second level of preconditioning called *polynomial preconditioning*. For more details see Saad [101, Chapters

12 and 13]. Another technique, similar to ILU but much more amenable to parallelization, is the sparse approximate inverse (SPAI) preconditioner. Further choice of parallelizable preconditioners include element-by-element preconditioners.

Another class of preconditioners that has received attention in the past years is based on domain decomposition methods. In particular, Additive Schwarz preconditioners are of interest in this study (details in Section 2.3.3). For a detailed overview of preconditioning techniques see, for instance, [16] and [101].

2.3.2 Conjugate Gradients

The method of conjugate gradients (CG) is often the method of choice for solving Equation (2.1) when A is a sparse symmetric positive definite (SPD[7]) matrix. It is remarkably fast when compared to other methods like steepest descent, does not require the specification of any parameters and, in the absence of round-off errors, will provide the solution in at most n steps. Furthermore, the matrix A does not need to be explicitly formed; it only requires the result of a matrix-vector product Au for a given vector u.

The method was first proposed by Hestenes and Stiefel [58] in 1952 and was originally conceived as a direct method. Unfortunately, the method was forgotten for several years because, in practice, the numerical properties of the algorithm differed from the theoretical ones [30]. Interest in the method resurged in the early 1970s when the work by John Reid [96] drew renewed attention to the algorithm and several authors proposed CG as an iterative method rather than a direct one.

[7] A matrix A is said *positive-definite* if $x^T Ax > 0$ for every nonzero vector x. If A is positive-definite the shape of its quadratic form is a paraboloid.

CG was developed in terms of a minimization problem of the quadratic function

$$f(x) = \frac{1}{2}x^T Ax - x^T b. \tag{2.3}$$

If A is symmetric and positive definite, $f(x)$ is minimized by the solution of $Ax = b$ (see Section 4.1.3, page 56). CG can also be used to minimize any continuous function $f(x)$ for which the gradient $\nabla f(x)$ can be computed. Applications include a variety of problems, such as engineering design, neural net training and nonlinear regression. The latter is called nonlinear CG and the former linear CG.

Several changes have been proposed to enforced orthogonality conditions and improve performance of CG, especially for the nonlinear case. Murty and Husain [83] proposed an algorithm that incorporates an orthogonality correction as well as an automatic restart. More recently, Day *et al.* [36] presented restart procedures. Chen and Sun [28] report on nonlinear conjugate gradient methods in which the line search procedure is replaced by a fixed formula for the stepsize. For the linear CG method such techniques have a limited improvement on the convergence rate (mainly numerical). However, for the nonlinear CG method the convergence can vary from linear to almost quadratic.

Concus *et al.* [30] presented a generalized conjugate gradient (GCG) method which was based on splitting off from A an approximating symmetric positive definite matrix M that corresponds to a system of equations easier to solve than the original one and then accelerating the associated iteration using CG. This method has become known as the preconditioned conjugate gradient algorithm.

The use of inexact preconditioning is analysed by Notay [84] for the method referred as flexible conjugate gradient (FCG). The main difference to the standard CG lies in the explicit orthogonalization of the search direction vectors. It is reported that depending on the problem, if compared to CG, FCG may bring

more stability at the price of a fairly small overhead. For other problems, FCG is stable with proper parameters.

There are some variants of CG available to solve unsymmetric systems of equations. For instance:

- Bi-CG - Bi-Conjugate, Lanczos, 1952 [70];
- CGS - Conjugate Gradient Squared, Sonneveld, 1989 [112];
- Bi-CGSTAB - Variant of Bi-CG and CGS, van der Vorst, 1992 [119];
- GCGS - Generalized CGS, Fokkema *et al.*, 1994 [47].

CGS is a variant of Bi-CG that often converges twice as fast as Bi-CG. However, large residual norms may appear, which may lead to inaccurate approximate solutions or may even deteriorate the convergence. CGS is often competitive with other established methods such as GMRES [102]. Bi-CGSTAB is seen as a more smooth variant of Bi-CG with the speed of convergence of CGS. GCGS is yet another variant which uses polynomials that are similar to the Bi-CG polynomial, i.e. CGS and Bi-CGSTAB are particular instances of GCGS.

2.3.3 Domain Decomposition

The term domain decomposition, as defined by Smith *et al.* [111, pp ix], has slightly different meanings to specialists within the discipline of PDEs:

- In parallel computing, it often means the process of distributing data from a computational model among the processors in a distributed memory computer. In this context, domain decomposition refers to techniques for decomposing a data structure and can be independent of the numerical solution

method. Data decomposition is perhaps a more appropriate term for this process.

- In asymptotic analysis, it means the separation of the physical domain into regions that can be modelled with different equations with the interfaces between the domains handled by various conditions (e.g. continuity). In this context, domain decomposition refers to the determination of which PDEs to solve.

- In preconditioning methods, domain decomposition refers to the process of subdividing the solution of a large linear system into smaller problems whose solutions can be used to produce a preconditioner (or solver) for the system of equations that results from discretizing the PDE on the entire domain. In this context, domain decomposition refers only to the solution method for the algebraic system of equations arose from the discretization.

In this context, domain decomposition methods refer to divide-and-conquer techniques to solve partial differential equations by iteratively solving subproblems defined on smaller subdomains. The earliest known iterative domain decomposition technique was proposed by H. A. Schwarz in 1870 to prove the existence of harmonic functions on irregular regions which are the union of overlapping subregions. Variants of Schwarz's method were later studied by several authors (see Chan *et al.* [25] for a survey). Domain decomposition methods have the capability of providing numerical solvers which are portable, efficient and highly parallelizable. An overview of these methods, their implementation, analysis and relation to multigrid methods may be found in the book by Smith *et al.* [111].

2.3.3.1 Schwarz Methods

The simplest way to apply the Schwarz method is to divide the domain into overlapping subdomains and solve the equations of each subdomain independently. The literature shows that the convergence rate and therefore the overall parallel efficiency of single-level Additive Schwarz (AS) methods are often dependent on subdomain granularity. The number of iterations required for convergence tends to increase as the number of subdomains increases. The reason lies in the fact that the only means for communicating information between subdomains is through the overlap region.

Two-level methods which employ a global coarse mesh solver to provide global communication are often used. The convergence rate of two-level Schwarz methods also depend on the ratio of the coarse mesh diameter and the size of the overlap. For structured meshes, a grid hierarchy is naturally available. However, for an unstructured mesh the coarse grid may not be easy to generate [26]. Thus, a procedure is needed that generates this grid, as well as the associated interpolation and restriction operators. Moreover, the amount of memory required to store a second, even coarser grid, is a disadvantage. Unlike recursive multilevel methods, a two-level Schwarz method may require too high a resolution on the coarse grid, which makes it less scalable overall. Parallelizing the coarse grid solver is ultimately necessary for high performance [67].

There are a number of variants of Schwarz preconditioner and they differ from each other mainly by the:

- Ordering of the coefficient matrix;
- Method used to treat the points on the interface of the subdomains; and
- Use of a coarse grid or restriction operator.

2.3.3.2 Substructuring Methods

A common ordering adopted is such that, for example

$$A = \begin{pmatrix} A_1 & & & & E_1 \\ & A_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & A_s & E_s \\ F_1 & F_2 & \cdots & F_s & C \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \\ g \end{pmatrix}, \tag{2.4}$$

where s is the number of subdomains; A_i and b_i represent the coefficients that are interior to each subdomain; E_i, F_i and g represent the interface coefficients at each subdomain; C represents interface coefficients common to two or more subdomains; x_i and y are the unknowns that are, respectively, in the interior and on the interface of each subdomain.

The sub-system at each subdomain can be rewritten as

$$\begin{pmatrix} A_k & E_k \\ F_k & C \end{pmatrix} \begin{pmatrix} x_k \\ y \end{pmatrix} = \begin{pmatrix} b_k \\ g \end{pmatrix}, \tag{2.5}$$

and in each subdomain the solution may be expressed as

$$x_k = A_k^{-1}(b_k - E_k y) \tag{2.6}$$

and

$$(C - F_k A_k^{-1} E_k) y = g - F_k A_k^{-1} b_k. \tag{2.7}$$

The matrix

$$S = C - F_k A_k^{-1} E_k \tag{2.8}$$

is called the *Schur complement.* The main difficulty here is how to approximate S. The matrix S is dense even if all constitutive matrices in (2.8) are sparse.

The ongoing research is mainly on how to solve (2.7) since S does not have a particular pattern. For instance, Smith [111] presents a new iterative substructuring algorithm for three dimensions which does not need to form S explicitly.

2.3.3.3 Multigrid

Simple iterative methods (such as the Jacobi method) tend to damp out high frequency components of the error fastest (see [12, pp 91]). This has led to the development of methods based on the following heuristic:

1. Perform some steps of a basic method in order to smooth out the error;
2. Restrict the current state of the problem to a subset of the grid points, the so-called "coarse grid", and solve the resulting projected problem;
3. Interpolate the coarse grid solution back to the original grid, and perform a number of steps of the basic method again.

Steps 1 and 3 are called *pre-smoothing* and *post-smoothing* respectively; by applying this method recursively to step 2 it becomes a true *multigrid* method. Usually the generation of subsequently coarser grids is halted at a point where the number of variables becomes small enough that the direct solution of the linear system is feasible.

The method outlined above is said to be a *V-cycle* method, since it descends through a sequence of subsequently coarser grids and then ascends this sequence in reverse order. A *W-cycle* method results from visiting the coarse grid twice, with possible some smoothing steps in between.

Multigrid methods can be distinguished between algebraic multigrid (AMG) and geometric multigrid (GMG) [126] methods. In algebraic multigrid methods, no

information is used concerning the grid on which the governing partial differential equations are discretized. Therefore, it might be better to refer to them as algebraic multilevel methods instead of multigrid methods. In geometric multigrid methods, coarse grids are constructed from the given fine grid and coarse grid corrections are computed using discrete systems constructed on the coarse grids. Constructing coarse grids by agglomeration of fine grid cells is easy when the fine grid is structured but not if the fine grid is unstructured. That is where AMG becomes useful. The AMG method accelerates convergence by reducing the low-frequency components of the residual using the coarse level correction. [41].

Geometric multigrid [126]

- Coarse grids are constructed from the given fine grid and coarse grid corrections are computed using discrete systems constructed on the coarse grids.
- Standard geometric multigrid has been very successful for the Euler equations but for the Navier-Stokes equations the situation is less satisfactory.

Algebraic multigrid [113]

- The AMG is best-developed for symmetric positive-definite (SPD) problems that arise from the discretization of scalar elliptic PDEs of second order. However, the potential range of applicability is much larger. In particular, AMG has successfully been applied to various nonsymmetric (e.g. convection-diffusion) and certain indefinite problems. Moreover, important progress has been achieved in the numerical treatment of systems of PDEs (mainly Navier-Stokes and structural mechanics applications). However, major research is ongoing and much remains to be done to obtain efficiency and robustness comparable to the case of scalar applications.

- AMG is also a very good preconditioner, much better then standard (one-level) ILU-type preconditioners. Heuristically, the major reason is due to the fact that AMG, in contrast to any one-level preconditioner, aims at the efficient reduction of all error components, short range as well as long range.

- The extra overhead of the setup phase is one reason for the fact that AMG is usually less efficient than geometric multigrid approaches (if applied to problems for which geometric multigrid can be applied efficiently). GMG also has a setup phase but is usually faster than the AMG setup phase.

- An efficient parallelization of classical AMG is rather complicated and requires certain algorithmical modifications in order to limit communication cost without sacrificing convergence significantly.

Linear vs nonlinear multigrid [76]

- Linear methods can deliver superior asymptotic convergence efficiency over nonlinear multigrid methods for fluid flow or radiation diffusion problems. When exact Jacobians are available, similar asymptotic convergence rates per multigrid cycles are observed for equivalent linear and nonlinear multigrid methods. The efficiency gains of the linear methods are largely attributed to the reduced number of costly nonlinear residual evaluations required and the ability to employ a linear Gauss-Seidel smoother in the place of a Jacobi smoother. Therefore, in cases where costly or complicated nonlinear discretizations are employed, the use of linear methods can be advantageous.

- Additional convergence acceleration can be achieved by using both linear and nonlinear methods as preconditioners to a Newton-Krylov method. This approach is particulary beneficial in cases where an inaccurate linearization is employed by the multigrid solvers.

- The required Jacobian storage for the linear multigrid approach can be prohibitive for many applications, particularly in three dimensions.

2.4 Summary

In this chapter, some of the methods used to generate and refine meshes, discretize partial differential equations and solve system of linear equations have been reviewed. It has been shown that the techniques to generate meshes in serial computers are powerful and mature nowadays. Parallel mesh generation is a relatively new area of research. Techniques specific for parallel computers have been developed but it is also possible to successfully adapt serial techniques to parallel meshing.

Another important areas related to meshing are mesh partition and mesh refinement. Many heuristics for the partition and repartitioning of meshes were developed during recent years. The refinement of meshes, mainly the adaptive mesh refinement (AMR), has been a very active area of research. It has been shown that ideally a mesh should only be refined in certain regions to avoid inefficiencies caused by global refinement. In parallel computing, the complexity of the AMR algorithms increases significantly. It involves mesh adaptation, repartitioning and re-balancing. The main goal is to develop algorithms that keep load balance without migrating a large number of elements.

Despite the analytical studies, on practical level almost all PDEs are solved computationally. Various mesh-based methods were proposed and are well developed. More recently, gridless methods have been proposed since meshing is still a delicate and expensive component of the solution, though meshing techniques are powerful and well developed. This work is mainly concerned with mesh-based

finite element methods (FEM). These methods are been used to solve CFD problems only since a few years. They have been proven successful for many problems. One area that has very little work is the direct numerical simulation of turbulence.

A very important aspect of numerical analysis is the solution of systems of linear equations. Problems of moderate size are commonly solved by direct methods whilst large complex problems are solved by iterative methods. Direct methods are not well suited for parallel computers due to their inherent sequential component. Iterative methods have as main advantages the arithmetic requirements and the low amount of storage needed if compared to direct methods. There are several methods available which performance depends on the problem to be solved. Which one to choose for a given problem is still an open question.

Iterative methods are usually accelerated by preconditioners. Preconditioning might be used to improve the rate of convergence of a given method or to prevent divergence. They can be purely back-box algebraic techniques or problem dependent. Most preconditioners are based on direct methods and hence are not well suited for parallel computers.

Domain decomposition methods are considered portable, efficient and highly parallelizable. There are several approaches, distinguished mainly by the ordering of the elements of the coefficient matrix and by the number of meshes adopted. In particular, multigrid methods have optimal properties but they are very difficult to parallelize and the need of multiple meshes is not ideal when using unstructured meshes. Simple domain decomposition methods, that use only one mesh, are more suitable for unstructured meshes. However, the convergence rate of these methods usually deteriorates quickly as the number of subdomains increases. One remedy to time-wise accelerate the convergence of simple domain decomposition methods is the use of inexpensive local solvers such as Krylov methods. This approach has received little attention and is the main subject of this research.

Chapter 3

Discretization of PDEs and the Finite Element Method

The process of obtaining the computational solution of flow problems consists, basically, of two stages: discretization and algebraic equation solver [44]. The first stage converts the continuous partial differential equations and auxiliary (boundary and initial) conditions into discrete algebraic equations. The second stage depends on the discretization technique used. Usually, the second stage requires the solution of a system of algebraic equations and, in some cases, the equation-solving stage is reduced to a marching algorithm.

The discretization process can be divided into spatial and temporal discretization. Time derivatives are resolved using numerical techniques for the solution of initial value problems (IVPs). Spatial derivatives are discretized by either the finite difference, finite element, finite volume, spectral methods or SPH[1] gridless methods, typically. Each of these methods has its advantages and disadvantages. In this work, the finite element method on unstructured meshes is the method of choice because it supports unstructured grids and hence allows easier study of complex geometries and also because it can have certain optimality properties.

[1]Smoothed Particle Hydrodynamics

The finite element method is a numerical procedure for solving physical problems governed by a differential equation or an energy balance. It has two characteristics that distinguish it from other numerical procedures [105]:

1. The method utilizes an integral formulation to generate a system of algebraic equations. Integral (variational) formulation of the problem allows a wider set of data to be admissible than the differential formulation.

2. The method generally uses continuous, piecewise smooth functions for approximating the unknown quantity or quantities.

The second characteristic distinguishes the finite element method from other numerical procedures that utilize an integral formulation. The continuous functions must have only enough continuity in the derivatives to allow the integrals to be evaluated. Functions without continuous first derivative terms can also be solved with a Galerkin method by modifying the higher derivative terms using integration by parts.

As in simple finite difference schemes the finite element method requires a problem defined in a geometrical space (or domain) to be subdivided into a finite number of smaller regions (a mesh). In finite differences, the mesh consists of rows and columns of orthogonal lines; in finite elements more general subdivisions can be used. For example, triangles or quadrilaterals can be used in two dimensions, and tetrahedra or hexahedra in three dimensions. Over each finite element, the unknown variables (e.g. temperature, velocity, etc.) are approximated using known functions; these functions can be linear or high-order polynomial expansions that depend on the geometrical locations (nodes) used to define the finite element shape. The governing equations are integrated over each finite element and the solution assembled over the entire problem domain. As a consequence of

these operations, a finite set of algebraic equations is obtained in terms of a set of unknown parameters over each element [92].

3.1 Domain Discretization - Unstructured Grids

There are two fundamental classes of grids in the solution of problems in multidimensional regions: structured and unstructured. These classes differ in the way the mesh points are locally organized. If the local organization of the grid points and the form of the grid cells are defined by a general rule the mesh is called structured [71], otherwise unstructured. A third class, called multiblock or block-structured, is derived from the two fundamental classes. Figure 3.1 shows a sketch of an unstructured mesh and the notation used to described it and a structured grid.

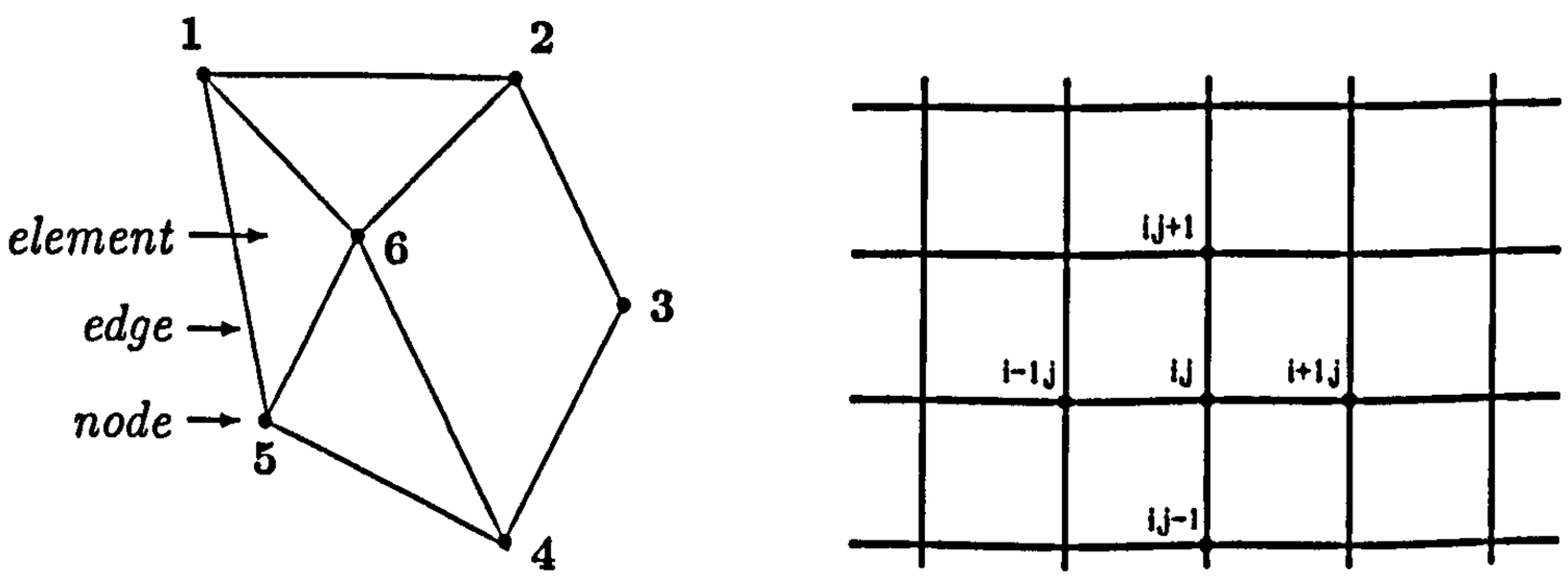


Figure 3.1: Sketch of an unstructured mesh (left) and a structured grid (right)

Unstructured grids consist, in contrast to structured grids, in a nearly absolute absence of any restrictions on grid cells, organization or structure, providing the most flexible tool for the discrete description of a geometry. The overall time required to produce unstructured grids in complex geometries is much shorter than

for structured or multiblock grids. The main advantage of unstructured grids lies in their ability to deal with complex geometries, while allowing one to provide natural grid adaptation by the insertion or removal of nodes. However, the numerical algorithms based on an unstructured grid topology are more complicated and are the most costly in terms of operations per time step and memory per grid point because memory access is indirect.

Since an unstructured grid inherently possesses more geometric flexibility than a structured grid, it may require many fewer cells to adequately model a given geometry than a structured grid, thereby decreasing memory requirements. Grid generation is qualitatively the same for complex as well as simple domains. Therefore, more of the grid generation can be automated, speeding the generation process considerably. For applications with complex geometry or requiring rapid turnaround time, the unstructured formulation appears to be the method of choice.

Unstructured grids can, in principle, be composed of cells of arbitrary shapes built by connecting a given point to an arbitrary number of other points, thereby allowing the application of a natural approach to local adaptation by either insertion or removal of nodes. They also allow excessive resolution to be removed by deleting grid cells locally over regions in which the solution does not vary appreciably.

In structured grids, the neighbours of the points and cells are defined directly from the indices. This is lost in unstructured grids and makes the coding and optimization harder. Location and connectivity of nodes need to be specified explicitly, i.e. a special procedure for numbering and ordering the nodes, edges, faces, and cells of the grid is needed. Extra memory is required to store information about the connectivity among cells of the mesh and because the linearized difference scheme operators are not usually band matrices, it is more difficult to use implicit schemes.

Unstructured grids are usually used with finite element methods and, increasingly, with finite volume methods. Computer codes for unstructured grids are more flexible since they do not need to be changed when the grid is locally refined or when elements or control volumes of different shapes are used [42]. To obtain accuracy in time higher-order formula on a fine grid might be used. However, from a practical perspective, the accuracy that can be achieved for a given execution time is more important than the accuracy alone. The use of low-order formulae and either grid refinement or grid adaptation can improve the accuracy and maintain computational efficiency.

3.2 Weighted Residuals Method

The general weighted residuals method (WRM) statement reads

$$\int_\Omega W^i L(\phi) d\Omega \phi_j = 0, \tag{3.1}$$

where W^i is a *weighting function*, Ω is the domain and $L(\phi)$ denotes an operator, for example, the Laplace operator $L(\phi) = \nabla^2\phi$. The idea of the weighted residuals method is that a residual $R(\phi, x)$ can be multiplied by a weighting function to force the integral of the weighted expression to vanish, i.e.,

$$\int_\Omega W(x) R(\phi, x) d\Omega = 0 \tag{3.2}$$

For example, for the equation

$$\frac{\partial \phi}{\partial x} = f(x) \tag{3.3}$$

$R(\phi, x)$ is given by

$$R(\phi, x) = \frac{\partial \phi}{\partial x} - f(x) \tag{3.4}$$

The unknown ϕ is approximated by

$$\phi(x) = a_1 N_1(x) + a_2 N_2(x) + ... = \sum a_i N_i(x), \tag{3.5}$$

where $N_i(x)$ are called *shape functions* and a_i are unknown parameters.

A system of linear equations in the unknown parameters a_i can be generated by choosing different weighting functions and replacing each in (3.2). This will determine an approximation ϕ on the form of the finite series given in Equation (3.5). Since the number of unknown parameters a is equal to the number of shape functions N, a system of linear algebraic equations with the same number of equations as unknowns will be generated. The existence and uniqueness of the solution to such a system of equations is guaranteed if the boundary conditions associated with the differential equation are correctly posed.

For instance, to approximate a differential equation

$$L\phi + f = 0 \text{ in } \Omega \tag{3.6}$$

with boundary conditions

$$m\phi + s = 0 \text{ on } \Gamma, \tag{3.7}$$

the discretized equations are obtained in the weighted form as

$$\int_\Omega W_i \left(L\phi + f\right) d\Omega + \underbrace{\int_\Gamma \bar{W}_i \left(m\phi + s\right) d\Gamma}_{\text{Boundary Conditions}} = 0 \tag{3.8}$$

3.3 Boundary Conditions

The boundary conditions can be divided into three types, namely Dirichlet, Neumann and mixed[2] boundary conditions. The latter two are known as natural (implicit) boundary conditions and the former as essential boundary condition.

There are two common methods used to apply Dirichlet boundary conditions

$$\phi = s \tag{3.9}$$

[2]The mixed boundary condition is also known as Robin boundary condition.

One is to eliminate the restrained parameters from the assembled equations and solve the reduced system. Its major disadvantage is that it is computationally difficult to program it efficiently. Another is to modify the assembled equations to reflect the boundary constrains and solve the complete system. This modification should consider symmetry, round-off error and the eigenvalue distribution. The former is specially important when Krylov methods are used to solve the linear system of equations.

The derivative boundary conditions can be described in a general form as

$$\frac{\partial \phi}{\partial n} = -m\phi + s, \tag{3.10}$$

where $\partial\phi/\partial n$ is the derivative normal to the boundary, m and s are scalars and ϕ the unknown. If $m = 0$ it is a Neumann boundary condition and if both $m \neq 0$ and $s \neq 0$ it is a mixed boundary condition. The inclusion of the derivative boundary conditions into the finite element analysis is done using an integral (or weak form) over the boundary (see Table 3.1). Details of how to define the components are given in Section 3.5.

Name	Condition	Integral Form
Dirichlet	$\phi = s$	-
Neumann	$\frac{\partial \phi}{\partial n} = s$	$\int_\Gamma N(s)d\Gamma$
Mixed	$\frac{\partial \phi}{\partial n} = -m\phi + s$	$\int_\Gamma N(m\phi - s)d\Gamma$

Table 3.1: Boundary conditions

3.4 Matrix Formulation

The finite element method is based on the numerical approximation of the dependent variable at specific nodal locations (elements) and on the solution of the system of linear equations that is produced. The coefficient matrix is formulated from evaluations performed locally over each element and assembled into a global Galerkin matrix. In other words, the local coefficient matrices obtained from each element are assembled into a large matrix which contains all the local element contributions.

The weighted residuals formulation of the Poisson and convection-diffusion equations in two dimensions is described in Sections 3.4.1 and 3.4.2 respectively. The extension to three dimensions is straightforward.

3.4.1 Poisson Equation

For the Poisson equation,

$$\mu \nabla^2 \phi = f \tag{3.11}$$

the weighted residuals method statement reads

$$\int_\Omega W^i (\mu \nabla^2 N^j - f) d\Omega \; \phi_j = 0. \tag{3.12}$$

Integrating by parts results in

$$\underbrace{-\mu \int_\Omega \nabla W^i \cdot \nabla N^j d\Omega \; \phi_j}_{\text{Diffusion term}} \underbrace{- \int_\Omega \nabla N^i f d\Omega}_{\text{Source term}} = 0. \tag{3.13}$$

Equation (3.11) in two dimensions is equivalent to

$$\mu \left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right) = f(x, y) \text{ in } \Omega \tag{3.14}$$

Adding boundary conditions, the discrete problem over each element is given by

$$T_D\phi_i - T_F - T_M\phi_i - T_S = 0 \tag{3.15}$$

where

$$\begin{aligned} T_D &= \int_{\Omega_e} \left(\frac{\partial W_i}{\partial x}\mu\frac{\partial N_j}{\partial x} + \frac{\partial W_i}{\partial y}\mu\frac{\partial N_j}{\partial y} \right) d\Omega \\ T_F &= \int_{\Omega_e} W_i f(x,y) d\Omega \\ T_M &= \int_{\Gamma_e} W_i m N_j d\Gamma \\ T_S &= \int_{\Gamma_e} W_i s d\Gamma \end{aligned}$$

The equations of each element (local system) may be written as

$$A_e x_e = b_e \tag{3.16}$$

where x_e is equivalent to the dependent variable ϕ,

$$A_e = \mu \int_{\Omega_e} \left(\frac{\partial W_i}{\partial x}\frac{\partial N_j}{\partial x} + \frac{\partial W_i}{\partial y}\frac{\partial N_j}{\partial y} \right) d\Omega - m \int_{\Gamma_e} W_i N_j d\Gamma$$

$$b_e = f \int_{\Omega_e} W_i d\Omega + s \int_{\Gamma_e} W_i d\Gamma$$

Given that ne is the number of elements of the mesh and

$$A = \sum_{e=1}^{ne} A_e, \quad b = \sum_{e=1}^{ne} b_e, \quad x = \sum_{e=1}^{ne} x_e,$$

the assembled system of linear equations (global system) is given by

$$Ax = b \tag{3.17}$$

As an example of assembling the system, consider the regular triangular mesh shown in Figure 3.2. The element connectivity data is given in Table 3.2. The

possible matrix entries that arise because of the mesh topology and the numbering of the nodes are shown in Figure 3.3. Given that A and b are initially zero and that the numbers represent the element number and not the value of the entries and a set of numbers represent summation (for instance $12 \equiv 1{+}2$), after evaluating all eight elements the assembled matrix A of Equation 3.17 is given by

$$\begin{bmatrix} 12 & 2 & 0 & 1 & 12 & 0 & 0 & 0 & 0 \\ 2 & 234 & 4 & 0 & 23 & 34 & 0 & 0 & 0 \\ 0 & 4 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 1 & 0 & 0 & 156 & 16 & 0 & 5 & 56 & 0 \\ 12 & 23 & 0 & 16 & 123678 & 38 & 0 & 67 & 87 \\ 0 & 34 & 4 & 0 & 38 & 348 & 0 & 0 & 8 \\ 0 & 0 & 0 & 5 & 0 & 0 & 5 & 5 & 0 \\ 0 & 0 & 0 & 56 & 67 & 0 & 5 & 567 & 7 \\ 0 & 0 & 0 & 0 & 78 & 8 & 0 & 7 & 78 \end{bmatrix}$$

and the right-hand side vector b by

$$\begin{bmatrix} 12 \\ 234 \\ 4 \\ 156 \\ 123678 \\ 348 \\ 56 \\ 57 \\ 78 \end{bmatrix}$$

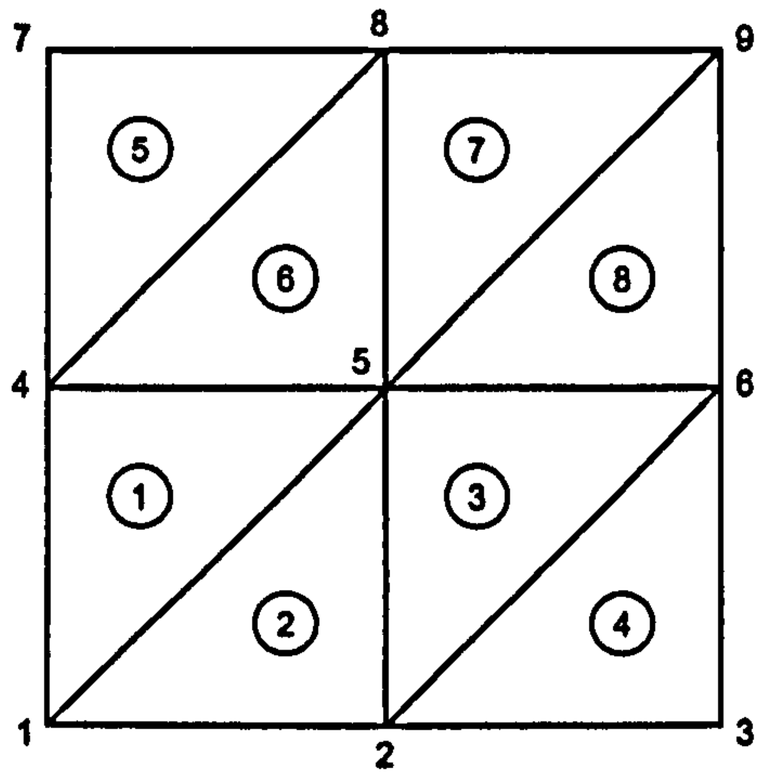


Figure 3.2: Example of mesh

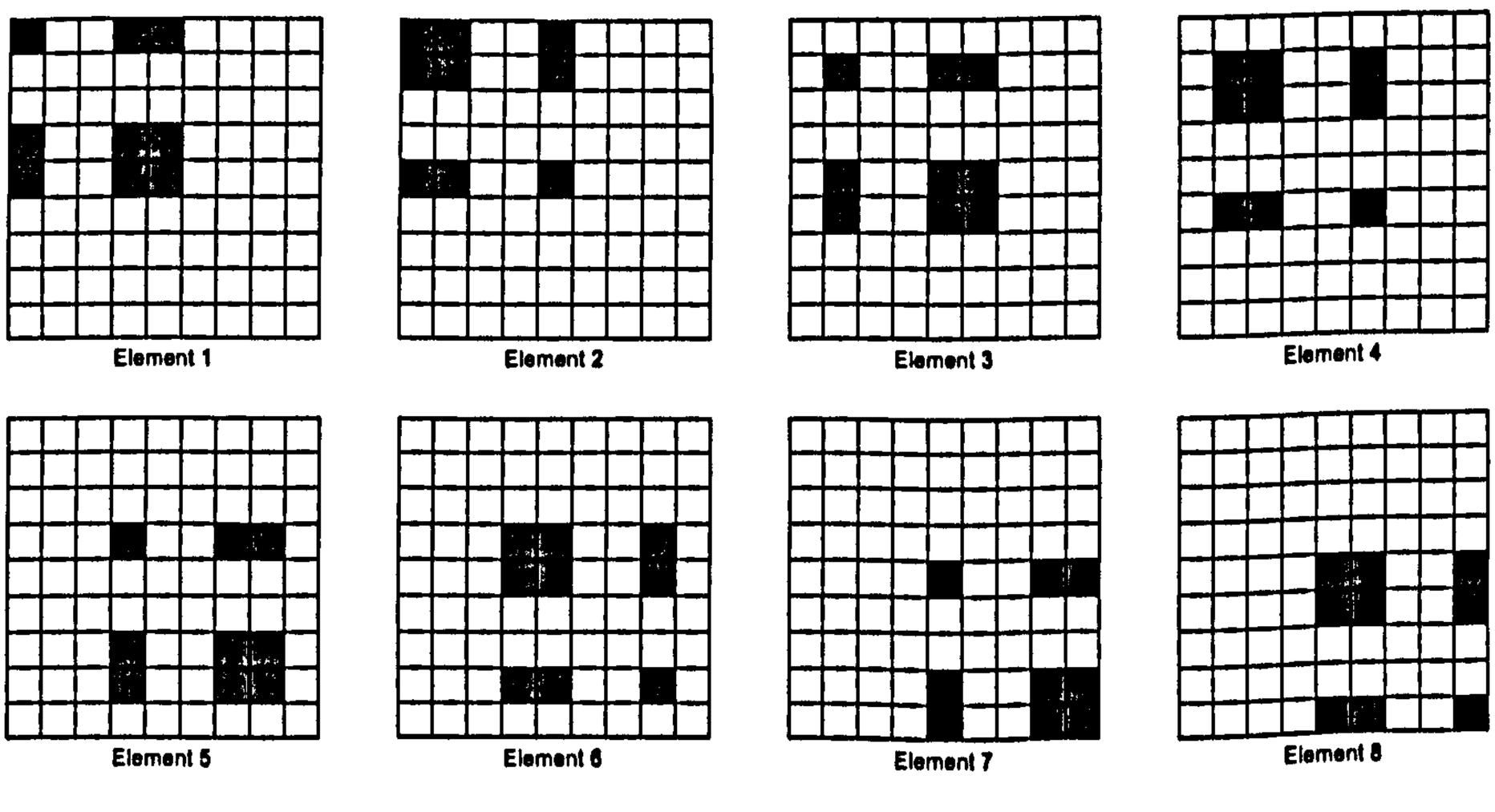


Figure 3.3: Matrix entries

Element	Node 1	Node 2	Node 3
1	1	5	4
2	2	5	1
3	2	6	5
4	2	3	6
5	4	8	7
6	4	5	8
7	5	9	8
8	5	6	9

Table 3.2: Element connectivity data

3.4.2 Convection-Diffusion Equation

For the convection-diffusion equation,

$$-\mu\nabla^2\phi + v\nabla\phi = f \tag{3.18}$$

the weighted residual method statement reads

$$\int_\Omega W^i(-\mu\,\nabla^2 N^j + v\nabla N^j - f)d\Omega\;\phi_j = 0. \tag{3.19}$$

Integrating by parts results in

$$\underbrace{-\mu\int_\Omega \nabla W^i\cdot\nabla N^j d\Omega\;\phi_j}_{\text{Diffusion term}} + \underbrace{v\int_\Omega N^i\cdot\nabla N^j d\Omega\phi_j}_{\text{Convection term}} - \underbrace{\int_\Omega \nabla N^i f d\Omega}_{\text{Source term}} = 0. \tag{3.20}$$

The diffusion and source terms of Equation (3.20) are the same as in Equation (3.13). Equation (3.18) in two dimensions is equivalent to

$$-\mu\left(\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}\right)+\left(v_1\frac{\partial\phi}{\partial x}+v_2\frac{\partial\phi}{\partial y}\right) = f(x,y)\text{ in }\Omega \tag{3.21}$$

Adding boundary conditions, the discrete problem over each element is given by

$$(T_D+T_C)\,\phi_i - T_F - T_M\phi_i - T_S = 0 \tag{3.22}$$

where

$$
\begin{aligned}
T_D &= \int_{\Omega_e} \left(\frac{\partial W_i}{\partial x} \mu \frac{\partial N_j}{\partial x} + \frac{\partial W_i}{\partial y} \mu \frac{\partial N_j}{\partial y} \right) d\Omega \\
T_C &= \int_{\Omega_e} W_i \left(v_1 \frac{\partial N_j}{\partial x} + v_2 \frac{\partial N_j}{\partial y} \right) d\Omega \\
T_F &= \int_{\Omega_e} W_i f(x,y) d\Omega \\
T_M &= \int_{\Gamma_e} W_i m N_j d\Gamma \\
T_S &= \int_{\Gamma_e} W_i s d\Gamma
\end{aligned}
$$

The terms of Equation (3.16) are given by

$$A_e = \mu \int_{\Omega_e} \left(\frac{\partial W_i}{\partial x} \frac{\partial N_j}{\partial x} + \frac{\partial W_i}{\partial y} \frac{\partial N_j}{\partial y} \right) d\Omega + \int_{\Omega_e} W_i \left(v_1 \frac{\partial N_j}{\partial x} + v_2 \frac{\partial N_j}{\partial y} \right) d\Omega - m \int_{\Gamma_e} W_i N_j d\Gamma$$

$$b_e = f \int_{\Omega_e} W_i d\Omega + s \int_{\Gamma_e} W_i d\Gamma$$

3.5 Galerkin and Petrov-Galerkin Methods

The type of weighting function chosen depends on the type of weighted residual technique selected. In the Galerkin procedure (GFEM) the shape function N_i is chosen as a polynomial and $W_i = N_i$. The Petrov-Galerkin (PGFEM) procedure represents a generalization of the Galerkin procedure and both N_i and W_i are taken polynomials, but $W_i \neq N_i$. For operators that exhibit first-order derivatives PGFEM may be superior to GFEM [72].

The Petrov-Galerkin weighting function in two dimensions is defined as

$$W_i = N_i + \beta_i \tag{3.23}$$

where

$$\beta_i = \frac{\alpha h}{2\|v\|_2}\left(v_1\frac{\partial N_i}{\partial x} + v_2\frac{\partial N_i}{\partial y}\right), \tag{3.24}$$

$$\alpha = coth Pe_h - \frac{1}{Pe_h}, \tag{3.25}$$

$$Pe_h = \frac{\|v\|_2 h}{2\mu}, \tag{3.26}$$

h is the element size, v_i are the velocities, Pe_h is the mesh Peclet number and μ the diffusion coefficient. Note that if β is zero the Petrov-Galerkin method is equivalent to the Galerkin method. The mesh Peclet number measures the strength of convection versus diffusion relative to the mesh size. The greater the mesh Peclet number the greater the dominance of convection. For the convection-diffusion equation GFEM exhibits problems with spurious oscillations whenever $Pe_h > 1$ [43].

The size of the element is the distance between one of the nodes to its opposite face or edge in the direction of the velocity vector v. The node and face or edge are chosen so that the vector v crosses the node and the opposite face or edge. Figure 3.4 sketches the size of a triangle.

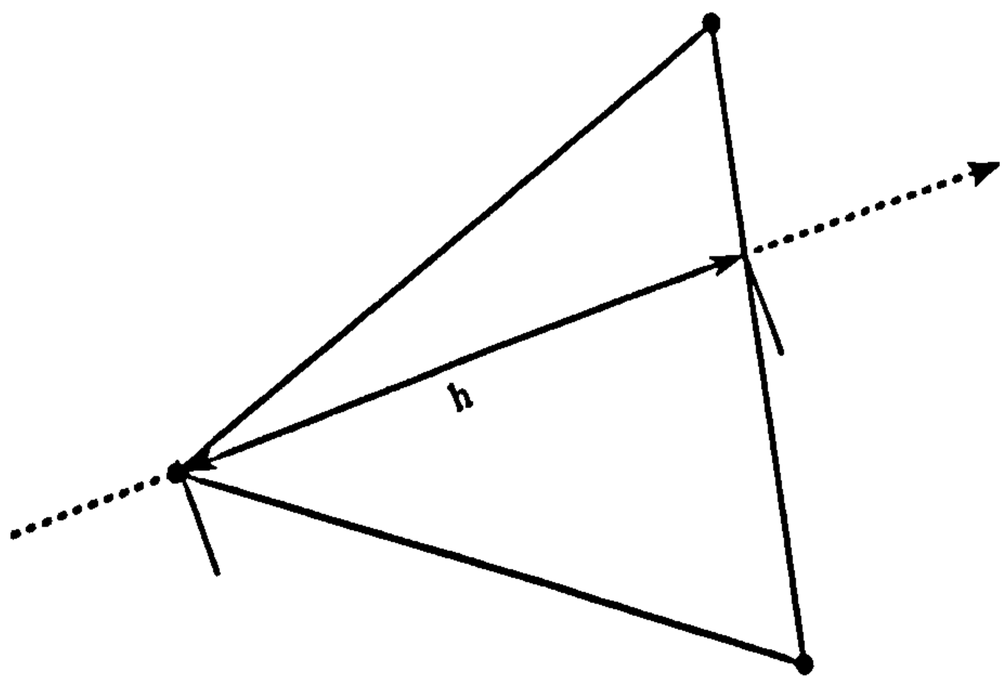


Figure 3.4: Element size h of a triangle

Defining

$$\mathcal{D} = \mu\left(\frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x} + \frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial y} + \frac{\partial N_i}{\partial z}\frac{\partial N_j}{\partial z}\right) \tag{3.27}$$

and

$$C = v_1 \frac{\partial N_j}{\partial x} + v_2 \frac{\partial N_j}{\partial y} + v_3 \frac{\partial N_j}{\partial z} \tag{3.28}$$

as respectively the diffusion and convection terms in three dimensions, the Galerkin and Petrov-Galerkin formulation for linear elements for the Poisson and convection-diffusion equations follows.

Poisson Equation - Galerkin

$$A = \mathcal{D} \int_{\Omega_e} d\Omega + m \int_{\Gamma_e} N_i N_j d\Gamma \tag{3.29}$$

$$b = f \int_{\Omega_e} N_i d\Omega + s \int_{\Gamma_e} N_i d\Gamma \tag{3.30}$$

Convection-Diffusion Equation - Galerkin

$$A = \mathcal{D} \int_{\Omega_e} d\Omega + C \int_{\Omega_e} N_i d\Omega + m \int_{\Gamma_e} N_i N_j d\Gamma \tag{3.31}$$

$$b = f \int_{\Omega_e} N_i d\Omega + s \int_{\Gamma_e} N_i d\Gamma \tag{3.32}$$

Convection-Diffusion Equation - Petrov-Galerkin

$$A = \mathcal{D} \int_{\Omega_e} d\Omega + C \left[\int_{\Omega_e} N_i d\Omega + \beta_i \int_{\Omega_e} d\Omega \right] + m \int_{\Gamma_e} N_i N_j d\Gamma \tag{3.33}$$

$$b = f \left[\int_{\Omega_e} N_i d\Omega + \beta_i \int_{\Omega_e} d\Omega \right] + s \int_{\Gamma_e} N_i d\Gamma \tag{3.34}$$

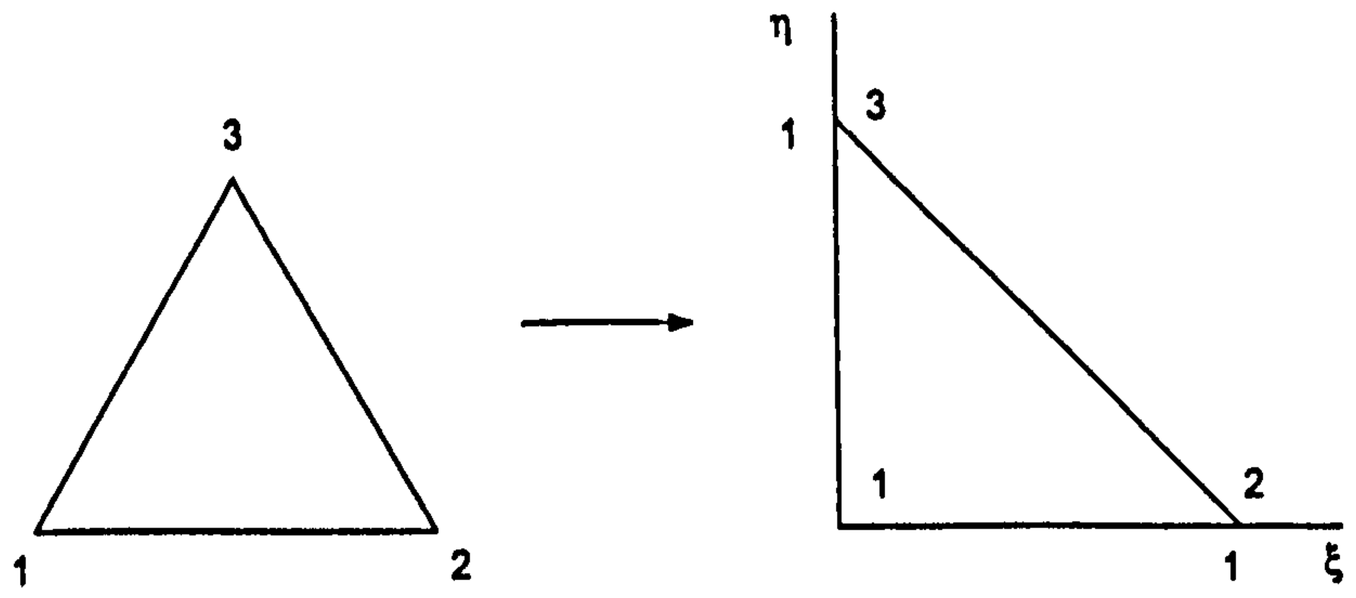


Figure 3.5: Linear triangle

3.6 Shape Functions

3.6.1 Linear Triangular Element

A general linear function in 2D is given by the form

$$f(x, y) = a + bx + cy.$$

The natural geometric object to represent the three unknowns (a, b, c) is the triangle. The shape-functions may be derived by observing the mapping from cartesian (x, y) to local (ξ, η) coordinates (Figure 3.5)

$$x = x_1 + (x_2 - x_1)\xi + (x_3 - x_1)\eta \tag{3.35}$$

or

$$x = N^i x_i = (1 - \xi - \eta)x_1 + \xi x_2 + \eta x_3, \tag{3.36}$$

implying that

$$N^1 = 1 - \xi - \eta, \quad N^2 = \xi, \quad N^3 = \eta. \tag{3.37}$$

The shape-function derivatives, which are constant over the element, can be derived analytically by making use of the chain rule and the derivatives with respect

to the local coordinates

$$N^i_{,x} = N^i_{,\xi}\xi_{,x} + N^i_{,\eta}\eta_{,x}. \tag{3.38}$$

From Equation (3.36), the Jacobian matrix of the derivatives is given by

$$\boldsymbol{J} = \begin{pmatrix} x_{,\xi} & x_{,\eta} \\ y_{,\xi} & y_{,\eta} \end{pmatrix} = \begin{pmatrix} x_{21} & x_{31} \\ y_{21} & y_{31} \end{pmatrix}, \tag{3.39}$$

its determinant by

$$det(\boldsymbol{J}) = |\boldsymbol{J}| = 2A_{el} = x_{21}y_{31} - x_{31}y_{21}, \tag{3.40}$$

and the inverse by

$$\boldsymbol{J}^{-1} = \begin{pmatrix} \xi_{,x} & \xi_{,y} \\ \eta_{,x} & \eta_{,y} \end{pmatrix} = \frac{1}{2A}\begin{pmatrix} y_{31} & -x_{31} \\ -y_{21} & x_{21} \end{pmatrix}. \tag{3.41}$$

The analytical derivatives of the shape-functions with respect to x and y are

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,x} = \frac{1}{2A}\begin{bmatrix} -y_{31} + y_{21} \\ y_{31} \\ -y_{21} \end{bmatrix},$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,y} = \frac{1}{2A}\begin{bmatrix} x_{31} - x_{21} \\ -x_{31} \\ x_{21} \end{bmatrix}.$$

The basic integrals are given by

$$\int_e N^i d\Omega = \frac{A}{3}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

and

$$M_e = \int N^i N^j d\Omega = \frac{\Delta_e}{12}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}.$$

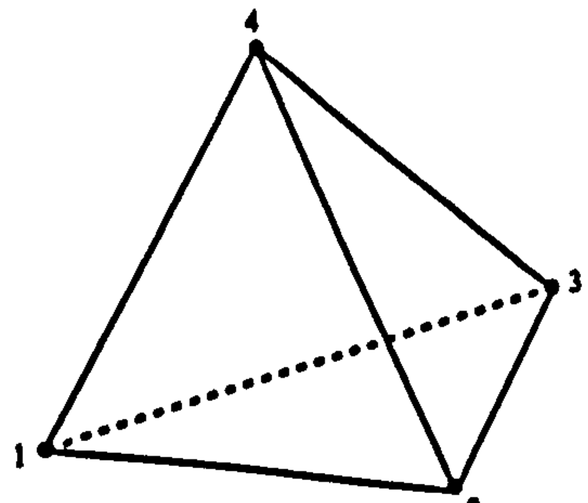


Figure 3.6: Tetrahedron

3.6.2 Linear Tetrahedral Element

The linear trial-function in 3D follows the derivation of the 2D function. The analytical derivatives of the shape-functions with respect to x, y and z are

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \\ N^4 \end{bmatrix}_{,x} = \frac{1}{6V} \begin{bmatrix} T_{11} \\ T_{21} \\ T_{31} \\ -T_{11} - T_{21} - T_{31} \end{bmatrix},$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \\ N^4 \end{bmatrix}_{,y} = \frac{1}{6V} \begin{bmatrix} T_{12} \\ T_{22} \\ T_{32} \\ -T_{12} - T_{22} - T_{32} \end{bmatrix},$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \\ N^4 \end{bmatrix}_{,z} = \frac{1}{6V} \begin{bmatrix} T_{13} \\ T_{23} \\ T_{33} \\ -T_{13} - T_{23} - T_{33} \end{bmatrix},$$

where V is the volume of the tetrahedron and

$$\begin{aligned}
T_{11} &= y_{24}z_{34} - y_{34}z_{24} \\
T_{21} &= y_{34}z_{14} - y_{14}z_{34} \\
T_{31} &= y_{14}z_{24} - y_{24}z_{14} \\
T_{12} &= x_{34}z_{24} - x_{24}z_{34} \\
T_{22} &= x_{14}z_{34} - x_{34}z_{14} \\
T_{32} &= x_{24}z_{14} - x_{14}z_{24} \\
T_{13} &= x_{24}y_{34} - x_{34}y_{24} \\
T_{23} &= x_{34}y_{14} - x_{14}y_{34} \\
T_{33} &= x_{14}y_{24} - x_{24}y_{14}
\end{aligned}$$

The basic integration is given as

$$\int_e N^i d\Omega = \frac{V}{4} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

3.7 Summary

There are two fundamental classes of grids, namely, structured and unstructured. The former is a natural choice for finite differences methods while the latter is most commonly associated with finite element methods. Nevertheless, both types and their variations can be used by different approximation techniques. Unstructured meshes have geometric flexibility and the inherent approach for local refinement that is not present on structured grids. However, approximation techniques using an underneath unstructured mesh are more difficult to code and optimize and are also computationally more expensive.

The process of discretizing a PDE using unstructured meshes and the weighted residual method has been outlined in this chapter. The formulation for the Galerkin and Petrov-Galerkin finite element methods for both the Poisson and convection-diffusion equations has been listed. Additionally, the linear shape function for the triangular element (two dimensions) and the tetrahedral element (three dimensions) were derived. For linear shape functions, the integrals needed might all be computed through analytical functions, i.e. there is no need for numerical integration, making the coding relatively simple.

Chapter 4

Systems of Linear Equations

There are a number of iterative methods for solving symmetric and nonsymmetric systems of linear equations

$$Ax = b, \tag{4.1}$$

where A is square and nonsingular, including: Jacobi, Gauss-Seidel, Successive Overrelaxation – SOR, Symmetric Successive Overrelaxation – SSOR, Conjugate Gradient – CG, CG on Normal Equations – CGNE and CGNR, Conjugate Gradient Squared – CGS, Bi-conjugate Gradient – BiCG, Stabilized Bi-Conjugate Gradient – Bi-CGSTAB, Symmetric LQ – SYMMLQ, ORTHODIR, Generalized Conjugate Residual – GCR, Minimal Residual – MINRES, Generalized Minimal Residual – GMRES, GMRES with Eigenvalues Estimation – GMRESEV, Quasi-Minimal Residual – QMR, Transpose-Free QMR – TFQMR, and Chebyshev Acceleration.

The first four methods are stationary methods and are often called basic iterative methods. See Section 4.2 for the definition of stationary methods. The other methods are nonstationary methods based on the idea of sequences of orthogonal vectors, except for the Chebyshev method that is based on orthogonal polynomials. Most of the nonstationary methods are projection methods based on Krylov subspaces. How to choose a method from the many available is still an open

question since a method may solve a particular system in very few iterations and diverge on another. For a comprehensive overview of the methods cited see for instance [12], [20], [35], [54] and [101].

In this chapter, a brief review of the basic iterative methods Jacobi, Gauss-Seidel and SOR is given. These methods are presented because they are the basis from where many iterative methods were developed. They also are quite easy to understand and implement. However, they are not very efficient by themselves but they can be very efficient when combined with other more-sophisticated methods.

An overview of the steepest descent (SD), conjugate gradient (CG) and domain decomposition (DD) methods is also introduced. The steepest descent method can be seen as a simplified version of CG and, hence, CG can be derived from SD. Before introducing the methods, the definition and relation of residual and error are presented. Following, Krylov subspaces methods, the quadratic form and the derivative of a quadratic form are defined. Most details about derivation and convergence analysis of the methods are omitted since there is an extended literature on the subject.

4.1 Definitions

In this section, the definition of residual, error, Krylov subspace, quadratic function and gradient of a quadratic function is given. These concepts are relevant for the understanding of the methods that follow.

4.1.1 Residual and Error

In solving (4.1) with any iterative method it is assumed that an initial approximation $x^{(0)}$ is given (e.g. $x^{(0)} = 0$). A sequence of approximations $x^{(1)}, x^{(2)}, \ldots$ is then computed. The solution vector is explicitly given by

$$x = A^{-1}b \tag{4.2}$$

if A is invertible. The error vector $\varepsilon^{(k)}$ associated with the $k-$th iteration is defined by

$$\varepsilon^{(k)} = x^{(k)} - x \tag{4.3}$$

which can only be assessed if x is known, i.e. the solution has to be known. In order to evaluate the accuracy of the approximation $x^{(k)}$ the residual

$$r^{(k)} = b - Ax^{(k)} \tag{4.4}$$

can be used. Note that $r^{(k)}$ can be computed at any iteration. Obviously it can be deduced that

$$r^{(k)} = -A\varepsilon^{(k)} \tag{4.5}$$

The error and the residual are two different measures of the accuracy of an approximation $x^{(k)}$. The error $\varepsilon^{(k)} = x^{(k)} - x$ is a vector that indicates how far the approximation $x^{(k)}$ is from the solution x. The residual $r^{(k)} = b - Ax^{(k)}$ indicates how far the approximation $x^{(k)}$ is from the correct value of b.

4.1.2 Krylov Subspace Methods

At present time, Krylov subspace methods are considered to be among the most important techniques for solving large linear systems of equations. These techniques are based on projection processes onto Krylov subspaces. For solving (4.1)

general projection methods seek an approximate solution $x^{(k)}$ from an affine subspace $x^{(0)} + \mathcal{K}_k$ of dimension k by imposing the Petrov-Galerkin condition

$$b - Ax^{(k)} \perp \mathcal{L}_k,$$

where $\mathcal{L}_k$ is another subspace of dimension k and $x^{(0)}$ is an arbitrary initial approximation. There are many projection methods that differ by the subspace $\mathcal{L}$ chosen and by the preconditioner applied [101, pp. 114-279]. A Krylov subspace method is a method for which the subspace $\mathcal{K}$ is the Krylov subspace

$$\mathcal{K}_k(A, r^{(0)}) = \text{span}\{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{k-1}r^{(0)}\}$$

where $r^{(0)} = b - Ax^{(0)}$. The approximations obtained are of the form

$$A^{-1}b \approx x^{(k)} = x^{(0)} + q_{k-1}(A)r^{(0)},$$

where q_{k-1} is a polynomial of degree $k-1$. In the simple case where $x^{(0)} = 0$

$$A^{-1}b \approx x^{(k)} = q_{k-1}(A)b,$$

i.e. $A^{-1}b$ is approximated by $q_{k-1}(A)b$.

Although all the Krylov techniques provide the same type of polynomial approximation the choice of $\mathcal{L}$, i.e. the constrains used to build these approximations, will have an important effect on the iterative technique. Two broad and best-known choices for $\mathcal{L}_k$ are $\mathcal{L}_k = \mathcal{K}_k$ and the minimum residual variation $\mathcal{L}_k = A\mathcal{K}_k$.

4.1.3 Quadratic Form

A quadratic form is a scalar, quadratic function of a vector with the form

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{4.6}$$

where A is a matrix, x and b are vectors, and c is a scalar constant [109, pp 2]. If A is symmetric and positive-definite (SPD[1]), $f(x)$ is minimized by the solution to $Ax = b$.

4.1.4 Gradient of a Quadratic Form

The gradient of a quadratic form is defined as:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix} \tag{4.7}$$

The gradient is a vector field that, for a given point x, points in the direction of the greatest increase of $f(x)$. The function $f(x)$ can be minimized by setting $\nabla f(x)$ equals to zero. Applying Equation (4.7) to Equation (4.6) gives

$$\nabla f(x) = \frac{1}{2} A^T x + \frac{1}{2} Ax - b. \tag{4.8}$$

If A is symmetric, this equation reduces to

$$\nabla f(x) = Ax - b. \tag{4.9}$$

Setting the gradient to zero Equation (4.1) is obtained, i.e. the solution to $Ax = b$ is a critical point of $f(x)$. If A is positive-definite as well as symmetric, then this solution is a minimum of $f(x)$, hence $Ax = b$ can be solved by finding an x that minimizes $f(x)$. If A is not symmetric Equation (4.8) gives the solution to the system

$$\cdot \quad \frac{1}{2}(A^T + A)x = b \tag{4.10}$$

Note that $\frac{1}{2}(A^T + A)$ is symmetric.

---

[1] A matrix A is said *positive-definite* if $x^T Ax > 0$ for every nonzero vector x. If A is positive-definite the shape of its quadratic form is a paraboloid.

4.2 Basic Iterative Methods

Iterative methods used before the age of high-speed computers were usually rather unsophisticated methods based on *relaxation of the coordinates.* In *Richardson's method*, for instance, the next approximation is computed as

$$x^{(k+1)} = x^{(k)} + \omega_k(b - Ax^{(k)}), \quad k = 0, 1, 2, \ldots, \tag{4.11}$$

where $\omega_k > 0$ are parameters to be chosen. Beginning with a given approximate solution, these methods modify the components of the approximation, one or a few at a time and in a certain order, until convergence is reached. Each of these modifications, called relaxation steps, is aimed at annihilation of one or a few components of the residual vector.

Presently, these techniques are rarely used separately. However, when combined with the more efficient methods such as Krylov subspace methods they can be quite successful. Moreover, there are a few application areas where variations of these methods are still quite popular.

4.2.1 Jacobi, Gauss-Seidel and SOR

Assuming that A is nonsingular and moreover that its diagonal elements a_{ii} are all nonzero the matrix A can be expressed as the sum

$$A = D - L - U \tag{4.12}$$

where D is the diagonal of A, $-L$ its strict lower triangular part and $-U$ its strict upper triangular part, as illustrated in Figure 4.1.

Equation (4.1) can then be written as

$$Dx = (L + U)x + b \tag{4.13}$$

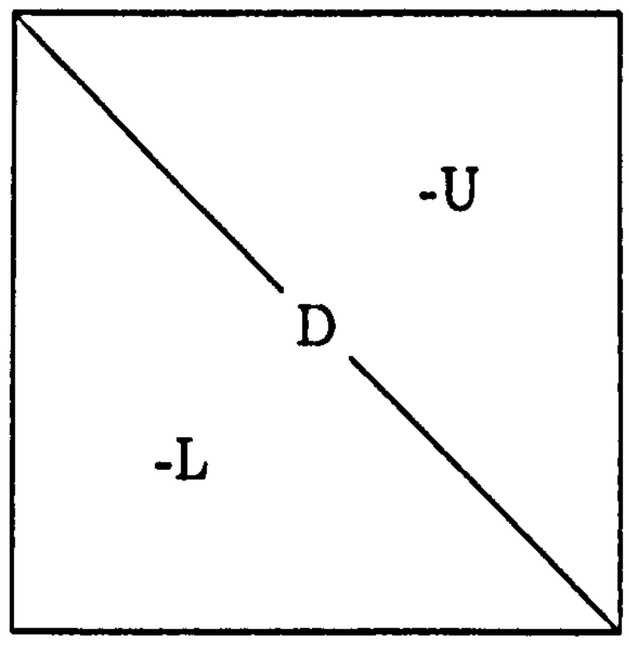


Figure 4.1: Initial partitioning of matrix A

Since all the diagonal entries of A are nonzero, the following iterative method can be derived from Equation (4.13):

$$a_{ii}x_i^{(k+1)} = -\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_i^{(k)} + b_i, \quad i = 1, 2, \ldots, n \tag{4.14}$$

Equation (4.14) can be rewritten as

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_i^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_i^{(k)}\right) \tag{4.15}$$

In matrix notation Equation (4.15) becomes

$$x^{(k+1)} = D^{-1}(L+U)x^{(k)} + D^{-1}b \tag{4.16}$$

The resulting iterative method is called the method of simultaneous displacements or *Jacobi method.* Note that all components of x can be updated simultaneously and the result does not depend on the sequencing of the equations.

The method of successive displacements or *Gauss-Seidel method* differs from the Jacobi method by using new values $x^{(k+1)}$ as soon as they are available as follows

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_i^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_i^{(k)}\right) \tag{4.17}$$

Here the components are successively updated and the sequencing of the equations does influence the result. In matrix notation Equation (4.17) becomes

$$x^{(k+1)} = (D - L)^{-1}Ux^{(k)} + (D - L)^{-1}b \tag{4.18}$$

A backward Gauss-Seidel iteration can also be defined as

$$x^{(k+1)} = (D - U)^{-1}Lx^{(k)} + (D - U)^{-1}b \tag{4.19}$$

The Jacobi, Gauss-Seidel and Richardson methods are special cases of a class of iterative methods which has a general form

$$Mx^{(k+1)} = Nx^{(k)} + b, \tag{4.20}$$

where

$$M - N = A \tag{4.21}$$

is a *splitting* of the coefficient matrix A with M nonsingular. For the method to be practical Equation (4.20) must be "easy" to solve. This is the case, for example, when M is triangular.

Iteration (4.20) is equivalent to

$$x^{(k+1)} = Bx^{(k)} + c \tag{4.22}$$

where

$$B = M^{-1}N = I - M^{-1}A \tag{4.23}$$

$$c = M^{-1}b \tag{4.24}$$

An iteration of the form of (4.22) is called a *stationary iterative method* and B the *iteration matrix*. The error vector $\varepsilon^{(k)}$ associated to the k-iteration is defined by

$$\begin{aligned} \varepsilon^{(k)} &= x^{(k)} - x \\ &= B(x^{(k-1)} - x) \\ &= B\varepsilon^{(k-1)} \end{aligned}$$

Table 4.1 lists the matrix splitting (M and N) and the iteration matrix (B) of the Jacobi and Gauss-Seidel methods.

Jacobi	$M = D$
	$N = L + U$
	$B = D^{-1}(L + U)$
Gauss-Seidel	$M = D - L$
	$N = U$
	$B = (D - L)^{-1}U$

Table 4.1: Jacobi and Gauss-Seidel matrix splitting and iteration matrix B

The Gauss-Seidel iteration is very attractive because of its simplicity. Unfortunately, if the spectral radius of $M^{-1}N$ is close to unity it may be prohibitively slow since the error tends to zero like $\rho(M^{-1}N)$. To rectify this a relaxation parameter ω can be introduced such that

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_i^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_i^{(k)} \right) + (1 - \omega) x_i^{(k)} \tag{4.25}$$

This defines the method of *successive overrelaxation - SOR*. In matrix notation Equation (4.25) becomes

$$x^{(k+1)} = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)x^{(k)} + (D - \omega L)^{-1}\omega b \tag{4.26}$$

The relaxation method has the general form

$$M_\omega x^{(k+1)} = N_\omega x^{(k)} + \omega b \tag{4.27}$$

where

$$M_\omega = D + \omega L \tag{4.28}$$

$$N_\omega = (1-\omega)D - \omega U \tag{4.29}$$

For a few structured but important problems the value of the relaxation parameter ω that minimizes $\rho(M_\omega^{-1}N_\omega)$ is known. However, in more complicated problems it may be necessary to perform a fairly sophisticated eigenvalue analysis in order to determine an optimal ω [49, pp 514].

4.2.2 Convergence Analysis

To each of the iterative methods described, an error vector $\varepsilon^{(k)}$, as defined by Equation (4.3), can be associated. From the very definition of each method the error vectors can be expressed in each case as

$$\varepsilon^{(k)} = B\varepsilon^{(k-1)} = \ldots = B\varepsilon^{(0)}, \tag{4.30}$$

where B is the corresponding iteration matrix for the specific method.

The necessary and sufficient condition for convergence of these methods is that the spectral radius of B must satisfy

$$\rho(B) < 1 \tag{4.31}$$

The asymptotic rate of convergence of the iterative method, or simply rate of convergence, is defined as

$$R_\infty = -\log \rho(B) \tag{4.32}$$

Asymptotically, a reduction by a factor of ϵ of the error ε will be obtained after k^* iterations where

$$k^* \geq -\frac{\log \epsilon}{\log \rho(B)} \tag{4.33}$$

If an iterative method is symmetrisable there always exists an extrapolated version of the basic iterative method such that convergence is guaranteed, since (4.31) is satisfied.

The stationary iterative method, Equation (4.22), is said to be symmetrisable if there is a nonsingular matrix W such that the matrix $W(I-B)W^{-1}$ is symmetric and positive-definite. For a symmetrisable method the matrix $I - B$ has real positive eigenvalues. A sufficient condition for a method to be symmetrisable is that both A and the splitting matrix M are symmetric-positive-definite, since then there is a matrix W such that $M = W^T W$ and

$$W(I - B)W^{-1} = WM^{-1}AW^{-1} = WW^{-1}W^{-T}AW^{-1} = W^{-T}AW^{-1},$$

which again is positive-definite.

The extrapolated method is obtained by

$$x^{(k+1)} = \gamma(Gx^{(k)} + f) + (1 - \gamma)x^{(k)} = G_\gamma x^{(k)} + \gamma f \tag{4.34}$$

where $G_\gamma = \gamma G + (1 - \gamma)I$ and γ is the extrapolation factor.

The optimum value of γ is the one that minimizes $\rho(G_\gamma)$ and is given by

$$\overline{\gamma} = \frac{2}{2 - M(G) - m(G)} \tag{4.35}$$

where $m(G)$ and $M(G)$ are the smallest and largest eigenvalues of G and the spectral radius of G_γ is then

$$\rho(G_\gamma) = \rho(\overline{\gamma}G + (1 - \overline{\gamma})I) = \frac{2\rho(G) - M(G) - m(G)}{2 - M(G) - m(G)} \tag{4.36}$$

Since A is SPD, $\rho(G) = M(G)$ and

$$\rho(G_\gamma) = \frac{M(G) - m(G)}{2 - M(G) - m(G)} < 1 \tag{4.37}$$

The optimum-extrapolated method is then defined by

$$x^{(k+1)} = G_{\overline{\gamma}}x^{(k)} + \overline{\gamma}f \tag{4.38}$$

and is always convergent under the above hypotheses since $\rho(G_{\overline{\gamma}}) < 1$.

4.3 Steepest Descent Method

In 1938-39 Temple [31, pp 31] showed that solving system (4.1) with A SPD is equivalent to the minimization of the quadratic functional

$$f(x) = \frac{1}{2}x^T Ax - b^T x \tag{4.39}$$

(see Section 4.1.4, page 57). One of the simplest strategies for minimizing f is the steepest descent (SD) method where x is iteratively defined by

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} r^{(k)} \tag{4.40}$$

If the residual is nonzero, then there exists a positive α that implies global convergence. The parameter α is chosen by an exact line search such that $x^{(k+1)}$ exactly minimizes $f(x^{(k+1)})$.

A *line search* is a procedure that chooses α to minimize f along a line. In the case of the steepest descent method the search lines are the residuals. From basic calculus, α minimizes f when the directional derivative $\frac{d}{d\alpha} f(x^{(k)})$ is equal zero. By the chain rule

$$\frac{d}{d\alpha} f(x^{(k+1)}) = \nabla f(x^{(k+1)})^T \frac{d}{d\alpha} x^{(k+1)} = \nabla f(x^{(k+1)})^T r^{(k)}$$

Setting this expression to zero, α should be chosen so that $r^{(k)}$ and $f'(x^{(k+1)})$ are orthogonal.

Therefore, given that $g(x) = f'(x)$ is the gradient of $f(x)$,

$$g(x^{(k+1)})^T r^{(k)} = 0 \tag{4.41}$$

Note that $g(x) = -r(x)$ and hence $r(x^{(k+1)})^T r^{(k)} = 0$, i.e. the residuals are orthogonal to each other.

Rewriting $g(x^{(k+1)})$ in terms of $x^{(k)}$ gives

$$g(x^{(k+1)}) = Ax^{(k+1)} - b = A(x^{(k)} + \alpha^{(k)} r^{(k)}) - b \tag{4.42}$$

Substituting the expression for $g(x^{(k+1)})$ into Equation (4.41) and after some algebraic manipulation α is determined by

$$\alpha^{(k)} = \frac{r^{(k)^T} r^{(k)}}{r^{(k)^T} A r^{(k)}} \tag{4.43}$$

Equations (4.4), (4.40) and (4.43) define the steepest descent method:

$$r^{(k)} = b - Ax^{(k)} \tag{4.44}$$

$$\alpha^{(k)} = \frac{r^{(k)^T} r^{(k)}}{r^{(k)^T} A r^{(k)}} \tag{4.45}$$

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} r^{(k)} \tag{4.46}$$

The algorithm, as written above, requires two matrix-vector multiplications per iteration. The computational cost of SD is dominated by matrix-vector products. Fortunately, one matrix-vector multiplication can be eliminated by premultiplying both sides of Equations (4.44) by $-A$ and adding b such that

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} A r^{(k)} \tag{4.47}$$

Although Equation (4.44) is still needed to compute $r^{(0)}$, Equation (4.47) can be used for every iteration thereafter. The product Ar, which occurs in both Equations (4.45) and (4.47), needs only to be computed once. The disadvantage of using this recurrence is that the sequence defined by Equation (4.47) is generated without any feedback from the value of $x^{(k)}$ and hence the accumulation of floating point round-off errors may cause $x^{(k)}$ to converge to some point near x. This effect can be avoided by periodically using Equation (4.44) to compute the correct residual. The method is described in Algorithm 4.3.1.

SD is very attractive for its simplicity but convergence may be prohibitively slow. If $x^{(0)}$ is a good estimate of the solution, e.g. $g(x^{(0)})^T r^{(0)} \approx 0$ and $\kappa(A)$ is small,

Algorithm 4.3.1 Steepest Descent

1. $r_0 = b - Ax_0$
2. for i=1,2,...
3. $\quad \alpha_i = (r_i, r_i)/(Ar_i, r_i)$
4. $\quad x_{i+1} = x_i + \alpha_i r_i$
5. $\quad r_{i+1} = r_i - \alpha_i A r_i$
6. end for

convergence is quadratic (see [31, pp 32]). By Equation (4.41) every two successive search directions are orthogonal to each other making the points $x^{(0)}, x^{(1)}, \ldots$ form a zig-zag path to the solution as shown in Figure 4.2. This results in small corrections to the current estimate $x^{(k)}$ when it is far from the solution and convergence becomes linear.

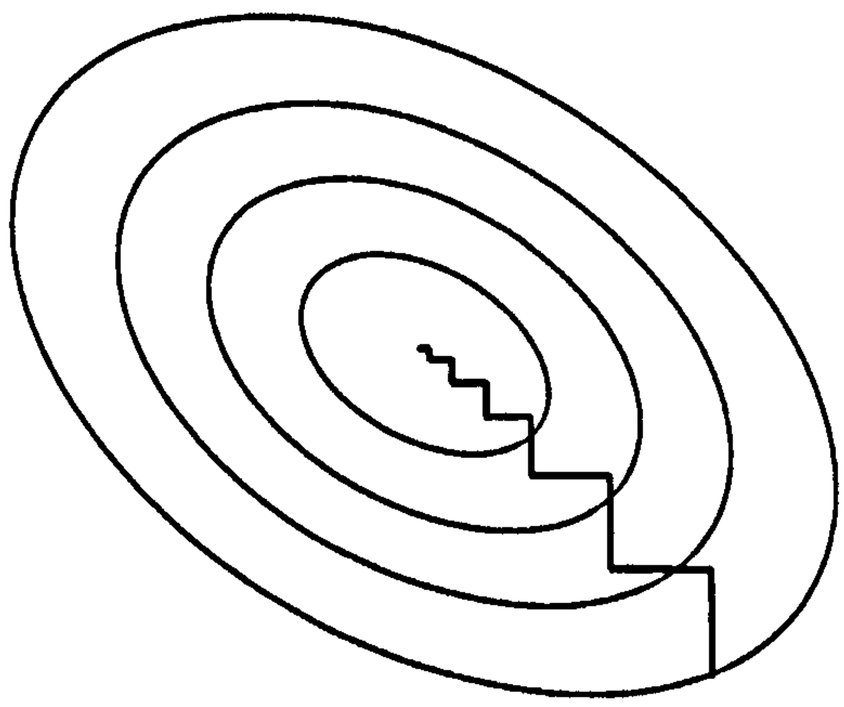


Figure 4.2: Typical zig-zag pattern of steepest descent approximations

4.4 Conjugate Gradient Method

In 1952, Hestenes and Stiefel [58] proposed the conjugate gradient method (CG) for solving systems of linear equations

$$Ax = b \tag{4.48}$$

where A is assumed to be an $n \times n$ symmetric-positive-definite (SPD) sparse matrix. CG has the same principle of SD but uses a different approach to define the search directions.

To increase the rate of convergence of steepest descent, i.e. to avoid taking steps in the same direction as earlier steps, the conjugate gradient method minimizes f along a set of orthogonal search directions $\{p^{(1)}, p^{(2)}, \dots\}$ that are constructed by conjugation of the residuals. The estimate of the solution is updated by

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)} \tag{4.49}$$

By the conjugacy of the search direction vectors $p^{(k)}$ the residual vectors must satisfy

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} A p^{(k)} \tag{4.50}$$

If the $r^{(k)}$'s are to be orthogonal, then it is necessary that $(r^{(k)} - \alpha^{(k)} A p^{(k)}, r^{(k)}) = 0$ and as a result

$$\alpha^{(k)} = \frac{(r^{(k)}, r^{(k)})}{(Ap^{(k)}, r^{(k)})} \tag{4.51}$$

The search directions are recursively defined as a linear combination of the residual and the previous search direction, except for $p^{(0)}$, such that

$$p^{(k)} = \begin{cases} r^{(k)} & \text{for } k = 0 \\ r^{(k)} + \beta^{(k-1)} p^{(k-1)} & \text{for } k > 0 \end{cases} \tag{4.52}$$

A first consequence of the above relation is that

$$(Ap^{(k)}, r^{(k)}) = (Ap^{(k)}, p^{(k)} - \beta^{(k-1)} p^{(k-1)}) = (Ap^{(k)}, p^{(k)})$$

because $Ap^{(k)}$ is orthogonal to $p^{(k-1)}$. Then, the parameter α can be rewritten as

$$\alpha^{(k)} = \frac{(r^{(k)}, r^{(k)})}{(Ap^{(k)}, p^{(k)})}$$

and $\beta^{(k-1)}$ is chosen so that the vector $p^{(k+1)}$ is A-orthogonal, or conjugate, to $p^{(k)}$. A well-known formula for β is the Fletcher-Reeves[2] formula [45], given by

$$\beta^{(k)} = -\frac{(r^{(k+1)}, Ap^{(k)})}{(p^{(k)}, Ap^{(k)})} \tag{4.53}$$

Note that from (4.50)

$$Ap^{(k)} = -\frac{1}{\alpha^{(k)}}(r^{(k+1)} - r^{(k)})$$

and, therefore

$$\beta^{(k)} = \frac{1}{\alpha^{(k)}} \frac{(r^{(k+1)}, (r^{(k+1)} - r^{(k)}))}{(Ap^{(k)}, p^{(k)})} = \frac{(r^{(k+1)}, r^{(k+1)})}{(r^{(k)}, r^{(k)})}$$

CG may be describe as in Algorithm 4.4.1. For more details about the derivation and convergence analysis see for instance [8], [49], [101] and [109][3].

Algorithm 4.4.1 Conjugate Gradient

1. $r_0 = b - Ax_0$
2. $p_0 = r_0$
3. for i=0,1,...
4. $\quad \alpha_i = (r_i, r_i)/(Ap_i, p_i)$
5. $\quad x_{i+1} = x_i + \alpha_i p_i$
6. $\quad r_{i+1} = r_i - \alpha_i Ap_i$
7. $\quad \beta_i = (r_{i+1}, r_{i+1})/(r_i, r_i)$
8. $\quad p_{i+1} = r_{i+1} + \beta_i p_i$
9. end for

[2]For alternate β see, for instance, [83] and [28].

[3]J. R. Schewchuck derives CG in a fairly simple and easy to understand way.

The conjugate gradient method can be remarkably fast when compared to other methods like steepest descent, does not require the specification of any parameters and, in the absence of round-off errors, will provide the solution in at most n steps. However, it is not always highly parallelisable because of the vector norms that have to be computed twice per iteration. Each vector norm needs a global reduction operation, i.e. the synchronization of all processors, compromising the overall speedup as the number of processors grows.

4.4.1 Convergence

In exact arithmetic, CG converges after at most n iterations. In practice, i.e. in finite arithmetic, round-off errors cause the residual to gradually lose accuracy and the search vectors to lose A-orthogonality (conjugacy). The former problem can be dealt with by computing the residual from its very definition $r^{(k)} = b - Ax^{(k)}$ instead of using the recurrence formula. However, the latter problem is not easily solvable. Because of this loss of conjugacy the mathematical community discarded CG during the 1960s. Interest only resurged when evidence for its effectiveness as an iterative procedure was published in the 1970s.

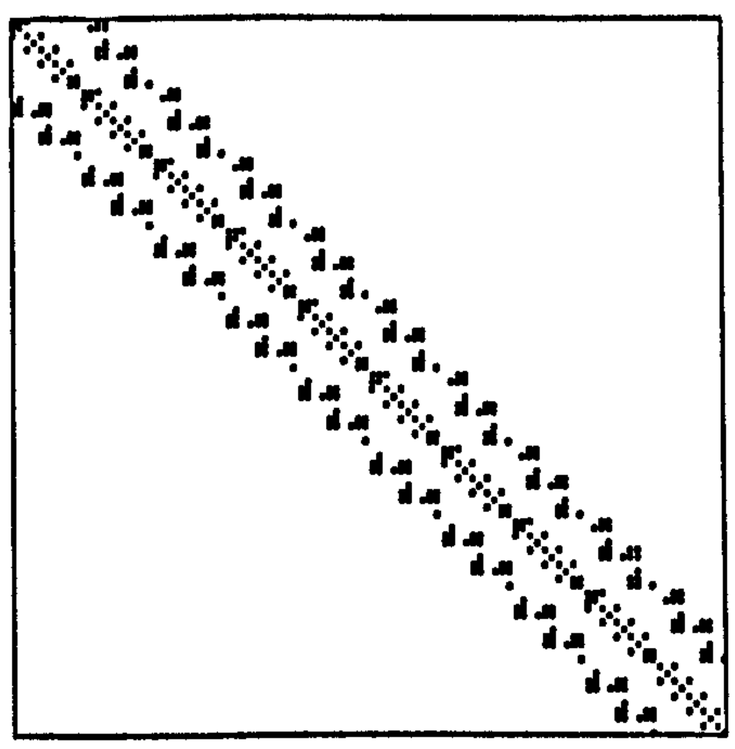
Figures 4.4 and 4.5 show the loss of orthogonality of the residual and search direction vectors for a system where A is the matrix NOS4 described in Figure 4.3 and b is such that the solution is a vector of ones. The same problem is solved with different number of digits of precision. The legend *MatLab* represents the precision of MatLab (double precision) while the numbers 4, 8 and 16 represents the number of digits used to truncate[4] the floating point numbers. Note that as the precision is reduced the solution becomes completely dominated by round-off errors and the approximation diverges as shown in Figure 4.6.

[4]For instance, 1.234567 is truncated and becomes 1.234500 with 4 digits of precision.

Matrix NOS4

Lanczos with partial reorthogonalization
Finite element approximation to a beam structure.
from set LANPRO, from the Harwell-Boeing Collection
`http://math.nist.gov/MatrixMarket`

Structure Plot



Matrix Statistics

Size	Entries	Type
100×100	347	real symmetric positive definite

Nonzeros

total	diagonal	below diagonal	above diagonal	$A - A'$
594	100	247	247	0

Conditioning

Frobenius norm	4.2
condition number (est.)	$2.7e+03$
2-norm (est.)	0.85
diagonal dominance	no

Figure 4.3: Matrix NOS4 from Harwell-Boeing Collection

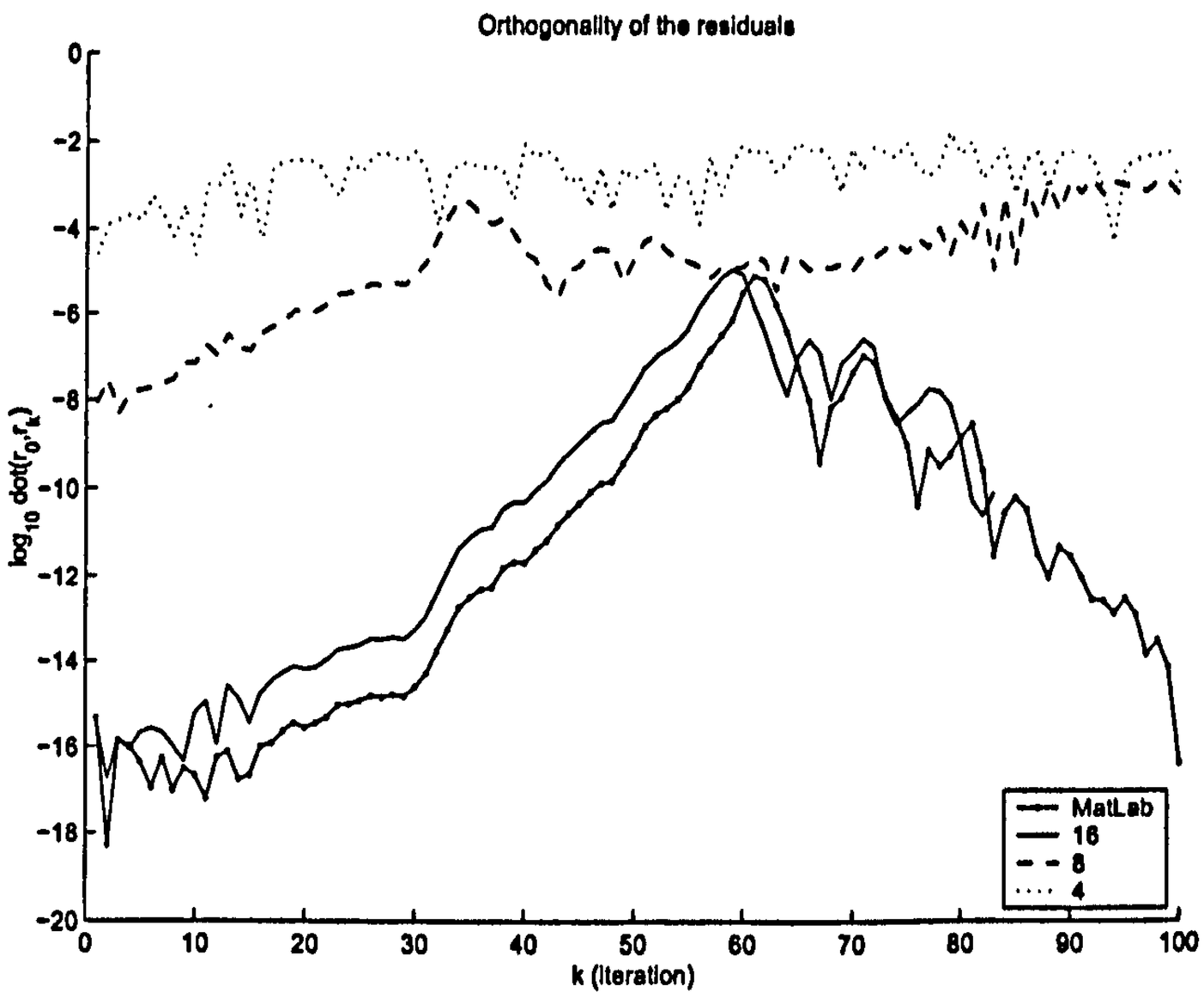


Figure 4.4: Problem NOS4; orthogonality of residual vectors - $r_0^T r_k$; floating point numbers truncated at 4, 8, 16 and double precision (MatLab default) digits.

Though the loss of accuracy of the residual can be reduced by using the definition of r it is not practical. CG using the recursive formula to compute r requires only one matrix-vector multiplication and a couple of saxpy[5] operations, i.e. the main cost of the algorithm is the matrix-vector multiplication. Computing r as $b - Ax$ requires one more matrix-vector multiplication which means that the algorithm requires almost twice the number of flops per iteration. The storage requirements of CG in terms of vectors of size n (x, r, p and Ap) and disregarding the storage of A and b is $4n$. Note that Ap is store to avoid multiplying A by p twice since Ap is needed twice in Algorithm 4.4.1. As it is described, Algorithm 4.4.1 requires the storage of three vectors of size n plus the matrix A and vector b and also requires A to be multiplied by p twice.

[5]Saxpy operation: $ax + y$, where a is a scalar and x and y are vectors

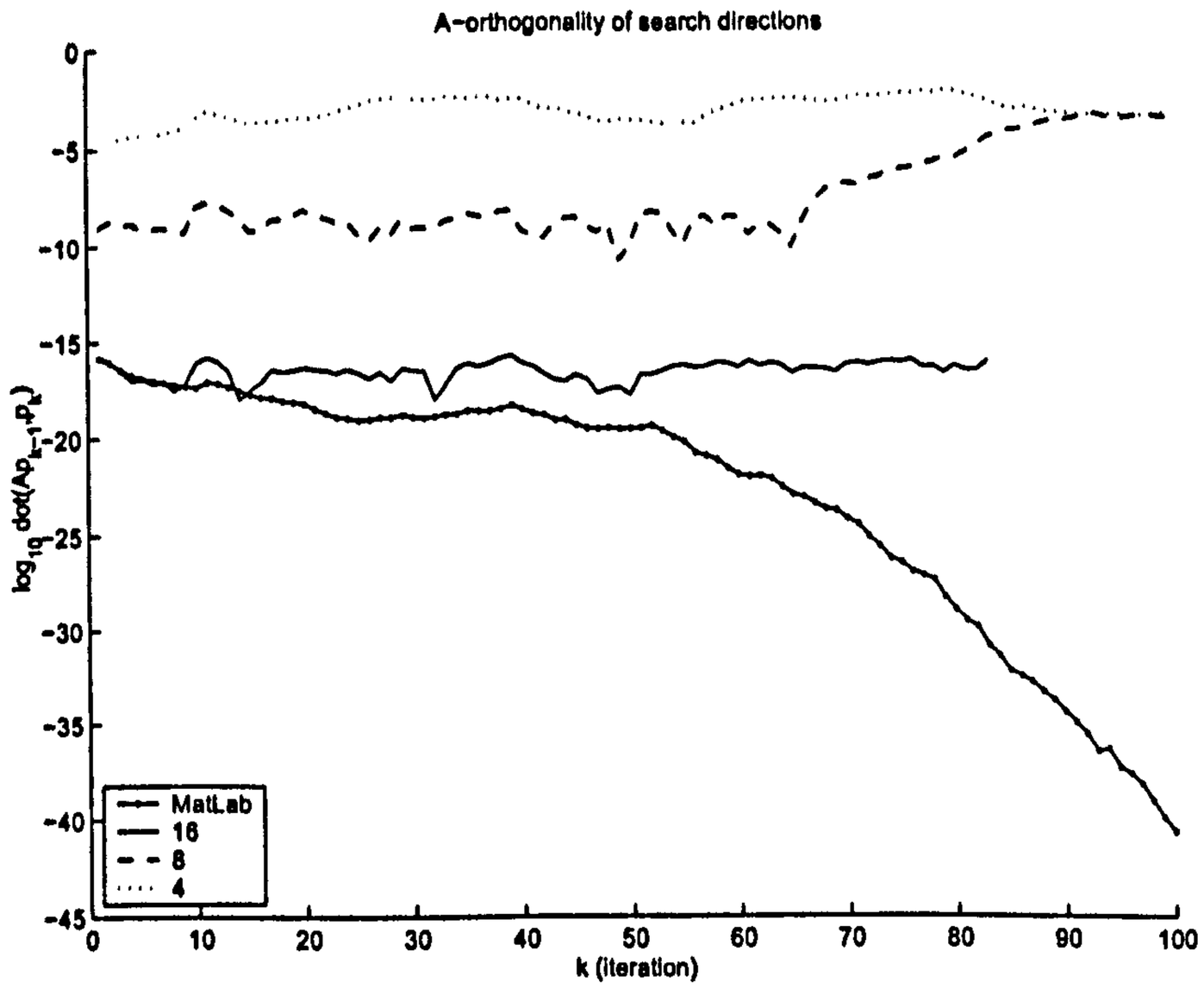


Figure 4.5: Problem NOS4; A-orthogonality of search direction vectors - $(Ap_{k+1})^T p_k$; floating point numbers truncated at 4, 8, 16 and double precision (MatLab default) digits.

4.5 Domain Decomposition Methods

In this section, a brief introduction to domain decomposition methods and the notation used in following chapters are presented. A more general overview can be found in [101, Chapter 13] and a comprehensive overview of the methods, their implementation, analysis and relation to multigrid methods can be found in the book by Smith *et al.* [111].

Domain decomposition methods can be defined as divide-and-conquer techniques to solve partial differential equations by iteratively solving subproblems defined on subdomains. These methods combine ideas of partial differential equations, linear

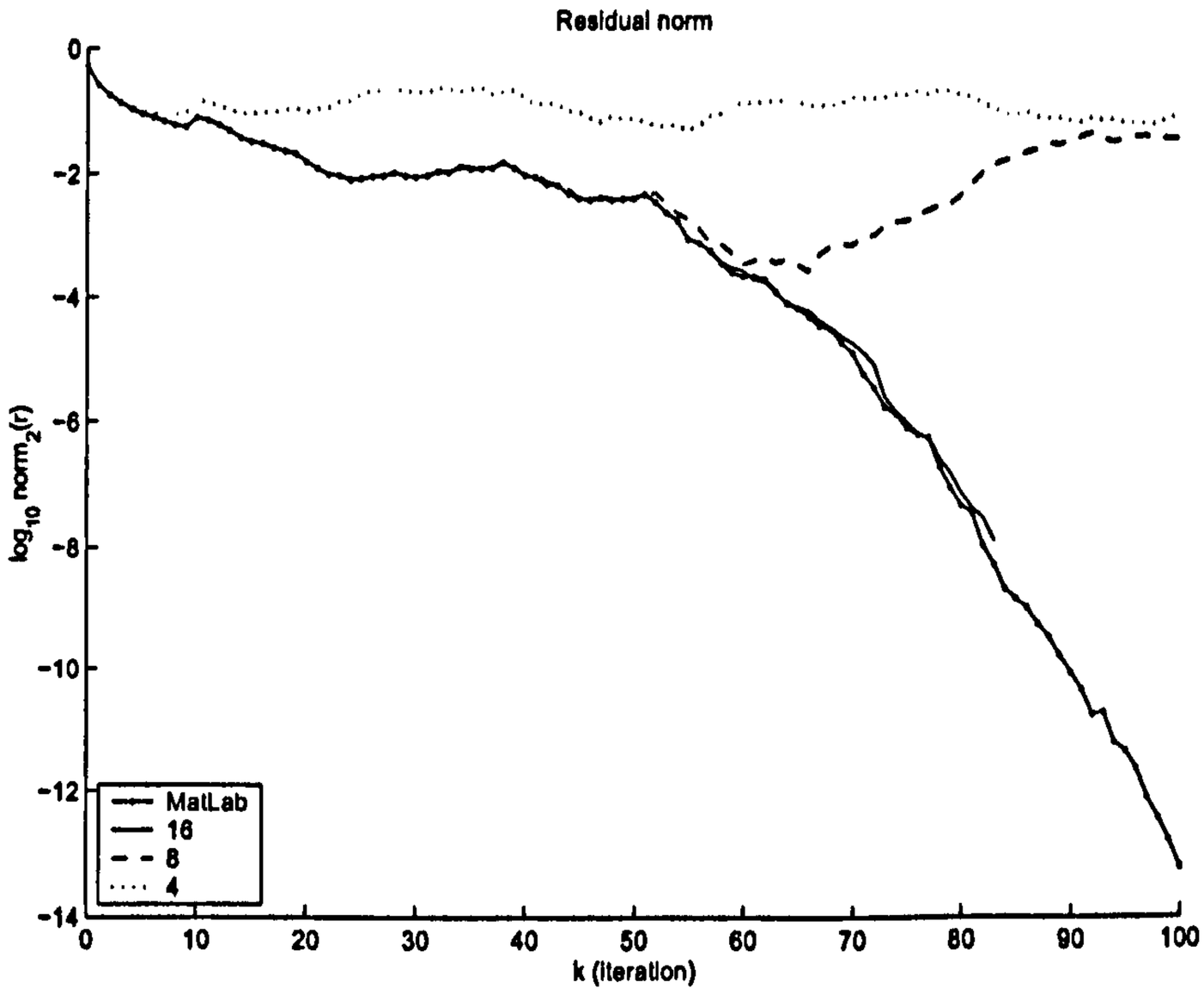


Figure 4.6: Problem NOS4; conjugate gradient residual norm; floating point numbers truncated at 4, 8, 16 and double precision (MatLab default) digits.

algebra, mathematical analysis and techniques from graph theory. Accordingly to Saad [101], among the new classes of numerical methods that can take advantage of parallelism, domain decomposition methods are undoubtedly the best known and perhaps the most promising for certain types of problems.

There are a number of variants of domain decomposition methods that differ mainly by five features:

- *Type of partitioning*: vertex-, edge- or element-based (see Figure 5.11). The type of partition will define the minimum amount of data replicated, i.e. the number of nodes or edges common to two or more subdomains without overlapping. Vertex-based partitions without overlapping do not have any common data among subdomains. Edge- and element-based partitions have

common nodes and common nodes and edges, respectively. The type of partition is especially important in terms of programming the algorithm in parallel since data needed by one process/subdomain may be held by another process/subdomain. See Sections 5.6.2 and 5.6.3 for more details.

- *Grid levels.* The grid levels refer to the number of grids used. Single-level methods use only one grid. Two-level methods make use of both a fine grid and a coarse grid. Multilevel methods use two or more levels of coarse grids. The coarse grids are used as a means of communicating within the subdomains.

- *Overlap.* Domain decomposition methods may be divided into two main classes named overlapping and nonoverlapping methods.

- *Treatment of interface nodes.* There are two main techniques used to resolve the interface nodes. One is to set these nodes as boundary nodes (artificial boundary) and apply boundary conditions to them. Another is to solve the interface nodes first and then solve each subproblem independently.

- *Subdomain solution.* The way that each subdomain problem is solved depends on how the interface nodes are treated. Direct or iterative methods may be used.

Generally, domain decomposition methods can be classified as Schwarz methods and substructuring, or Schur complement, methods. The former are also known as overlapping methods and the latter as nonoverlapping methods. Both classes of methods attempt to solve the problem on the entire domain Ω from problem solutions on the subdomains Ω_i.

4.5.1 Schwarz Methods

Perhaps the simplest domain decomposition algorithms are the one level, explicitly overlapping Schwarz methods. The solution on the whole domain problem is obtained by iteratively solving subproblems on subdomains by updating the interface unknowns which are also called artificial or virtual boundary. There are a number of techniques to define the artificial boundary.

The literature shows that for such methods the number of iterations usually increases as the number of subdomains increases. The standard solution is to turn the method into a two level method, using a coarse grid to provide global communication among subdomains, given by

$$M^{-1} = Q_H^T A_H^{-1} Q_H + \sum_{i=1}^{m} R_{h,i}^T A_{h,i}^{-1} R_{h,i} \tag{4.54}$$

where Q_H is a restriction operator from the fine grid into the coarse grid and $R_{h,i}$ is the restriction operator from the fine grid into the i-th subdomain. The transpose operators Q_H^T and $R_{h,i}^T$ map the coarse grid and the i-th subdomain into the fine grid. On unstructured grids, how to build the coarse grid is not obvious and could be a tedious task. Besides, the amount of memory required to store a second grid, even coarser, could be prohibitive.

4.5.2 Substructuring or Schur Complement Methods

Substructuring or Schur complement methods are a class of domain decomposition methods with no overlapping. Such methods have an interface solver that is not used in overlapping methods. The domain can be partitioned and the nodes

numbered such that the system of equations is written as

$$A = \begin{pmatrix} A_1 & & & E_1 \\ & A_2 & & E_2 \\ & & \ddots & \vdots \\ & & A_s & E_s \\ F_1 & F_2 & \cdots & F_s & C \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \\ g \end{pmatrix}, \tag{4.55}$$

where s is the number of subdomains; A_i and b_i represent the coefficients that are interior to each subdomain; E_i, F_i and b_i represent the interface coefficients at each subdomain; C represents interface coefficients common to two or more subdomains; x_i and y are the unknowns that are in the interior and on the interface of each subdomain, respectively.

The sub-system at each subdomain can be written as

$$\begin{pmatrix} A_k & E_k \\ F_k & C \end{pmatrix} \begin{pmatrix} x_k \\ y \end{pmatrix} = \begin{pmatrix} b_k \\ g \end{pmatrix}, \tag{4.56}$$

and in each subdomain the solution may be expressed as

$$x_k = A_k^{-1}(b_k - E_k y) \tag{4.57}$$

and

$$(C - F_k A_k^{-1} E_k) y = g - F_k A_k^{-1} b_k \tag{4.58}$$

The matrix

$$S = C - F_k A_k^{-1} E_k \tag{4.59}$$

is called the *Schur complement.* The main difficulty here is how to approximate S. A number of Schur complement methods have been proposed over the last few year and research is still ongoing.

4.6 Summary

In this Chapter, a brief review of the basic iterative methods Jacobi, Gauss-Seidel and SOR was presented. An overview of the steepest descent method, conjugate gradient method and domain decomposition methods was also introduced. The definition and relation of residual an error were stated; Krylov subspaces methods and quadratic functions defined.

From the theory presented, it is clear that basic iterative methods lack in efficiency to solve the kind of problems being solved currently. The conjugate gradient method can be very efficient and domain decomposition methods seem a very good option for parallel processing. It was also showed that the convergence rate and sometimes the convergence itself depend strongly on round-off errors. CG, for instance, may lose the orthogonality properties of the residual and search directions vectors due to round-off errors, compromising the efficiency of the method.

Chapter 5

Parallel PDE Solver

In this chapter, general parallel issues such as some of the terminology associated with parallel computing and the classification of parallel computers is addressed. There are many ways to classify parallel computers based on their structure or behaviour. The major classification schemes consider the number of instructions and operands sets that can be processed simultaneously, the internal organization of the processors, the interprocessor connection structure, or the methods used to control the flow of instructions and data through the system. Two of the most common categorizations, namely Flynn's taxonomy and structural classification, are presented. Flynn's taxonomy aims to separate the computers in classes accordingly to the notion of stream of information (instruction and data streams) while the structural classification is based on how the processors are connected to the memory.

In Section 5.3 two main metrics used to measure the performance of parallel algorithms, namely *speedup* and *efficiency*, are defined. Theoretical limits of parallel performance are discussed based on Amdahl's Law. Besides, the concepts of scalability, locality and superlinearity are introduced. In the following section, parallel computational models are presented and the models used in this work (MPI and SPMD) are further elaborated.

Section 5.5 presents empirical and theoretical data about the scalability of the Sun Fire 15K system installed at Cranfield University. A comparison of the parallel performance of the Sun system and an IBM 9076 system running a parallel RRQR factorization is given as well as the description of both systems. The actual latency and transfer rate of a point-to-point (round-trip) and collective communication (broadcast) on the Sun system are computed based on the data obtained experimentally.

In the last section key factors of the development of the parallel PDE solver used in this work are highlighted. This includes the conceptual design of the solver, mesh partitioning, data partitioning and the communication schemes implemented.

5.1 Concepts and Terminology

Some of the terms associated with parallel computing are listed below [53, 55].

Processes and Processors A process is a software executable module and represents an address space and one or more threads. A processor is a piece of hardware containing a central processing unit (CPU) capable of executing a program.

Communication Data exchange among processes. The actual event of data exchange is commonly referred to as communication regardless of the method employed.

Synchronization The coordination of parallel tasks in real time, very often associated with communication. This is often implemented by establishing a synchronization point within an application where a task may not proceed further until another task reaches the same or logically equivalent point. Synchronization

usually involves waiting by at least one task and can therefore cause the wall-clock execution time of a parallel application to increase.

Granularity In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. Coarse granularity is such that relatively large amounts of computational work are done between communication events. Fine granularity is such that relatively small amounts of computational work are done between communication events.

Speedup and efficiency Metrics used to measure the performance of parallel programs (see Section 5.3).

Parallel overhead The extra amount of time required to coordinate parallel tasks.

Massively parallel Refers to the hardware that comprises a given parallel system with many processors. The meaning of "many" has changed over the years and currently is often interpreted as more than 1000.

5.2 Classification of Parallel Computers

Parallel computers are systems consisting of multiple processing units connected via some interconnection network and the software needed to make the processing units work together. Parallel computers can be classified by various aspects of their architectures. A major factor that can be used to categorize such systems is how the processors are connected to the memory. The most popular taxonomy of computer architectures was defined by Flynn in 1966.

Flynn's classification scheme is based on the notion of stream of information (instruction and data streams). The instruction stream is defined as the sequence

of instructions performed by a processing unit. The data stream is defined as the data traffic exchange between the memory and the processing unit. Accordingly to Flynn's classification both the instruction and data streams can be single or multiple and the computer architecture can be classified in four distinct categories: SISD – single-instruction single-data, SIMD – single-instruction multiple-data, MISD – multiple-instruction single-data, and MIMD – multiple-instruction multiple-data.

Conventional single-processor von Neumann[1] computers are classified as SISD systems. Parallel computers are either SIMD or MIMD. When there is only one control unit and all processors execute the same instruction in a synchronized fashion, the parallel machine is classified as SIMD. In a MIMD machine each processor has its own control unit and can execute different instructions on different data, i.e. the processors are independent. In the MISD category, the same stream of data flows through a linear array of processors executing different instructions streams. In practice, there is no viable MISD machine and the class has been added for completeness only. However, some authors have considered pipelined machines as examples of MISD computers [40, Section 1.2] and a concept of systolic arrays from the early 1980s may be viewed as a prototype of a MISD machine.

SIMD (single instruction stream, multiple data stream) refers to a parallel execution model in which all processors execute the same operation at the same time, but each processor is allowed to operate upon its own data. This model naturally fits the concept of performing the same operation on every element of an array, and thus is often associated with vector or array manipulation. Because all oper-

[1]John Von Neumann was a Hungarian Mathematician (1903-1957) who conceived the computer architecture that most modern computer systems are based on today. The Von Neumann architecture consists of inputs, outputs, a CPU and memory.

ations are inherently synchronized, interactions among SIMD processors tend to be easily and efficiently implemented.

MIMD (multiple instruction stream, multiple data stream) refers to a parallel execution model in which each processor is essentially acting independently. This model most naturally fits the concept of decomposing a program for parallel execution on a functional basis; for example, one processor might update a database file while another processor generates a graphic display of the new entry. This is a more flexible model than SIMD execution, but it is achieved at the risk of debugging difficulties called race conditions, in which a program may intermittently fail due to timing variations reordering the operations of one processor relative to those of another.

In terms of memory-processor organization three main groups of architectures can be distinguished: shared-memory architectures, distributed-memory architectures and distributed-shared-memory architectures. Each group has advantages and disadvantages. The former is characterized by their comparative ease of programming with limited scalability. Distributed-memory architectures have potential for high scalability but are more difficult to program. The third group attempts to combine the advantages of both shared and distributed-memory architectures.

Shared-Memory Architectures

The main property of shared-memory architectures is that all processors in the system have access to the same memory, i.e. there is only one global address space. Typically, the main memory consists of several memory modules whose number is not necessarily equal to the number of processors. The processors are connected to the memory modules via some interconnection network as sketched in Figure 5.1.

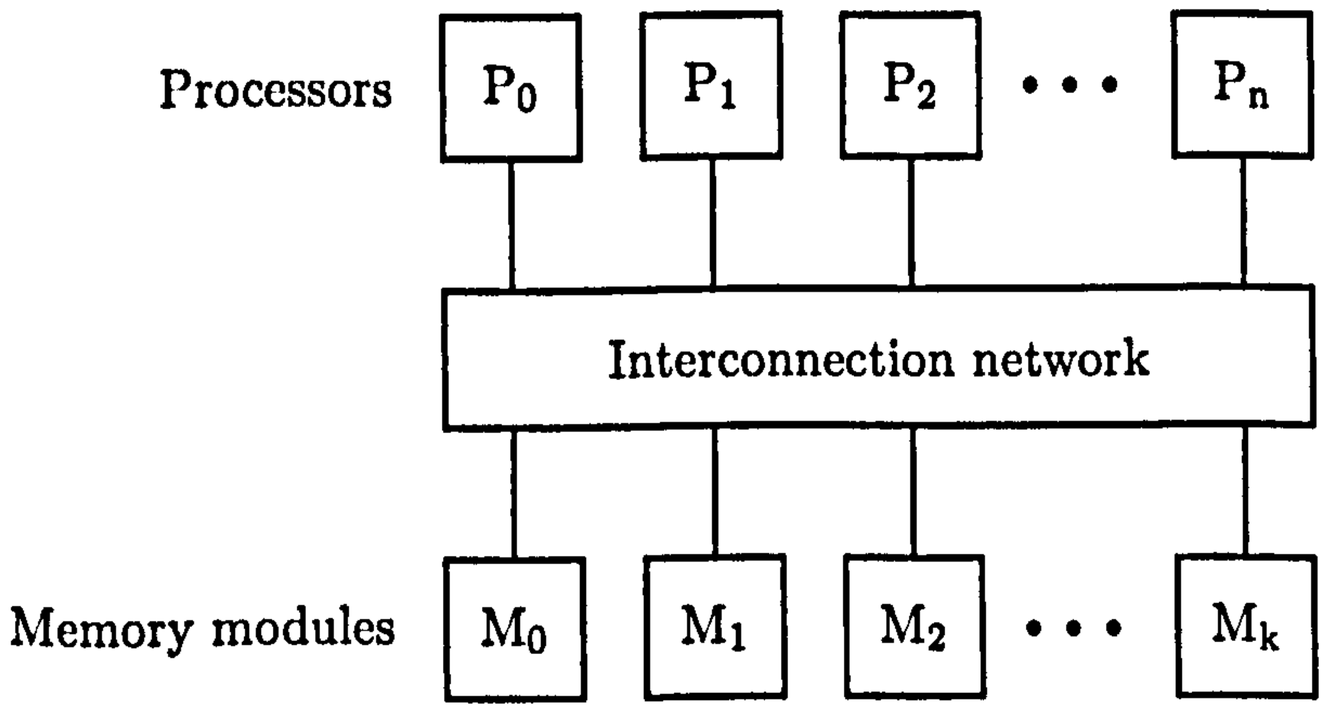


Figure 5.1: Shared Memory Architecture

Shared-memory computers are relatively easy to program since the same data can be accessed by all processors though a method has to be used to control the data access. It is common to have blocks of processors and a portion of the total memory associated with a block on a single board, although each processor can technically access the whole address space (e.g. the SGI Origin architecture). Many applications use message-passing methods as in a distributed system to control the data flow. A main disadvantage of shared-memory machines is the limited number of processors. Most systems do not have more than 64 processors due to both the centralized memory and the interconnection network.

Distributed-Memory Architectures

In distributed-memory computers each processor has its own, private memory. There is no common address space which means that the processors can only access their own memories. Communication and synchronization among processors is done by exchanging messages over the interconnection network (Figure 5.2). In contrast to a shared-memory architecture a distributed-memory machine scales very well since all processors have their own local memory which means that

there are no memory access conflicts.

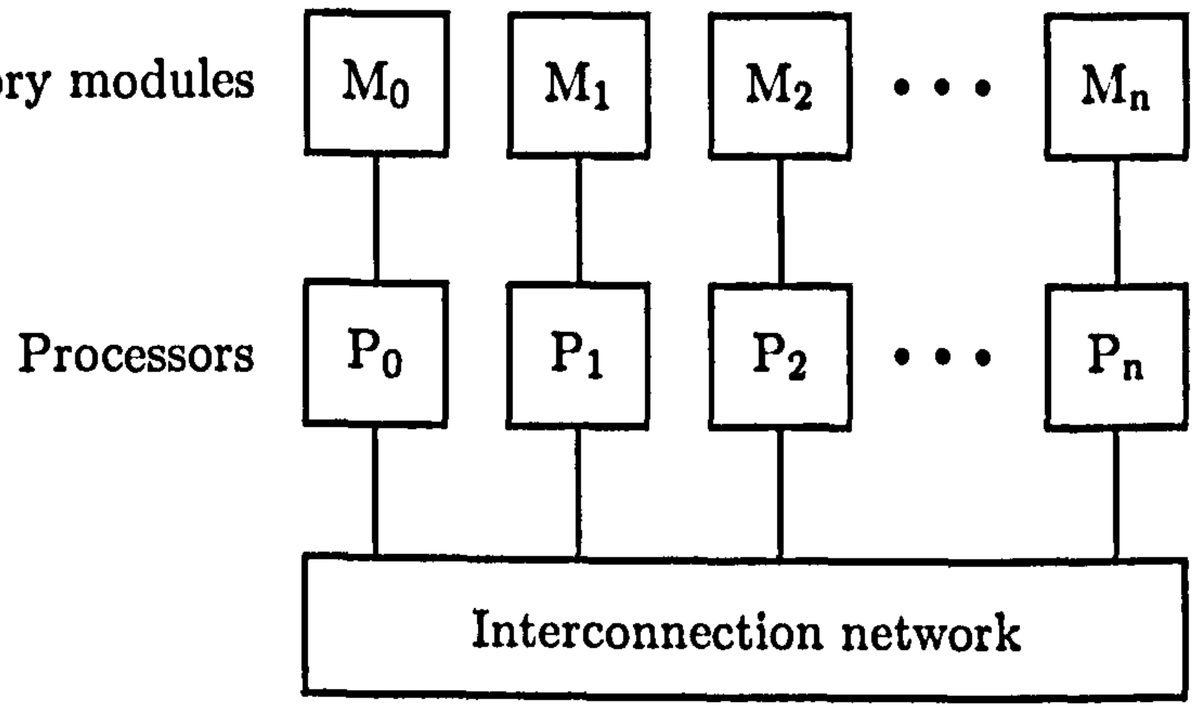


Figure 5.2: Distributed Memory Architecture

Typical representatives of a pure distributed-memory architecture are clusters of computers. In a cluster each node is a complete computer. These computers are connected through a low-cost commodity network (e.g. Ethernet or Myrinet). A major advantage of clusters is the cost to performance ratio. However, to date, clusters are outperformed by most parallel systems.

Distributed-Shared-Memory Architectures

Distributed-shared-memory machines attempt to combine the advantages of the two architectures described above (ease of programming of shared-memory systems and high scalability of distributed-memory systems). In such systems each processor has its own local memory but, contrary to the distributed-memory architecture, all memory modules form one common address space, i.e. each memory cell has a system-wide unique address. In order to avoid the disadvantage of shared-memory computers, namely the low scalability, each processor uses a cache memory which keeps the number of memory access conflicts and the network contention low (Figure 5.3). The use of cache memories requires sophisticated cache

coherence and consistency protocols in order to keep the data in the memory and the copies in the caches updated.

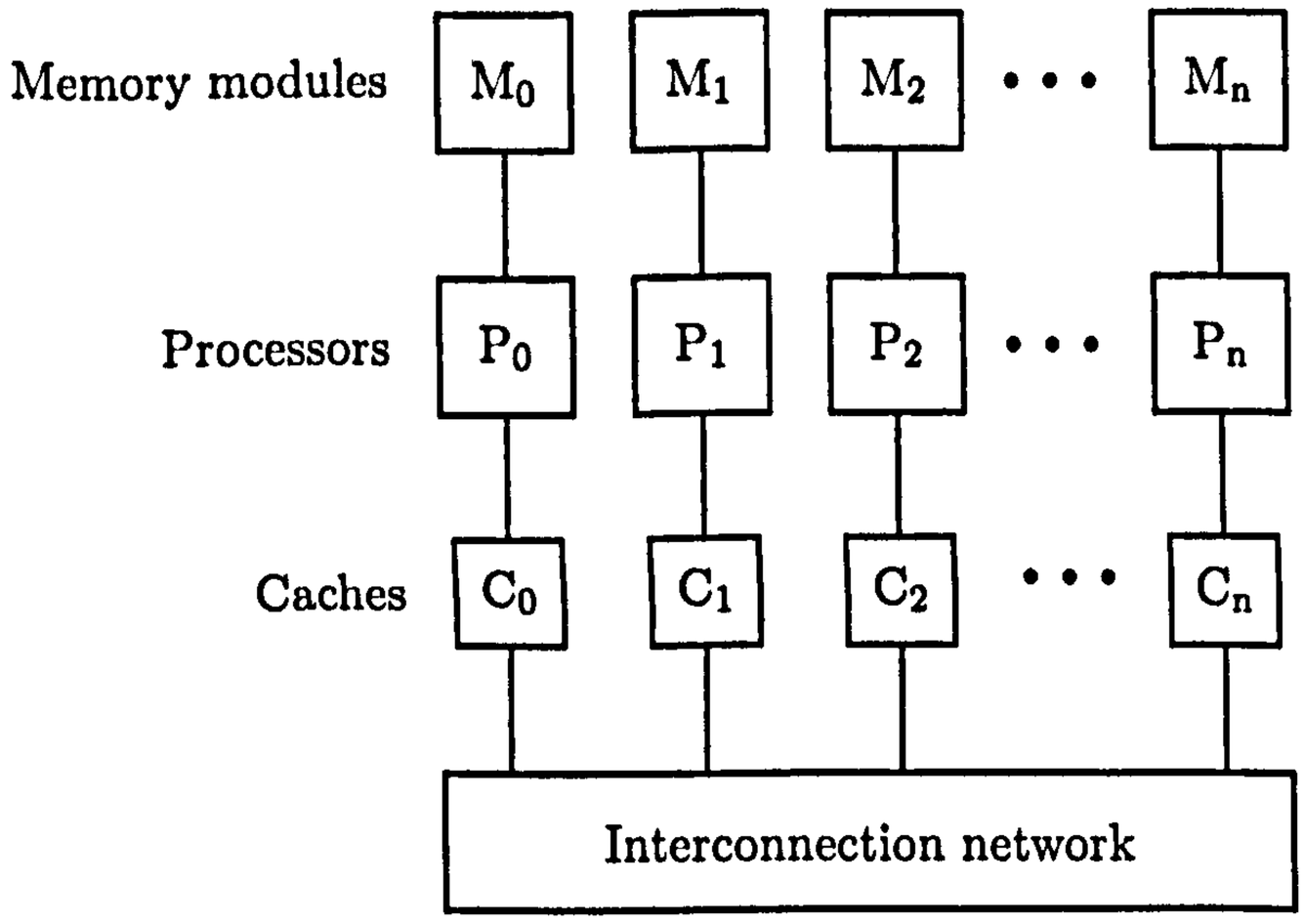


Figure 5.3: Distributed Shared Memory Architecture

5.3 Parallel Performance Metrics

There are two main metrics used to measure the performance of parallel algorithms, namely *speedup* and *efficiency* [80, pp. 36]. Given that T_p is the time required to perform some calculation using $p \geq 1$ processors, the speedup is defined as

$$S_p = \frac{T_1}{T_p}$$

and the efficiency as

$$E_p = \frac{S_p}{p}$$

where T_1 is the time of the serial algorithm.

Theoretically the upper bounds of S_p and E_p are p and 1, respectively. Also theoretically, these upper bounds can only be achieved if the algorithm does not

have any sequential component and the computation is equally distributed among all processors. Accordingly to *Amdahl's Law* [39, 17] if a fraction f of a calculation is inherently sequential the speedup is above limited by

$$S_p \leq \frac{1}{f + (1-f)/p},$$

where $0 \leq f \leq 1$ (sequential fraction). Amdahl's Law states that as p grows to infinity the maximum speedup depends on the fraction of sequential code f and not on the number of processors p (see Figures 5.4 to 5.7) since

$$\lim_{p \to \infty} \frac{1}{f + (1-f)/p} = \frac{1}{f}$$

However f is not constant; it depends on the number of processors among other parameters. In order to achieve maximum speedup f must not increase as the number of processors increases. This behaviour is called *scalability* and is a major condition for parallel computing [97]. Another important condition is *locality*. The data needs to be stored in the main memory as close as possible to the corresponding CPU in order to minimize the work of cache-coherence and loading/storing data.

Despite the fact that theoretically S_p can only be less than p (sublinear speedup), the actual speedup can be either less than p (sublinear speedup), equals to p (linear speedup) or greater than p (superlinear speedup). Linear or superlinear speedups happen, for instance, when the data fits into cache-memory after being distributed. Given the fact that potential speedup is often affected by problem size and that the Amdahl's Law does not capture this effect, other laws have been proposed. These laws are called Gustafson's Law [110] and Ni's Law [37, pp 247]. Gustafson's Law addresses how the increasing sizes of a program affects its scalability. Ni's Law summarizes the interaction between increasing problem size and the ability to execute the program in parallel.

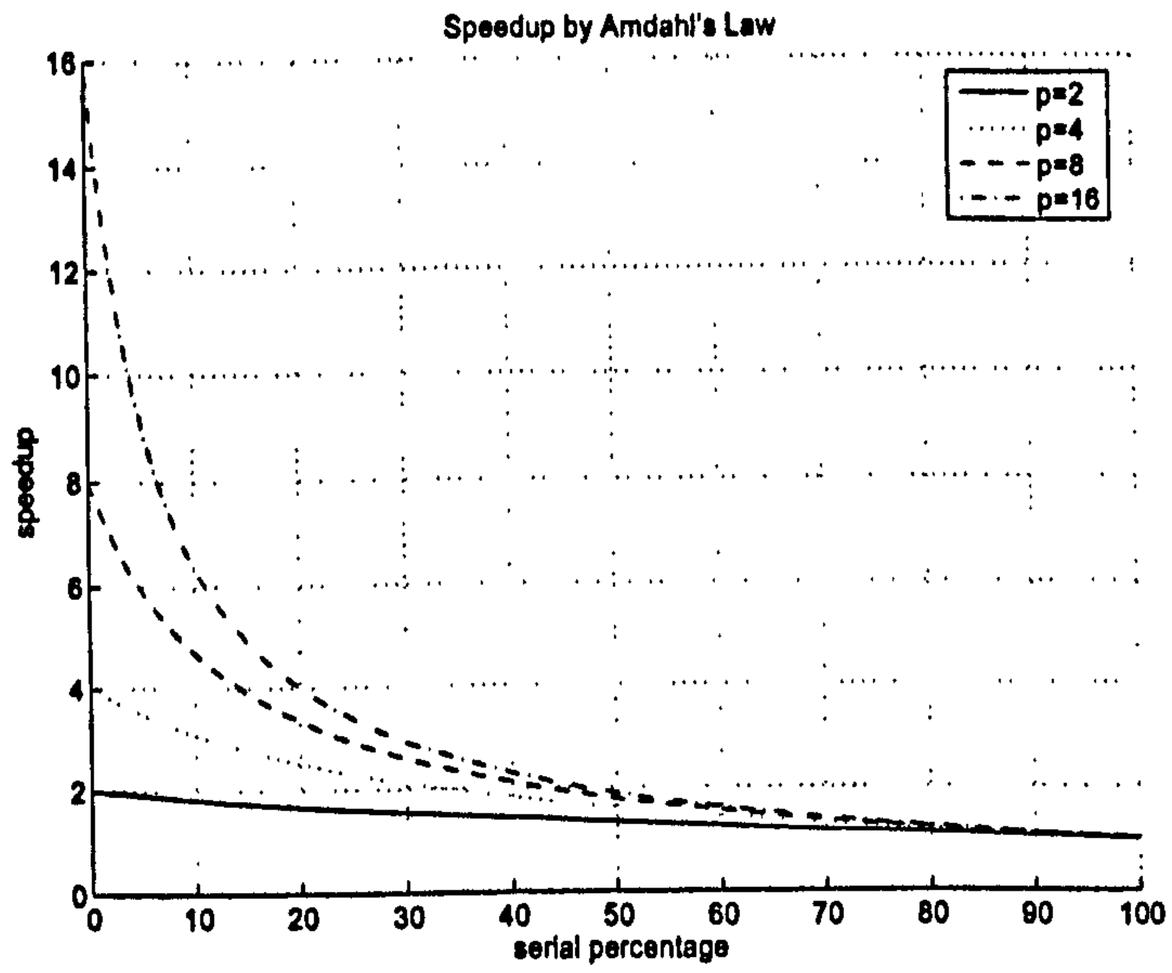


Figure 5.4: Amdahl's Law predicted speedup (2, 4, 8 and 16 processors).

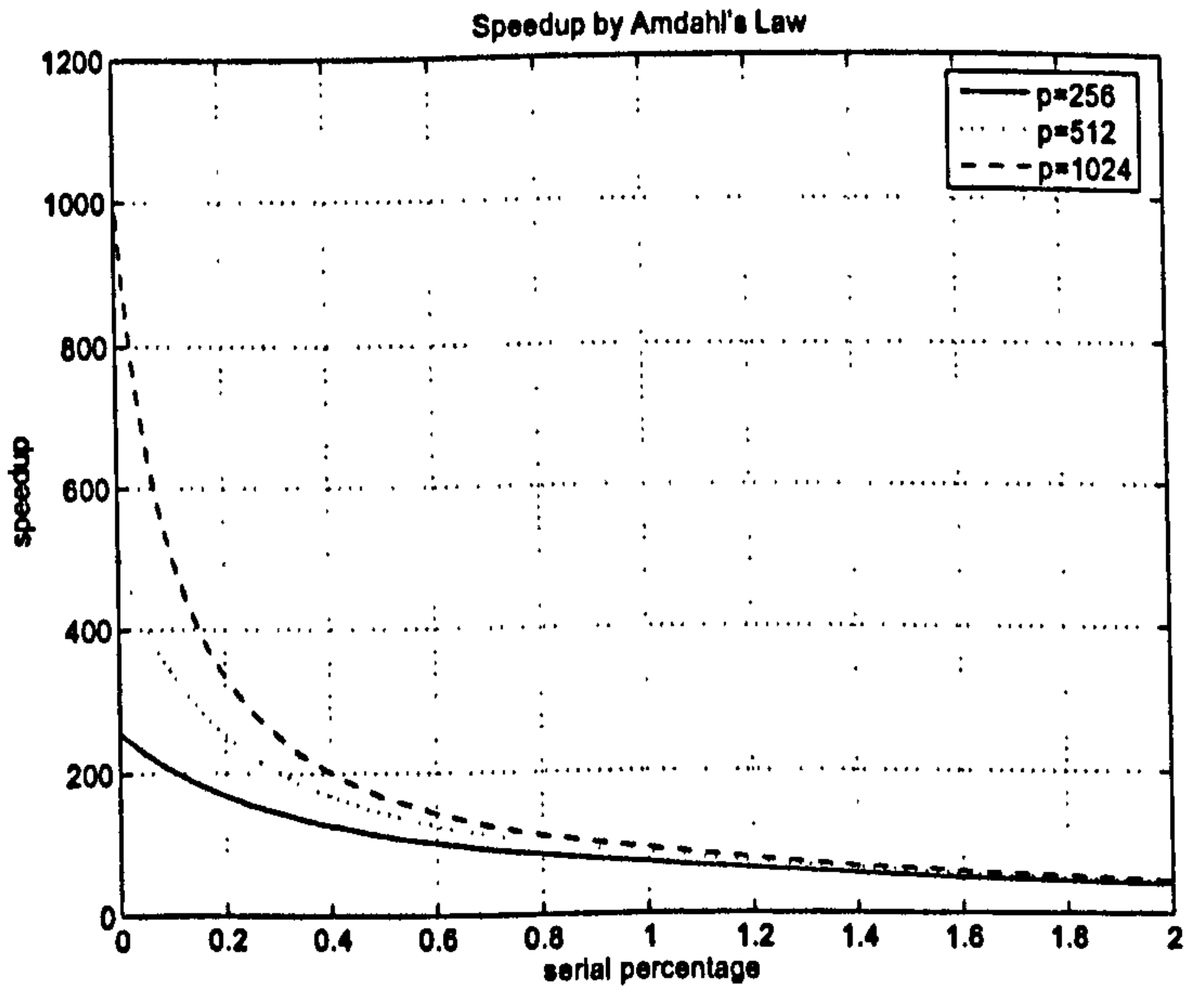


Figure 5.5: Amdahl's Law predicted speedup (256, 512 and 1024 processors).

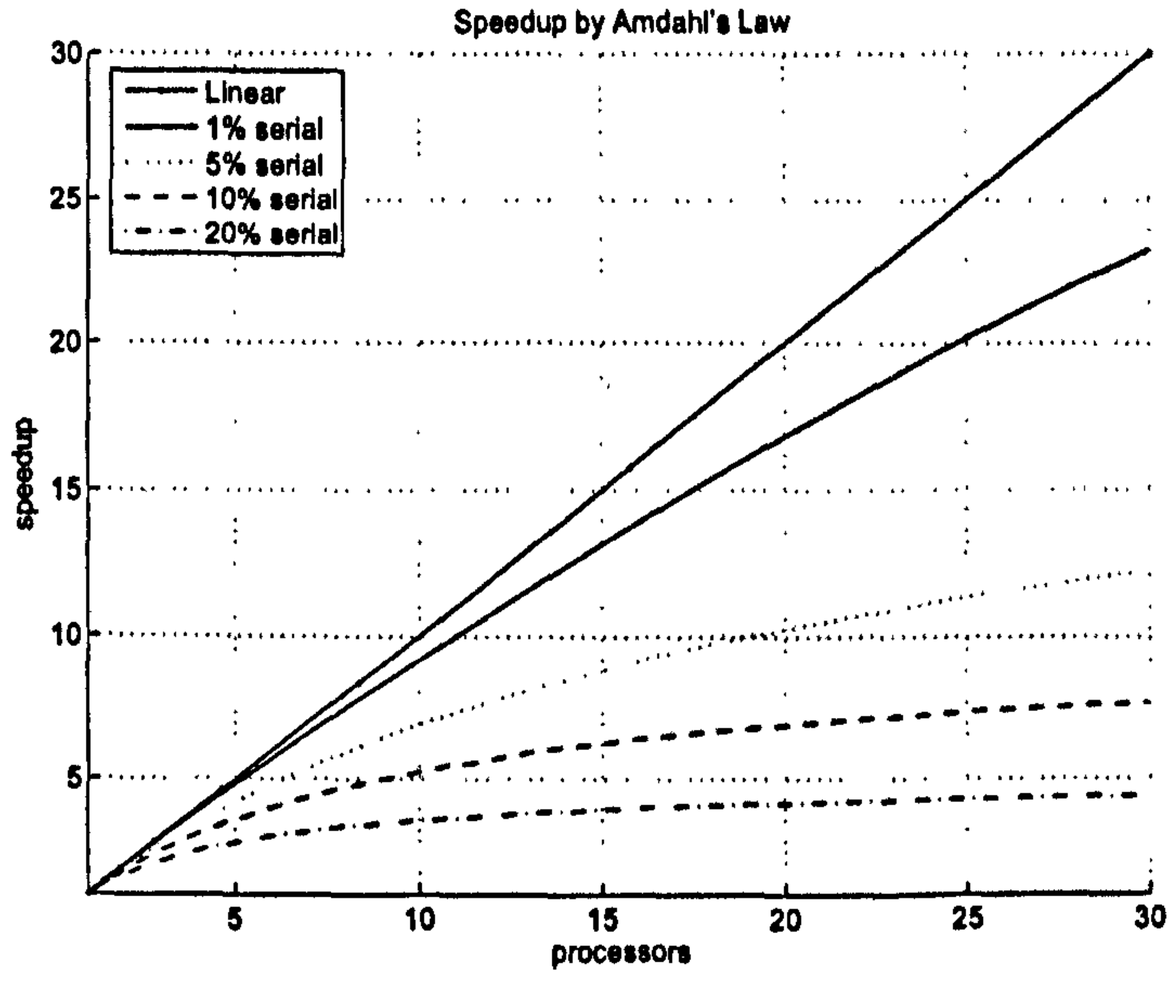


Figure 5.6: Amdahl's Law predicted speedup (linear, 1, 5, 10 and 20% serial).

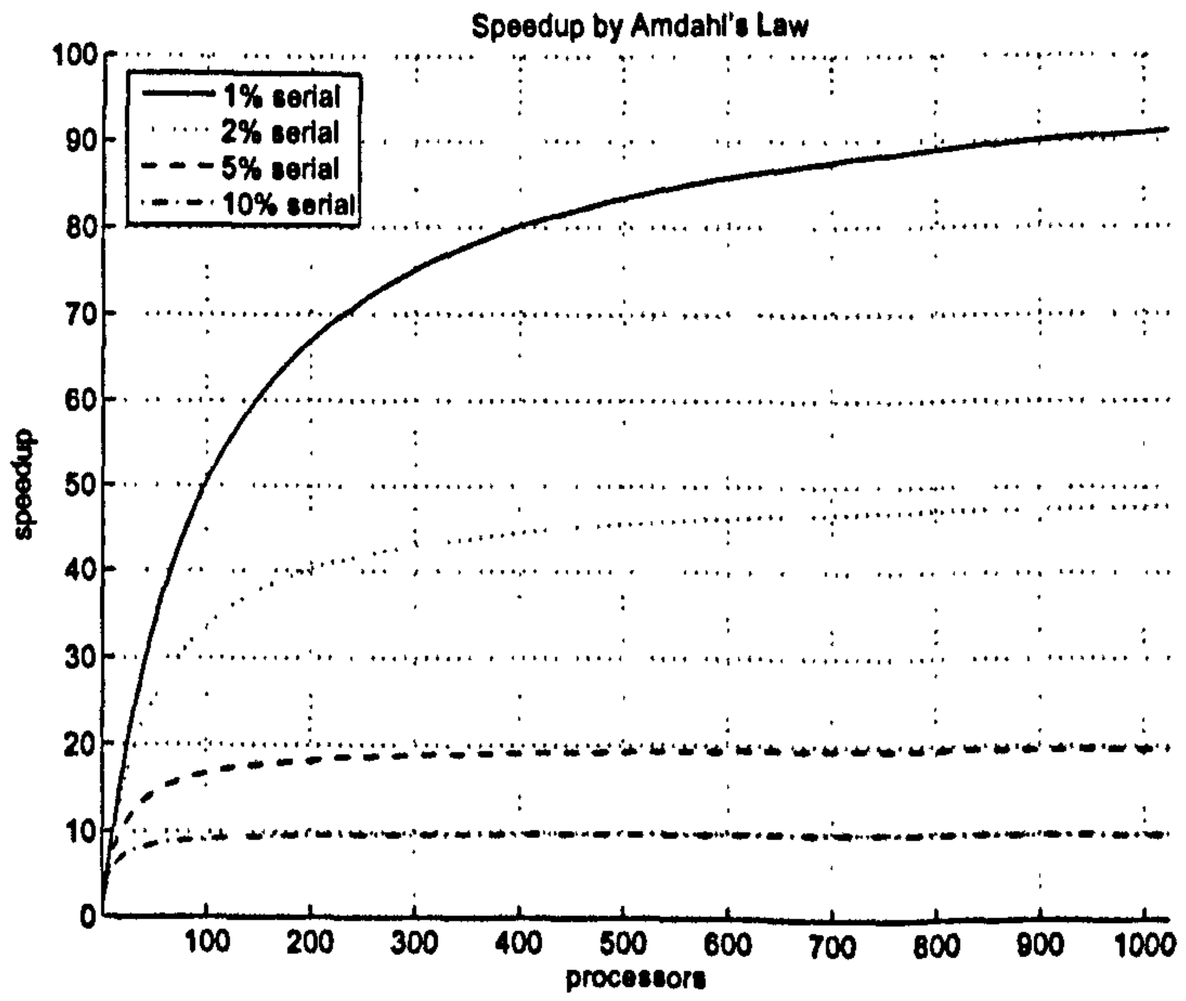


Figure 5.7: Amdahl's Law predicted speedup (1, 5, 10 and 20% serial).

5.4 Parallel Computational Models

Gropp *et al.* [53, pp 3] define a computational model as a conceptual view of the types of operations available to the program. It does not include syntax of a particular programming language or library and it is almost independent of the underlying hardware that supports it. In other words, any of the models can be implemented on any modern parallel computer given the right support from the operating system. However, the effectiveness of such an implementation depends on the gap between the model and the machine. A brief description (summarized from [53]) of some of the currently used models follows. A more comprehensive overview of the models can be found in [53] and [55] for instance.

Message passing Message passing is a model for interactions between processors within a parallel system. In general, a message is constructed by software on one processor and is sent through an interconnection network to another processor, which then must accept and act upon the message contents. MPI (Message Passing Interface) is a message passing model and has been used in this work – see Section 5.4.2.

Data parallelism The parallelism comes entirely from the data and the program itself looks very much like a sequential program. A set of tasks work collectively on the same data structure but each task works on a different partition of the same data structure. The partition of the data that underlies this model may be done by a compiler, the High Performance Fortran (HPF) [69] for instance, or by the use of an interface such as OpenMP [87].

Shared memory On the shared-memory model each processor has access to all of a single, shared address space at the usual level of load and store operations. Access to locations manipulated by multiple processes is coordinated by some form of locking, although high-level languages may hide the explicit use of locks.

Threads The most common version of the shared-memory model specifies that all memory be shared. This allows the model to be applied to multithreaded systems in which a single process has associated with it several program counters and execution stacks. The model allows fast switching from one thread to another and requires no explicit communication. The difficulty imposed by the thread model is that any state of the program defined by the program variables is shared by all threads simultaneously, although in most thread systems it is possible to allocate thread-local memory. The most widely used thread model is specified by the POSIX Standard [93]. A higher-level approach in programming with threads is also offered by OpenMP [87].

Combined models Combinations of the above models are also possible. These combined models are also called hybrid models. Hybrid models lead to software complexity in which a shared-memory approach (OpenMP, for instance) is combined with a message-passing approach (MPI, for instance).

Currently, some of the most common used parallel programming interfaces are OpenMP and MPI (Message Passing Interface). Among Fortran users HPF (High Performance Fortran) [37, pp 289] has also been often used. HPF is an extension to the Fortran 90 language to support data parallel programming and is based on data parallelism. OpenMP is targeted only at shared-memory systems. It is relatively easy and simple to use but its performance if compared to MPI is usually inferior (see Reuter [97] for a comparison).

The parallel solver presented in this thesis was implemented under the message-passing model, using the MPI library, and the SPMD programming model (see Sections 5.4.1 and 5.4.2). Despite the fact that most of the development was done on a shared-memory system (see Section 5.5.1), the program was written to be portable to distributed-memory architectures. The message-passing paradigm was chosen for its universality (it is available on most parallel machines) and

performance.

5.4.1 SPMD programming model

One of the most popular styles of programming parallel computers is the SPMD (Single Program Multiple Data) programming model. The same program, though not necessarily the same instruction stream, is executed on all processors, each processor operating on a part of the data. The SPMD model can also be seen as a restricted version of MIMD in which all processors run the same program. Hence, as opposed to SIMD, each processor executing SPMD code may take a different control flow path through the program. For the definition of MIMD and SIMD see Section 5.2.

5.4.2 Message Passing Interface - MPI

MPI is a message-passing library that supports both distributed- and shared-memory systems. Historically, a variety of message-passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.

In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message-passing implementations. Part 1 of the Message Passing Interface (MPI) was released in 1994 and part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web [81].

MPI is currently considered the *de facto* standard for message passing, replacing virtually all other message-passing implementations. Most, if not all parallel computing platforms offer at least one implementation of MPI and very few have a

full implementation of MPI-2. For shared-memory architectures, MPI implementations usually do not use a network for inter-process communications. Instead, they use shared memory (memory copies) for performance reasons.

From a programming perspective, message-passing implementations commonly comprise a library of subroutines that are imbedded in the source code. The programmer is responsible for determining all parallelism. A parallel program written with MPI has a single source code that, after being compiled and linked to the MPI library, can be launched with the command `mpirun`[2]. The command provides many optional flags and the mandatory flag `-np` that defines the number of processes. After being launched the MPI application is replicated among the processes and each one of these processes should operate on a different set of data, i.e. the same program is executed in several processes with different sets of data. Hence, a program written with MPI is classified as SPMD (see Section 5.2) [32].

5.5 Experiments in Processor Scalability

Experiments in processor scalability on a Sun Fire 15K parallel computer were performed in order to measure the actual latency and transfer rate. The parallel performance of a rank-revealing QR factorization [33] was also measured and compared to the results obtained on an IBM 9076 SP/2 parallel computer. A brief description of the parallel computers used follows and the results obtained are presented in the next sections.

[2]The command to launch an MPI application may vary depending on the system. The most common command used is `mpirun`.

5.5.1 Sun Fire 15K Parallel Computer

The Sun Fire 15K parallel computer is equipped with UltraSPARC™ III+ CPU and the Sun Fireplane Interconnect architecture running the binary-compatible Solaris 8 UNIX® operating environment. The Sun Fire 15K is a cache coherent Non-Uniform Memory Access (ccNUMA) architecture. Four processors and 16Gb of memory are placed on a so-called Uniboard CPU/Memory Board and a maximum of 18 Uniboard CPU/Memory boards are connected through the Sun Fireplane Interconnect which comprises of an address and a data bus. Each processor has equal access to the memory located on the same board but slower access to the memory on remote boards. The interconnect peak latency and bandwidth reported on the documentation [114] are briefly quantified on Tables 5.1 and 5.2, respectively.

Memory Access	Bandwidth Gb/sec
Same CPU as requester	9.6
Same board as requester	6.7
Separate board from requester	2.4

Table 5.1: Sun Fire 15K peak interconnect bandwidth [114, pp 4-7]

The system installed at Cranfield University, part of the Cambridge-Cranfield High Performance Computing Facility (CCHPCF), called `robinson`, has 72 1.2GHz CPUs with 288Gb of shared memory, which gives a 144 Gflops peak performance. The FORTE DEVELOPER 7 FORTRAN 95 compiler and the LAM implementation of MPI are available.

Location in Memory	Clock Count
Same board (requester local memory)	180 ns, 27 clocks
Same board (other CPU on the same dual CPU data switch)	193 ns, 29 clocks
Same board (other side of data switch)	207 ns, 31 clocks
Other board (coherency cache directory hit)	333 ns, 50 clocks
Other board (coherency cache directory miss)	440 ns, 66 clocks

Table 5.2: Sun Fire 15K pin-to-pin latency for data in memory [114, pp 4-8]. Pin-to-pin latency is calculated by counting clocks in the interconnect logic design between the address request from a CPU and the completion of the data transfer back into the CPU. It is independent of what the CPU does with the data.

5.5.2 IBM 9076 SP/2 Cluster

The IBM 9076 SP/2 RISC-based distributed-memory multi-processor cluster, housed at the Brazilian *Laboratório Nacional de Computação Científica - LNCC* (National Scientific Computing Laboratory), has 8 WIDE RS6000 (model 590) processors and 32 THIN RS6000 (model 390) processors connected through a SWITCH Fiber Channel with multiple connections capability at 266 Mbits/s each. The theoretical peak performance of the IBM 9076 SP/2 is of 640 Mflops per processor[3]. The machine is equipped with the AIX 4.1.5 operating system (IBM Unix variant) and all the programs used in this work were compiled with the IBM *xlf* Fortran compiler and the message-passing library MPICH (MPI IBM implementation that allows message passing through the high speed SWITCH).

[3]Reported at http://www.top500.org/orsc/1998/sp2.html

5.5.3 Parallel RRQR Factorization Scalability

A double-precision implementation of the parallel RRQR (PRRQR) factorization by da Cunha *et al.* [33] was used to investigate the parallel performance of the Sun Fire 15K system. The PRRQR algorithm computes the QR factorization of a matrix $A_{m\times n}$ intended for use when $m \gg n$. The algorithm uses a reduction strategy to perform the factorization which in turn allows a good degree of parallelism. It is then integrated into a parallel implementation of the QR factorization with column pivoting algorithm due to Golub and Van Loan [48, pp. 233-236], which allows the determination of the rank of A. The algorithms were coded in FORTRAN 90 using the MPI library.

Table 5.3 presents results of tests carried out on the Sun Fire 15K and the IBM 9076 SP/2 parallel computers described in Sections 5.5.1 and 5.5.2. The results for the IBM computer were obtained from the paper by da Cunha *et al.* [33]. Table 5.3 shows the running times (in seconds) and the respective speedups ($S_p = T_1/T_p$) of PRRQR for two large matrices, with $m \gg n$. Here T_1 is the time taken by a sequential implementation of the RRQR algorithm where the QR factorization is obtained via Householder reflections [48, pp. 193-220]. The Sun system is faster than the IBM 9076 SP/2 parallel computer whilst the speedups are quite similar but slightly higher on the IBM 9076.

5.5.4 Latency and Transfer Rate

The actual latency and transfer rate of the Sun Fire 15K system were obtained running a Fortran 90 message-passing (MPI) implementation of a round-trip point-to-point communication procedure and a broadcast collective communication procedure, both with double-precision arithmetic (64-bit words). The original codes

	M × N	p=1	p=2	p=4	p=8	p=16	p=32
Sun	20000 × 10	0.14	0.12	0.06	0.05	0.06	0.12
			1.16	2.33	2.80	2.33	1.16
IBM 9076	20000 × 10	0.73	0.43	0.25	0.16	0.10	–
			1.70	2.92	4.56	7.30	–
Sun	800000 × 10	23.02	12.37	6.59	4.60	2.97	1.11
			1.86	3.49	5.00	7.75	20.73
IBM 9076	800000 × 10	28.39	14.79	7.56	4.55	3.30	–
			1.92	3.76	6.24	8.60	–

Table 5.3: Timings (in seconds) and speedups for PRRQR.

in Fortran 77 were developed by Professor R. D. da Cunha (UFRGS/Brazil). The Sun One Studio 8 Fortran 95 compiler (`mpf90`) and the LAM implementation of MPI were used.

The round-trip point-to-point communication time is measured by first sending a message from processor *master* to processor *slave* and then sending a message from processor *slave* to processor *master*. Here, master is the processor with lowest rank and slave the processor with highest rank, i.e. for 4 processors, master and slave are the processors of rank 0 and 3, respectively. This implies that the measurements are taken using processors on different boards when applicable. The messages are sent and received using, respectively, the MPI commands `MPI_SEND` and `MPI_RECV`, i.e. a blocking point-to-point communication. The resulting time is divided by 2, since actually two send/receive pairs are performed. The measured bandwidth and time are listed on Tables A.3 and A.4, Appendix A.

The broadcast collective communication time is measured by broadcasting a message of some length from processor 0 to all other processors. This is done by all

processors calling the MPI command MPI_BCAST with 0 as the root processor. In MPI, communication involving more than two processes is collective and all participating processes call the same routine. MPI_BCAST is an example of a collective communication routine. Notice that the broadcast could be performed by a series of send and receive calls. The use of an intrinsic routine that is implemented as a collective communication allows the message-passing library to take advantage of its knowledge of the structure of the machine to optimize and increase the parallelism in these operations. The measured bandwidth and time are listed on Tables A.1 and A.2, Appendix A.

The data transfer model is assumed to be linear for the same number of processors or pairs of processors. Hence, the communication time (T_c) is given by $\alpha + \beta w$, where w is the number of words being transferred; α is the latency (in seconds) and β is the transfer rate (in seconds per word). The latency and transfer rate for a given number of processors can be obtained by linear regression on a set of communication times for messages of variable lengths. For the results that follow, the coefficients α and β where calculated using the MatLab function POLYFIT (fit polynomial to data). As described in the MatLab manual [75], `p = polyfit(x,y,n)` finds the coefficients of a polynomial $p(x)$ of degree n that fits the data y best in a least-squares sense.

Tables 5.4 and 5.5 depict the reckoned latency and transfer rate of the round-trip and broadcast routines described above. An average execution time over a series of repetitions is considered to compute the coefficients. In order to evaluate the closeness of the empirical data to the linear model, two latency data sets are presented. The first set (namely latency) uses the linear data transfer model described above (linear regression). The second set (namely latency*) uses a second degree polynomial for fitting the data sample. Except for the case with two processors, both models (linear and quadratic) estimate similar latency coefficients, i.e. the

data sample exhibits a linear behaviour for most of the cases.

The latency of the Sun Fire 15K system is relatively low. For the two cases tested, it is equivalent to approximately 800 words (51200 bits). Therefore, it is possible to transfer a relatively small number of words without a major overhead. Notice that the round-trip latency and transfer rate are roughly the same for different numbers of processors. However, the broadcast latency and transfer rate increase as the number of processors increases. This behaviour was expected since the round-trip is a point-to-point communication and hence is executed between only two processors (independently of the actual number of processors) whilst the broadcast is a collective communication and therefore transfers data among all processors.

Processors	Rate	Latency	Latency*
2	3.02e-001	1.41e-005	9.67e-006
4	3.78e-001	1.94e-005	1.52e-005
8	4.97e-001	2.59e-005	2.23e-005
16	9.50e-001	3.58e-005	3.55e-005
32	1.50e+000	7.44e-005	6.85e-005

Table 5.4: Sun Fire 15K system actual latency (seconds) and transfer rate (seconds/gigabit) of a broadcast collective communication. Latency is computed by a polynomial of degree 1 and latency* by a polynomial of degree 2.

Figures 5.8 and 5.9 show the bandwidth of the broadcast and round-trip routines described above. The broadcast rate decreases as the number of processors increases. However, this increase is not linear due to the capability of the message-passing library to take advantage of its knowledge of the structure of the machine to optimize and increase the parallelism in these operations. If the topology of the network of processors is a binary-tree (or another into which a binary-tree

Processors	Rate	Latency	Latency*
2	1.93e-001	9.81e-006	8.29e-006
4	2.17e-001	9.76e-006	9.46e-006
8	4.02e-001	1.70e-005	8.60e-006
16	2.29e-001	1.05e-005	9.76e-006
32	2.32e-001	1.02e-005	8.77e-006

Table 5.5: Sun Fire 15K system actual latency (seconds) and transfer rate (seconds/gigabit) of a round-trip point-to-point communication. Latency is computed by a polynomial of degree 1 and latency* by a polynomial of degree 2.

can be mapped, e.g. a fully connected network) the number of steps to complete a broadcast using p processors is $\lceil \log_2 p \rceil$.

5.6 Development of the Parallel Solver

A computer code, called PUPS (Parallel Unstructured PDE Solver) has been developed as part of the present work. The conceptual design of PUPS proposes a parallel solver capable of solving a variety of partial differential equations, in n spatial dimensions, using the finite element method with a choice of several methods to solve systems of linear equations. Despite the conceptual design, the actual implementation of the solver comprises:

- Equations: Poisson and convection-diffusion equations with Dirichlet, Neumann or mixed boundary conditions.

- Meshes: triangular (2D) and tetrahedral (3D) unstructured meshes.

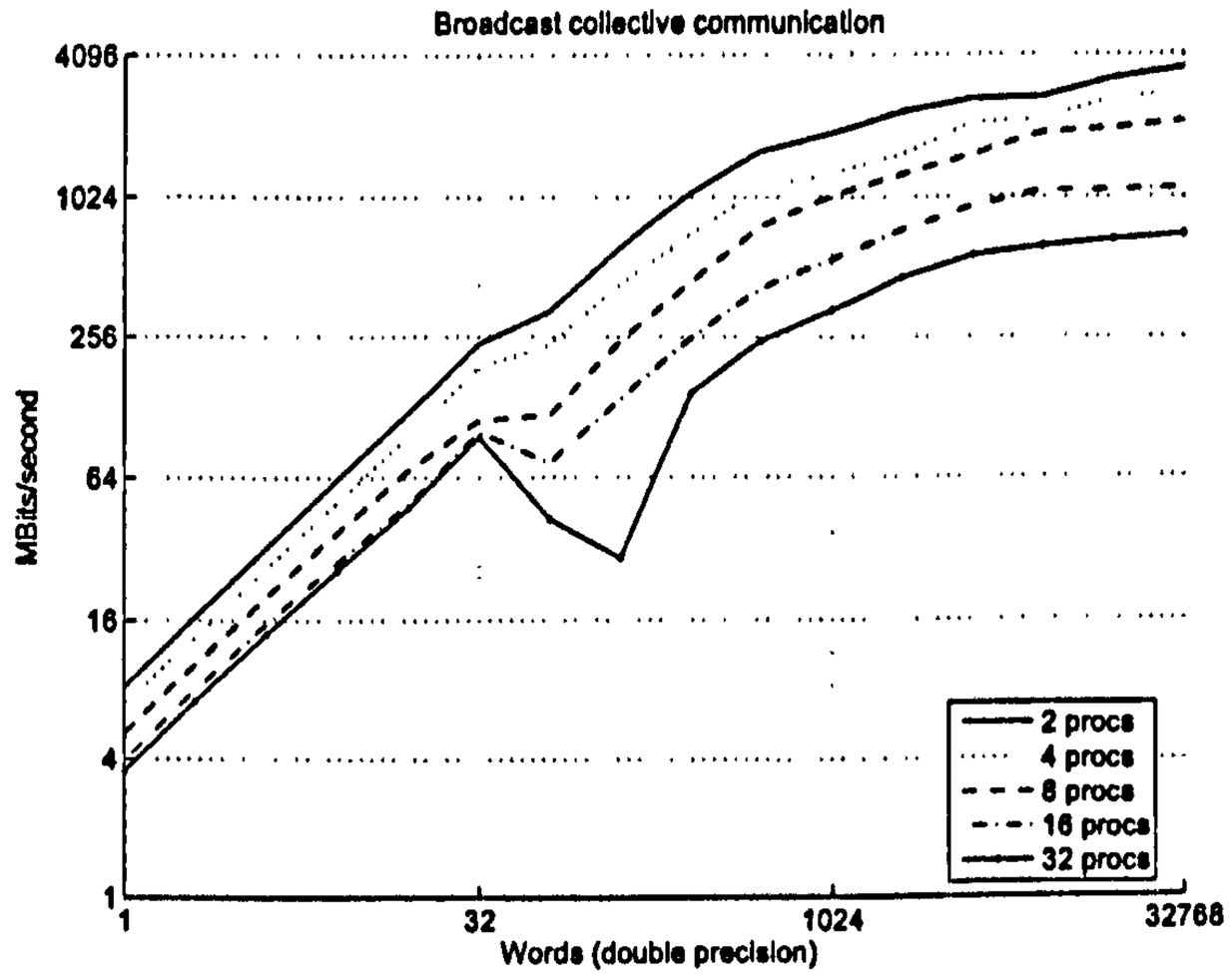


Figure 5.8: Sun Fire 15K system measured bandwidth (in megabits/second) of a broadcast collective communication ($\log_2$ scale plot).

- Iterative methods: distributive conjugate gradient (DCG) [15], conjugate gradient (CG) [58], flexible conjugate gradient (FCG) [84], BICGSTAB [119], CGS [112], restarted generalized minimum residual (GMRES) [102], flexible GMRES (FGMRES) [99].

- Preconditioners: Jacobi, Neumann polynomial, weighted least-squares polynomial, unweighted least-squares polynomial.

The ultimate goal is to develop a tool to solve CFD problems on parallel computers attaining accuracy, scalability on inhomogeneous computing environments, geometric flexibility, expansibility to new areas and problems, and modularity allowing easy upgrade. The immediate goal was to develop a parallel unstructured Poisson solver to validate and analyse the method proposed in this dissertation.

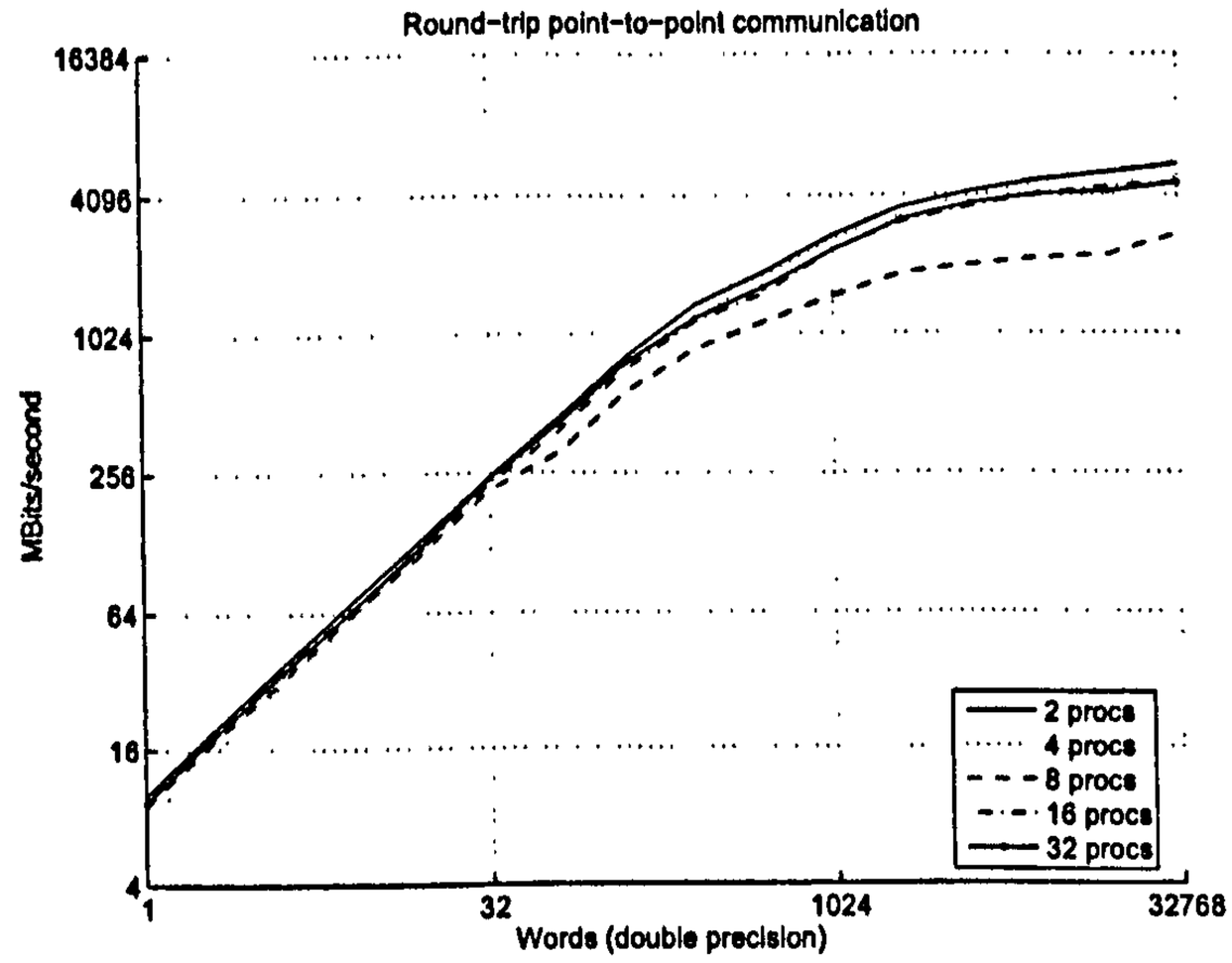


Figure 5.9: Sun Fire 15K system measured bandwidth (in megabits/second) of a round-trip point-to-point communication ($\log_2$ scale plot).

5.6.1 Solver Overview

PUPS has been developed following an object-oriented methodology and using the Single Program Multiple Data (SPMD) model (see Section 5.4.1). The object-oriented methodology allows to design reusable and easily extended softwares. SPMD involves writing a single code that will run on all the processors cooperating on a task. The data are partitioned among the processors which know what data portions they will work on.

The solver deals with three different data types: scalar values, vectors and matrices. Only the latter two are liable to data partitioning. If a scalar value is derived from some computation over data distributed among the processors, the use of a SPMD model may result in a high communication overhead while the result is sent to all processors. For instance, such communication is necessary

when computing inner products and vector norms. Sometimes it is easier and faster to compute BLAS-1 type operations on all processors simultaneously than to parallelize them.

The actual implementation of PUPS can be fragmented into six main parts, namely:

1. Mesh loading: stores information about the mesh in a variable called `mesh`. This procedure is entirely serial and depends on the application used to generate the mesh. The user can supply their own routine.

2. Mesh partition: splits the mesh into a given number of parts using Metis.

3. Discretization: transforms the PDE into a discrete problem using FEM.

4. Natural boundary conditions: sets Neumann and mixed (Robin) boundary conditions.

5. Essential boundary conditions: sets Dirichlet boundary conditions.

6. Algebraic solver: solves the system of linear equations arising by the previous steps.

The data partition adopted renders the discretization step to be altogether local, i.e. it does not involve data transfer among subdomains. The boundary conditions are set in two stages. First, the code loops through all of the boundary nodes storing the values of all nodes with essential (Dirichlet) boundary conditions and effectively setting the natural (Neumann and mixed) boundary conditions. Then, the essential boundary conditions are assigned by deleting the redundant coefficients and updating the right-hand side. This approach is used to retain the symmetry of A whenever applicable. Note that the size of the discrete problem is not reduced.

The solver is written in FORTRAN 90 using the message-passing library MPI. A sequential version that does not need MPI to be compiled is also available. The arithmetic precision is defined by a parameter through the Fortran intrinsic `kind` and the precision range depends on the characteristics of each system. PUPS has been successfully compiled and executed using the compilers Sun ONE Studio 8 Fortran 95, GNU Fortran, The Portland Group Inc. (PGI) Fortran 90/95 and NAGWare Fortran 95.

Geometry Optimization

PUPS has been successfully used by Bo Xu (PhD student at the Applied Mathematic and Computing Group, Cranfield University) on his research on shape optimization. The main goal of Bo Xu's project is to compute a geometry that optimizes the objective function either subject to constraints or without constraints. For example, to define the shape of an airfoil that minimizes the drag and preserves the lift.

A shape optimization problem may be split into four main tasks:

- Representation of the geometry: in this work a cubic B-spline with a nominal uniform knot is used to parameterize the geometry. This allows the objective function to be described as a function of the control points rather than a function of shape. The control points become the design variables in this optimization scheme.

- Mesh generator: a mesh generator is needed to produce a mesh for any suitable geometry.

- Poisson solver: a Poisson equation with any acceptable mesh has to be solved. PUPS is the current solver integrated with the geometry optimization solver.

- Optimization scheme: the optimization scheme is the core of the geometry optimization solver.

The optimization scheme used by Bo Xu and a brief review on other shape optimization schemes are presented in [127]. Figure 5.10 is one of the examples presented in [127] where the constrain was to preserve the area. As reported in [127], this final shape is not the best that could be obtained. It could be improved by reducing the error tolerance on the magnitude of the gradient vector. However, this implies in an increase on the computational cost.

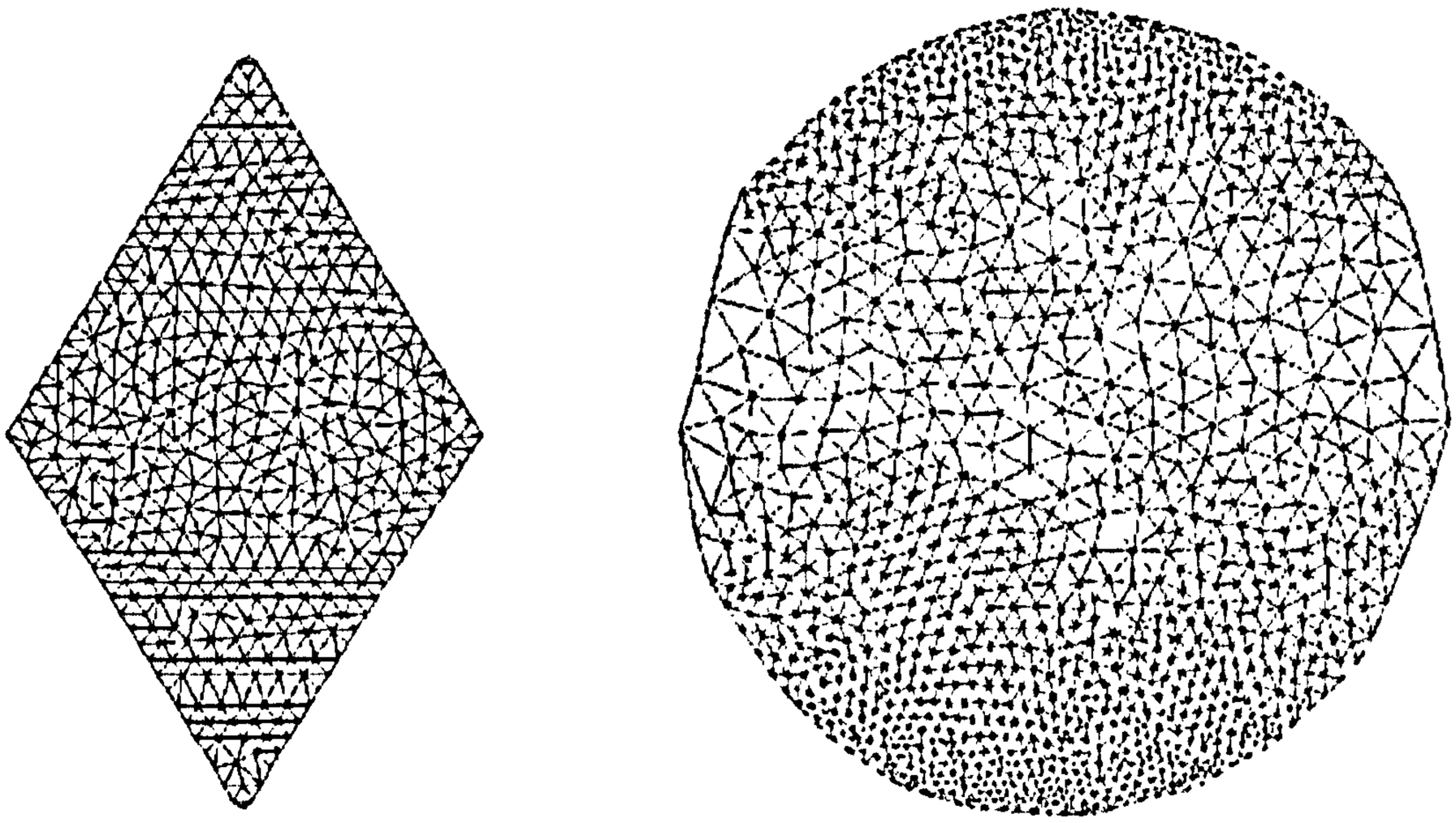


Figure 5.10: Shape optimization example. Initial shape nearly rectangular (left) and optimal shape without using mesh refinement (right).

5.6.2 Mesh Partitioning

It is common to use a graph representation when partitioning a domain. In spite of the fact that theoretically there are no restrictions on how to partition a domain, a few issues have to be addressed concerning the information that should be kept together, i.e. in the same processor (subdomain) on parallel environments. Figure 5.11 shows three types of partitioning commonly used. The vertex-based partition divides the original set of vertices in subsets of vertices and has no restriction on the edges, i.e. both edges or elements are allowed to straddle among subdomains. Edge-based is a somewhat more restrictive partitioning that does not allow edges to be split between subdomains. The element-based partitioning does not allow elements and therefore edges to be split among subdomains, i.e. all information related to a given element is mapped to the same subdomain. Both edge- and element-based partitions introduce overlap.

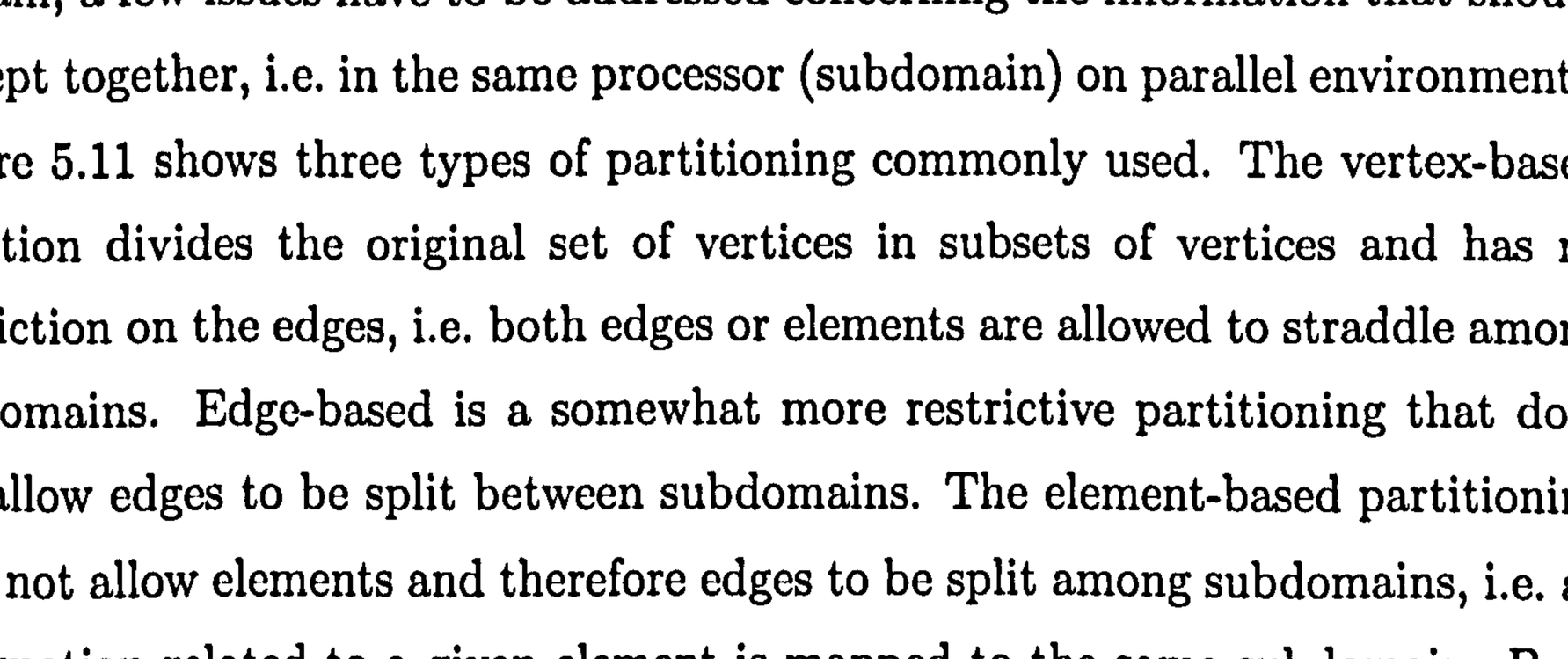


Figure 5.11: Types of partitioning: (a) vertex-based, (b) edge-based, (c) element-based

For the approach used in this work, initially the mesh has to be vertex-based decomposed into p non-overlapping subdomains Ω_i such that the number of nodes

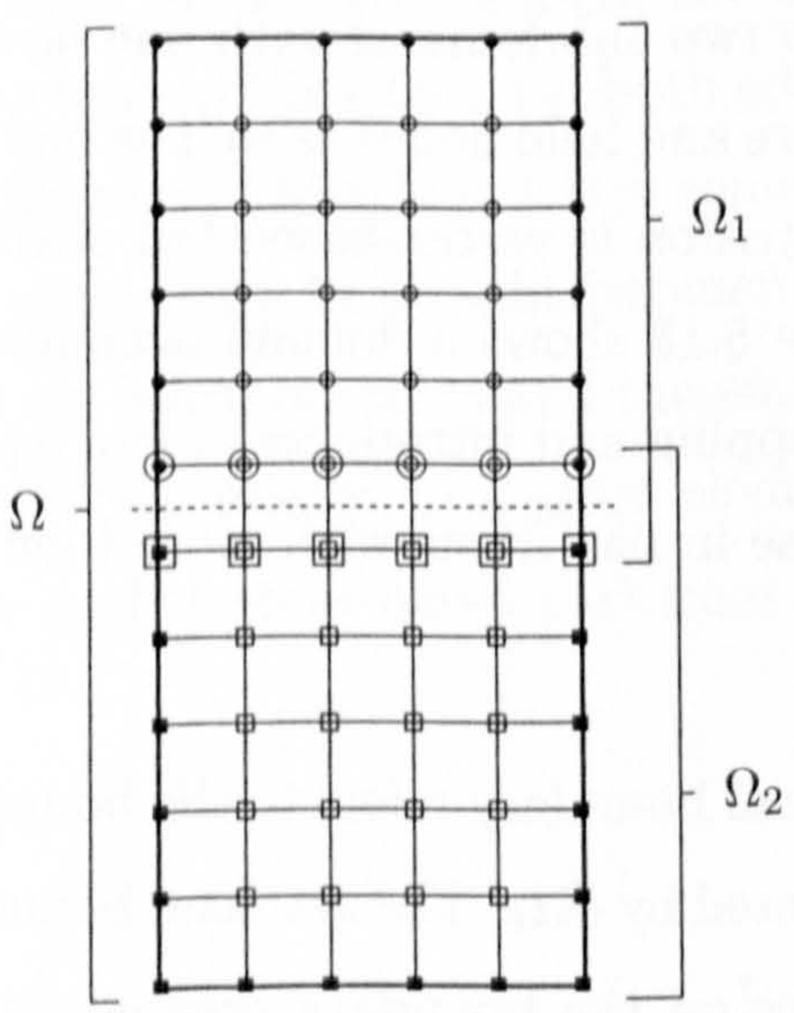
assigned to each subdomain is roughly the same and the number of adjacent elements assigned to different processors is minimized. The goals of the first and second conditions are, respectively, to balance the computation among processors and to minimize communication. Afterwards, an overlap has to be algebraically introduced by enlarging the subdomains to contain the vertices within at least one edge from the original subdomain. Even though the initial partition is vertex-based, the final partition is element-based and overlapped.

Figure 5.12 shows a domain decomposed into two subdomains with one layer of overlapping and introduces the definition of core and halo nodes as well as real and artificial boundaries. Note that the initial partition is vertex-based but with the overlapping it becomes element-based. Figure 5.13 shows a domain decomposed into two subdomains with two layers of overlapping and introduces a new type of node called *ghost-node*. Ghost nodes only arise in partitions with more then one layer of overlapping.

There are two types of boundary nodes. The real boundary refers to the boundary nodes of the original domain Ω and are represented by $\partial\Omega_i$. The artificial boundary or interface nodes, named Γ_i, are those nodes on the boundary created by the partition, i.e. the part of the boundary of Ω_i that is interior to Ω.

If a node that belongs to Ω_i is on Γ_i it is called a *halo-node* (or interface-node), otherwise it is called a *local-node*. Local nodes can be either core, ghost, boundary or ghost boundary nodes. Local nodes placed on the original partition (without overlapping) are called *core-nodes* if they are in the interior of the subdomain and *boundary-nodes* if they are on the real boundary. Nodes added by the overlapping and that are not halo nodes are called *ghost-nodes*. *Remote-nodes* are those nodes that belong to another subdomain and are not connected to any of the nodes that belong to a given subdomain.

In term of matrix notation, a domain is represented by edges and not by isolated nodes. An edge that belongs to a given subdomain may have two local-nodes, one local- and one halo-node, two halo-nodes, one halo- and one remote-node or two remote-nodes. Edges composed purely by local nodes are called *local-edges*. Edges that have a local node connected to a halo node are called *halo-edges* and all the other edges are called *remote-edges*. Note that an edge that has a halo node connected to a local node is considered a remote edge (not a local edge).

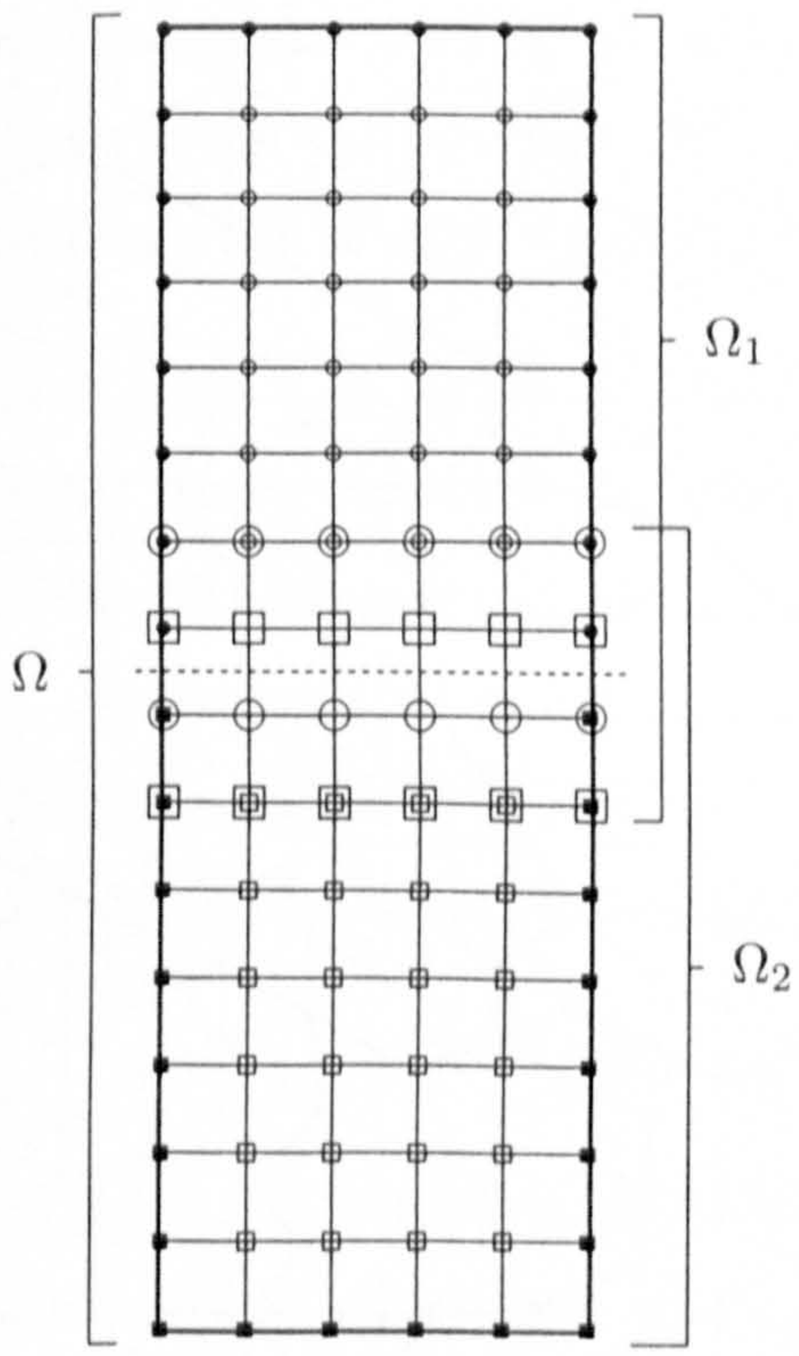


Ω = Domain
Ω_i = Subdomain i
$\partial\Omega_i$ = Real Boundary of Ω_i
Γ_i = Artificial Boundary of Ω_i

Symbol	Ω_1	Ω_2
○	Core	Remote
◎	Core	Halo/Γ_2
•	Core $\partial\Omega_1$	Remote
◉	Core $\partial\Omega_1$	Halo $\partial\Omega_2$
□	Remote	Core
▣	Halo/Γ_1	Core
■	Remote	Core $\partial\Omega_2$
▣	Halo $\partial\Omega_1$	Core $\partial\Omega_2$

Figure 5.12: Nomenclature of nodes of a domain decomposed into two subdomains with one layer of overlapping.

It is relatively simple to partition a structured grid. However, the partition of unstructured meshes is not trivial and is an area of research by itself. A large number of efficient partitioning heuristics have been developed during recent years and there are several mesh partition programs available (see Section 2.1.2). Metis was chosen for the present work mainly due to its availability and approval in the research community. Figure 5.14 shows an example of a mesh partitioned by Metis into 2 to 5 parts using an element-based algorithm.



Ω = Domain
Ω_i = Subdomain i
$\partial\Omega_i$ = Real Boundary of Ω_i
Γ_i = Artificial Boundary of Ω_i

Symbol	Ω_1	Ω_2
○	Core	Remote
◎	Core	Halo/Γ_2
•	Core $\partial\Omega_1$	Remote
◉	Core $\partial\Omega_1$	Halo $\partial\Omega_2$
□	Core	Ghost
▣	Core $\partial\Omega_1$	Ghost $\partial\Omega_2$
▫	Remote	Core
▣	Halo/Γ_1	Core
▪	Remote	Core $\partial\Omega_2$
▣	Halo $\partial\Omega_1$	Core $\partial\Omega_2$
○	Ghost	Core
◉	Ghost $\partial\Omega_1$	Core $\partial\Omega_2$

Figure 5.13: Nomenclature of nodes of a domain decomposed into two subdomains with two layers of overlapping.

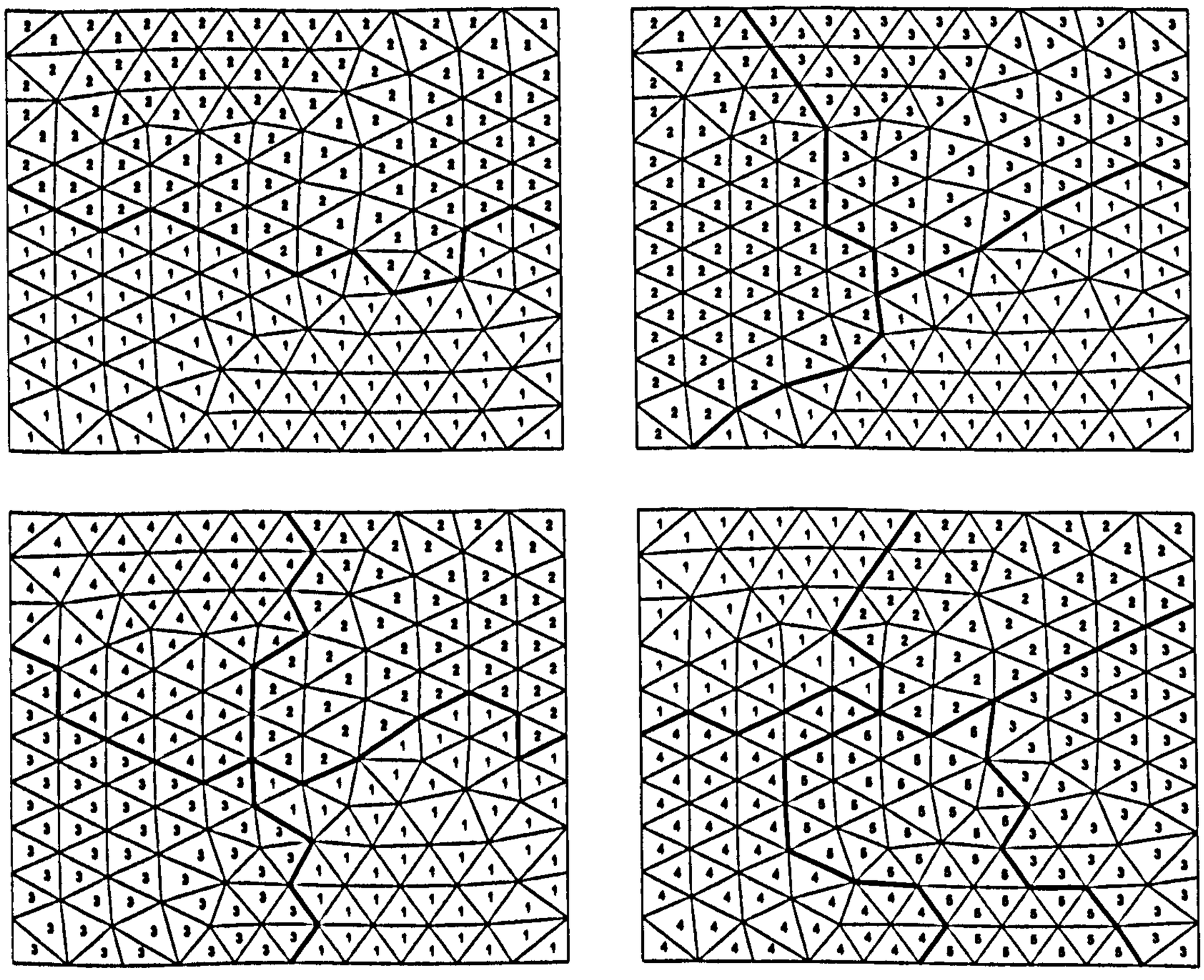


Figure 5.14: Metis partitioning into 2 to 5 parts of an unstructured squared mesh composed of 208 elements and 125 nodes

5.6.3 Data Partitioning

Data partitioning among the processors plays a major role in the design and efficiency of parallel algorithms. There are two main reasons for partitioning the data. The first is the need to solve a problem in a smaller amount of time. If a problem of data size n takes t units of time to be completed on a single processor then p processors working on subproblems of size n/p would ideally solve the same problem in t/p units of time. The second reason is the need to solve problems with a large amount of data which do not fit into the memory of a single computer.

Figures 5.15, 5.16 and 5.17 depict three of the most common data partitioning schemes: row-, column- and block-partitioning. In this work, the matrix A of Equation (6.6) is row-wise partitioned, i.e. only edges whose first node is local are kept. Assuming that n is the number of nodes of the mesh, p is the number of subdomains and n_i is the number of local nodes per subdomain, each subdomain is represented by an $n_i \times n$ matrix

$$A_i = [H_{i,1} \cdots H_{i,i-1} \; L_i \; H_{i,i+1} \cdots H_{i,p}],$$

where L_i is the matrix of local-edges and $H_{i,j}$, for $j = 1, \ldots, p$ and $j \neq i$, are matrices of halo-edges from neighbour j. If j is not a neighbour of i then $H_{i,j} = 0$. Note that each subdomain has all columns of A and there are no replicated data only if a single layer of overlapping is used. Vectors contain only the elements corresponding to local-nodes, i.e. they are of size n_i.

5.6.4 Communication Among Adjacent Subdomains

There are three basic types of communication that can happen: between two subdomains (point-to-point), among all subdomains (collective) and among adjacent subdomains. The two first cases are handle efficiently by the message-passing

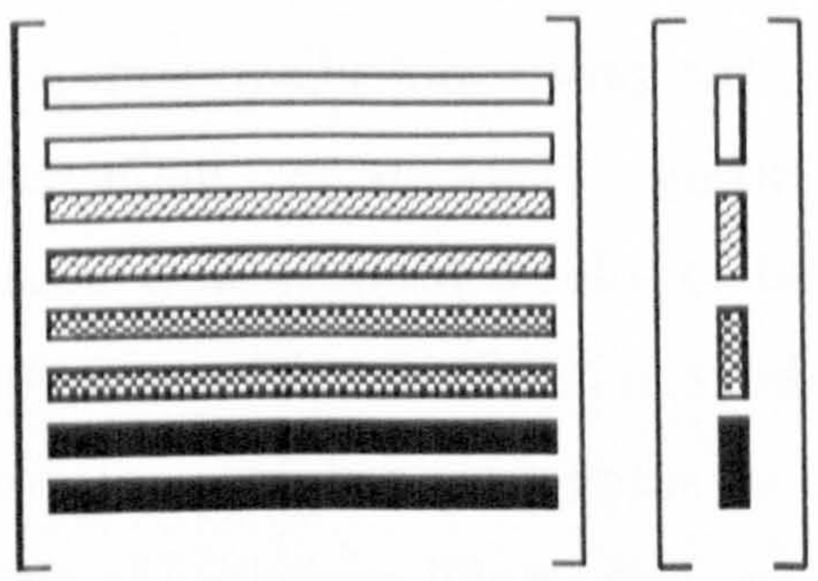


Figure 5.15: Matrix and vector row partitioning.

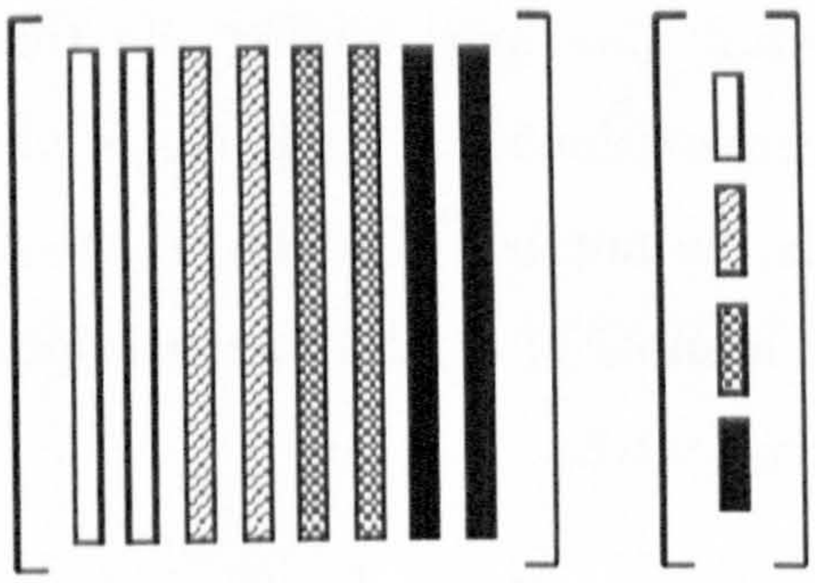


Figure 5.16: Matrix and vector column partitioning.

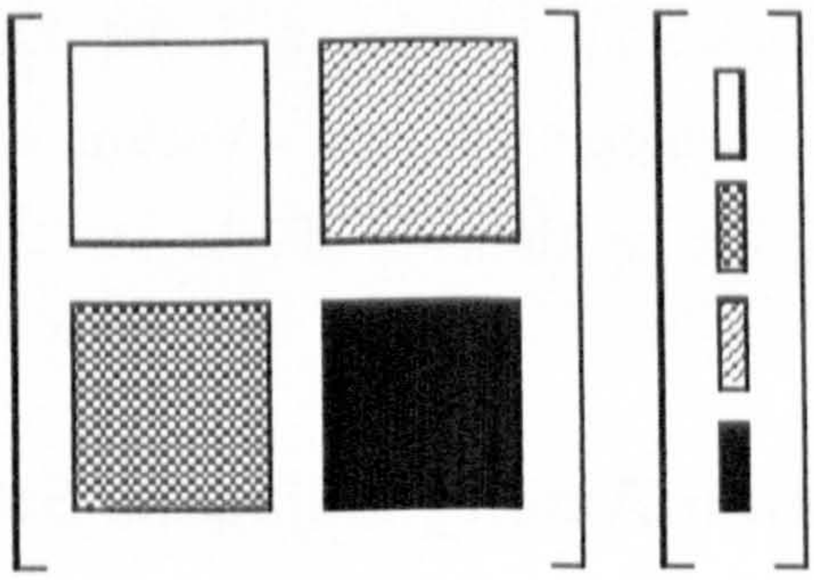


Figure 5.17: Matrix and vector block partitioning.

library but the latter case is not directly supported. In two-dimensional domains, the number of adjacent subdomains is relatively low. In three dimensions, this number can be quite high and many of the subdomains may be joined by only a few nodes.

Two communication schemes were implemented and compared. The first scheme transfers data directly to all adjacent subdomains, i.e. all adjacent subdomains are considered neighbours. The second scheme uses an algorithm to identify adjacent subdomains that are common to two or more subdomains and allocates the common adjacent subdomains as the neighbour of only one subdomain. The data transfer to adjacent subdomains that are not flagged as neighbours is made through a neighbour. These schemes are called direct and indirect communication schemes respectively.

For instance, in the domain sketched in Figure 5.18 every subdomain is connected to the other three subdomains. The communication pattern for the direct scheme requires that each one of the four subdomain sends and receives data from three subdomains. In the indirect scheme data is sent from s_1 to s_4 through s_2, for instance. Both communication schemes are outlined in Figure 5.19. Despite the fact that the number of messages exchanged among subdomains is reduced by the indirect scheme, the execution time is not consistently lower than for the direct scheme. This happens because the former has to operate in a synchronous mode imposed by the algorithm whilst the latter does not. In the example, s_2 has to receive data from s_1 before sending data to or receiving data from s_4. A series of tests showed that none of the schemes is consistently faster. Figure 5.20 shows the number of adjacent subdomains that are dropped for a cubic mesh composed of 152,118 nodes and 934,778 elements.

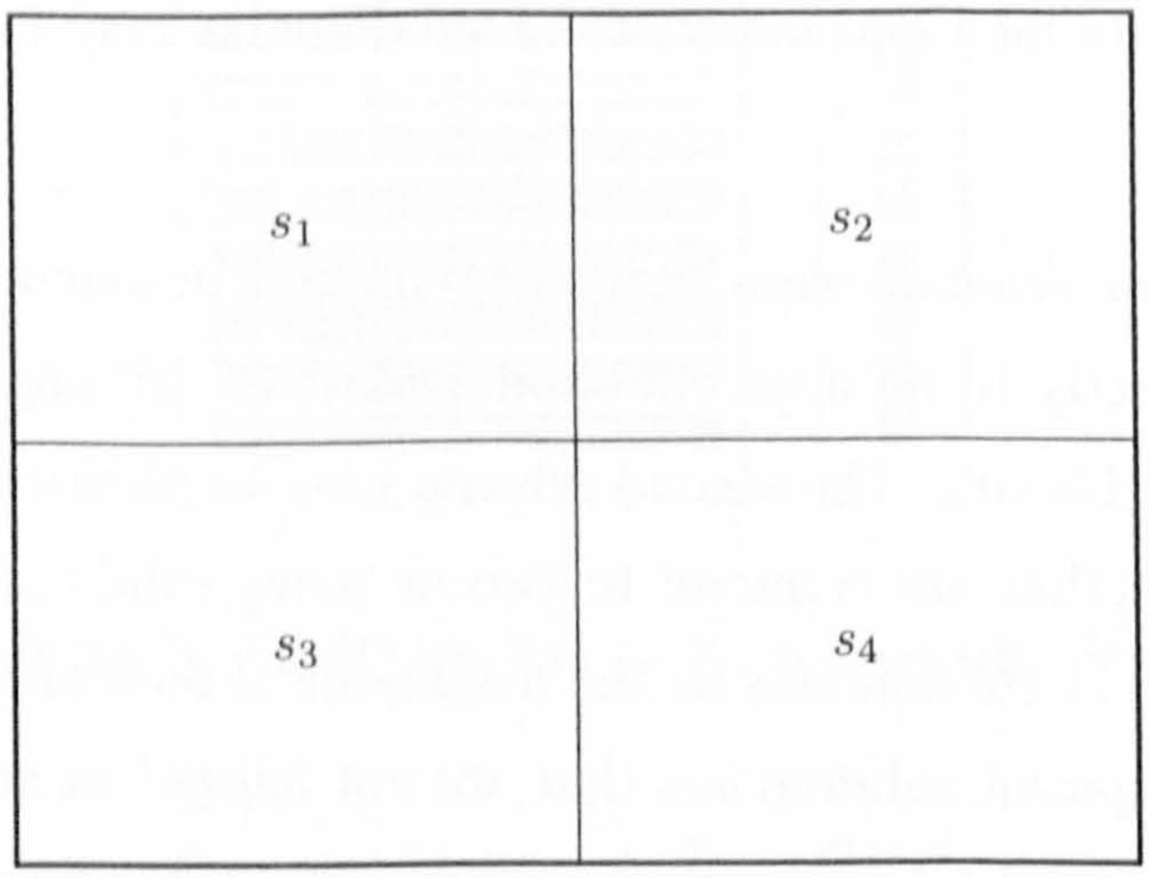


Figure 5.18: Squared domain divided into four parts.

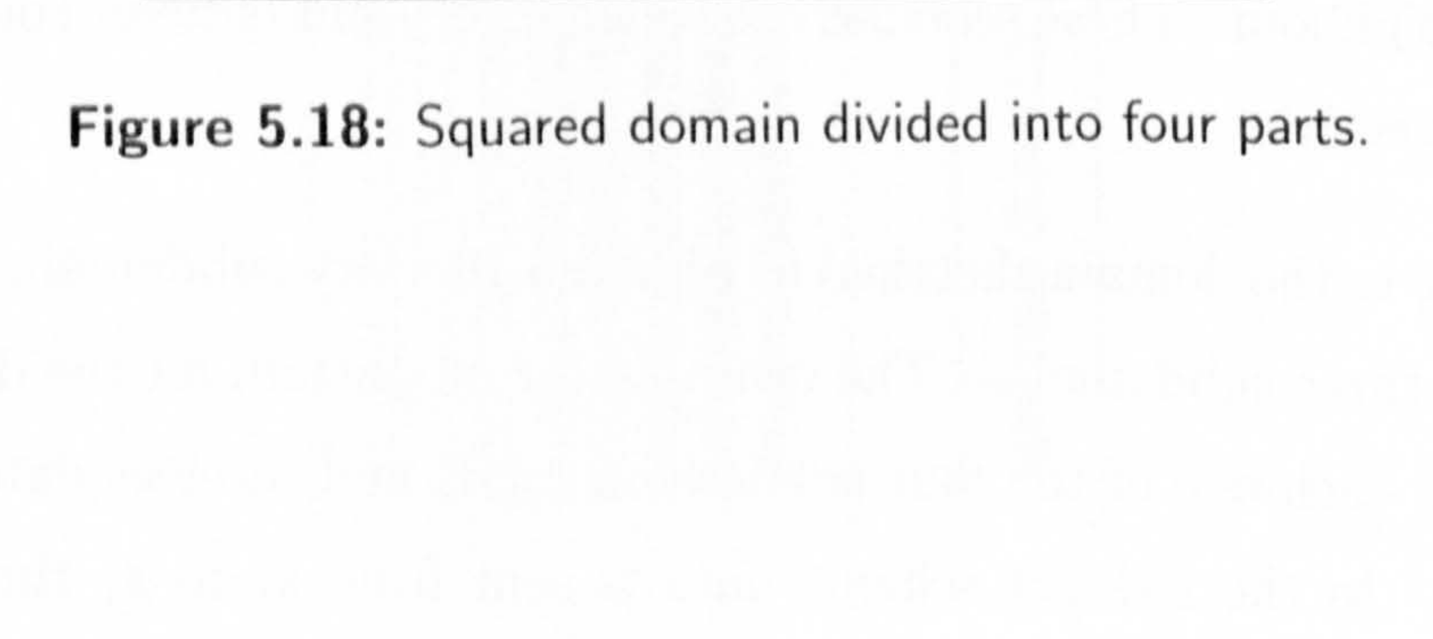


Figure 5.19: Processors connectivity for two communication schemes.

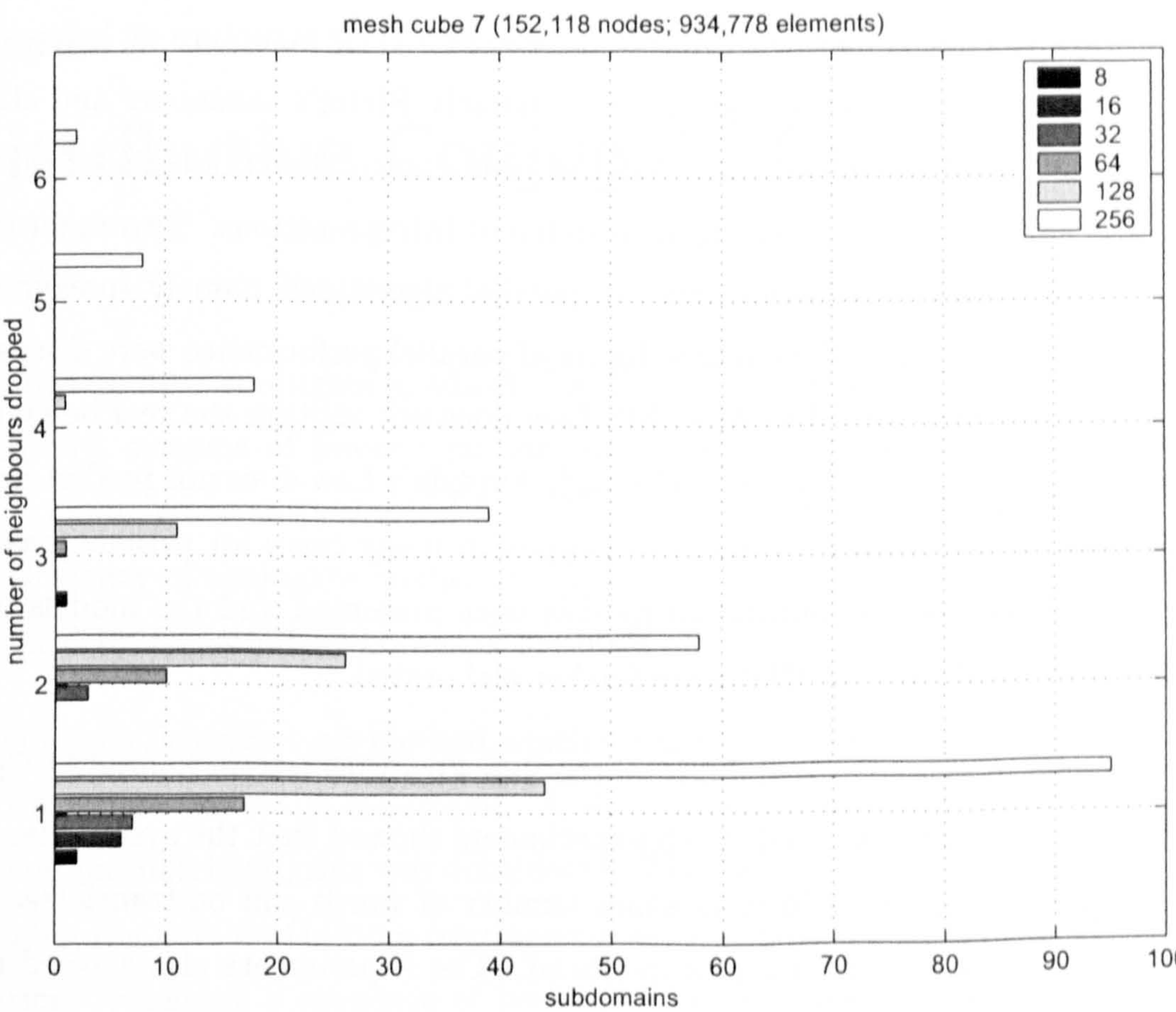


Figure 5.20: Number of adjacent subdomains dropped as neighbours by the indirect communication scheme. Cubic mesh composed of 152,118 nodes and 934,778 elements and partitioned into 8, 16, 32, 64, 128 and 256 parts (subdomains).

5.7 Summary

General parallel issues such as some of the terminology associated with parallel computing and the classification of parallel computers were addressed. There are many ways to classify parallel computers based on their structure or behaviour. Two of the most common categorizations, namely Flynn's taxonomy and structural classification, were presented. It has been shown that these classifications are quite ambiguous and might lead to different interpretations. Two main metrics used to measure the performance of parallel algorithms, namely *speedup* and *efficiency*, were defined. Theoretical limits of parallel performance were discussed and arguments that aim that Amdahl's Law does not capture the real behaviour of parallel computers were given. Although Amdalh's Law does not predict linear or superlinear speedups they actually happen in many cases for various reasons. Afterwards, parallel computational models were presented and the models used in this work (MPI and SPMD) were further elaborated.

Experiments in processor scalability of the Sun Fire 15K system housed at Cranfield University were reported. Such experiments showed that the system has low latency and therefore a relatively small number of words can be transferred between processors without a major overhead. The experiments also showed that the bandwidth has a roughly linear behaviour for the same number of processors or pairs of processors and hence a linear model can be used to calculate the actual latency and transfer rate of the system.

A general overview of the parallel PDE solver developed as part of this research project was given. Issues like the conceptual design of the solver, mesh partitioning, data partitioning and the communication schemes used were discussed. In summary, the solver was written in Fortran 90 with MPI, following an object-oriented methodology and using the SPMD model. Results that validate and probe the solver performance are reported in Chapter 6.

Chapter 6

Distributive Conjugate Gradient

The distributive conjugate gradient (DCG) method is a novel method to solve symmetric systems of linear equations on parallel computers. DCG is a main result of the research performed for this thesis. The primary idea is to combine the efficiency of conjugate gradient methods and the high parallelism that can be obtained by domain decomposition methods.

The use of a Schwarz type method which allocates one subdomain to each processor was set as an initial constraint. A single-level overlapping additive Schwarz method on matching grids was considered. The method consists basically of partitioning the domain Ω into p overlapping subdomains Ω_i and approximating the solution by solving a sequence of boundary value problems in each subdomain. The conjugate gradient method was initially considered to approximate the system of linear equations at each subdomain. A preliminary study has shown that the conjugate gradient method must be modified in order to achieve an acceptable rate of convergence.

Before describing DCG, the problem is formulated from an algebraic standpoint and the notation used is stated. A preliminary study that contains an analysis of the CG and GMRES methods applied to a decomposed domain is presented. This study has led to the DCG algorithm that has its final version presented in

Section 6.3.1. In Section 6.3.3 the parallelization of DCG is described highlighting the procedure used to evaluate the stopping criteria. An extended version of the algorithm is also presented.

6.1 Problem Formulation

From an algebraic standpoint the procedure to solve a system of linear equations divided among subdomains (processors) could be stated as follows. Given a certain matrix A divided row-wise into p blocks of size $n_i \times n$ and vectors x and b also divided into p blocks of size n_i, where $\sum_{i=1}^{p} n_i = n$, Equation (4.1) can be written as

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{bmatrix}_{n\times n} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}_{n\times 1} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{bmatrix}_{n\times 1}$$

and therefore

$$\begin{bmatrix} A_i \end{bmatrix}_{n_i\times n} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}_{n\times 1} = \begin{bmatrix} b_i \end{bmatrix}_{n_i\times 1}$$

for $i = 1, 2, \ldots, p$. Further dividing each A_i such that

$$A_i = [H_{i,1} \cdots H_{i,i-1} \; L_i \; H_{i,i+1} \cdots H_{i,p}],$$

Equation (4.1) can be written as

$$\begin{bmatrix} L_1 & H_{1,2} & \cdots & H_{1,p} \\ H_{2,1} & L_2 & \cdots & H_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ H_{p,1} & H_{p,2} & \cdots & L_p \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{bmatrix}$$

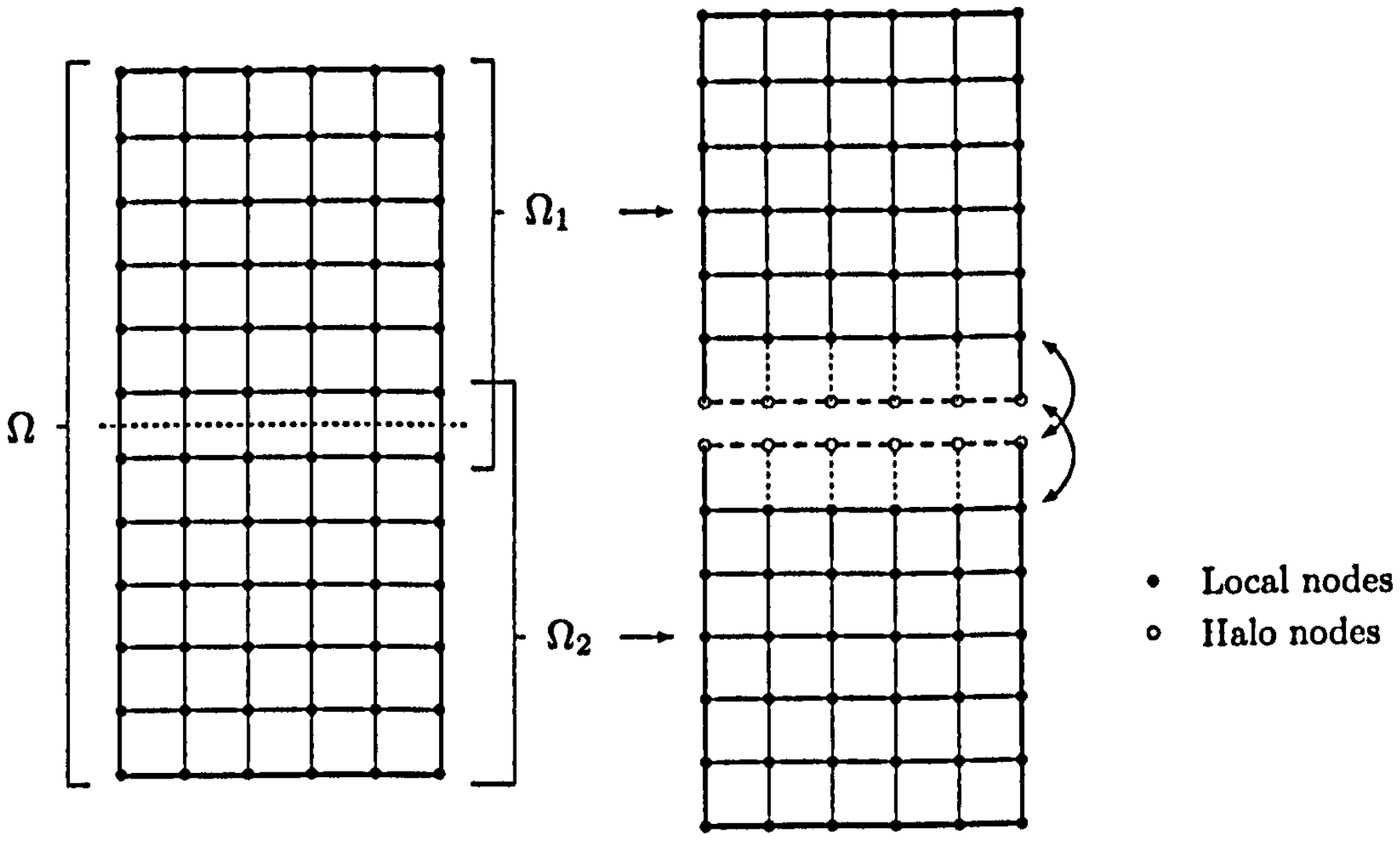


Figure 6.1: Domain decomposed into two subdomains - local and halo nodes

Such a system can be the algebraic representation of a domain partitioned into p subdomains with one layer of overlapping (halo nodes) as illustrated in Figure 5.12 (see Section 5.6.2 for the nomenclature description). According to the domain decomposition, L_i is a matrix of local-edges, i.e. an edge formed by two local nodes, and $H_{i,j}$, is a matrix of halo-edges, i.e. edges formed by a local node of Ω_i and a halo node that is local to Ω_j. If Ω_j is not adjacent to Ω_i then $H_{i,j} = 0$. Note that each subdomain holds all columns of A_i and no data are replicated and the vectors contain only local-nodes, i.e. they are all of size n_i.

Letting

$$\hat{H}_i = \begin{bmatrix} H_{i,1} & \cdots & H_{i,i-1} & H_{i,i+1} & \cdots & H_{i,p} \end{bmatrix} \tag{6.1}$$

and

$$\hat{x_i} = \begin{bmatrix} x_1 & \cdots & x_{i-1} & x_{i+1} & \cdots & x_p \end{bmatrix}^T \tag{6.2}$$

the problem to be solved at each subdomain may be written as

$$\left[\begin{array}{c|c} \mathrm{L}_i & \hat{H}_i \\ \hline 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{c} \mathrm{x}_i \\ \hline \tilde{x}_i \end{array}\right] = \left[\begin{array}{c} \mathrm{b}_i \\ \hline \hat{x}_i \end{array}\right]$$

or

$$\begin{cases} L_i x_i + \hat{H}_i \tilde{x}_i &= b_i \\ \tilde{x}_i &= \hat{x}_i \end{cases} \tag{6.3}$$

Assuming that $\hat{x}_i$ is known, Equation (6.3) can be written as

$$\begin{aligned} L_i x_i &= b_i - \hat{H}_i \tilde{x}_i \\ &= b_i - \hat{H}_i \hat{x}_i \\ &= b_i - \sum_{\substack{j=1 \\ j \neq i}}^{p} H_{i,j}\, x_j \end{aligned}$$

The solution of Equation (4.1) can be obtained by iteratively solving at each subdomain

$$L_i x_i^{(k)} = b_i - \hat{H}_i \hat{x}_i^{(k-1)} \tag{6.4}$$

where L_i, $\hat{H}_i$ and b_i are constant, and $\hat{x}_i^{(k-1)}$ is updated at each step by data exchange among the blocks or neighbour subdomains. This is the basic approach of Schwarz methods.

The same approach can be applied in the case of further overlapping, i.e. two or more layers of overlapping from a geometric viewpoint (see Figure 5.13). The main difference is that, in this case, there are replicated data and hence extra storage is needed, as depicted in Figure 6.2. The number of rows of each array is given by the number of core nodes plus the number of ghost nodes (for one layer of overlapping the number of ghost nodes is zero). Therefore, each array can be split into two main parts namely a core part and a ghost part, such as

$$v = \left[\begin{array}{c} \bar{v} \\ \hline \tilde{v} \end{array}\right]$$

where $\bar{v}$ represents the core nodes and $\tilde{v}$ the ghost nodes.

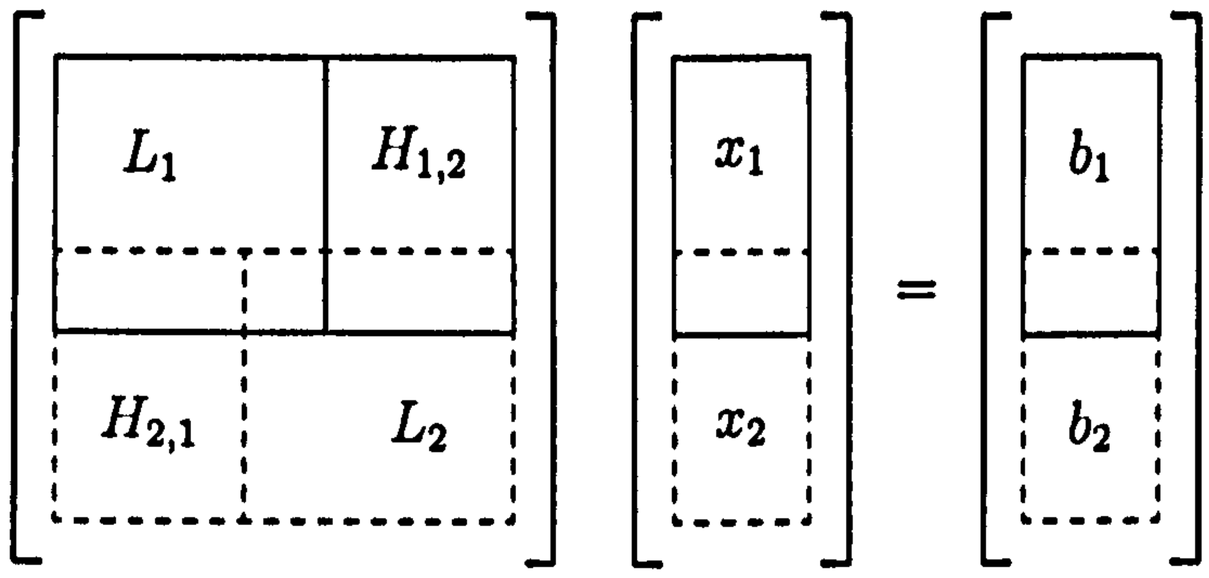


Figure 6.2: System sketch of a domain partitioned with more then one layer of overlapping.

6.2 Preliminary Study

In order to develop the algorithm presented in Section 6.3.1 a study on the behaviour of the conjugate gradient (CG) method as well as the generalized minimal residual (GMRES) method in a decomposed domain was performed. This study can be divided into three main cases. The first case only mimics parallelism and aims to examine the number of iterations for a given set of parameters. The second case has actual parallelism and aims to analyse both the number of iterations and the execution time for a given set of parameters. In both cases the algorithm used was called KASO - Krylov Additive Schwarz with Overlapping. The third case addresses directly the development of a procedure to accelerate the rate of convergence of CG as a distributed method.

The Parallel Iterative Methods (PIM) package by da Cunha and Hopkins [35, 34] was chosen as the linear solver to perform this preliminary study. PIM was chosen mainly due to its openness of design and also because the author had used it before. There are many other packages available. PIM routines can be used with user-supplied preconditioners (left-, right- and symmetric-preconditioning are supported), matrix-vector multiplication, dot product, among other routines.

In others words, a variety of routines can be tested using PIM as the main solver. A brief description of the CG and GMRES methods contained in PIM follows.

Conjugate Gradient

The CG method is used mainly to solve symmetric positive-definite (SPD) systems. It minimizes the residual in the A-norm and in exact arithmetic it terminates in at most n iterations. The method does not require the coefficient matrix; only the result of matrix-vector products Au is needed. It also requires a relatively small number of vectors to be stored per iteration since its iterates can be expressed by short, three-term vector recurrences. For more details, see [101, 48, 109], among many others, and Section 4.4.

The preconditioned conjugate gradient (PCG) algorithm is obtained by replacing the inner product (r_j, r_j) of the (non-preconditioned) CG algorithm (Algorithm 4.4.1, page 68) by (r_j, z_j), denoting by $r_j = b - Ax_j$ the original residual and by $z_j = M^{-1}r_j$ the residual of the preconditioned system. Algorithm 6.2.1, from PIM User's Guide [34], presents the PCG method which solves the system $Q_1AQ_2x = Q_1b$, where Q_1 and Q_2 are preconditioning matrices. See Section 6.3.2 for the derivation of PCG.

Restarted GMRES (RGMRES)

The GMRES method is a very robust method to solve nonsymmetric systems. The method uses the Arnoldi process to compute an orthonormal basis $\{v_1, v_2, \ldots, v_k\}$ of the Krylov subspace[1] $\mathcal{K}(A, v_1)$. The solution of the system is taken as $x_0 + V_k y_k$

[1] $\mathcal{K}_m = \mathcal{K}_m(A, v) \equiv span\{v, Av, A^2v, \ldots, A^{m-1}v\}$

Algorithm 6.2.1 Preconditioned Conjugate Gradient

1. $r_0 = Q_1(b - AQ_2x_0)$
2. $p_0 = r_0$
3. $\varrho_0 = r_0^T r_0$
4. $w_0 = Q_1AQ_2p_0$
5. $\xi_0 = p_0^T w_0$
6. for $k = 1, 2, \ldots$
7. $\quad \alpha_{k-1} = \varrho_{k-1}/\xi_{k-1}$
8. $\quad x_k = x_{k-1} + \alpha_{k-1}p_{k-1}$
9. $\quad r_k = r_{k-1} - \alpha_{k-1}w_{k-1}$
10. $\quad$ Check stopping criterion
11. $\quad s_k = Q_1AQ_2r_k$
12. $\quad \varrho_k = r_k^T r_k$
13. $\quad \delta_k = r_k^T s_k$
14. $\quad \beta_k = \varrho_k/\varrho_{k-1}$
15. $\quad p_k = r_k + \beta_k p_{k-1}$
16. $\quad w_k = s_k + \beta_k w_{k-1}$
17. $\quad \xi_k = \delta_k - \beta_k^2 \xi_{k-1}$
18. end for

where V_k is a matrix whose columns are the orthonormal vectors v_i, and y_k is the solution of the least-squares problem $H_k y_k = \| r_0 \|_2 e_1$, where the upper Hessenberg matrix H_k is generated using the Arnoldi process and $e_1 = (1, 0, 0, \ldots, 0)^T$. This least-squares problem can be solved using a QR factorization of H_k.

A problem that arises in connection with GMRES is that the number of vectors of order n that need to be stored grows linearly with k and the number of multiplications grows quadratically. This may be avoided by using a restarted version of GMRES (RGMRES). Instead of generating an orthonormal basis of dimension k, one chooses a value c, $c \ll n$, and generates an approximation to the solution using an orthonormal basis of dimension c, thereby reducing considerably the amount of storage needed. Algorithm 6.2.2 is the RGMRES presented in PIM User's Guide [34], where Q_1 and Q_2 are preconditioning matrices.

Although the restarted GMRES does not breakdown [102, pp. 865], it may, depending on the system and the value of c, produce a stationary sequence of residuals, thus not achieving convergence. Increasing the value of c usually cures this problem and may also increase the rate of convergence. Besides reducing the amount of storage and possibly the amount of computation needed, the restarted method can also speedup the solution as instanced in Figure 6.3.

6.2.1 Study 1: KASO MatLab

We consider the solution of a left-preconditioned system of n linear equations, derived from a finite element discretization (Galerkin scheme) of the Poisson equation

$$\nabla^2 \phi = -2\pi^2 \sin(\pi x) \sin(\pi y) \tag{6.5}$$

in the square $[-1, -1] \times [1, 1]$, subject to Dirichlet boundary conditions.

Algorithm 6.2.2 Restarted GMRES (RGMRES)

1. $r_0 = Q_1(b - AQ_2x_0)$
2. $\beta_0 = ||r_0||_2$
3. **for** $k = 1, 2, \ldots$
4. $\quad g = (\beta_{k-1}, \beta_{k-1}, \ldots)^T$
5. $\quad V_{:,1} = r_{k-1}/\beta_{k-1}$
6. $\quad$ **for** $j = 1, 2, \ldots,$ *restart*
7. $\quad\quad R_{i,j} = V_{:,i}^T Q_1 A Q_2 V_{:,j}, \quad i = 1, \ldots, j$
8. $\quad\quad \hat{v} = Q_1 A Q_2 V_{:,j} - \sum_{i=1}^{j} R_{i,j} V_{:,i}$
9. $\quad\quad R_{j+1,j} = ||\hat{v}||_2$
10. $\quad\quad V_{:,j+1} = \hat{v}/R_{j+1,j}$
11. $\quad\quad$ Apply previous Givens's rotations to $R_{:,j}$
12. $\quad\quad$ Compute Givens's rotation to zero $R_{j+1,j}$
13. $\quad\quad$ Apply Givens's rotation to g
14. $\quad\quad$ **if** $|g_{j+1}| <$ RHSSTOP **then**
15. $\quad\quad\quad$ Perform steps 20 and 21 with *restart*$\equiv j$
16. $\quad\quad\quad$ Stop
17. $\quad\quad$ **end if**
18. $\quad$ **end for**
19. $\quad$ Solve $Ry = g$ (solution to least-squares problem)
20. $\quad x_k = x_{k-1} + Vy$ (form approximate solution)
21. $\quad r_k = Q_1(b - AQ_2x_k)$
22. $\quad \beta_k = ||r_k||_2$
23. **end for**

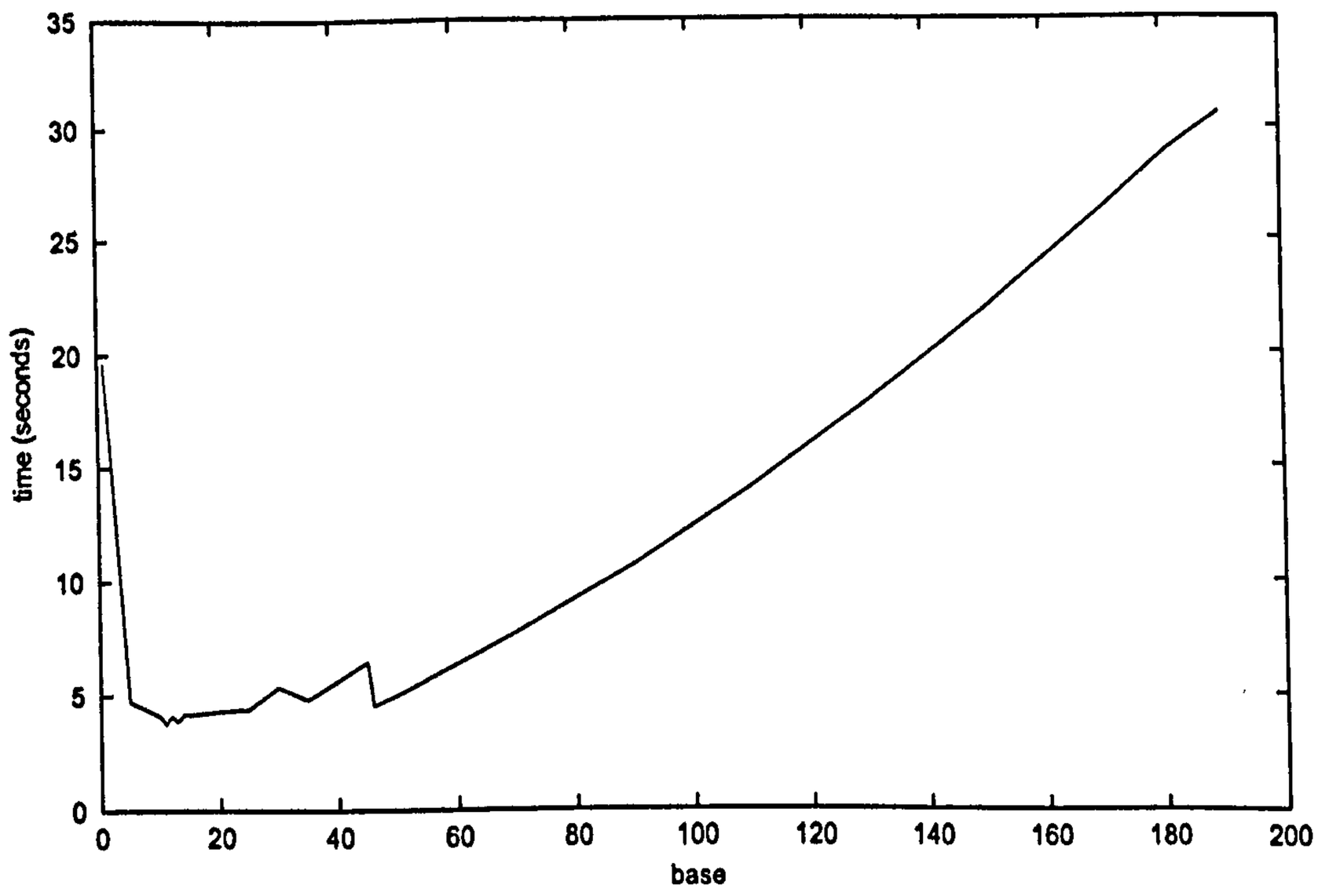


Figure 6.3: RGMRES execution time with different base sizes [14, Figure 4.16]. Matrix PLAT362: Platzman's oceanographic models, North Atlantic submodel; set PLATZ from the Harwell-Boeing Collection (http://math.nist.gov/MatrixMarket)

The results that follow were obtained running a MatLab implementation of Algorithm 6.2.3. This implementation mimics parallelism but is not actually executed in parallel. The test problem was discretized on a 40×40 regular grid of triangles and solved using several sets of parameters for the linear solver.

Algorithm 6.2.3 KASO

1. **for** Each processor p
2. Set $x_0 = 0$, ω, FB, C
3. **for** k=0,1,...
4. **if** $\|b - Ax\|_2 < 1e-6$ **then** $\omega = 0.6$
5. **if** $k > 1$ **then** $x_k = x_k + \omega(b - Ax_k)$
6. **if** C **then** Update x_{k_p} of artificial boundary
7. **for** $i = 1, 2, ..., nb$
8. Compute $x_{k_p}^{(i)}$ of interior nodes
9. **end for**
10. **if** FB **then**
11. **for** $i = nb-1, nb-2, ..., 1$
12. Compute $x_{k_p}^{(i)}$ of interior nodes
13. **end for**
14. **end if**
15. Average x_{p_k} on common nodes
16. **if** $\|b - Ax_k\|_2 < tol$ **then** Stop
17. **end for**
18. **end for**

Algorithm 6.2.3 depends on a set of parameters:

- np: number of processors or subdomains;
- nb: number of blocks; i.e. number of parts each subdomain is divided;

- ω: relaxation parameter;

- FB: true or false. If false only loops through the blocks from 1 to nb else first loops from 1 to nb and then from $nb-1$ to 1;

- C: true or false. If true the values of x on the artificial boundary are exchanged among processors hence the only means of communication among processors is the relaxation step.

The graphs are presented using the following aliases to simplify the reading:

`< method > < np x nb > [ C ] < F,FB > < ω > [ its ]`

where

- `method`:
  - `GMRES`: GMRES method without preconditioner
  - `KASO`: KASO method without preconditioner
  - `KASO-GMRES`: GMRES method preconditioned by KASO
- `np`: number of processors or subdomains
- `nb`: number of blocks per subdomain
- `C`: with communication among processors (optional)
- `F`: FB is false
- `FB`: FB is true
- `its`: number of iterations performed

The vector $x_{k_p}^{(i)}$, steps 8 and 12 of Algorithm 6.2.3, is calculated at each iteration using one iteration of the GMRES method. Note that this means that one sweep of the outer loop (step 3) of Algorithm 6.2.2 is performed and the number of inner loops is defined by the restarting parameter used.

Figure 6.4 shows the residual for GMRES and KASO with several sets of parameters. It shows that with the right choice of parameters KASO can even outperform GMRES. Figure 6.5 shows the residual for GMRES preconditioned by KASO. It is clear that the number of iterations is greatly reduced by using the preconditioner for any set of parameters. Figures 6.6, 6.7 and 6.8 show the residual for KASO with different sets of parameters and several relaxation parameters $-\omega$. Also in this case the efficiency of the algorithm depends strongly on the choice of parameters.

Figure 6.9 shows the number of iterations performed to achieve a given accuracy running KASO on 1 processor with 4 blocks per subdomain. For the 4 cases presented, only one converges for all the 9 relaxation parameters tested. This shows that the relaxation parameter might improve the rate of convergence but has to be carefully chosen. Figures 6.10, 6.11, 6.12 and 6.13 show the number of iterations for the GMRES method and KASO preconditioner for different sets of parameters. It is not clear from this graphs which set of parameters is the most suitable.

This first case study shows that it is possible to solve a system of linear equations distributed over several processors (subdomains). However, the algorithm used does not perform well for all sets of parameters and it was not possible to establish a best set of parameters for any case. In the case study presented in the next section a slightly different algorithm is analysed from an execution time standpoint.

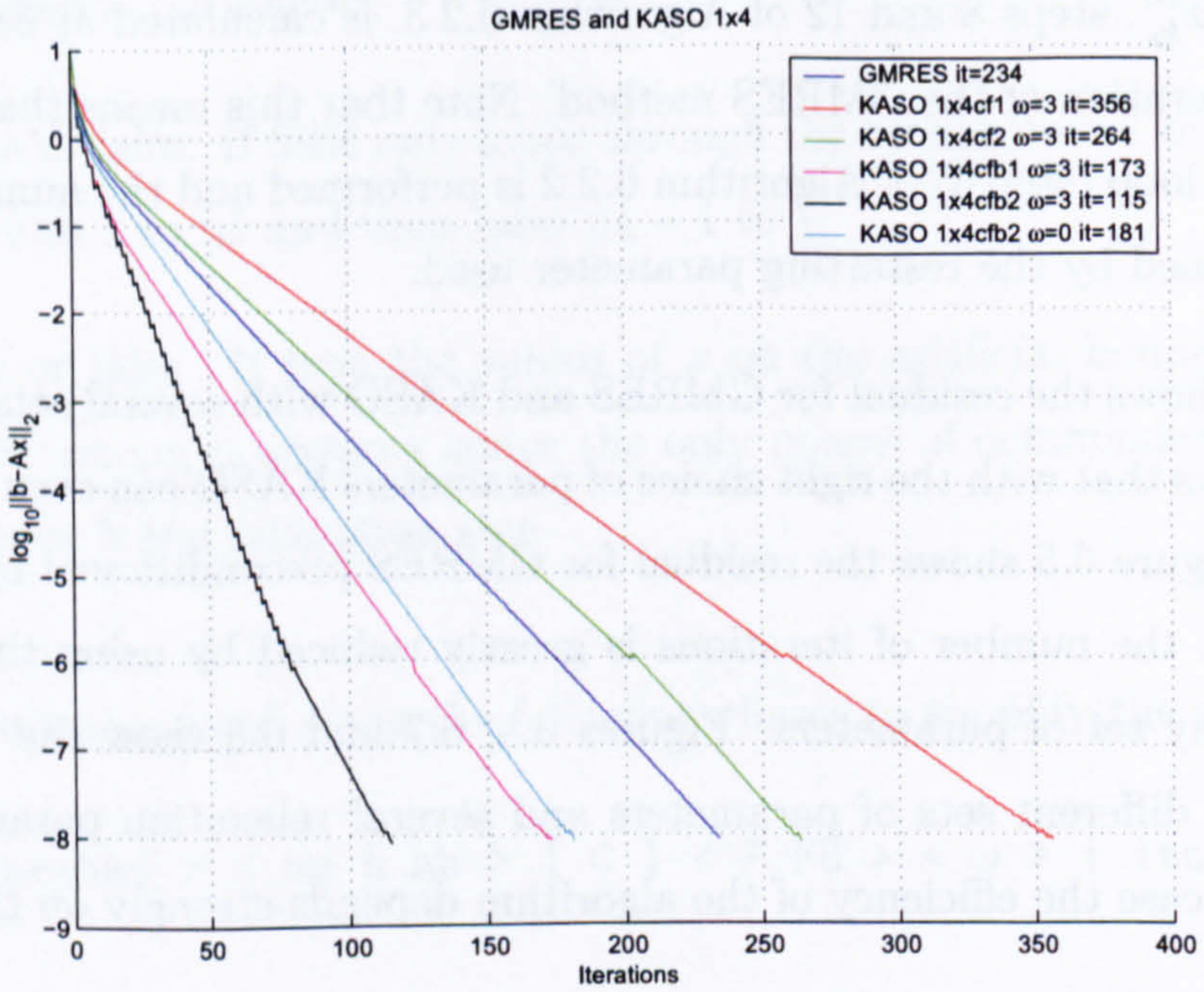


Figure 6.4: GMRES and KASO 1×4

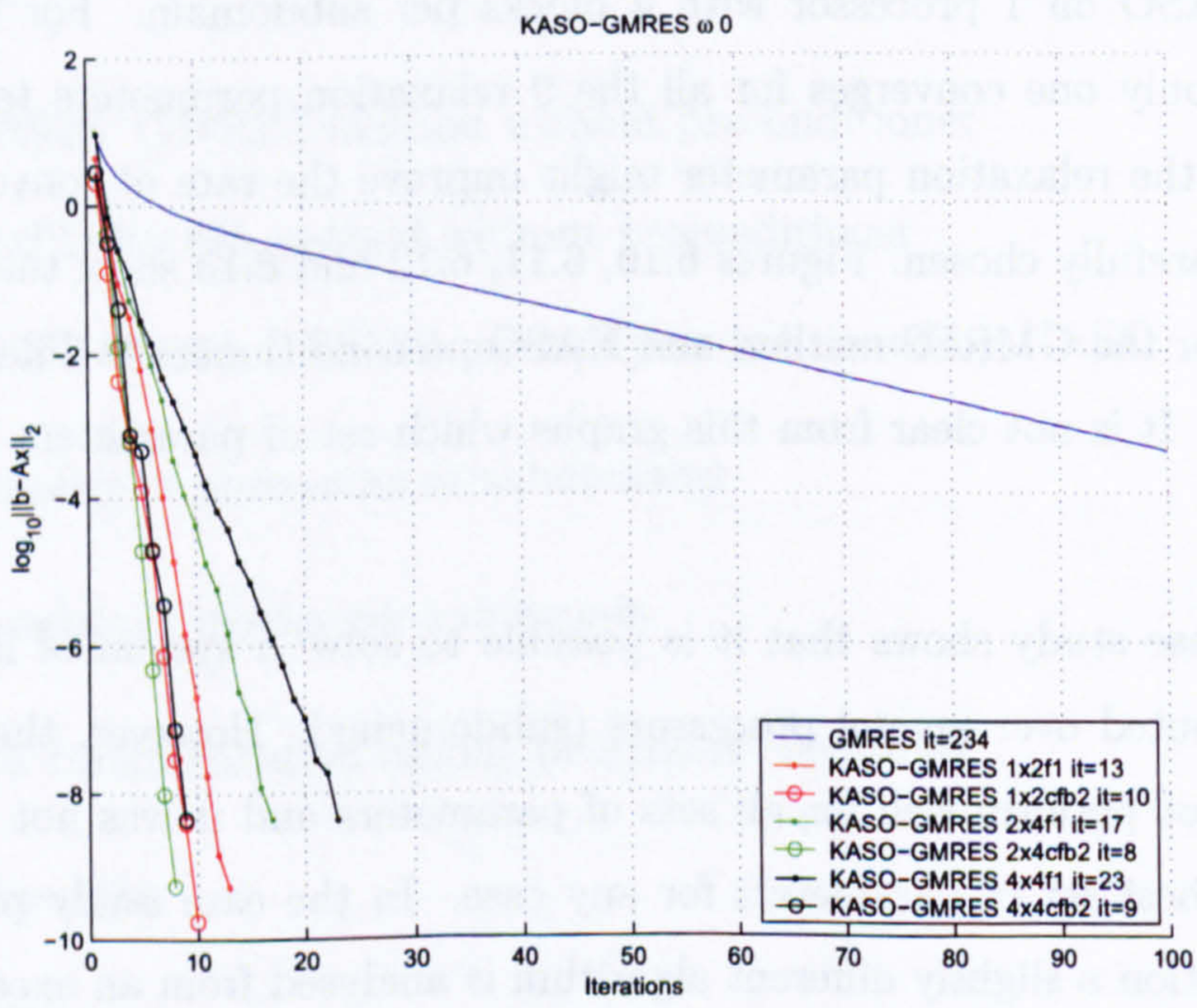


Figure 6.5: KASO-GMRES $\omega = 0$

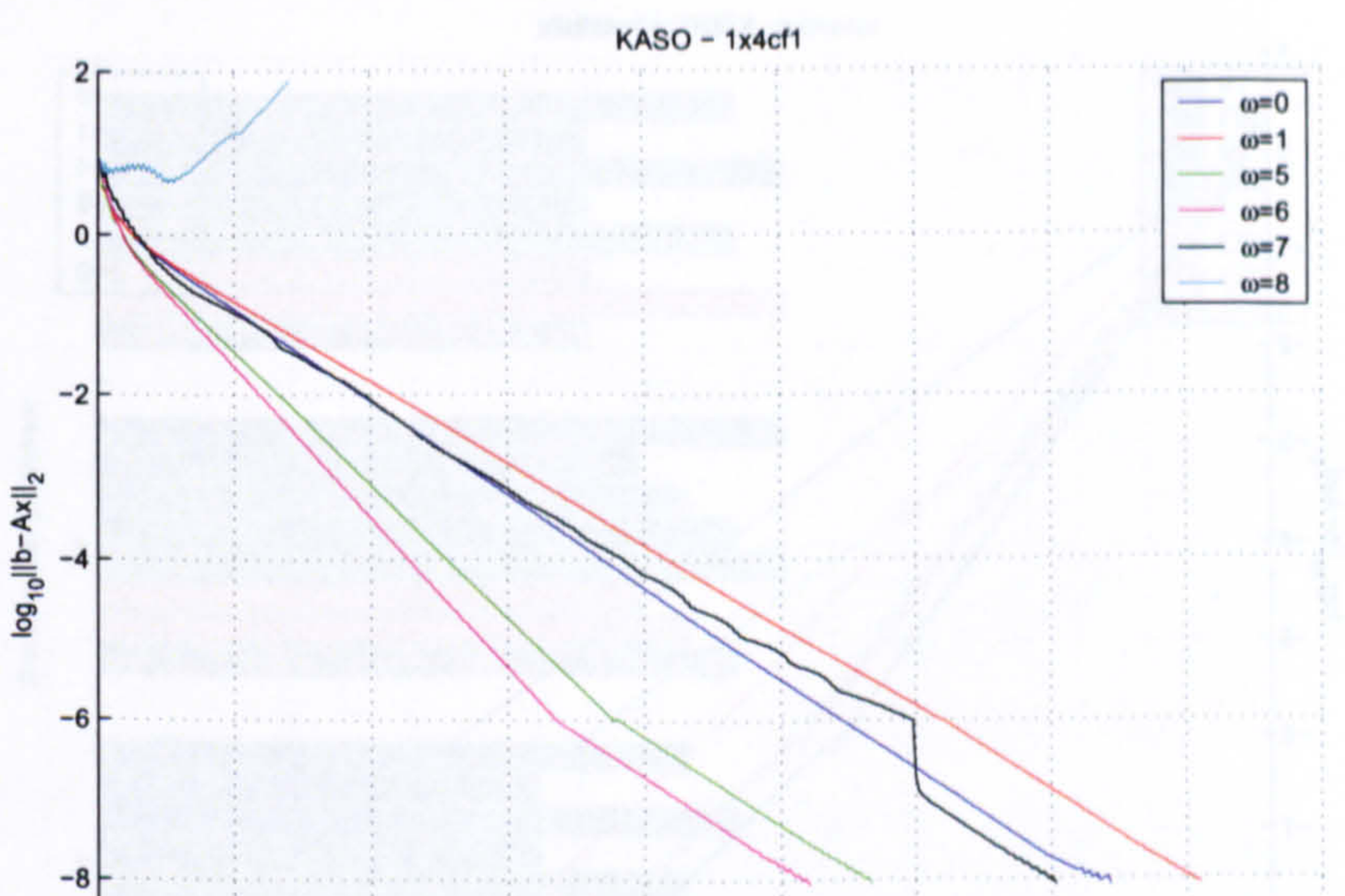


Figure 6.6: KASO 1×4cf1

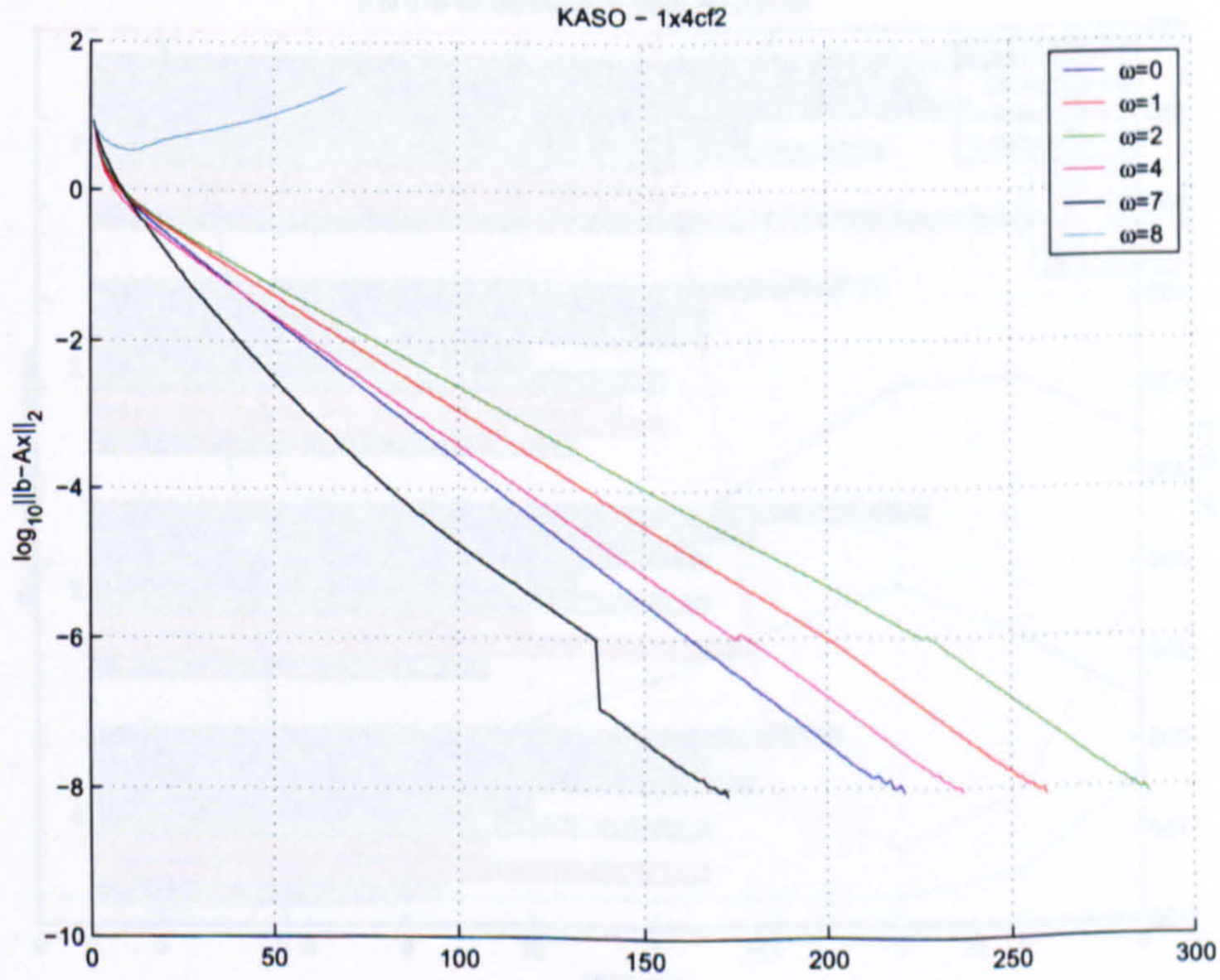


Figure 6.7: KASO 1×4cf2

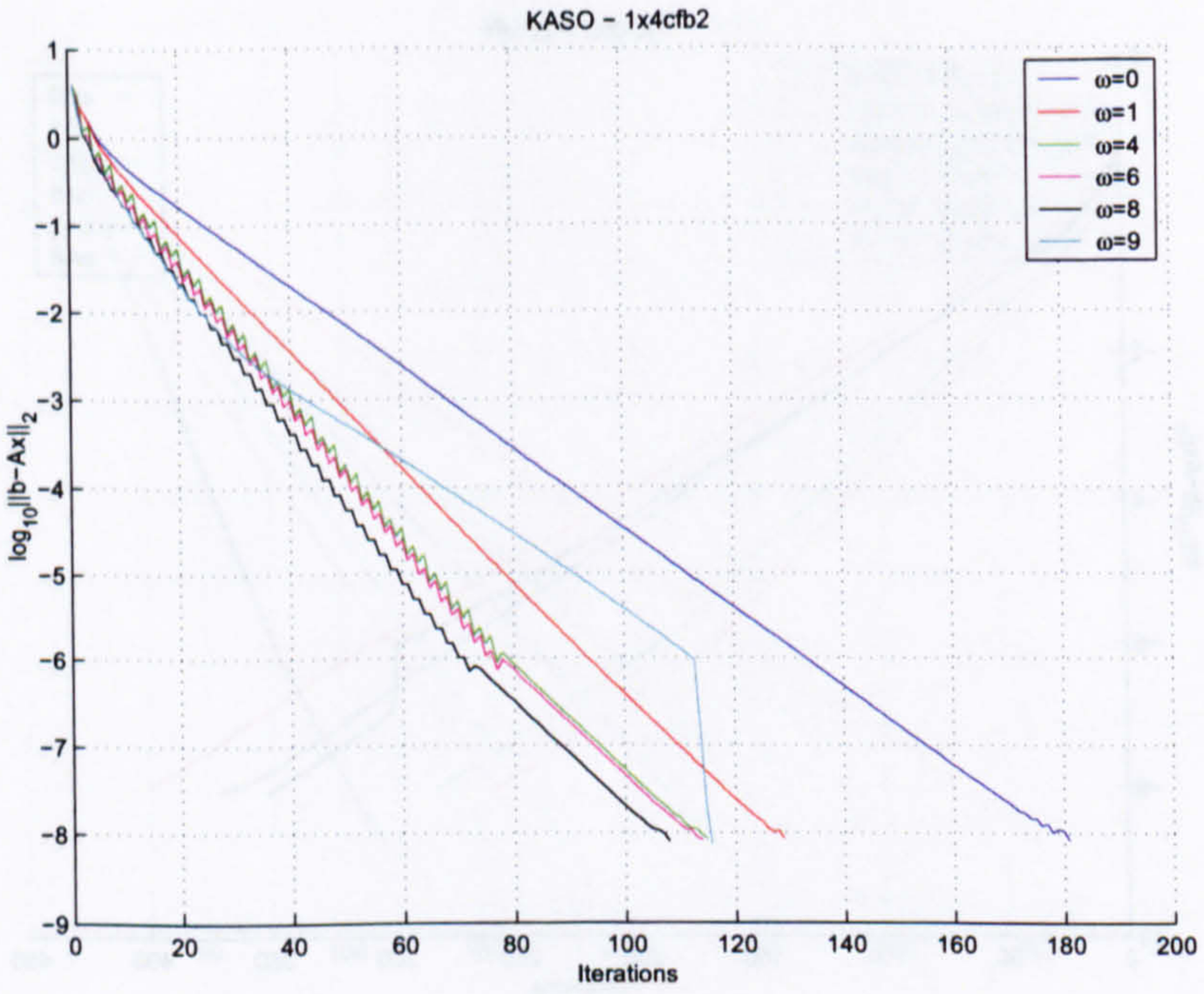


Figure 6.8: KASO 1×4cfb2

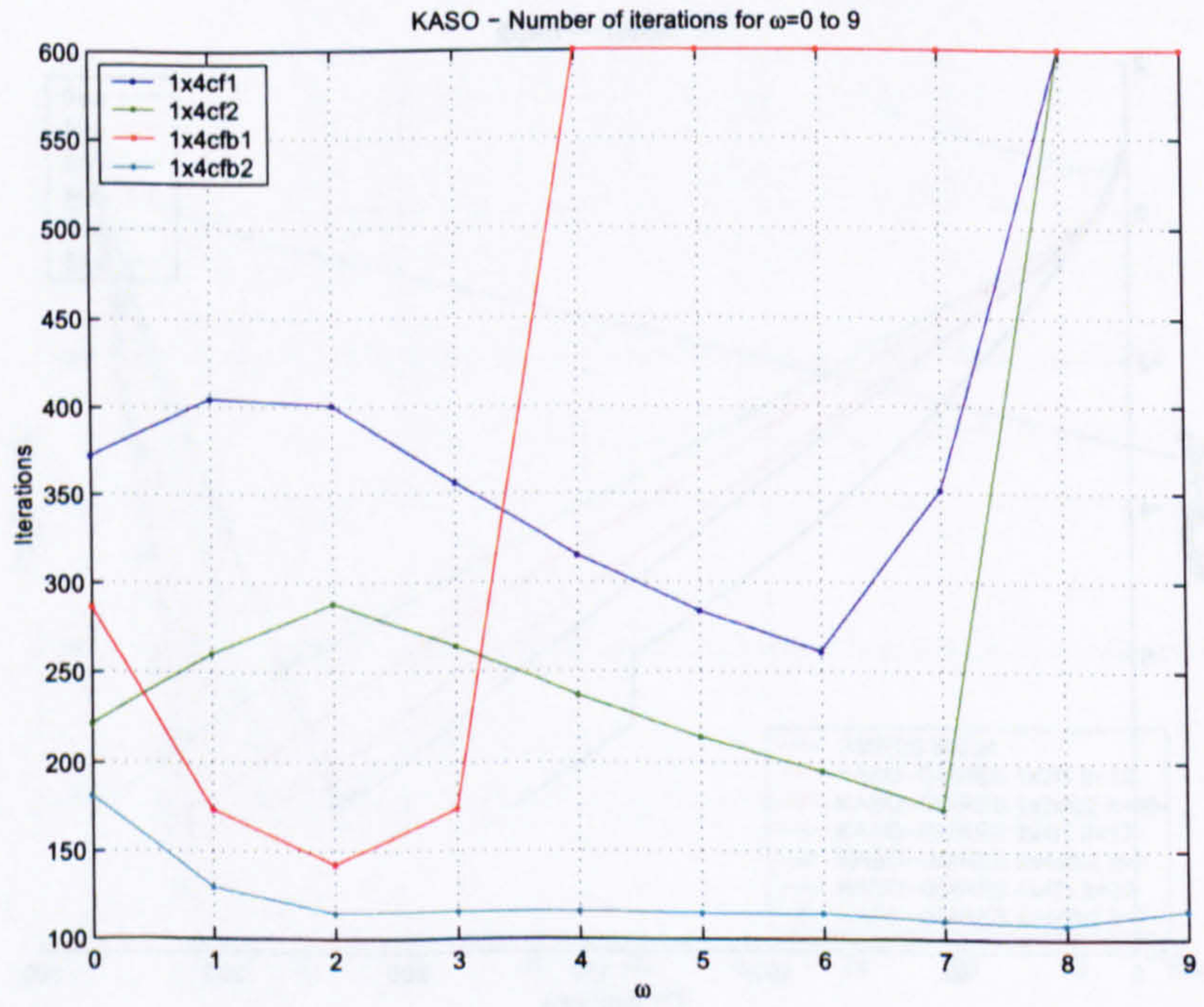


Figure 6.9: KASO - Number of iterations for ω=0 to 9

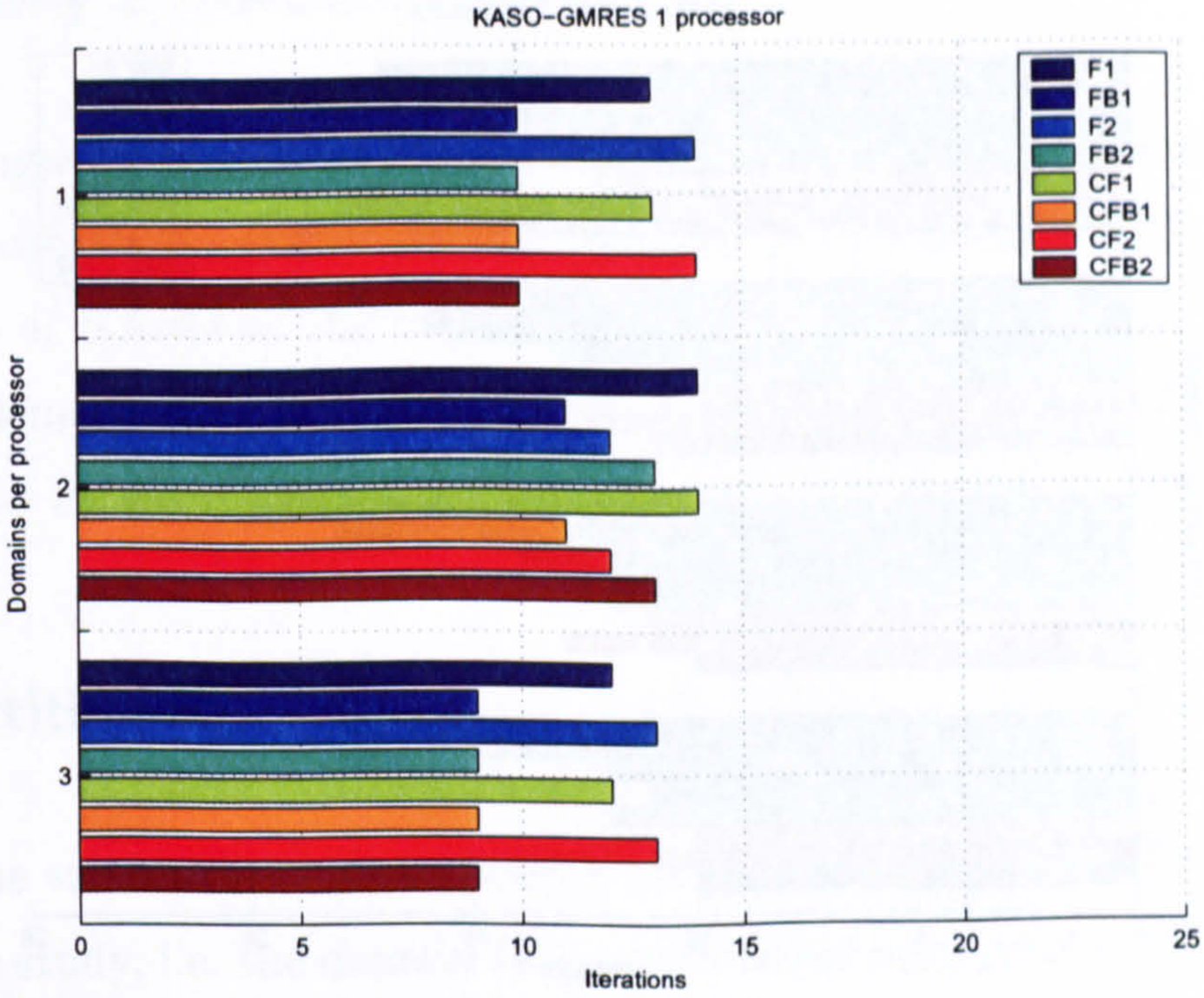


Figure 6.10: KASO-GMRES 1 processor

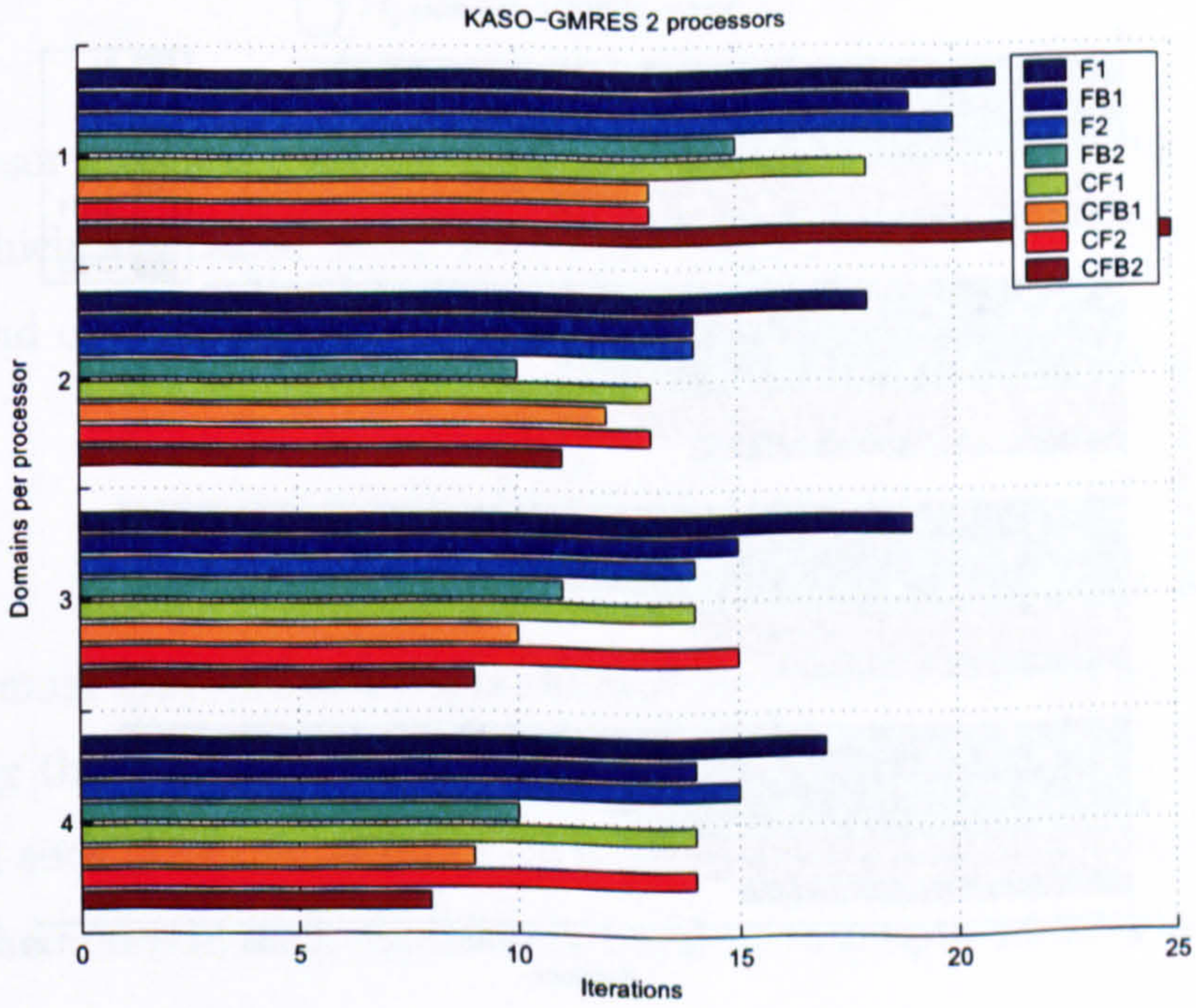


Figure 6.11: KASO-GMRES 2 processors

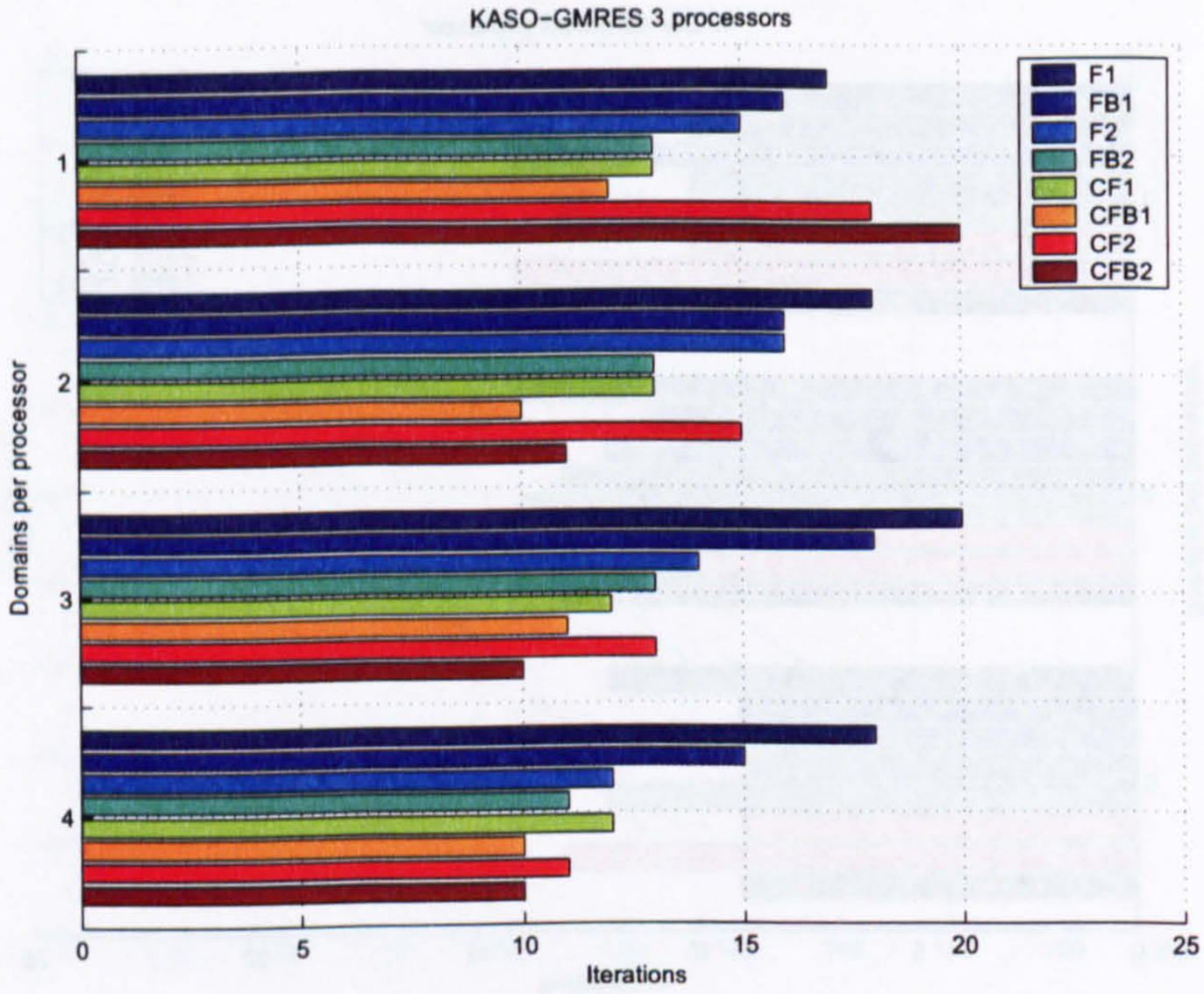


Figure 6.12: KASO-GMRES 3 processors

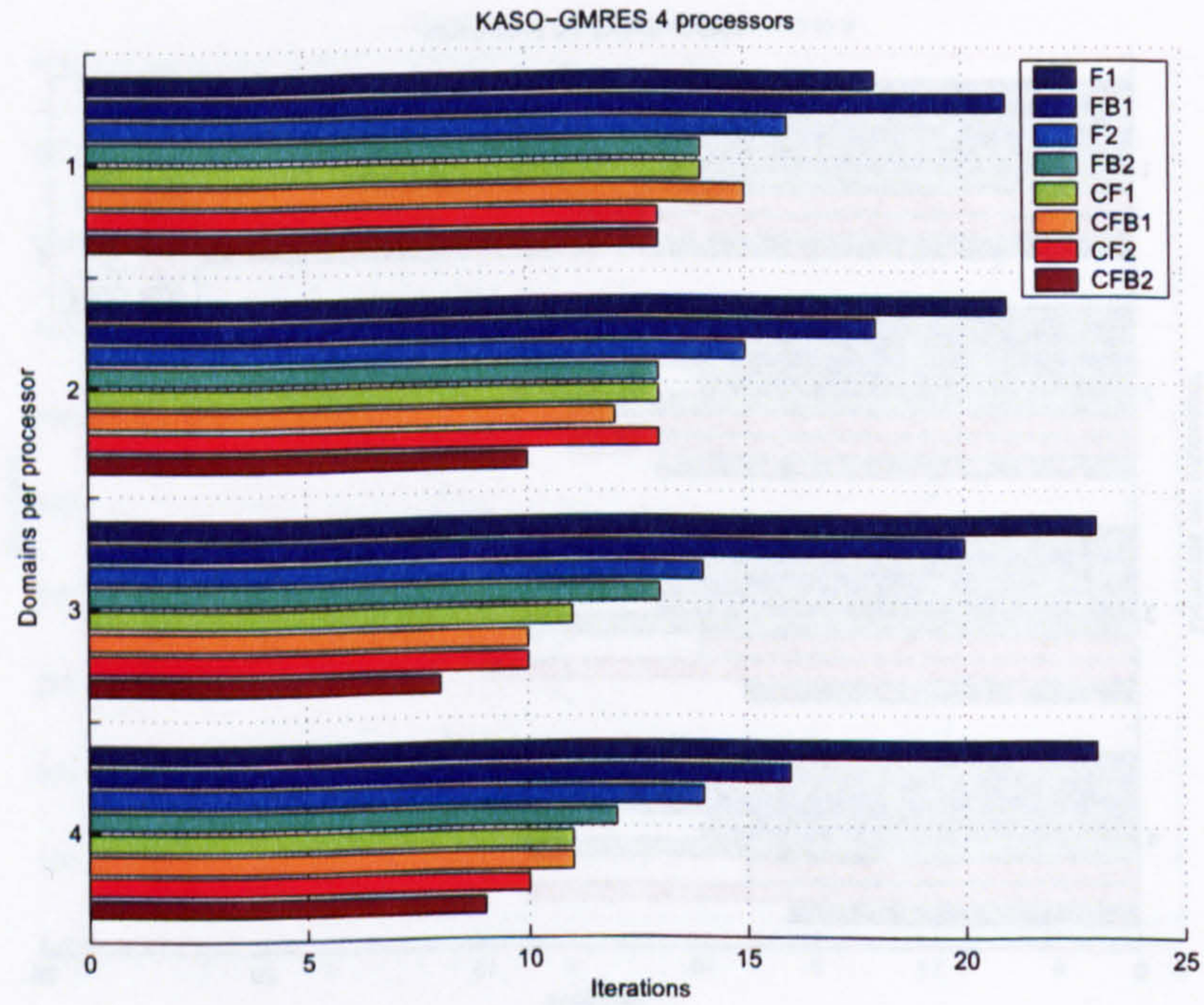


Figure 6.13: KASO-GMRES 4 processors

6.2.2 Study 2: KASO Fortran

A Fortran implementation of Algorithm 6.2.3 was developed to analyse the behaviour of both the GMRES and CG methods preconditioned by KASO or the performance of KASO as the solver. The Fortran implementation is an actual parallel implementation and therefore parallelism can be examined. The results were obtained by solving some of the test problems described in Section 7.1.

Mesh Partitioning

Basically, the same procedure to partition the mesh described in Section 5.6.2 is used on this study, i.e. the domain Ω is vertex-based partitioned into np (number of processors) subdomains Ω_p, such that

$$\bigcup_{p=1}^{np} \Omega_p = \Omega \quad \text{and} \quad \bigcap_{p=1}^{np} \Omega_p = \emptyset$$

Each processor stores the nodes of its own Ω_p and the nodes directly connected to them, which are called local- and halo-nodes, respectively. The halo nodes introduce and overlap and hence on the final partition

$$\bigcap_{p=1}^{np} \Omega_p \neq \emptyset$$

Each subdomain may be further partitioned into blocks. The partitioning is done by repeating the procedure described above on each subdomain. The nodes are renumbered such that the nodes of each block have a continuous numbering and, therefore, the nodes of each subdomain have a continuous numbering.

The partitioning into subdomains and blocks are distinguished because each subdomain can be allocated to a different processor whilst all blocks of a given subdo-

main should belong to the same processor, i.e. communication may happen among subdomains but not among blocks.

Data Partitioning

The data partitioning is an important aspect of the parallelization of any method. In this context, the matrix of coefficients A is row-wise partitioned which means that only the edges whose first node is local are kept. Edges can have two local nodes or one local and one halo node, which are called local- and halo-edges, respectively. Each processor stores

$$A_p = [H_{p,1} \cdots H_{p,p-1} \;\; L_p \;\; H_{p,p+1} \cdots H_{p,p}],$$

where L_p is the matrix of local-edges and $H_{p,i}$ are the matrices of halo-edges. If i is not a neighbour of p then $H_{p,i} = 0$.

Vectors and matrices rows are stored locally starting from position 1 to number of local nodes. This numbering can be translated to a global numbering if required. Note that each processor has all columns of A. Matrix A and a vector v are given by

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_{np} \end{bmatrix} = \begin{bmatrix} L_1 & H_{1,2} & \cdots & H_{1,np} \\ H_{2,1} & L_2 & \cdots & H_{2,np} \\ \vdots & \vdots & \ddots & \vdots \\ H_{np,1} & H_{np,2} & \cdots & L_{np} \end{bmatrix} \qquad v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{np} \end{bmatrix}$$

where np is the number of processors. This is the same data partition presented in Section 6.1. Only the matrix of local nodes L_p is partitioned into blocks, such that

$$L_p = \begin{bmatrix} \mathrm{D}_{p_1} & \mathrm{C}_{p_{1,2}} & \cdots & \mathrm{C}_{p_{1,nb}} \\ \mathrm{C}_{p_{2,1}} & \mathrm{D}_{p_2} & \cdots & \mathrm{C}_{p_{2,nb}} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{C}_{p_{nb,1}} & \mathrm{C}_{p_{nb,2}} & \cdots & \mathrm{D}_{p_{nb}} \end{bmatrix}$$

where nb is the number of blocks, D_{p_i} are matrices of local-edges, and $C_{p_{i,j}}$ are matrices of halo-edges. This gives us a set of local and halo nodes at a block level, that will be called block-local-edges and block-halo-edges, respectively. For instance, with 2 processors and 3 blocks per processor

$$A = \left[\begin{array}{ccc|ccc} \mathrm{D}_{1_1} & \mathrm{C}_{1_{1,2}} & \mathrm{C}_{1_{1,3}} & & & \\ \mathrm{C}_{1_{2,1}} & \mathrm{D}_{1_2} & \mathrm{C}_{1_{2,3}} & & \mathrm{H}_{1,1} & \\ \mathrm{C}_{1_{3,1}} & \mathrm{C}_{1_{3,2}} & \mathrm{D}_{1_3} & & & \\ \hline & & & \mathrm{D}_{2_1} & \mathrm{C}_{2_{1,2}} & \mathrm{C}_{2_{1,3}} \\ & \mathrm{H}_{2,1} & & \mathrm{C}_{2_{2,1}} & \mathrm{D}_{2_2} & \mathrm{C}_{2_{2,3}} \\ & & & \mathrm{C}_{2_{3,1}} & \mathrm{C}_{2_{3,2}} & \mathrm{D}_{2_3} \end{array}\right]$$

Derivation

The algorithm used in this case initially follows the same steps presented in Section 6.1. Therefore, given the system of n linear equations

$$Ax = b, \tag{6.6}$$

the partitioned system for each processor may be written as

$$\left[\begin{array}{ccccccc} H_{p,1} & \cdots & H_{p,p-1} & L_p & H_{p,p+1} & \cdots & H_{p,np} \end{array} \right] x = b_p \tag{6.7}$$

which is undetermined.

Considering halo nodes as Dirichlet-type boundary nodes, Equation (6.7) can be written as

$$L_p\, x_p = b_p - \left[\begin{array}{ccccccc} H_{p,1} & \cdots & H_{p,p-1} & 0 & H_{p,p+1} & \cdots & H_{p,np} \end{array} \right] \left[\begin{array}{cccc} x_1 & x_2 & \cdots & x_{np} \end{array} \right]^T$$

or

$$L_p\, x_p = b_p - \sum_{\substack{i=1 \\ i \neq p}}^{np} H_{p,i}\, x_i = y_p \tag{6.8}$$

In the same way, assuming that block-halo nodes are Dirichlet-type boundary nodes, the solution for each block is given by

$$D_{p_k}\, x_{p_k} = y_{p_k} - \sum_{\substack{i=1 \\ i \neq k}}^{nb} C_{p_{k,i}}\, x_{p_i} \tag{6.9}$$

where

$$\left[\begin{array}{cccc} y_{p_1} & y_{p_2} & \cdots & y_{p_{nb}} \end{array} \right]^T = y_p \tag{6.10}$$

The solution is given solving a new problem each time, with boundary conditions updated from the most recent subdomain or block solution

$$y_p^{(j)} = b_p - \sum_{\substack{i=1 \\ i \neq p}}^{np} H_{p,i}\, x_i^{(j-1)} \tag{6.11}$$

$$x_{p_k}^{(j)} = D_{p_k}^{-1} \left(y_{p_k}^{(j)} - \sum_{\substack{i=1 \\ i \neq k}}^{nb} C_{p_k,i}\ x_{p_i}^{(j-1)} \right) \tag{6.12}$$

Results

A series of tests were performed in order to understand how both CG and GMRES work when combined to KASO as a preconditioner. Some tests were also performed to evaluate KASO as a solver. All tests were run using RGMRES with the restarting parameter set to 5 and a tolerance of 1.0E-10. In order to identify the method and preconditioner used, either the label *method-preconditioner* or *method* is applied. The latter meaning that no preconditioner is used. The parameters listed on the tables (e.g. `maxit` and `restart`) refer to the preconditioner. The letters B, S, and M are used as alias for blocks, steps and maxit, respectively. The maximum number of iterations executed is called `maxit` and `steps` is the number of iterations of KASO.

Tables 6.1 and 6.2 show the number of iterations and execution time of a problem defined on the unit square domain. Compared to the number of iterations without preconditioning and for most of the sets of parameters, the number of iterations is quite small. However, the total time is even higher then the non-preconditioned solution. This shows that KASO is still computationally too expensive.

Table 6.3 shows the number of iterations and execution time of a problem solved by GMRES-KASO using GMRES, Jacobi and Jacobi with relaxation to solve the preconditioning problem. For most of the cases, the preconditioner reduces the number of iterations quite drastically and is faster then the non-preconditioned solution. However, it is only faster when the right set of parameters is used and how to choose the set of parameters is still not understood.

For this specific problem, Jacobi with relaxation was the fastest solver. The same technique was used on other problems without success. In some situations, the results were similar to the ones without relaxation. Note that the introduction of a relaxation step means extra parallel operations, since the residual has to be calculated.

The same set of tests was run on a regular grid. Table 6.4 shows the results. Note that the behaviour is similar to the one on irregular grids and the fastest solution is quite faster than the non-preconditioned solution. Once more, the main drawback is how to choose the right set of parameters.

The residual of a two-dimensional problem is shown on Figures 6.14 and 6.15. The convergence is monotonic for both the non-preconditioned solution and the preconditioned case. Note that the more steps the fewer iterations to solve the overall problem. However, more steps means also more time per iteration and, therefore, not necessarily a faster solution in terms of total execution time.

Figures 6.16 and 6.17 show the convergence of GMRES and KASO (as a solver not a preconditioner). GMRES needs fewer iterations than KASO (as expected) and both methods convergence is monotonic. Figure 6.16 also shows that the convergence is slightly worse for 10 blocks than for a single block. As can be seen on Figure 6.17, the convergence rate may be improved increasing the number of iterations per step. The last case, named *maxit=1+*, starts with 1 iteration per step and increases it by 1 at each step, i.e. maxit is equal the step number. The result is not satisfactory.

Figure 6.18 shows the residual of the approximation using CG and GMRES on the first iteration, second step, mesh `square1`. For CG, the magnitude of the residuals are closely related to the magnitude of the solution (see Figure 7.1 for a contour plot of the exact solution) while for GMRES the greater residuals are located on

Steps	Maxit	Procs	Restart=2	Restart=5	Restart=10
nonpre	-	2	-	147	-
	-	4	-	147	-
	-	6	-	147	-
1	1	2	73	65	-
		4	90	45	-
		6	168	125	-
	2	2	41	-	-
		4	44	-	-
		6	45	-	-
	4	2	23	-	-
		4	29	-	-
		6	27	-	-
2	1	2	26	21	-
		4	25	20	-
		6	25	21	-
	2	2	20	19	18
		4	21	20	23
		6	22	22	26
	4	2	19	18	18
		4	19	20	26
		6	20	22	29
4	1	2	21	20	-
		4	21	21	-
		6	21	19	-
	2	2	20	-	-
		4	19	-	-
		6	19	-	-
	4	2	21	13	13
		4	21	15	16
		6	21	15	-

Table 6.1: GMRES-KASO iterations, two-dimensional problem

Steps	Maxit	Restart=2	Restart=5	Restart=10
1	1	1.0	1.0	-
	2	1.0	-	-
	4	1.0	-	-
2	1	0.5	1.0	-
	2	0.5	1.0	2.0
	4	1.0	2.0	3.0
4	1	1.5	1.0	-
	2	1.5	-	-
	4	1.0	1.5	4.5

Table 6.2: GMRES-KASO - time per iteration for 2 processors, two-dimensional problem

Blocks	Steps	GMRES(5)		Jacobi		Jacobi ω=1.8	
		Its	Time	Its	Time	Its	Time
1	nonpre	375	2.0				
1	2	13	1.5	20	1.5	13	1.0
	4	9	1.5	20	1.0	13	1.0
	16	5	4.0	9	2.0	-	-
10	2	15	1.5	29	1.0	27	1.5
	4	9	1.5	21	1.5	14	1.5
	16	5	3.0	9	2.0	-	-

For all cases, GMRES maxit=2 and Jacobi maxit=4

Table 6.3: GMRES-KASO execution time, two-dimensional problem

Steps	Maxit	Procs	Its	Time
nonpre	-	2	14094	3065
		4	14094	877
		8	14094	410
2	1	2	-	-
		4	-	-
		8	2456	696
	2	2	826	2700
		4	1071	1237
		8	1085	555
	4	2	443	2418
		4	498	780
		8	588	416
4	1	2	969	3619
		4	1122	1390
		8	1333	723
	2	2	569	3437
		4	629	1238
		8	788	709
	4	2	177	1769
		4	217	653
		8	256	331*
	8	2	151	2869
		4	214	1110
		8	247	575
8	4	2	136	2631
		4	132	746
		8	146	397

* Fastest

Table 6.4: GMRES-KASO iterations, two-dimensional problem, uniform grid, n=160,000

the interface of the subdomains. Figures 6.19 and 6.20 also show the residual but for a three-dimensional problem. The same behaviour of the two-dimensional problem is observed. This suggests that CG might be suitable to be used at each subdomain almost independently. This possibility is further discussed in the next sections.

In summary, is has been shown that KASO is efficient if the right set of parameters is chosen. However, it was not possible to define how to choose these best parameters. The number of iterations is always reduced quite drastically by using KASO as a preconditioner but the total execution time is not necessarily reduced, i.e. the time per iteration of KASO is too high. The relaxation factor speedups the convergence in some cases but is not reliable. Moreover, the distribution of the residuals for the CG method suggests that it might be possible to adapt the method to solve each subdomain middling independently.

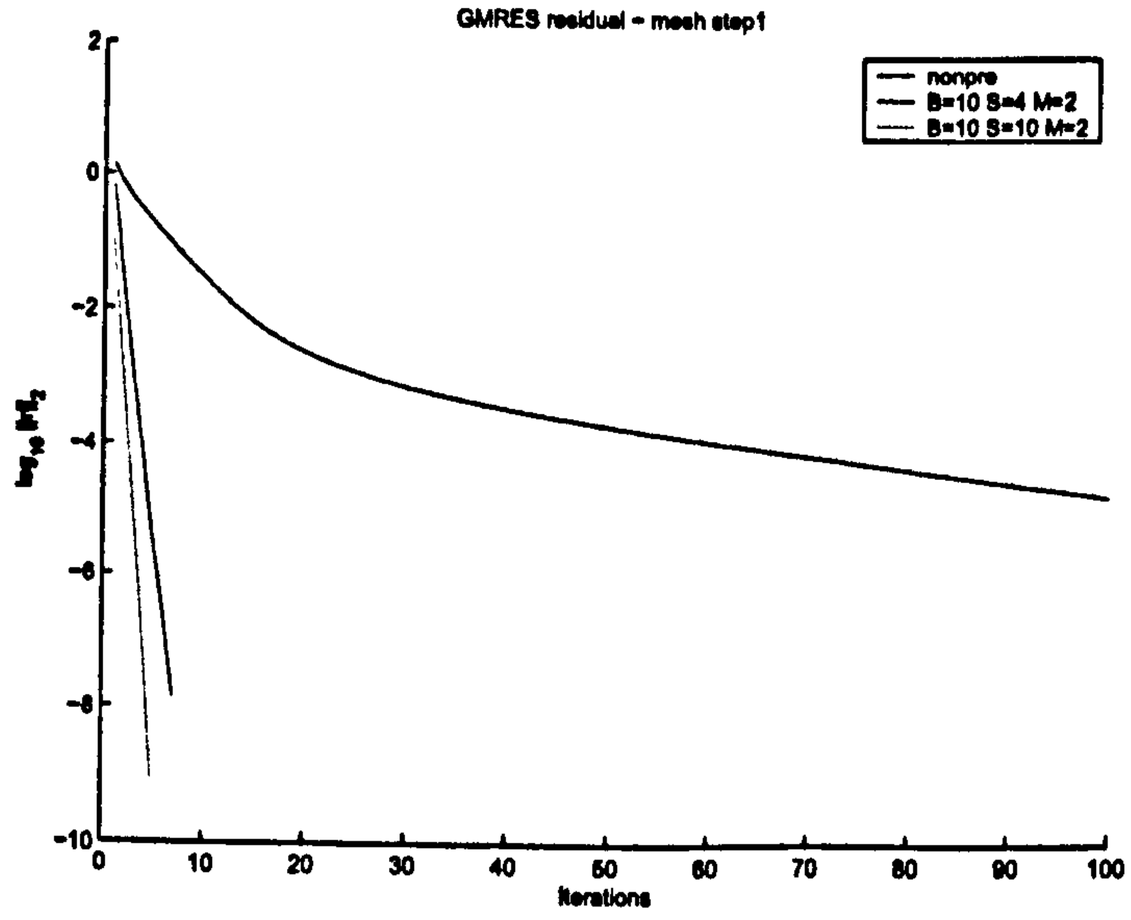


Figure 6.14: GMRES and GMRES-KASO residual, two-dimensional problem

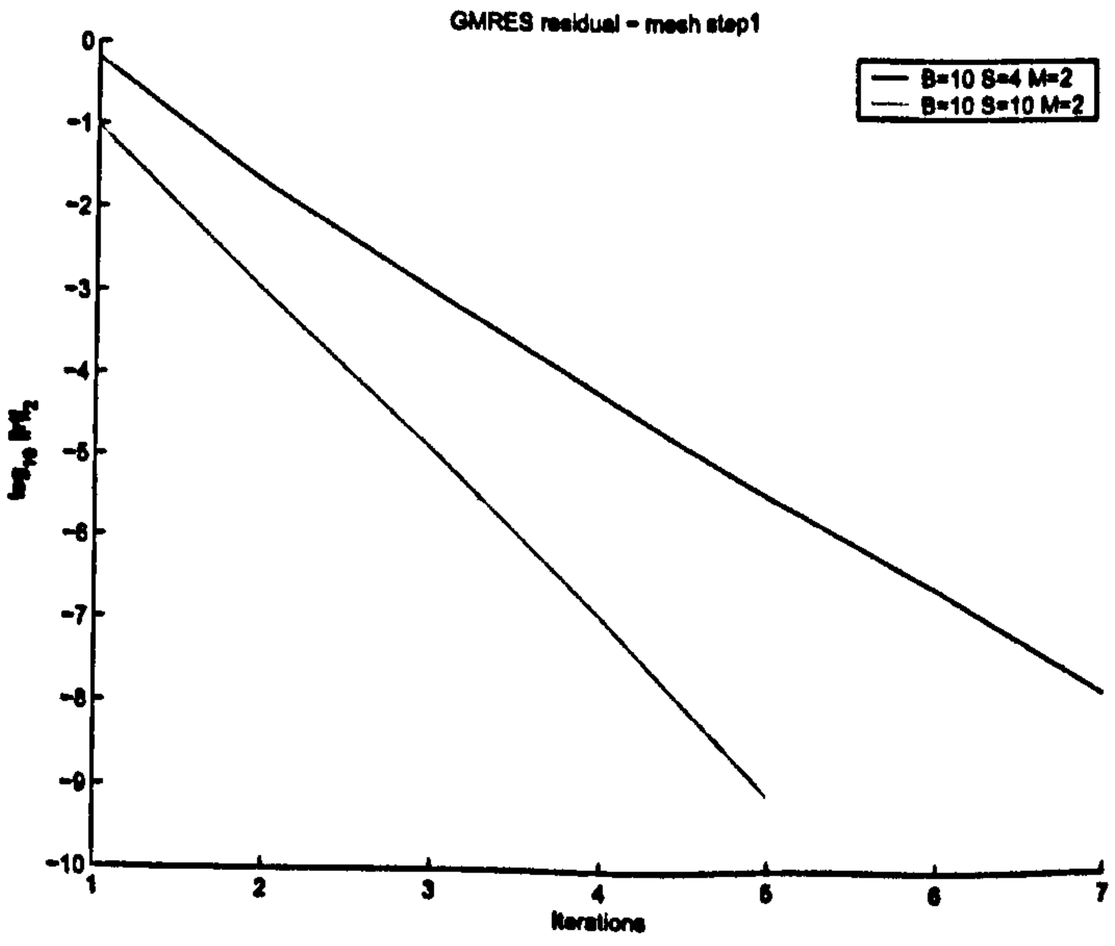


Figure 6.15: GMRES-KASO residual, two-dimensional problem

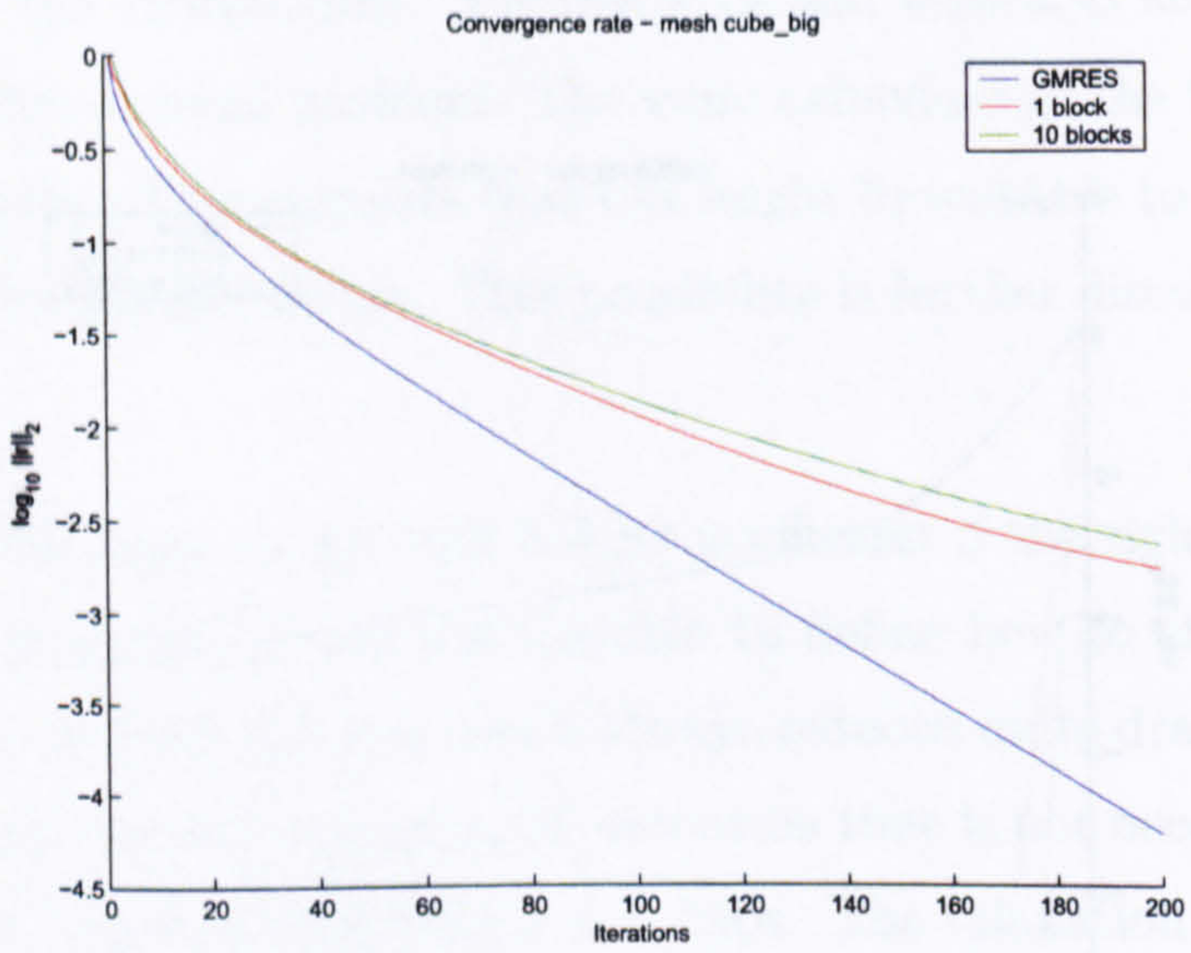


Figure 6.16: Residual of GMRES and KASO - 2 iterations, three-dimensional problem

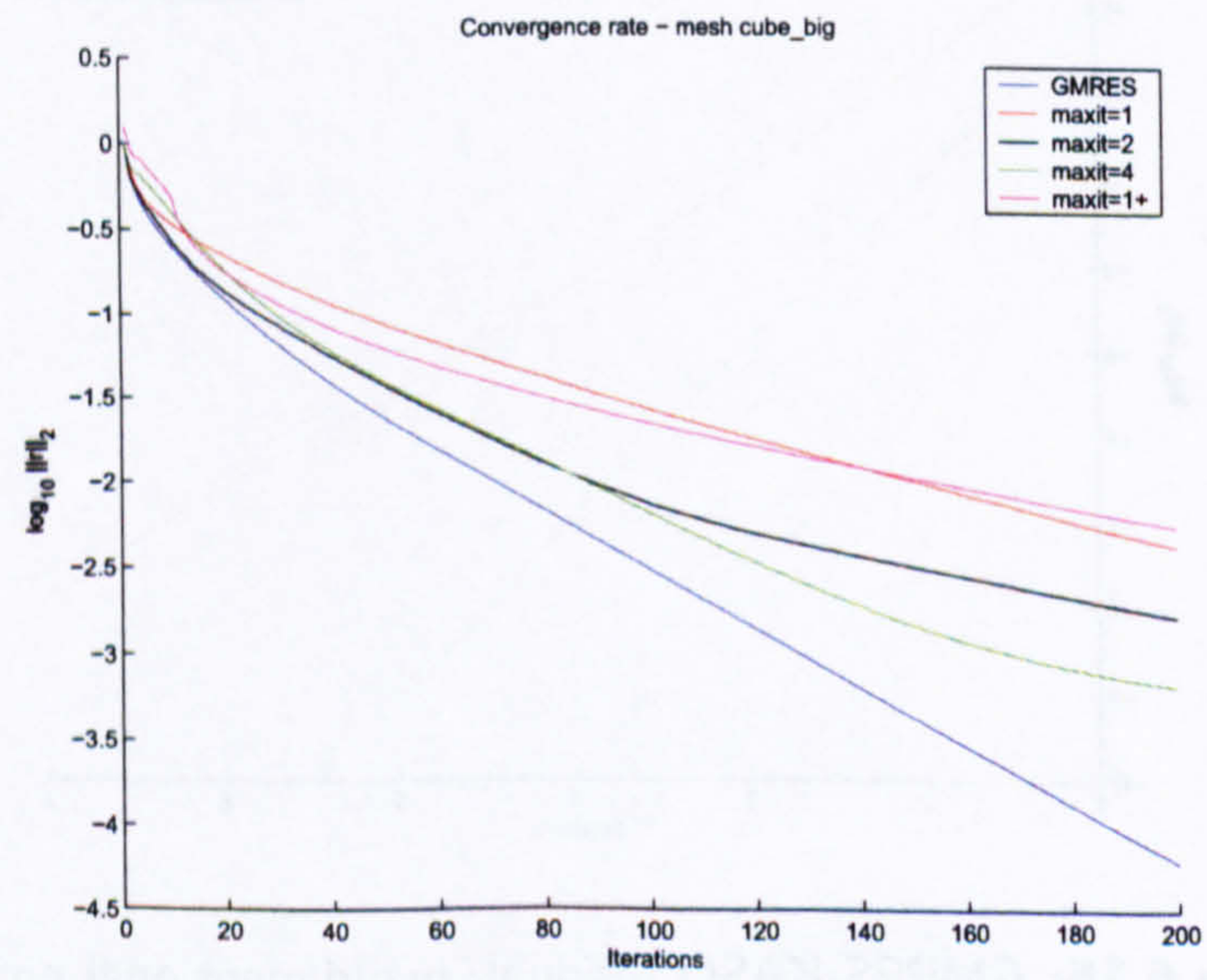


Figure 6.17: Residual of GMRES and KASO - 4 blocks, three-dimensional problem

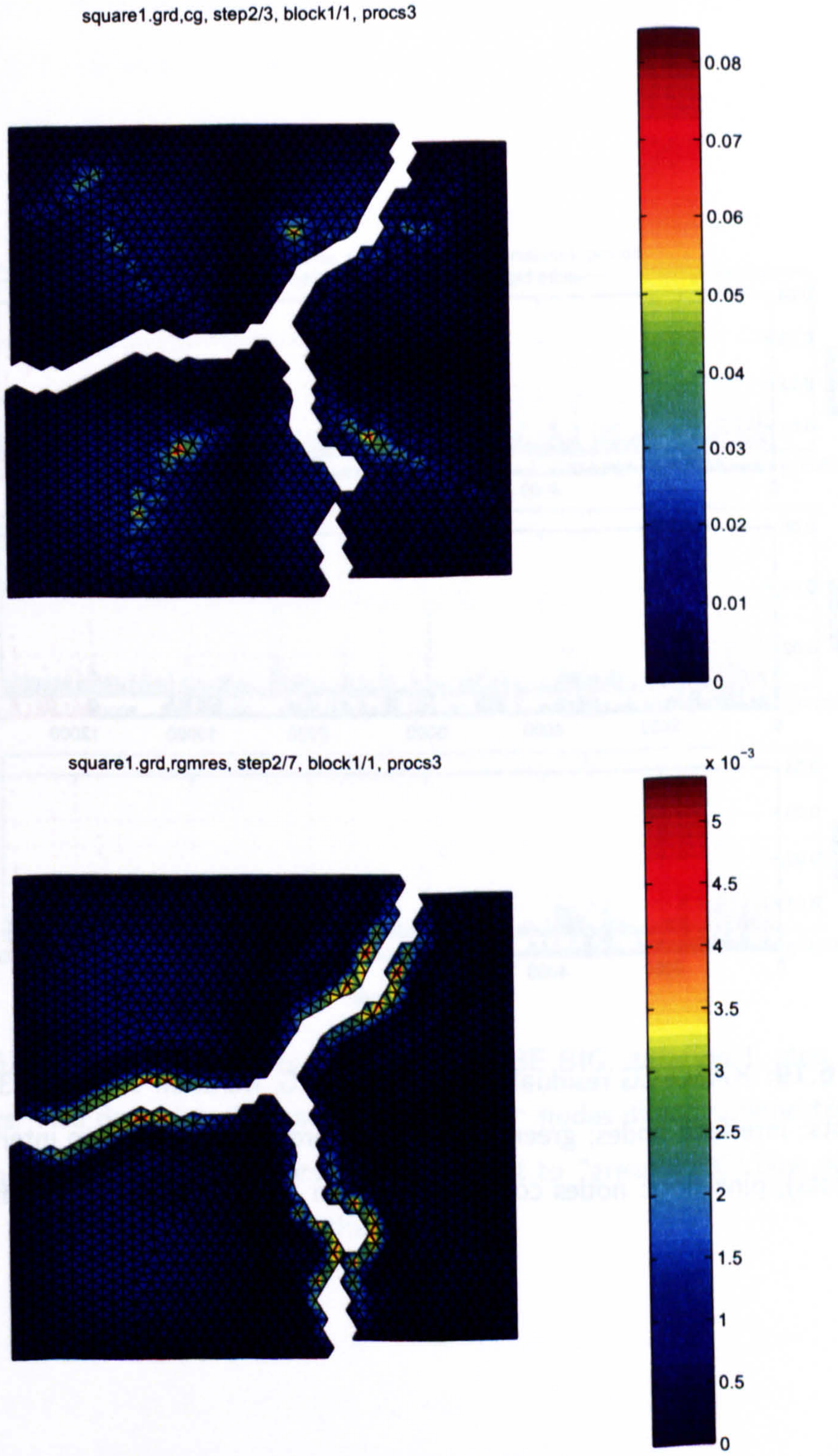


Figure 6.18: KASO-CG (top) and KASO-RGMRES (bottom) residual, mesh SQUARE1, iteration 1, step 2

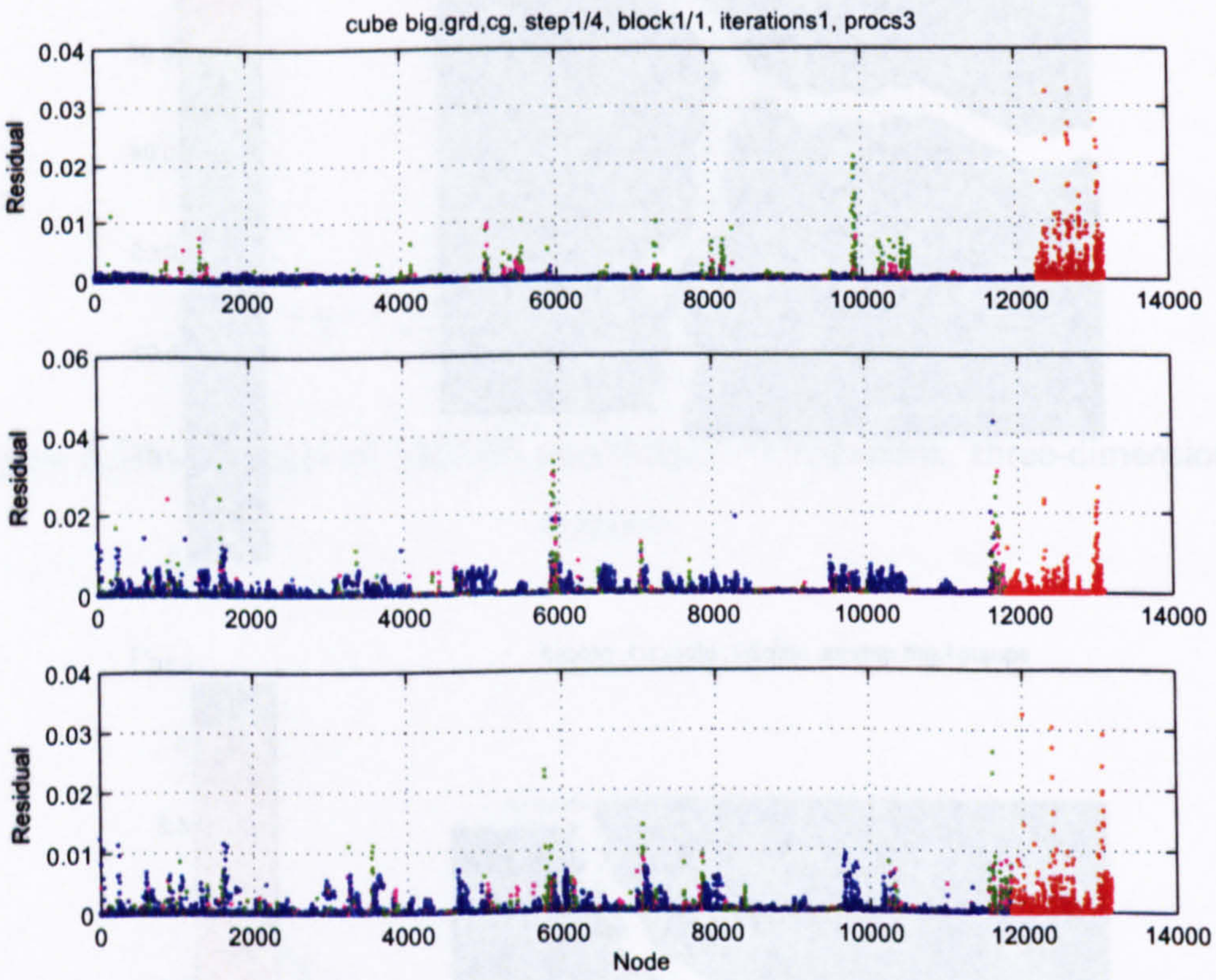


Figure 6.19: KASO-CG residual, mesh CUBE BIG, iteration 1, step 1, 3 processors. Red dots: interface nodes; green dots: nodes directly connected to interface nodes (red dots); pink dots: nodes connect to "green dots"; blue dots: all other nodes.

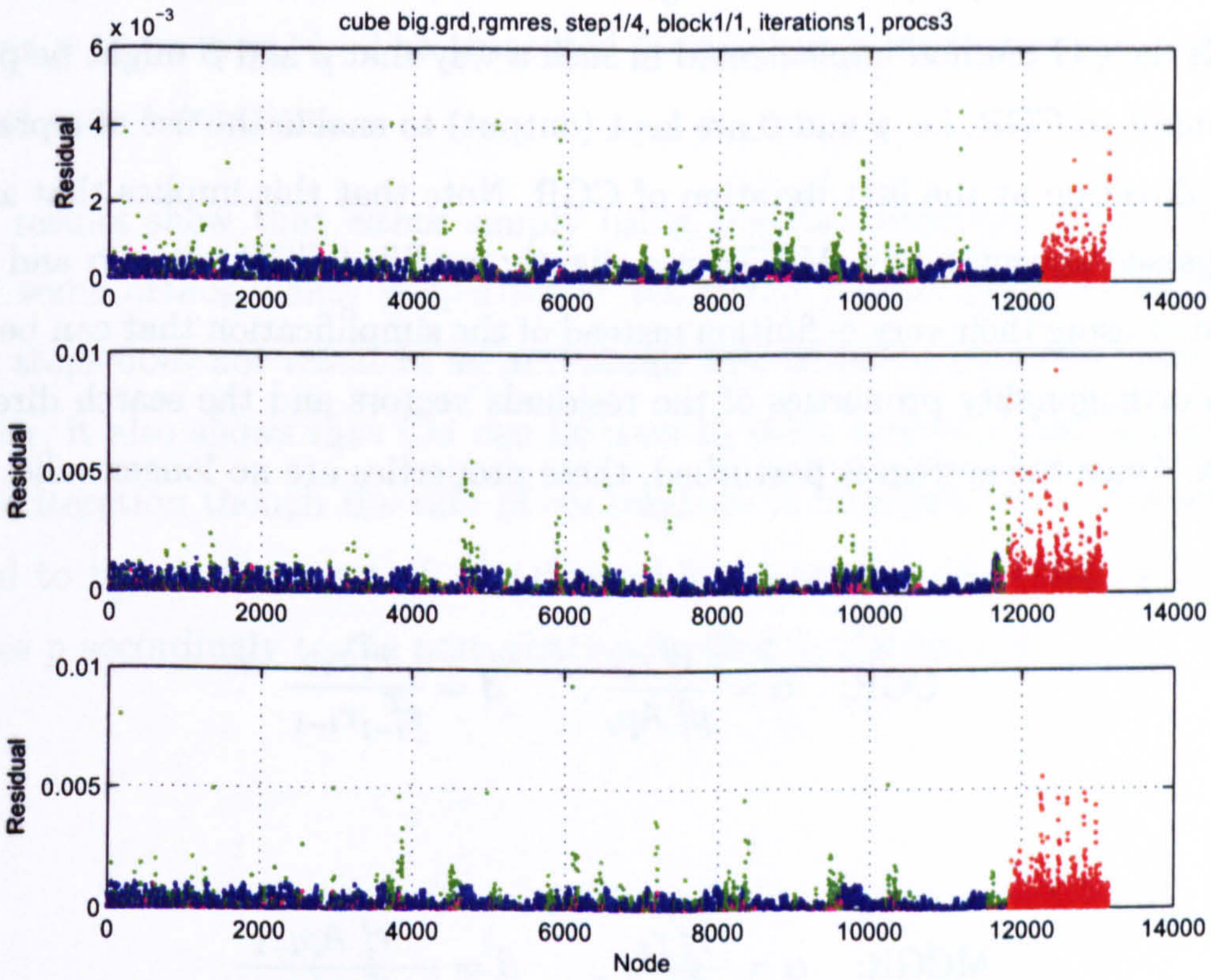


Figure 6.20: KASO-GMRES residual, mesh CUBE BIG, iteration 1, step 1, 3 processors. Red dots: interface nodes; green dots: nodes directly connected to interface nodes (red dots); pink dots: nodes connect to "green dots"; blue dots: all other nodes.

6.2.3 Study 3: CG restarted and truncated

In this section, a series of tests to analyse the convergence rate of the conjugate gradient method when used to solve a system which right-hand-side is perturbed at each iteration is presented. The algorithms used were called CGR and MCGR. CGR is the CG method implemented in such a way that p and β might be passed as an input to CGR, i.e. p and β are kept (output) to enable the use of a previous search direction at the first iteration of CGR. Note that this implies that x_0 has to be passed appropriately. MCGR is a slightly modified CGR where α and β are computed using their very definition instead of the simplification that can be done due to orthogonality properties of the residuals vectors and the search direction vectors. Once the system is perturbed, these properties are no longer valid.

$$\text{CGR:} \quad \alpha = \frac{r_i^T r_i}{p_i^T A p_i}, \qquad \beta = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$\text{MCGR:} \quad \alpha = \frac{r_i^T r_i}{r_i^T A p_i}, \qquad \beta = -\frac{r_i^T A p_{i-1}}{p_{i-1}^T A p_{i-1}}$$

The system to be solved is given by

$$Ax = b + \varepsilon_i$$

where ε_i is the perturbation applied to the system at each iteration. Notice that if $\varepsilon = 0$ both algorithms have $r_i^T r_j = 0$ and $p_i^T A p_j = 0 \ \forall\, i \neq j$, i.e. the orthogonality properties of the residuals and search directions are maintained. If $\varepsilon \neq 0$, MCGR only guarantees the orthogonality for $j = i - 1$.

Figures 6.21 to 6.32 show the A-norm of the residual ($r^T Ar$) for a series os tests. The matrix of coefficients used is the Poisson matrix from the MatLab gallery[2]. The right-hand-side vector was chosen such that the solution is a vector of ones. The graphs show the solution of the system without perturbation (b), the system perturbed by a random ε with p always set as r (b+e, p=r) and with p set as the previous p (b+e, p=r+). The truncated term refers to setting $r = b - Ax$ and $p = r$ at each t iterations.

These results show that either simply using the very definition of α and β to match some orthogonality properties or restarting (truncating) the solution at each t steps does not result in an acceptable rate of convergence for most cases. However, it also shows that CG can be used to solve a system that is perturbed at each iteration though the rate of convergence is relatively low. This analyses has led to the development of Algorithm 6.3.1, presented in Section 6.3.1, that updates p accordingly to the perturbation applied to the system.

[2]MatLab help: GALLERY('POISSON',N) is the block tridiagonal (sparse) matrix of order N^2 resulting from discretizing Poisson's equation with the 5-point operator on an N-by-N mesh.

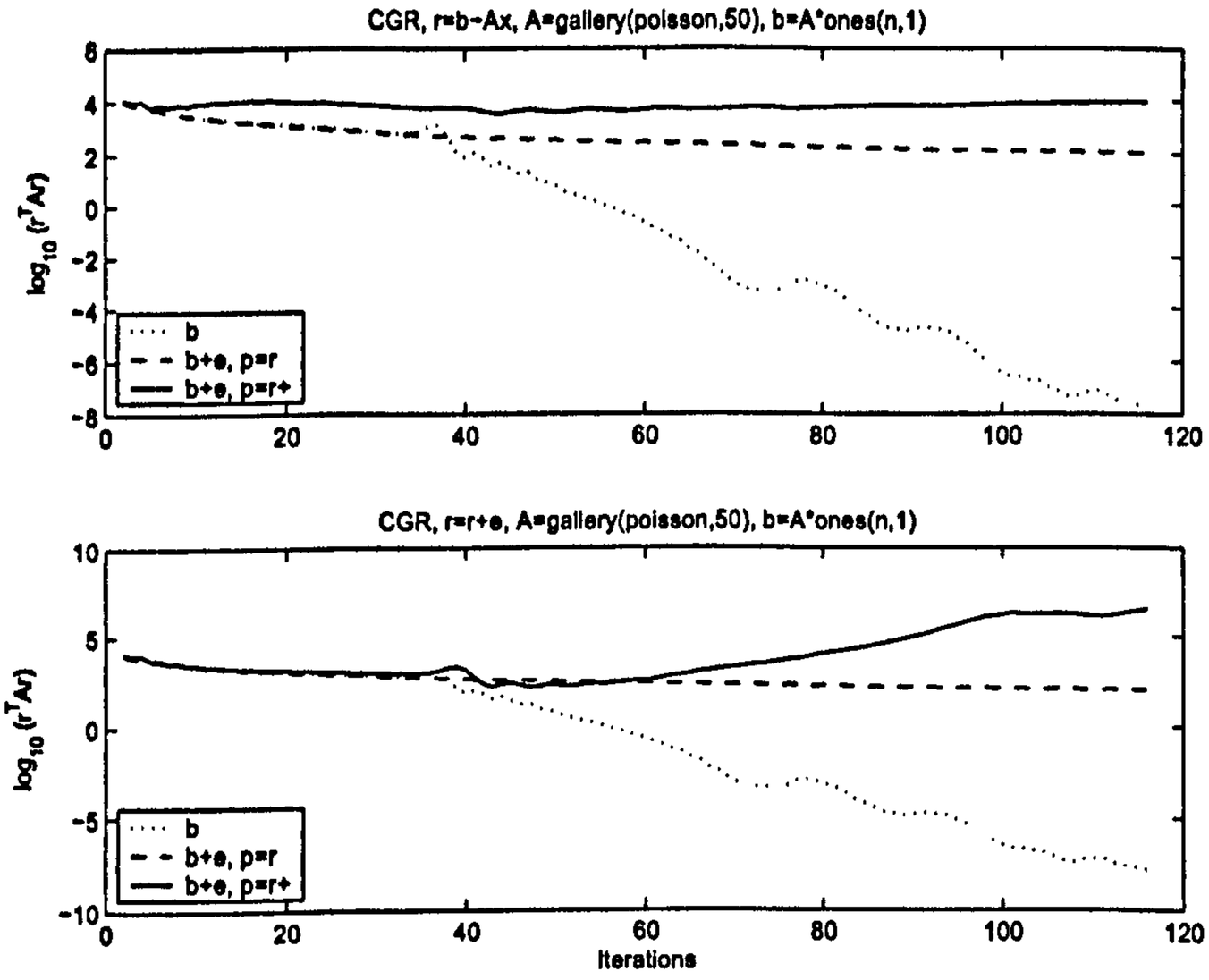


Figure 6.21: CGR - Non-truncated

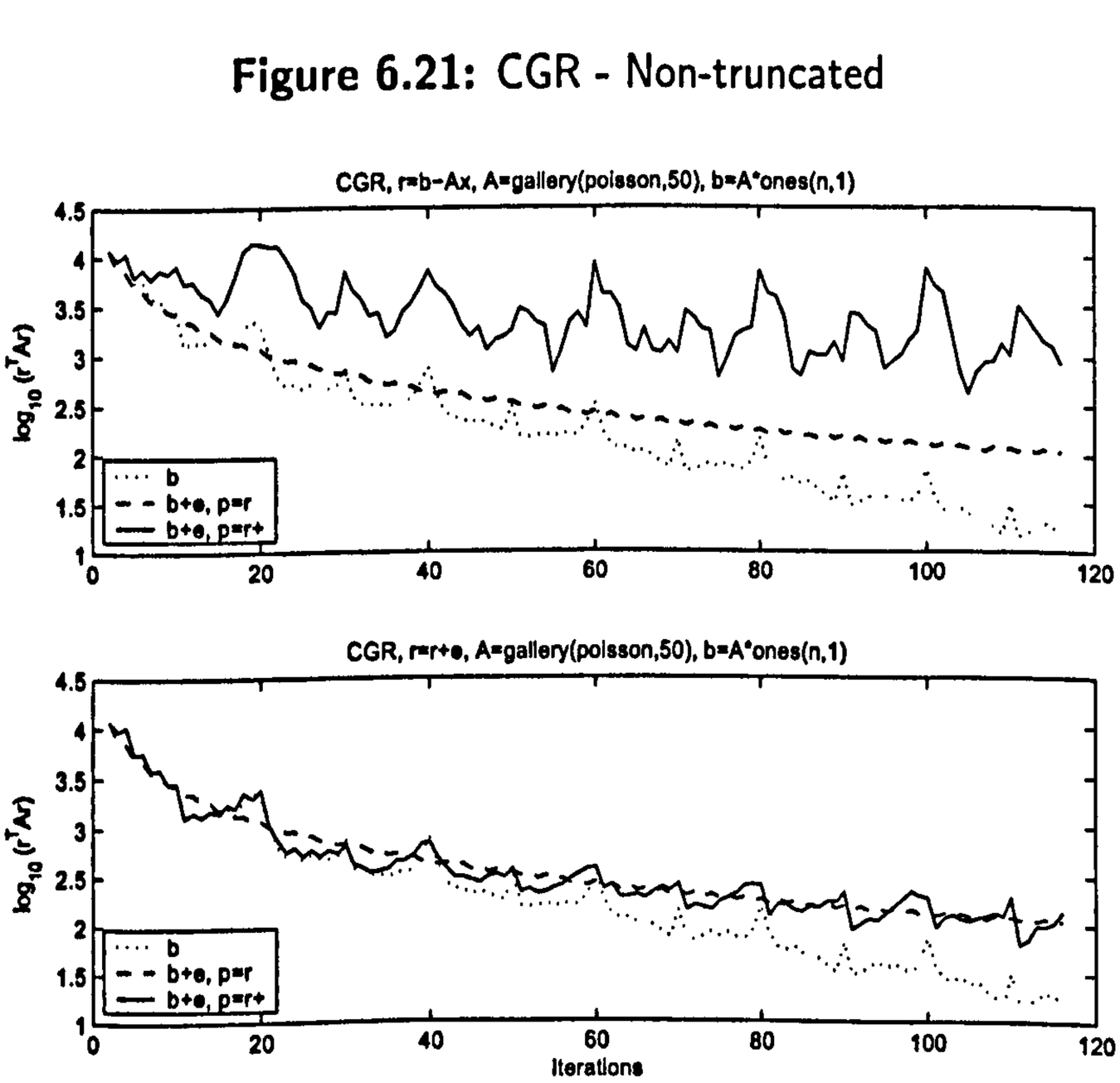


Figure 6.22: CGR - Truncated at each 10 iterations

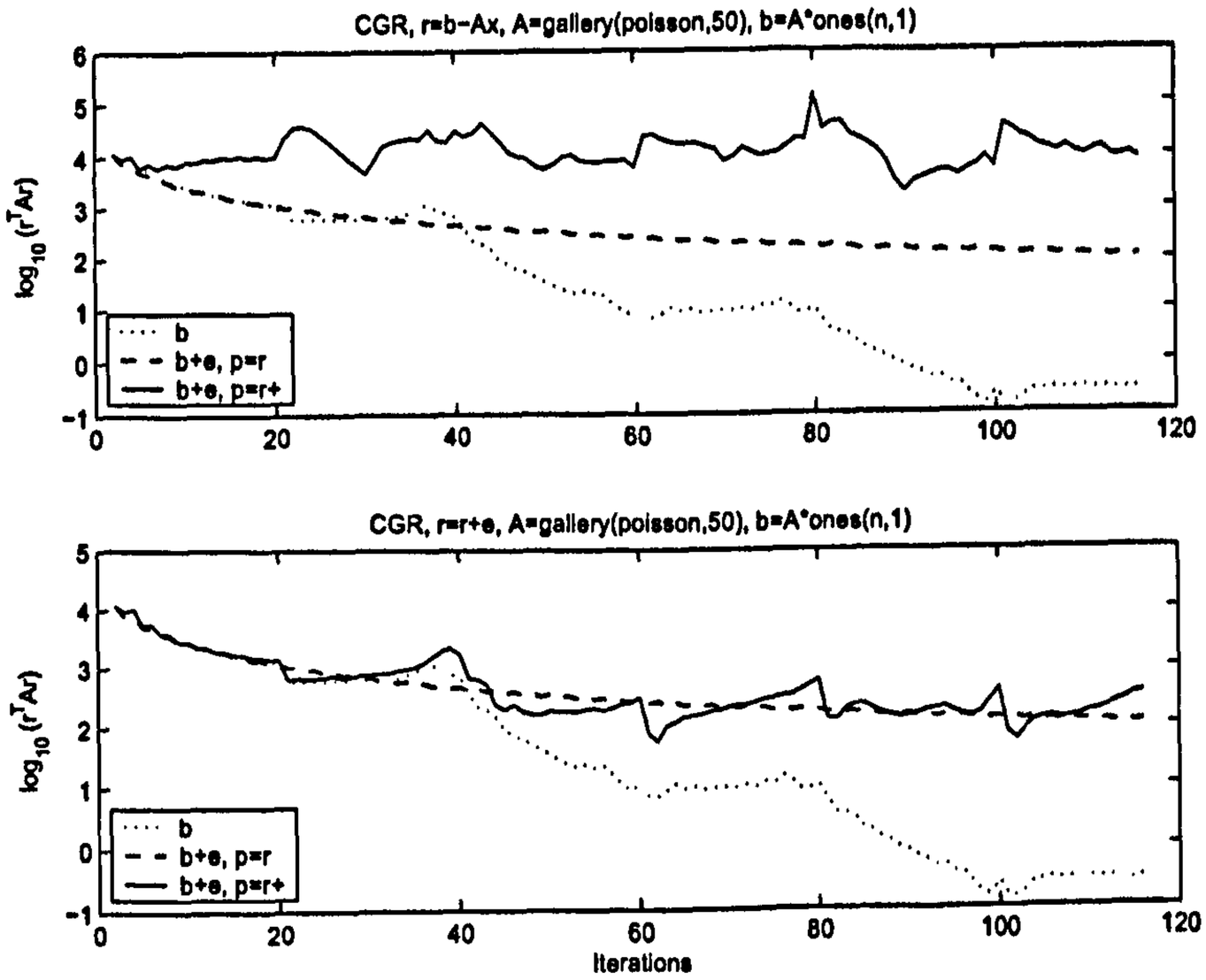


Figure 6.23: CGR - Truncated at each 20 iterations

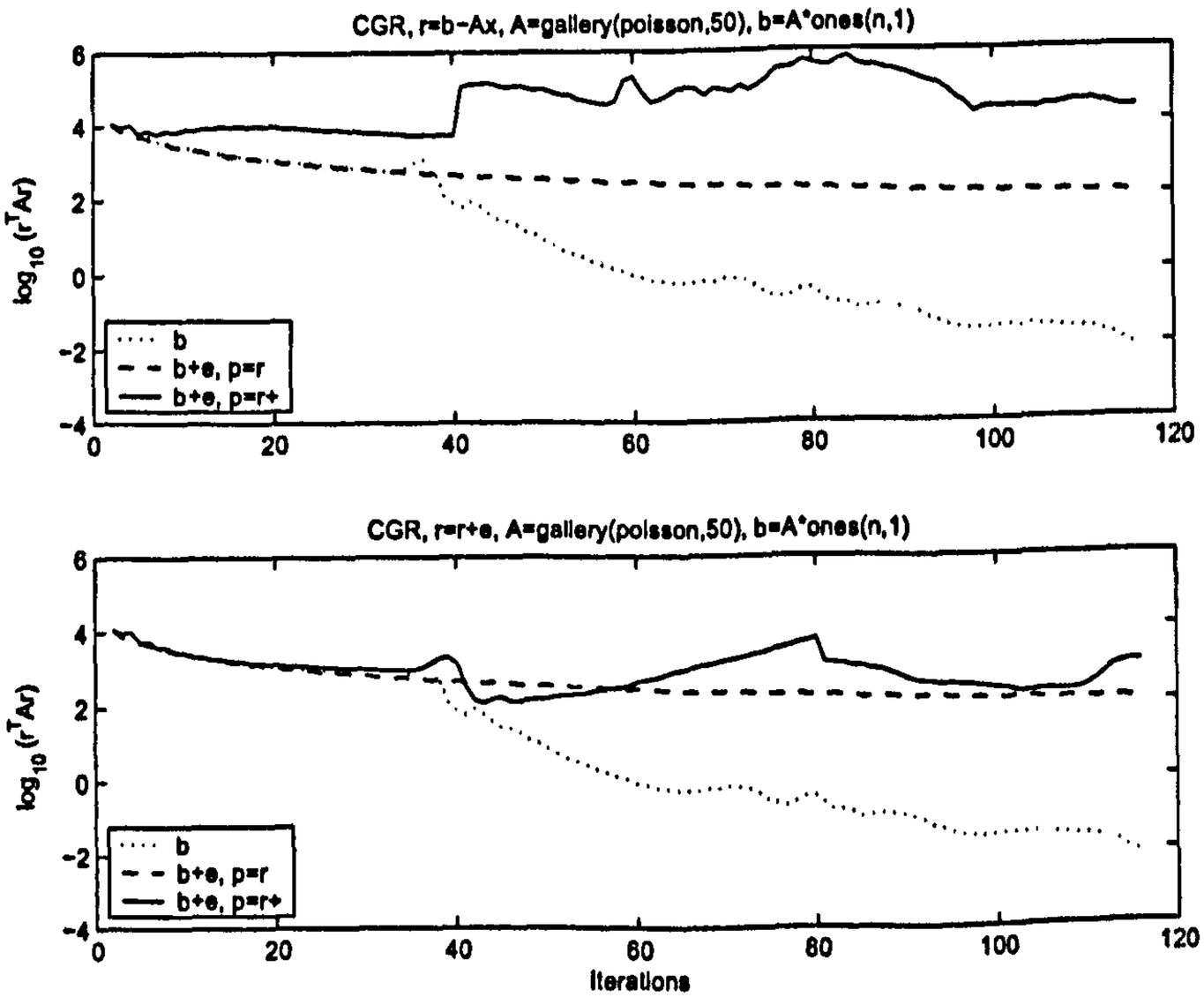


Figure 6.24: CGR - Truncated at each 40 iterations

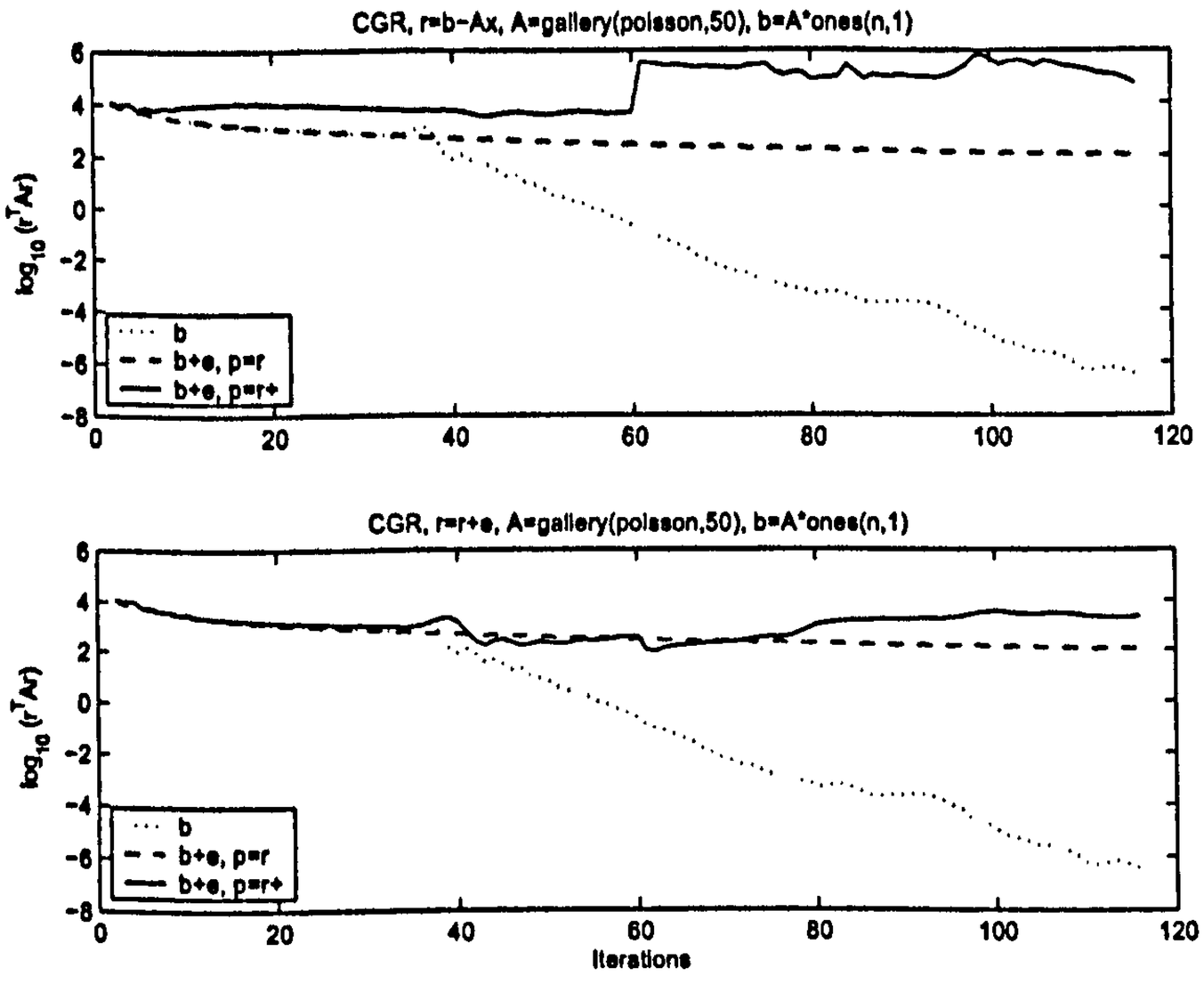


Figure 6.25: CGR - Truncated at each 60 iterations

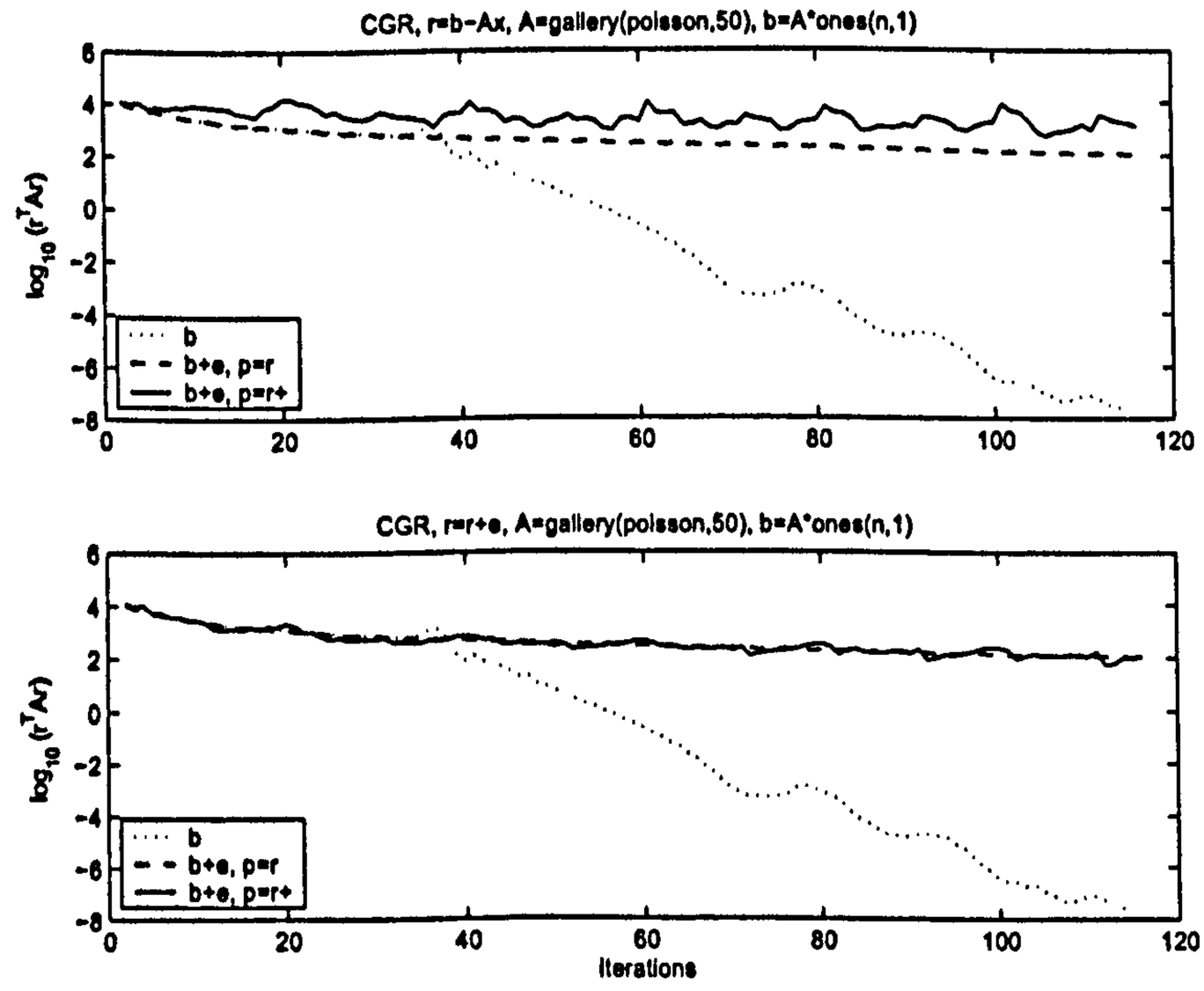


Figure 6.26: CGR - Truncated at each 10 iterations if $|(r_j + e, r_{j-1})| > 1e-4$

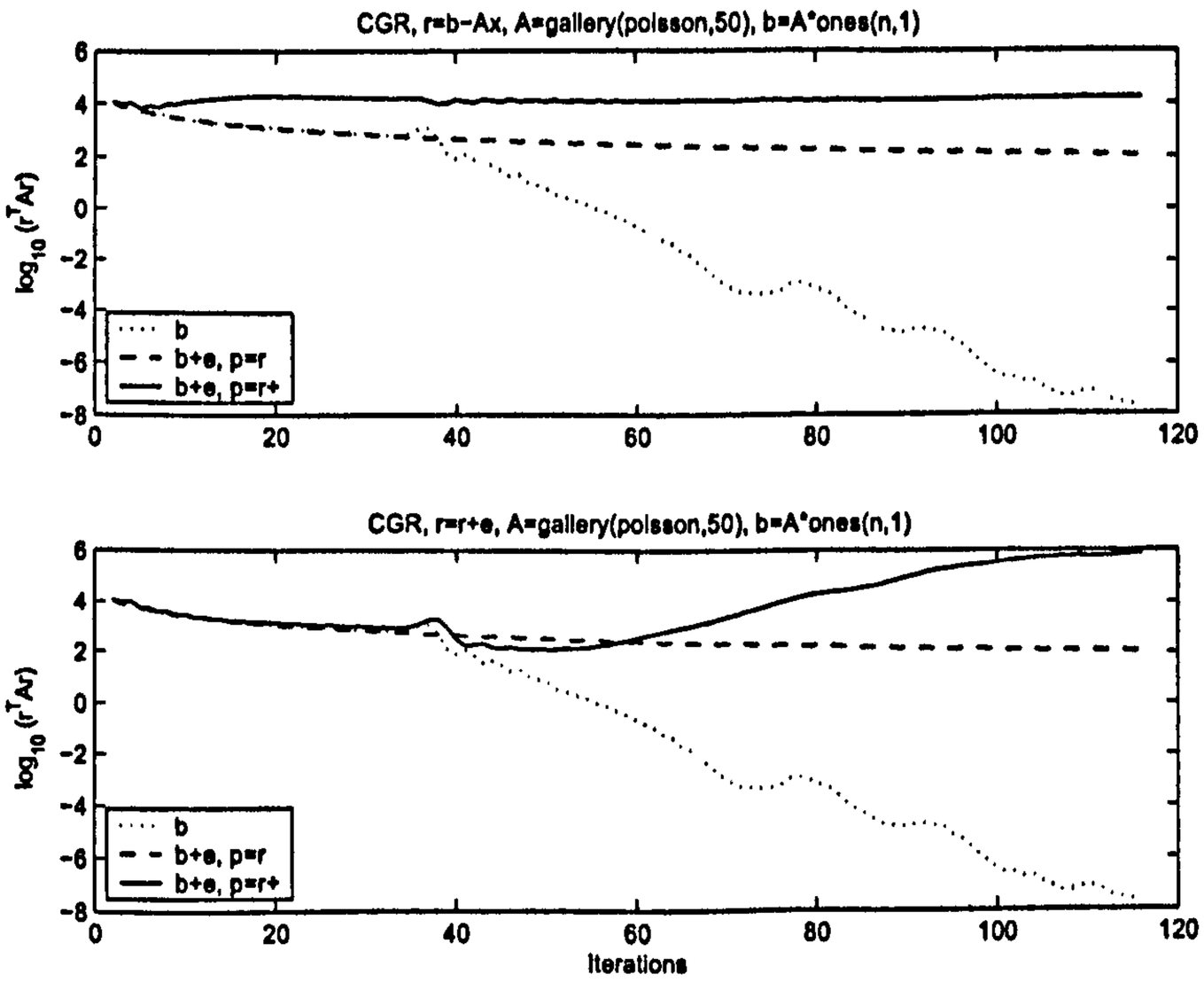


Figure 6.27: MCGR - Non-truncated

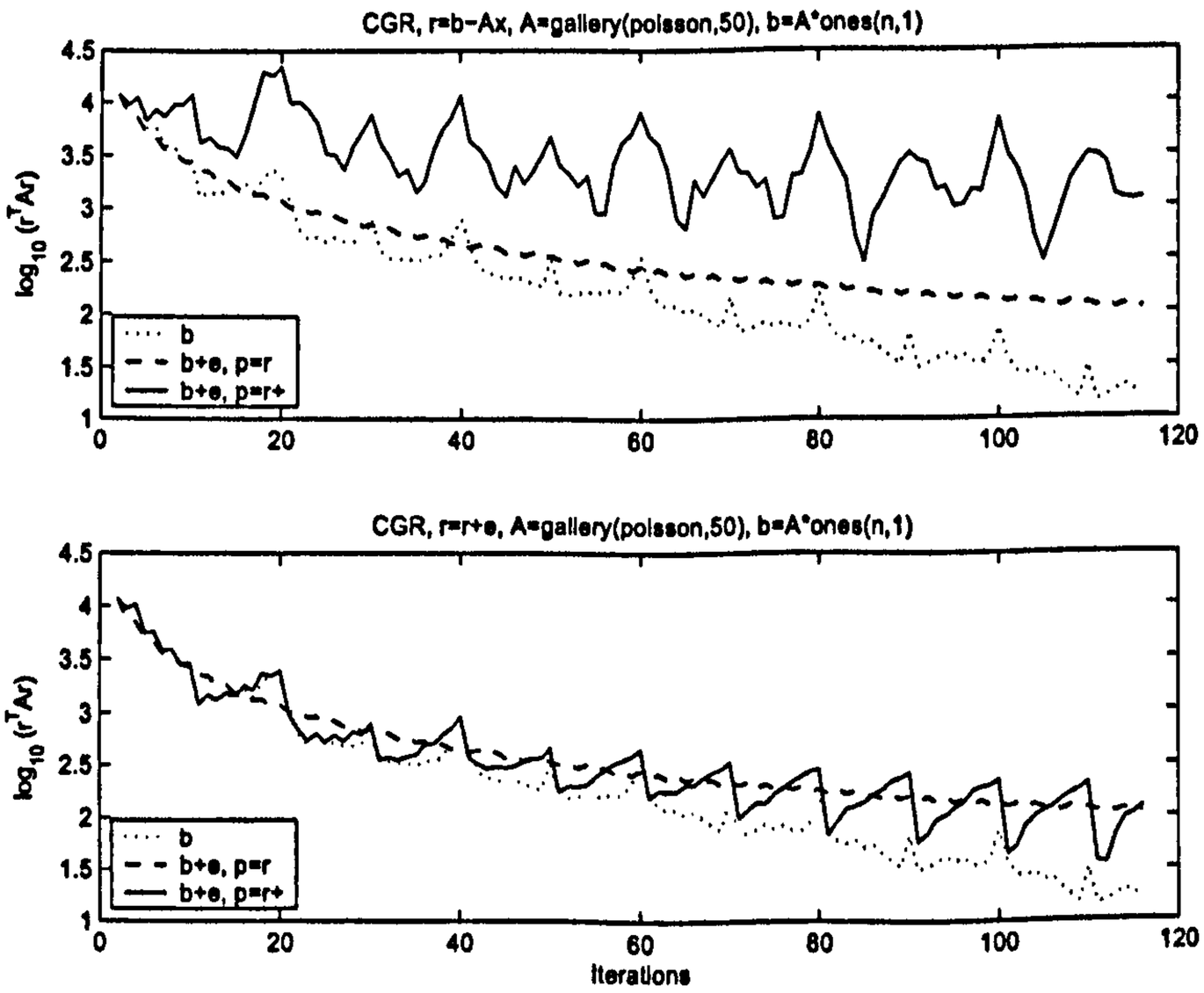


Figure 6.28: MCGR - Truncated at each 10 iterations

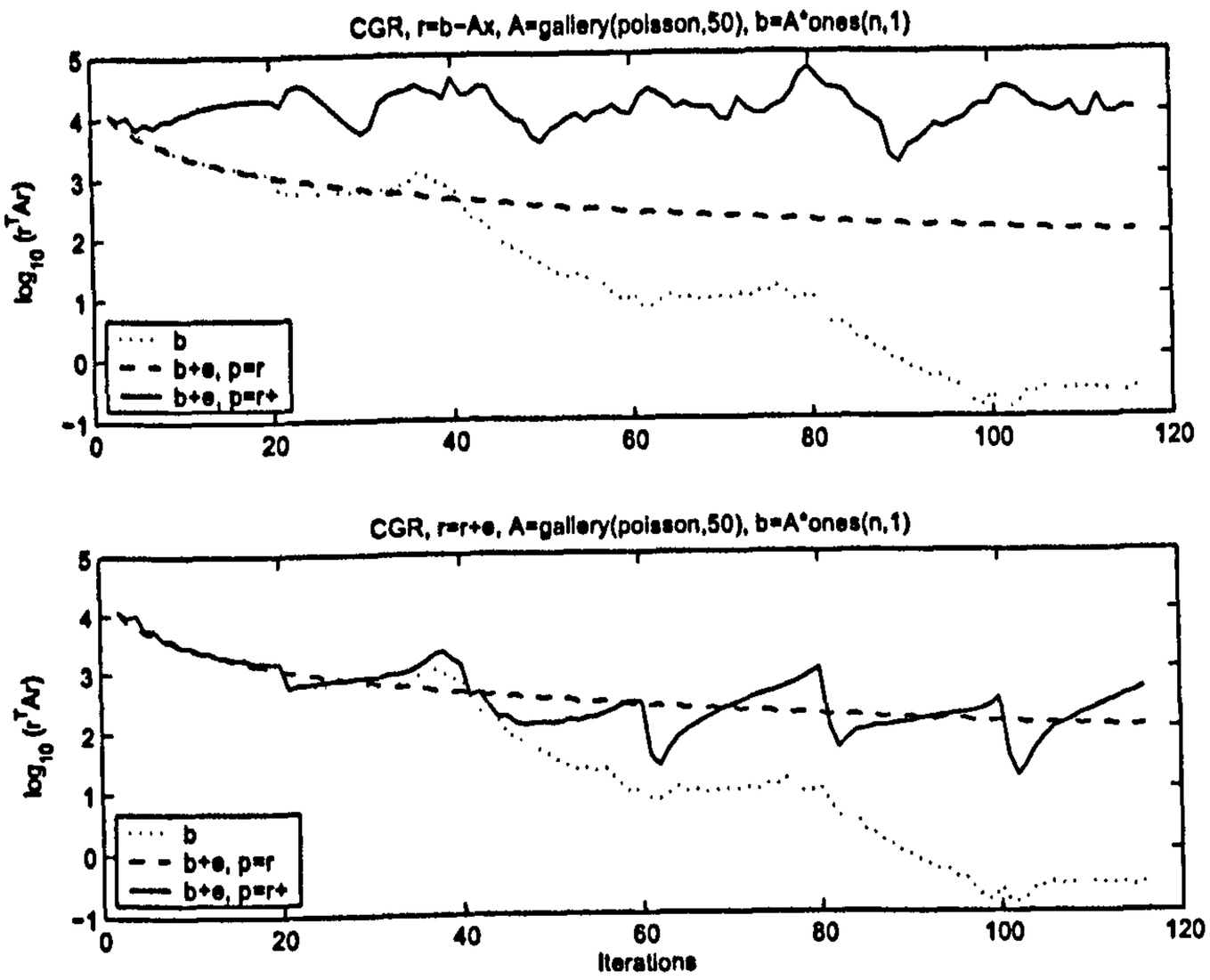


Figure 6.29: MCGR - Truncated at each 20 iterations

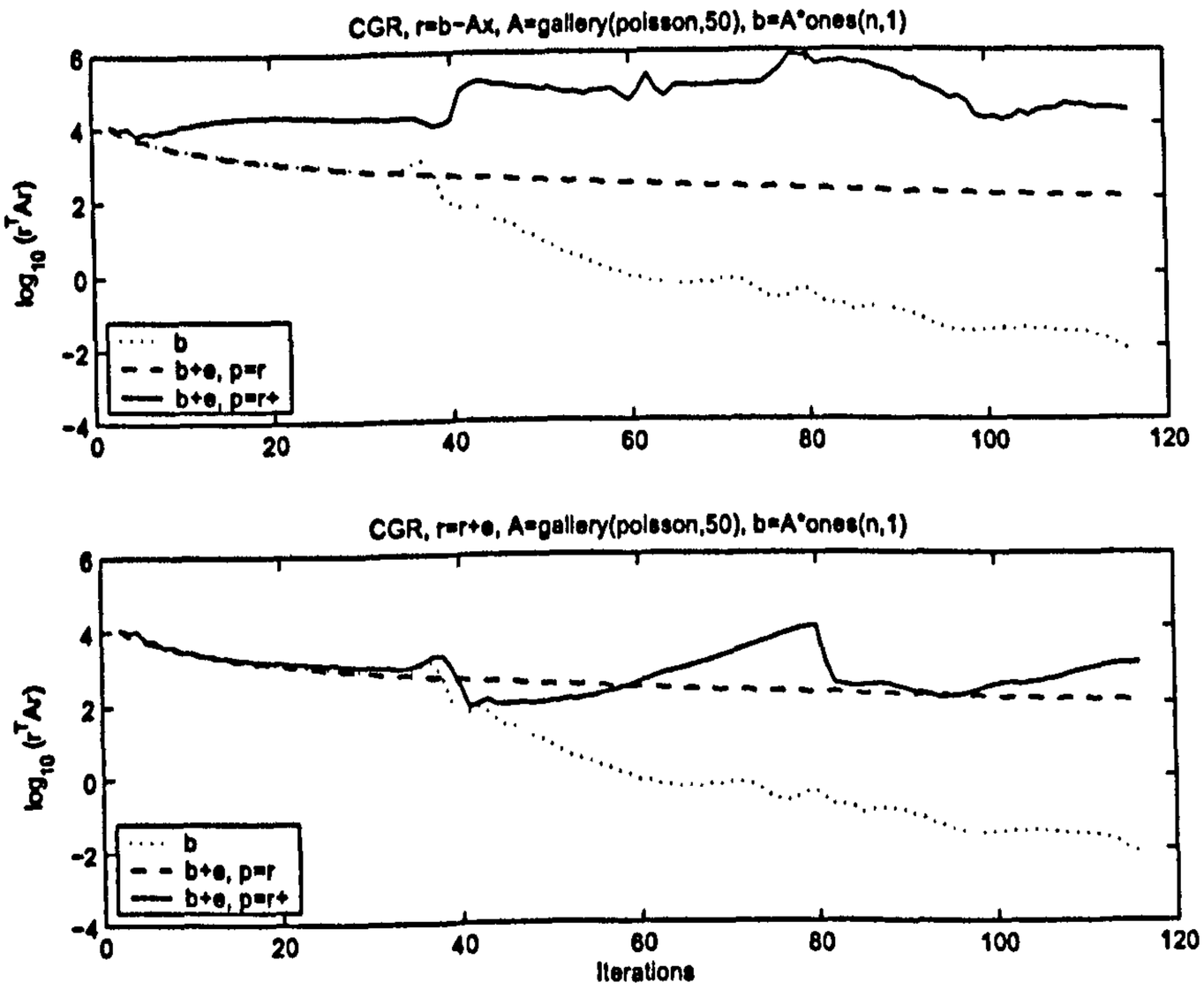


Figure 6.30: MCGR - Truncated at each 40 iterations

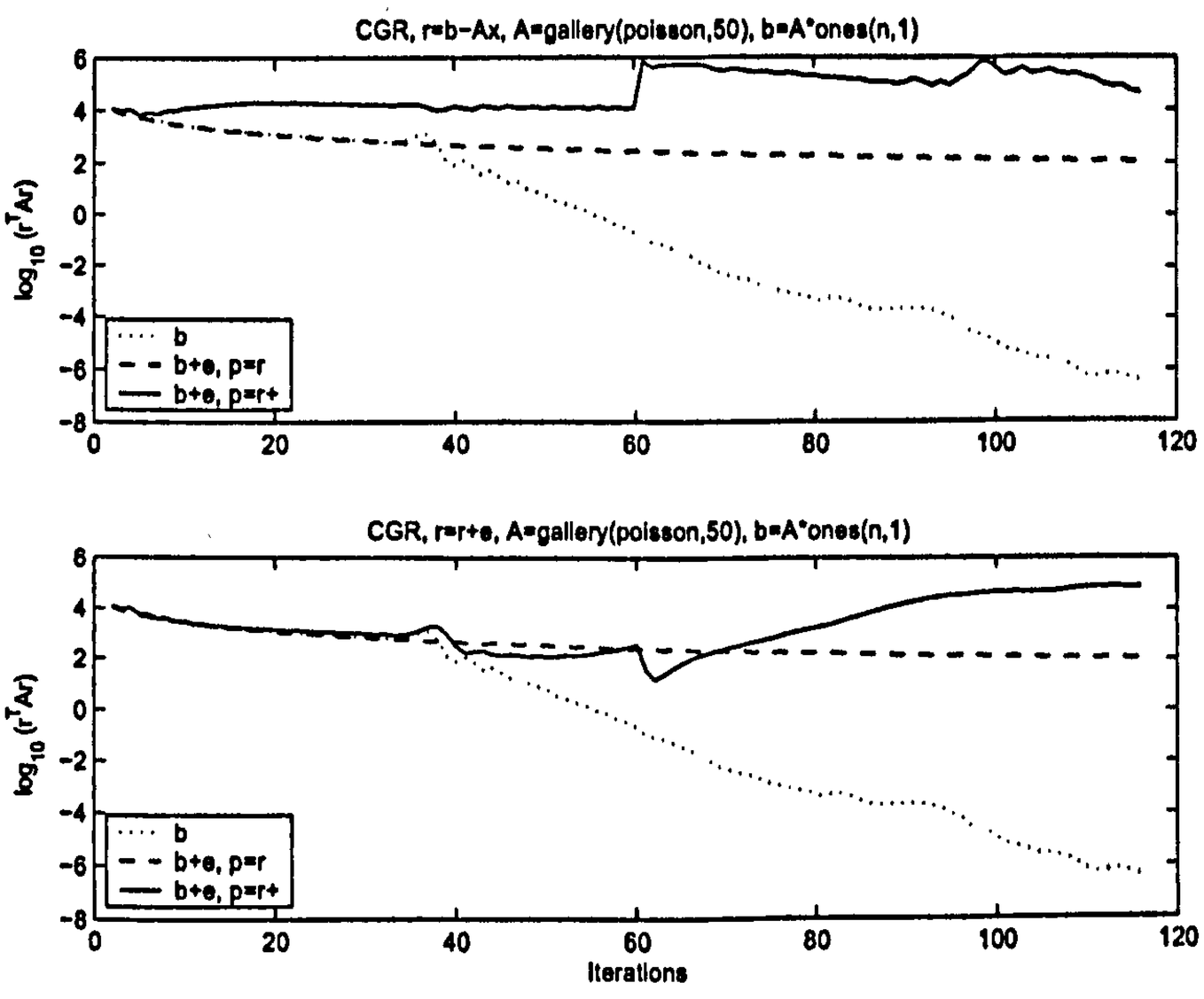


Figure 6.31: MCGR - Truncated at each 60 iterations

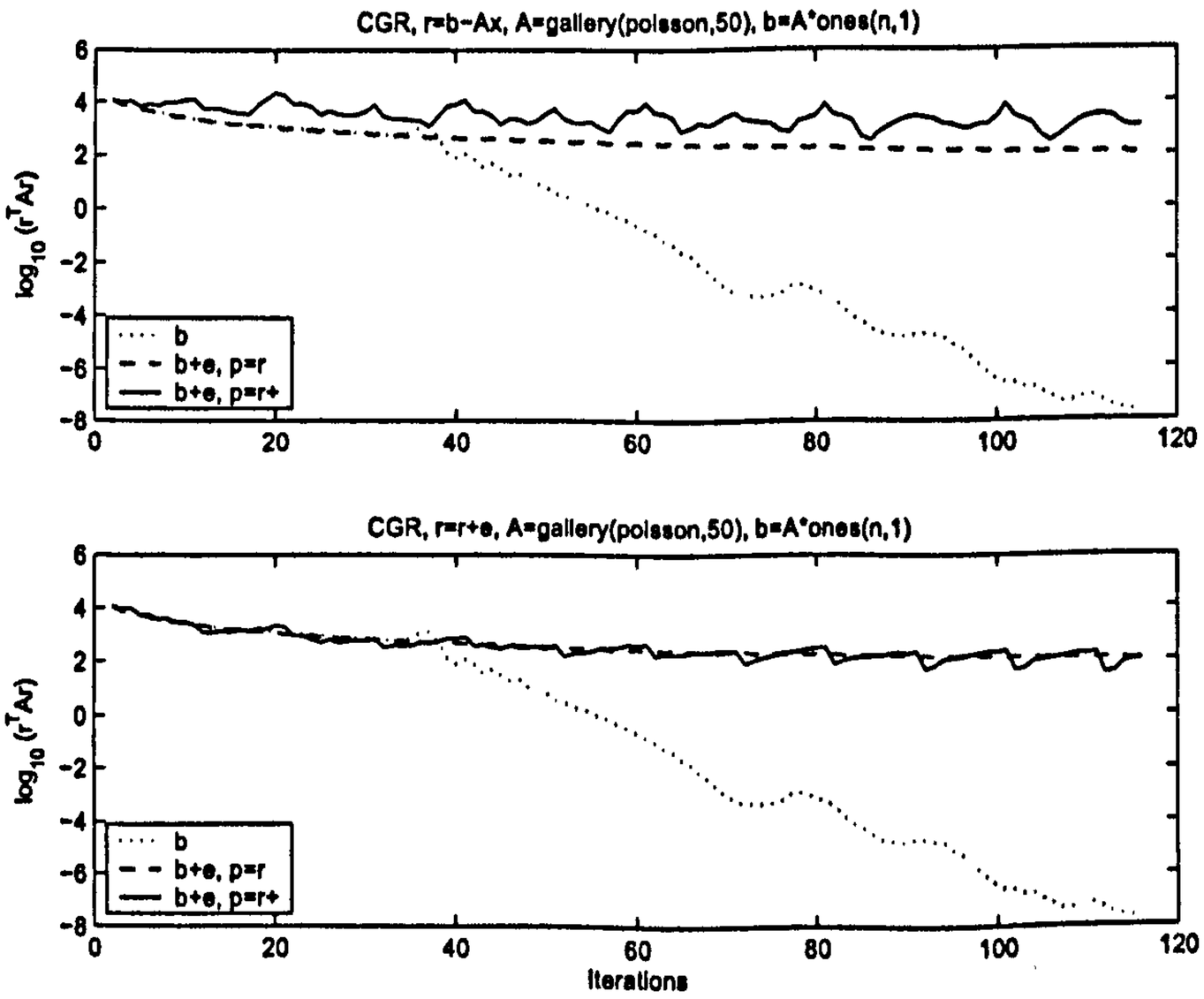


Figure 6.32: MCGR - Truncated at each 10 iterations if $|(r_j + e, r_{j-1})| > 1e-4$

6.3 Distributive Conjugate Gradient

The distributive conjugate gradient (DCG) method is a novel method to solve symmetric systems of linear equations on parallel computers. DCG is a main result of the research performed for this thesis. In the next sections, the derivation of DCG is presented, a preconditioned version of DCG (PDCG) is introduced and the parallelization of DCG is described, highlighting the procedure used to evaluate the stopping criteria.

6.3.1 Derivation and Algorithm

Consider the solution of Equation (6.4) in one subdomain. For simplicity, the subdomain subscripts were dropped and all the arrays that refer to the global system will be flagged with the symbol $\check{}$, for instance $Ax = b$ will be written as $\check{A}\check{x} = \check{b}$. Equation (6.4) is written in a general form as

$$Lx^{(k)} = b - \hat{H}\hat{x}^{(k-1)} \tag{6.13}$$

Every time $\hat{x}$ is updated a new problem with a perturbed right-hand-side has to be solved. Note that solving each subdomain problem accurately does not necessarily mean that the overall solution is accurate.

The algorithm proposed in this thesis updates $\hat{x}$ at each iteration which means that the right-hand side changes at each step. Solving (6.13) using one iteration of a conjugate gradient method per step is exactly the same as performing one steepest descent iteration, which degrades the convergence rate since CG has not fully been used. Therefore, a technique that uses information from previous iterations of CG needs to be adopted.

In the so-called distributive conjugate gradient (DCG) procedure, at each step k, the approximate solution x, residual r and search direction p are kept from the previous step. After updating $\hat{x}$ both r and p are updated according to the perturbation on the right-hand side: $\hat{H}(\hat{x}^{(k)} - \hat{x}^{(k+1)})$. The residual is updated such that

$$r^{(k+1)} = b - \hat{H}\hat{x}^{(k+1)} - Lx^{(k+1)} \tag{6.14}$$

Given that $p^{(k)}$ is the search direction computed before the perturbation, the updated search direction $\tilde{p}^{(k)}$ is chosen so that the conjugacy of p and r is reinstated. The conjugacy between the residual and the search direction vectors is perhaps the fundamental characteristic of CG and is exploited to avoid taking steps in the same direction as earlier steps. Therefore, the updated search direction $\tilde{p}$ is taken as an orthogonal vector of p projected onto r. The updated search direction $\tilde{p}^{(k)}$ is defined by

$$\tilde{p}^{(k)} = p^{(k)} - \frac{\left(r^{(k+1)}, p^{(k)}\right)}{\left(r^{(k+1)}, r^{(k+1)}\right)} r^{(k+1)} \tag{6.15}$$

which leads to Algorithm 6.3.1.

This approach makes $\tilde{p}^{(k)}$ conjugate to $p^{(k+1)}$ but does not guarantee any other orthogonality properties that CG has. However, it improves the convergence rate quite significantly if compared to the steepest descent method. For a given accuracy, the additive Schwarz procedure with DCG as a subdomain solver achieves an overall convergence rate comparable to the CG method, as reported in Chapter 7.

It is worth highlighting that the only step of Algorithm 6.3.1 that requires communication among processors to compute the solution is step eight, where each processor has to exchange (send and receive) the new values of $\hat{x}$ with the neighbour subdomains. This communication happens only among neighbour subdomains. Additionally, a global reduce operation has to be performed to evaluate the residual over the whole domain. However, the residual does not need to be computed at each iteration. A function that attempts to predict the number of

iterations needed to achieved a certain accuracy can be used reducing the number of global reduce operations to a fraction of the number of iterations. This function is explained in Section 6.3.4.

Algorithm 6.3.1 Distributive Conjugate Gradient

1. $r_0 = b - Lx_0$
2. $p_0 = r_0$
3. $\hat{x}_0 = 0$
4. for i=0,1,...
5. $\quad \alpha_i = (r_i, r_i)/(Lp_i, p_i)$
6. $\quad x_{i+1} = x_i + \alpha_i p_i$
7. $\quad r_{i+1} = r_i - \alpha_i Lp_i$
8. $\quad$ Update $\hat{x}_{i+1}$
9. $\quad r_{i+1} = r_{i+1} - \hat{H}(\hat{x}_{i+1} - \hat{x}_i)$
10. $\quad p_i = p_i - (r_{i+1}, p_i)/(r_{i+1}, r_{i+1})r_{i+1}$
11. $\quad \beta_i = -(r_{i+1}, Lp_i)/(p_i, Lp_i)$
12. $\quad p_{i+1} = r_{i+1} + \beta_i p_i$
13. end for

6.3.2 Preconditioned DCG

The preconditioned DCG (PCG) algorithm was obtained by initially considering the derivation of the preconditioned conjugate gradient method presented by Saad [101, pp 246-247]. The system of linear equations to be solved is given by

$$M^{-1}Ax = M^{-1}b \tag{6.16}$$

where M is a preconditioner matrix. This system is no longer symmetric in general. In order to preserve symmetry, observe that MA is self-adjoint for the

M-inner product,

$$(x, y)_M \equiv (Mx, y) = (x, My)$$

since

$$(MAx, y)_M = (Ax, y) = (x, Ay) = (x, M^{-1}(MA)y) = (x, MAy)_M$$

where $(\cdot, \cdot)$ is the Euclidean inner product and $(\cdot, \cdot)_M$ is the M-inner product. Therefore, an alternative to preserve symmetry is to replace the Euclidean inner product by the M-inner product in the conjugate gradient algorithm (Algorithm 4.4.1, page 68). If the CG algorithm is rewritten for the M-inner product, denoting by $r_i = b - Ax_i$ the original residual and by $z_i = Mr_i$ the residual for the preconditioned system, the following sequence of operations is obtained, ignoring the initial step:

$$\begin{aligned}
\alpha_i &= (z_i, z_i)_M / (MAp_i, p_i)_M \\
x_{i+1} &= x_i + \alpha_i p_i \\
r_{i+1} &= r_i - \alpha_i Ap_i \\
z_i &= Mr_{i+1} \\
\beta_i &= (z_{i+1}, z_{i+1})_M / (z_i, z_i)_M \\
p_{i+1} &= z_{i+1} + \beta_i p_i
\end{aligned}$$

Since $(z_i, z_i)_M = (r_i, z_i)$ and $(MAp_i, p_i)_M = (Ap_i, p_i)$, the M-inner products do not need to be computed explicitly. With these observations, Algorithm 6.3.2 is obtained.

This same approach can be used to derive a preconditioned version of DCG. In addition to the modifications introduced in the conjugate gradient algorithm, the formula used to update the search direction, Equation (6.15), has to be rewritten. The vector p has to be projected onto z instead of r and the inner products must

Algorithm 6.3.2 Preconditioned Conjugate Gradient (PCG)

1. $r_0 = b - Ax_0$
2. $z_0 = M^{(-1)}r_0$
3. $p_0 = z_0$
4. for i=0,1,...
5. $\quad \alpha_i = (r_i, z_i)/(Ap_i, p_i)$
6. $\quad x_{i+1} = x_i + \alpha_i p_i$
7. $\quad r_{i+1} = r_i - \alpha_i Ap_i$
8. $\quad z_{i+1} = M^{(-1)}r_{i+1}$
9. $\quad \beta_i = (r_{i+1}, z_{i+1})/(r_i, z_i)$
10. $\quad p_{i+1} = z_{i+1} + \beta_i p_i$
11. end for

be appropriately replaced. Given that

$$(z, p)_M \equiv (Mr, p)_M = (r, p)$$

and

$$(r, r)_M \equiv (r, Mr)_M = (r, z),$$

Equation (6.15) for the preconditioned DCG becomes

$$\tilde{p}^{(k)} = p^{(k)} - \frac{\left(r^{(k+1)}, p^{(k)}\right)}{\left(r^{(k+1)}, z^{(k+1)}\right)} z^{(k+1)}$$

Using this equation and the derivation of PCG, the preconditioned distributive conjugate gradient method can be described as Algorithm 6.3.3. Notice that the preconditioning step is serial and therefore preconditioning methods that are not well-suited for parallelism can be used without loss of parallel efficiency. Numerical results are presented in Section 7.4.4.

Algorithm 6.3.3 Preconditioned Distributive Conjugate Gradient

1. $r_0 = b - Lx_0$
2. $z_0 = Mr_0$
3. $p_0 = z_0$
4. $\hat{x}_0 = 0$
5. **for** i=0,1,...
6. $\quad \alpha_i = (r_i, z_i)/(Lp_i, p_i)$
7. $\quad x_{i+1} = x_i + \alpha_i p_i$
8. $\quad r_{i+1} = r_i - \alpha_i Lp_i$
9. $\quad$ Update $\hat{x}_{i+1}$
10. $\quad r_{i+1} = r_{i+1} - \hat{H}(\hat{x}_{i+1} - \hat{x}_i)$
11. $\quad z_{i+1} = Mr_{i+1}$
12. $\quad p_i = p_i - (r_{i+1}, p_i)/(r_{i+1}, z_{i+1})z_{i+1}$
13. $\quad \beta_i = -(z_{i+1}, Lp_i)/(p_i, Lp_i)$
14. $\quad p_{i+1} = z_{i+1} + \beta_i p_i$
15. **end for**

6.3.3 Parallelization

The parallelization of Algorithm 6.3.1 becomes quite straightforward due to the characteristics of the algorithm. DCG has been designed to be highly parallelizable and hence restrain the parallel operations to a minimal. This has been achieved by restricting the communication among processors to exchanging data of the interface nodes once per step.

In actuality, only step eight of Algorithm 6.3.1 has to be adequately parallelized since all the other operations are performed locally. The parallelization of step eight consists of the communication schemes described in Section 5.6.4. The 2-norm of the residual vector has also to be computed in parallel in order to evaluate the accuracy of the approximate solution. This operation is however directly supported by MPI. Note that the norm is not calculated at every step as explained in Section 6.3.4 and that this same approach can be used on parallelizing Algorithm 6.3.3.

An extended version of Algorithm 6.3.1, in pseudo-code, follows. This extended version describes with some detail the implementation used to produce the results reported in Chapter 7. The main difference from Algorithm 6.3.1 is that $\hat{x}$ may not be updated at every iteration. Table 6.5 contains the description of the symbols used on the pseudocode.

DCG Pseudocode

Initialization

1. $r = b - Lx_0$
2. $p = r$
3. $w = Lp$
4. $u_A = 0$

5. $\mu = 0$
6. $\eta = r^T r$

Calculate global residual and stopping tolerance

7. $\gamma_0 = \|\check{r}\|_2$
8. $\bar{\varepsilon} = \varepsilon \|\check{b}\|_2$

Set initial convergence rate and calculate iteration to check convergence

9. $\bar{\gamma} = 0.5$
10. $\kappa = \min(\lfloor \log_2 (\bar{\varepsilon}/\gamma_0) / \log_2 (\bar{\gamma}) \rfloor, t)$

Loop

11. **for** $i = 1, 2, \ldots$
12. $\alpha = \eta/(r^T w)$
13. $x = x + \alpha p$
14. $r = r - \alpha w$

Update solution on interface nodes and residual

15. **if** mod $(i, h) = 0$ **then**
16. Update $\hat{x}$
17. $\mu = |\mu - 1|$
18. **if** $\mu = 0$ **then**
19. $u_A = \hat{H}\hat{x}_i$
20. $r = r - (u_A - u_B)$
21. **else**
22. $u_B = \hat{H}\hat{x}_i$
23. $r = r - (u_B - u_A)$
24. **endif**
25. **endif**

Calculate residual norm

26. $\eta_* = \eta$
27. $\eta_C = r_C^T r_C$

28. $\eta_G = r_G^T r_G$

29. $\eta = \eta_C + \eta_G$

Check stopping criterion

30. **if** $i = \kappa$ **then**

31. $\gamma = \|\check{r}\|_2$

32. **if** $(\gamma < \bar{\varepsilon})$ or $(i = t)$ **then** stop

33. $\bar{\gamma}_* = \bar{\gamma}$

34. $\bar{\gamma} = (\gamma/\gamma_0)^{1/i}$

35. $\hat{\gamma} = \bar{\gamma}_*/\bar{\gamma}$

36. **if** $|\hat{\gamma} - 1| < 0.02$ **then** $\hat{\gamma} = 1.0$

37. $\kappa = \min(\max(\lfloor \hat{\gamma} \log_2(\bar{\varepsilon}/\gamma_0)/\log_2(\bar{\gamma}) \rfloor, i+1), t)$

38. **endif**

Set new search direction vector

39. **if** mod $(i, h) = 0$ **then**

40. $\xi = (r^T p)/\eta$

41. $p = p - \xi r$

42. $v = Lr$

43. $w = w - \xi v$

44. $\beta = -(r^T w)/(p^T w)$

45. $p = r + \beta p$

46. $w = v + \beta w$

47. **else**

48. $\beta = \eta/\eta_*$

49. $p = r + \beta p$

50. $w = Lp$

51. **endif**

52. **endfor**

Symbol	Size	Description
n_L	1	Number of local nodes ($n_L = n_C + n_G$)
n_H	1	Number of halo nodes
n_C	1	Number of core nodes
n_G	1	Number of ghost nodes
nz_L	1	Number of non-zeros in L
nz_H	1	Number of non-zeros in H
np	1	Number of processors or subdomains
t	1	Maximum number of iterations
h	1	Updates $\hat{x}$ each h iterations
$\bar{\varepsilon}$	1	Stopping tolerance
γ	1	2-norm of the residual of the whole system
$\bar{\kappa}$	1	Estimation of the rate of convergence
κ	1	Estimated iteration number to check stop criterion
r	n_L	Residual vector of local nodes
$\bar{r}$	n_C	Core nodes of r
$\hat{r}$	n_G	Ghost nodes of r
p	n_L	Search direction vector
x	n_L	Approximated solution of local nodes
$\hat{x}$	n_H	Approximated solution of halo nodes
$\check{r}$	n	Residual vector of the global system ($\check{r} = \check{b} - A\check{x}$)
$\hat{H}$	$n_H \times n_H$	Matrix of halo edges (Equation 6.1)
$\hat{L}$	$n_L \times n_L$	Matrix of local edges

Table 6.5: Distributive Conjugate Gradient pseudocode symbols description

6.3.4 Stopping Criteria

The norm of the residual is usually used as the stopping criterion for iterative methods. This norm has to be calculated in a global manner implying a global reduce operation at each time it is calculated. These global reduce operations might compromise the parallel performance of the algorithm.

In order to reduce the number of global reduce operations in DCG a procedure is used to anticipate the number of iterations needed to achieve a given accuracy and the norm is only computed at these iterations. The probable iteration to stop the computation is updated each time the anticipated iteration is reached and the stopping criterion is still not satisfied. This prediction is made based on the worst scenario, i.e. it prefigures that the convergence rate might degrade.

Given that $\gamma_i = \|r_i\|_2$ and that $\tilde{\gamma}_i = \gamma_i/\gamma_{i-1}$ (rate of convergence of iteration i), the average rate of convergence at iteration k is

$$\bar{\gamma}_k = \left(\sum_{j=1}^{k} \tilde{\gamma}_j\right)/k$$

The residual norm at iteration k can be written as

$$\gamma_k = \gamma_0 \prod_{j=1}^{k} \tilde{\gamma}_j$$

and therefore the average rate of convergence at iteration k can be written as

$$\bar{\gamma}_k = \left(\frac{\gamma_k}{\gamma_0}\right)^{1/k}$$

The probable number of iterations needed to achieve a certain accuracy, namely t, can be anticipated by using the average rate of convergence:

$$\bar{\gamma}_k = \left(\frac{\gamma_t}{\gamma_0}\right)^{1/t} \implies \bar{\gamma}_k{}^t = \frac{\gamma_t}{\gamma_0} \implies t = \frac{\ln(\gamma_t/\gamma_0)}{\ln(\bar{\gamma}_k)}$$

The stopping criterion is satisfied if $\gamma_i \leq \bar{\varepsilon}$ and hence $\gamma_t \geq \bar{\varepsilon}$ at iteration t. Therefore

$$t \geq \frac{\ln(\bar{\varepsilon}/\gamma_0)}{\ln(\bar{\gamma}_k)}$$

If the convergence rate is approximately stable throughout the entire solution, i.e. $\tilde{\gamma}_1 \simeq \tilde{\gamma}_2 \simeq \cdots \simeq \tilde{\gamma}_k$, the prediction is fairly accurate. However, the rate of convergence can change significantly especially in the first few iterations. Therefore, a correction factor $\hat{\gamma}$ was introduced to DCG such that the changes on the rate are considered on calculating t.

Table 6.6 lists the predicted number of iterations needed and the total number of iterations actually needed for three test problems. The number of global reduce operations is considerably reduced. Notice that if the initial rate ($\bar{\gamma}$) is set too high the actual number of iterations performed can be greater than the number of iterations actually needed for a given accuracy. Therefore, it is advisable to start using a low rate and let the procedure correct it as the calculation advances. It is also important to notice that the advantage of using this procedure is chiefly in terms of parallel performance since the inner product of the residual vectors are calculated at every iteration independently of computing the global norm.

6.4 Conclusion

In this chapter a study that led to the distributed conjugate gradient method as well as the method itself were presented. It has been shown that the parallelization of DCG became straightforward due to the design of the algorithm and that the parallel performance depends mainly on the routine used to exchange data among neighbour subdomains and on the number of global reduce operations executed. The procedure to select the iterations where the stopping criteria might have been

Initial $\bar{\gamma}$	0.2	0.5	0.8	0.2	0.5	0.8	0.5*	0.5*
Iterations	14	33	103	14	33	103	33	47
with global	15	53		26	79	165	255	426
reduce	43	59		126	154	215	784	1295
	52	63		188	205	237	1051	1722
	63	65		228	234	243	1149	1857
	65	66		240	242	245	1175	
	66			244	245	246	1185	
				246	246	247	1190	
				247	247	248	1191	
				248	248	249		
				249	249			
Total	7	6	1	11	11	10	9	5
Iterations	66	66	103	249	249	249	1191	1857

* Different stopping criteria

Table 6.6: Number of iterations predicted to satisfied the stopping criterion.

satisfied reduces considerably the number of global reduce operations needed by the method. The numerical performance of DCG is reported in Chapter 7.

Chapter 7

Numerical Results

In this chapter the more relevant results obtained with a Fortran implementation of DCG are reported.

7.1 Test Problems

The test problems are stated as:

$$\begin{aligned} \text{Problem 2D:} \quad \nabla^2\phi &= -2\pi^2 \sin(\pi x)\sin(\pi y) \text{ in } \Omega \\ \phi &= 0 \text{ on } \partial\Omega \\ \Omega &= \{\, (x,y) \in \mathbf{R^2} : -1 < x,y < 1 \,\} \end{aligned}$$

$$\begin{aligned} \text{Problem 3Da:} \quad \nabla^2\phi &= 12c(x^2+y^2+z^2)^{2c-1} + 4c(2c-1)(x^2+y^2+z^2)^{2c} \text{ in } \Omega \\ \phi &= (x^2+y^2+z^2)^{2c} \text{ on } \partial\Omega \\ \Omega &= \{\, (x,y,z) \in \mathbf{R^3} : 0 < x,z < 20\ ,\ 0 < y < 120 \,\} \end{aligned}$$

$$\begin{aligned} \text{Problem 3Db:} \quad \nabla^2\phi &= 3\pi^2 cos(\pi x)cos(\pi z)cos(\pi z) \text{ in } \Omega \\ \phi &= cos(\pi x)cos(\pi z)cos(\pi z) \text{ on } \partial\Omega \\ \Omega &= \{\, (x,y,z) \in \mathbf{R^3} : 0 < x,y,z < 100 \,\} \end{aligned}$$

where c is a scalar and the solution of problem 2D is sketched in Figure 7.1. The numerical experiments have been carried out on a SUNFIRE 15K parallel com-

puter (host name robinson) with 72 × 1200MHz CPUs and 288GB of shared memory (4GB/CPU) and on a Sun Fire 15K parallel computer (host name franklin) with 9 × 100 × 1200MHz CPUs and 288GB of shared memory. The meshes used are listed in Table 7.1. See Section 5.5.1 for an overview of the Sun Fire 15K computer.

7.2 Matrices Sparsity

This section presents a simple model that relates the matrix sparsity (number of non-zeros) to the number of equations (n) and the number of subdomains or processors (p). The matrices A, L and H cited are defined in Section 6.1. The equations were obtained using the data fitting function `polyfit` of MatLab and the data of one two-dimensional square mesh and one three-dimensional cubic mesh.

Figures 7.2 and 7.3 show the ratio nz to n of matrix A. The ratio is roughly linear and hence can be described by a linear equation. The number of non-zeros for the three- and two-dimensional meshes are given by $nz = 14.4n$ and $nz = 6.5n$, respectively.

The number of non-zeros of matrices L and H for a given number of processors (p) can be represented by power functions as shown in Figures 7.4 and 7.5. Another important relation is the ratio nz_L to nz_H (number of non-zeros of L to number of non-zeros of H) shown in Figure 7.6. The number of non-zeros of L decreases faster than the number of non-zeros of H, i.e. the amount of computation to be done locally by DCG decreases faster than the amount of data to be transferred among processors. In other words, the ratio computation to communication decreases as the number of processors increases for a given size of problem.

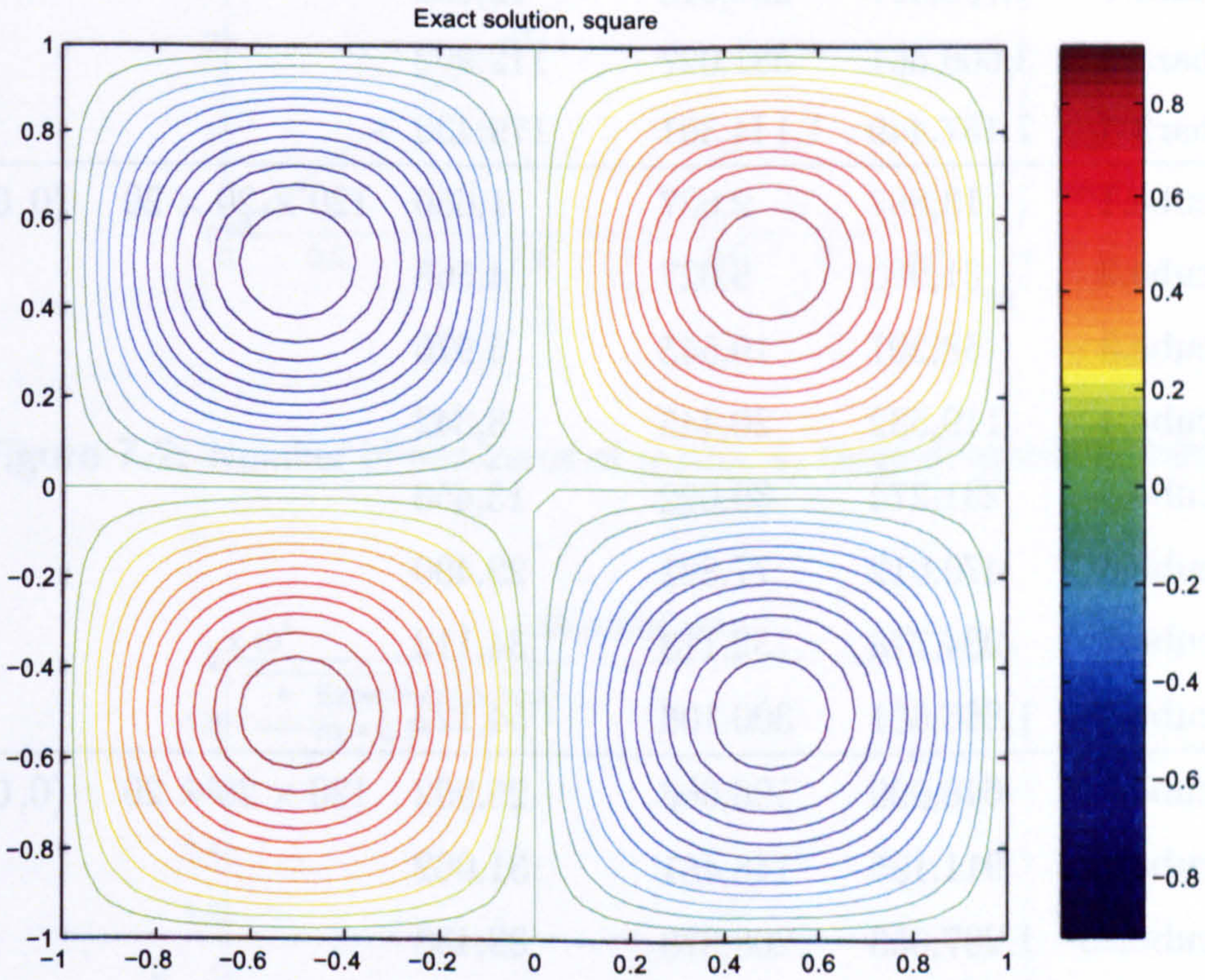


Figure 7.1: Problem 2D - Exact solution: $\phi = \sin(\pi x)\sin(\pi y)$

ID	Name	Elements	Nodes	Boundaries	Size (mm)	Origin
1	square1	3,598	1880	160	20×20	$(-1,-1)$
2	square4	208	125	40		
30	bar3.0	897,845	149,798	45,684	$120 \times 20 \times 20$	$(0,0,0)$
31	bar3.1	1,795,497	294,518	72,094		
32	bar3.2	3,606,561	581,027	112,902		
33	bar3.3	7,157,449	1,144,361	179,436		
41	cube.1	13,897	2,637	1,650	$120 \times 20 \times 20$	$(0,0,0)$
42	cube.2	31,980	6,027	4,568		
43	cube.3	58,201	10,545	5,976		
44	cube.4	119,532	20,445	8,342		
45	cube.5	231,274	39,022	13,458		
46	cube.6	470,612	77,821	22,490		
47	cube.7	934,778	152,118	34,112		
48	cube.8	1,866,651	300,106	54,122		
51	cube2.1	648,090	106,666	26,902	$120 \times 20 \times 20$	$(0,0,0)$
52	cube2.2	911,195	148,481	31,002		
53	cube2.3	1,287,653	208,373	39,438		
54	cube2.4	1,835,868	295,489	52,660		
55	cube2.5	2,592,411	415,270	65,326		
56	cube2.6	3,671,199	586,949	86,822		
57	cube2.7	5,187,042	825,563	108,020		
58	cube2.8	7,333,751	1,160,972	131,762		

Table 7.1: Meshes description

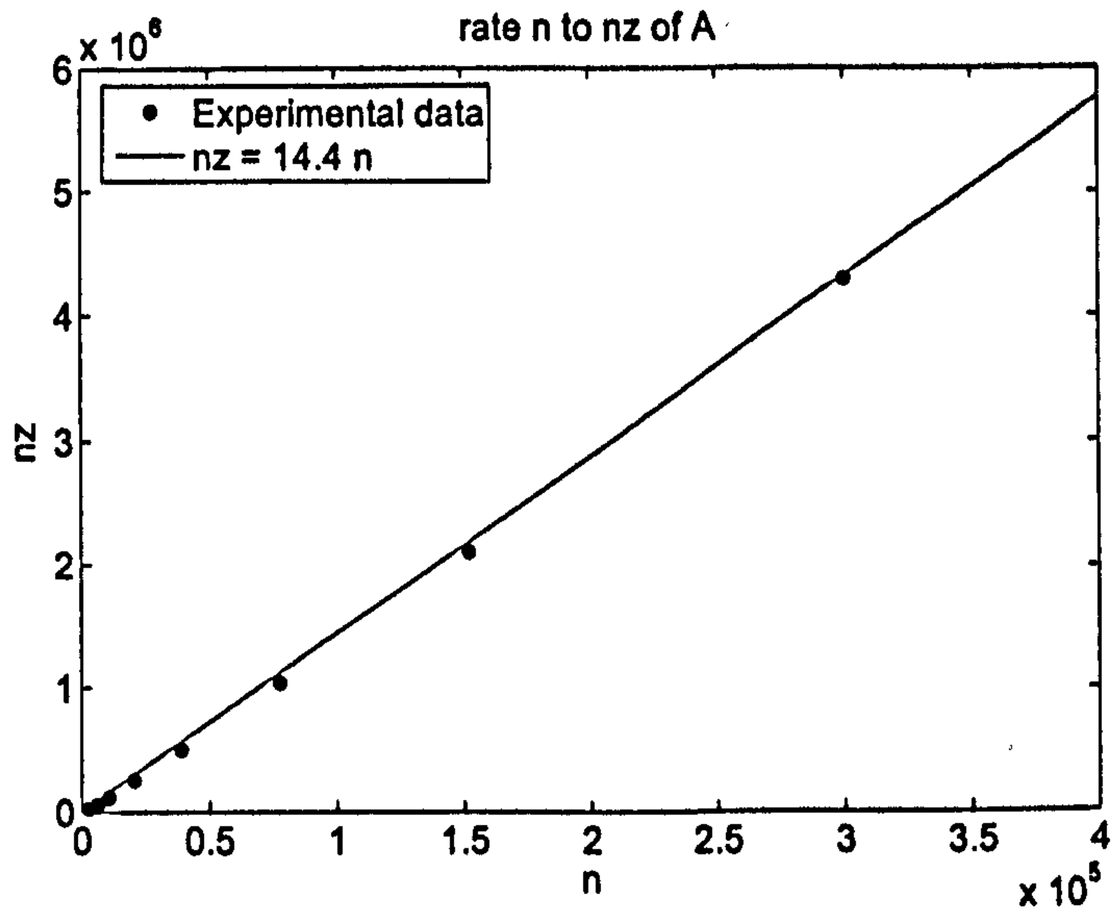


Figure 7.2: Number of non-zeros of matrix A, three-dimensional mesh.

Figure 7.3: Number of non-zeros of matrix A, two-dimensional mesh.

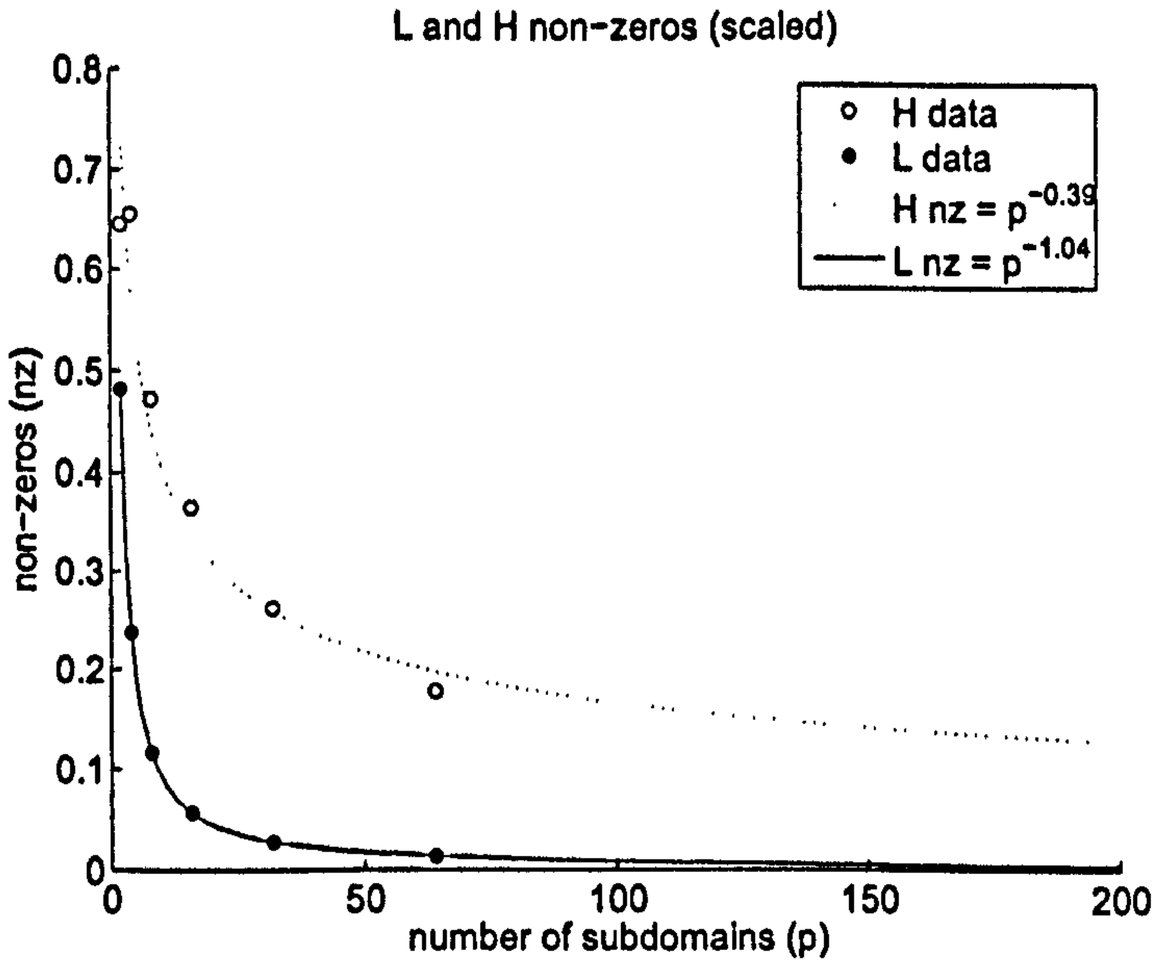


Figure 7.4: Number of non-zeros of matrices L and H, three-dimensional mesh.

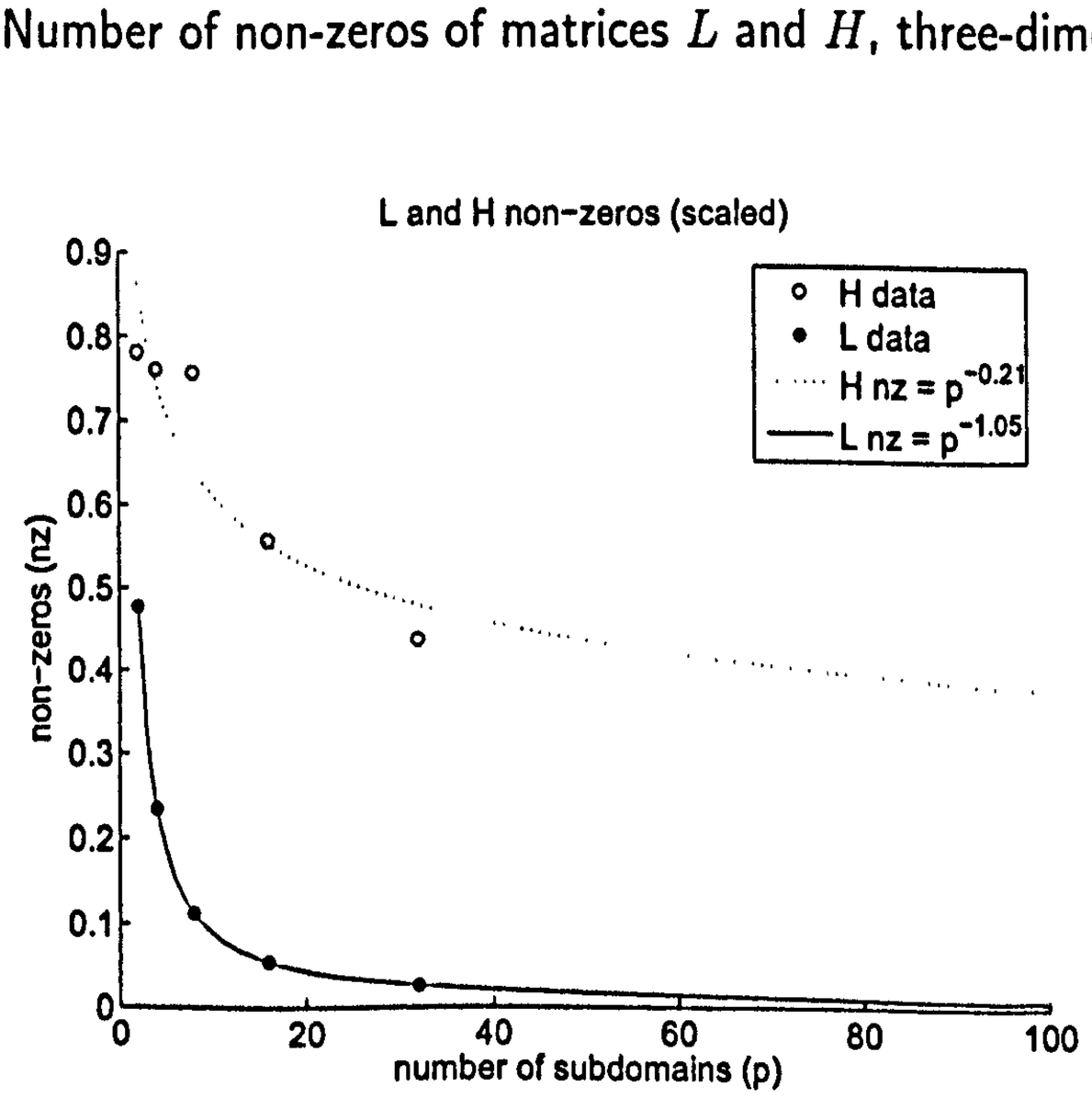


Figure 7.5: Number of non-zeros of matrices L and H, two-dimensional mesh.

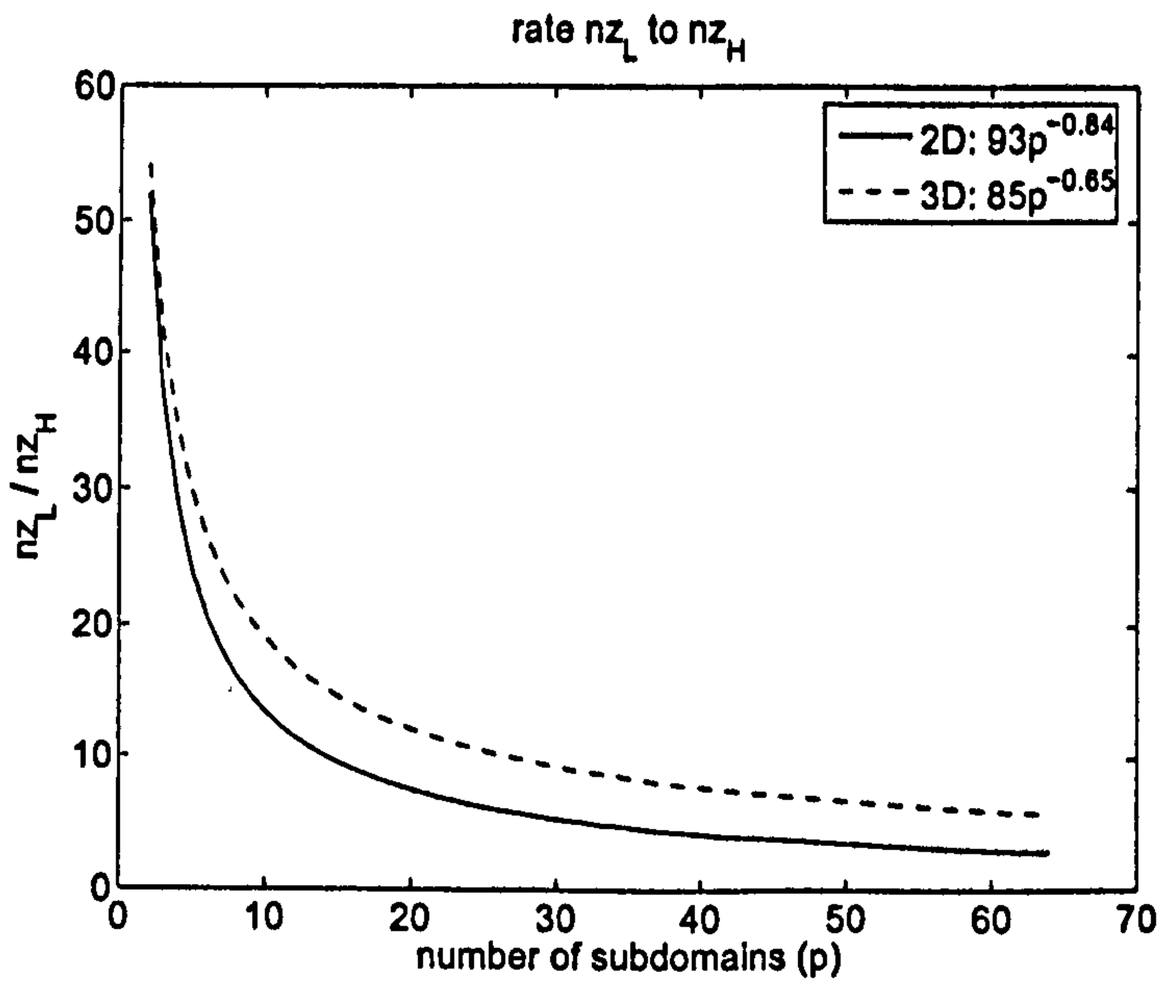


Figure 7.6: Ratio of number of non-zeros of matrix L to matrix H.

7.3 Spatial Discretization - FEM

In this section, a comparison between the Galerkin (GFEM) and Petrov-Galerkin (PGFEM) finite element methods for solving the convection-diffusion equation is presented. Some results addressing mesh refinement and boundary conditions for the Poisson and convection-diffusion equations are also reported. The formulation of both the Galerkin and Petrov-Galerkin methods for the Poisson and convection-diffusion equations can be found in Chapter 3 (pages 33–52). The results were obtained using a code that was implemented with the main purpose of understanding the finite element method. The solvers were coded initially in MatLab and then in Fortran 90.

The systems of linear equations are solved using the function `LA_GESV` of LAPACK95 – Fortran 95 Interface to LAPACK [4]. LAPACK is a library written in Fortran77 which provides routines for solving systems of linear equations,

least-squares solutions of linear systems of equations, eigenvalue problems and singular value problems. LAPACK95 improves upon the original user-interface to the LAPACK package, taking advantage of the considerable simplifications which Fortran 95 allows. More information on both LAPACK and LAPACK95 libraries my be found in `http://www.netlib.org`. The `LA_GESV` routine computes the solution to a real or complex linear system of equations $AX = B$, where A is a square matrix and X and B are rectangular matrices or vectors. Gaussian elimination with row interchanges is used to factor A as $A = PLU$, where P is a permutation matrix, L is unit lower triangular and U is upper triangular. The factored form of A is then used to solve the above system.

The partial differential equations being solved are the Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{7.1}$$

and the convection-diffusion equation

$$\mu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + v \left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} \right) = g(x, y) \tag{7.2}$$

on a two-dimensional domain Ω with boundary Γ.

7.3.1 Galerkin and Petrov-Galerkin FEM

For the first set of tests performed the functions $f(x, y)$ and $g(x, y)$ of Equations (7.1) and (7.2), respectively, where chosen to be

$$f(x, y) = 2\pi^2 \sin(\pi x) \sin(\pi y)$$

and

$$g(x, y) = 2\,\mu\pi^2 \sin(\pi x) \sin(\pi y) + v_1 \pi \cos(\pi x) \sin(\pi y) + v_2 \pi \sin(\pi x)$$

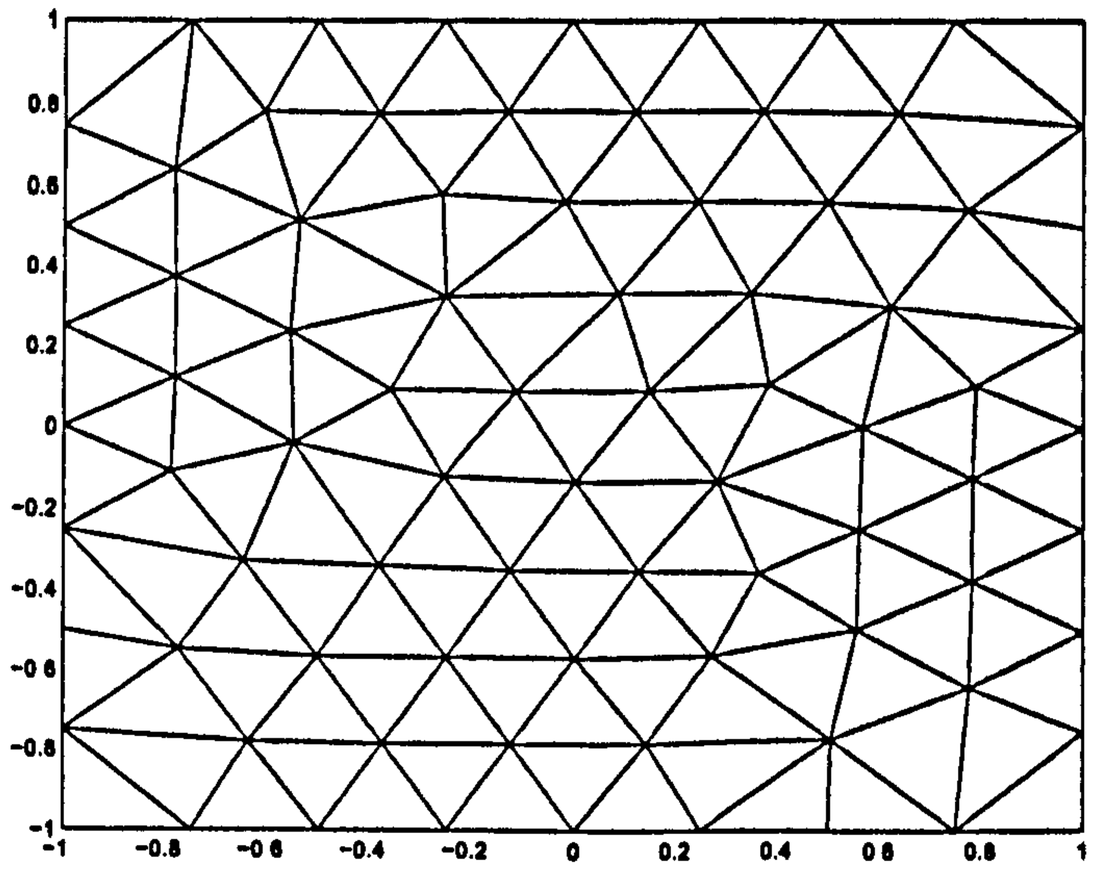


Figure 7.7: Mesh square8

so that for $u(x, y) = 0$ on Γ the exact solution is given by

$$u(x, y) = \sin(\pi x)\sin(\pi y)$$

The results were obtained on four meshes. The first mesh or coarser mesh was generated by a mesh generation program (Figure 7.7). The other three meshes are a result of global refinement. At each refinement step each triangle is divided into four triangles (classical method) by bisecting each edge (see Figure 7.8). This scheme preserves the proprieties of the elements and also does not generate hanging nodes. Table 7.2 lists the number of elements, nodes and boundary nodes of the four meshes used. Note that the number of elements and boundary nodes quadruple at each refinement whilst the number of nodes does not since the internal edges are shared by two triangles. The element size of the smallest and biggest element and the mesh Peclet number (Equation (3.26)) of the meshes used are listed on Table 7.3.

Tables 7.4 to 7.8 report the maximum and average error obtained for the problems described above, where **v** is kept constant and equals to 1 and μ varies, in order to

	Mesh 1	Mesh 2	Mesh 3	Mesh 4
Nodes	83	297	1121	4353
Elements	132	528	2112	8448
Boundary Nodes	32	64	128	256

Table 7.2: Number of elements, nodes and boundary nodes of mesh `square8` and the three meshes obtained by refining mesh `square8`

analyse the accuracy of the Galerkin and Petrov-Galerkin schemes under diffusion and convection dominance. Average error stands for the summation of the error of all nodes divided by the number of nodes of the mesh.

Table 7.4 shows the error of the results obtained by solving the Poisson equation on the four meshes described above. The error decreases by a factor of almost four as the mesh is refined which indicates that the error is proportional to h^2, since h decreases by a factor of 2. These results were expected and agree with the theory. The solution was obtained on all four meshes with different levels of accuracy.

Tables 7.5 and 7.6 list the error of the solution of the convection-diffusion equation using the Galerkin method. Tables 7.7 and 7.8 list the error of the same problem

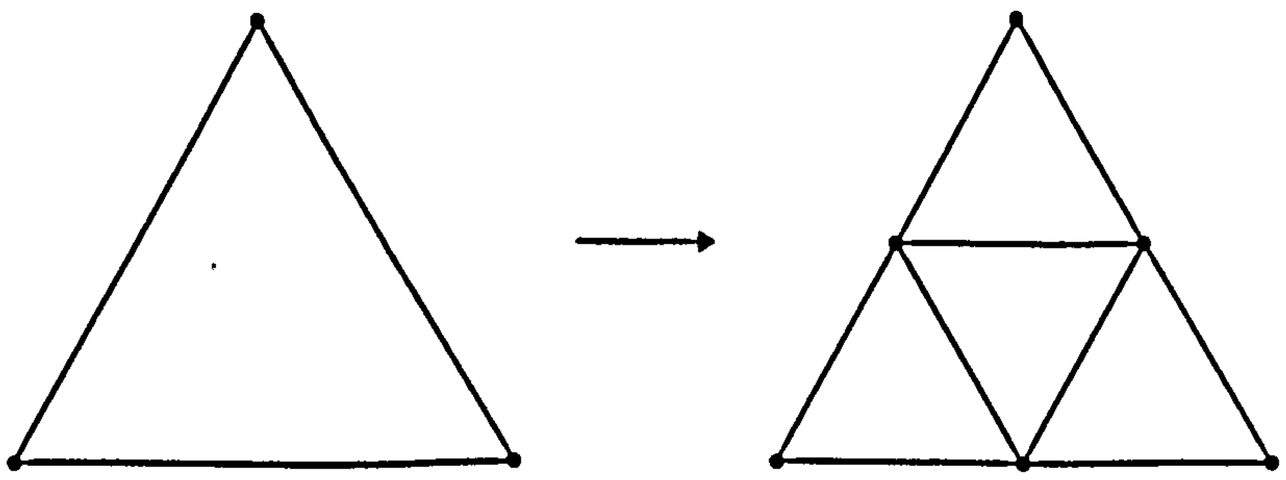


Figure 7.8: Refinement of a triangle

Mesh	h		Pe_h	
	Minimum	Maximum	Minimum	Maximum
1	0.152190	0.353553	$0.107615/\mu$	$0.250000/\mu$
2	0.076095	0.176777	$0.053807/\mu$	$0.125000/\mu$
3	0.038049	0.088388	$0.026904/\mu$	$0.062500/\mu$
4	0.019024	0.044194	$0.013452/\mu$	$0.031250/\mu$

Table 7.3: Mesh element size (h) and mesh Peclet number (Pe_h) for $\boldsymbol{v} = 1$

solved by the Petrov-Galerkin approximation. For $\mu \geq 10^{-1}$, the errors of both approximations are similar and, as for the Poisson equation, decrease by a factor of almost four, which agrees with the theory. For $\mu < 10^{-1}$, the error of the Petrov-Galerkin approximation almost does not increase when μ decreases whilst it decreases by a factor between two and three when the mesh is refined, as expected. On the coarsest mesh, the error of the Galerkin approximation increases by almost the same factor that μ decreases. However, as the mesh is refined the error becomes similar to the Petrov-Galerkin approximation error.

For problems dominated by convection the Galerkin approximation solution is oscillatory. For coarser grids, these oscillations completely dominate the solution and the Galerkin approximation does not perform well. On finer grids, the Galerkin and Petrov-Galerkin methods achieve similar error profiles. This means that refining the mesh, besides improving the accuracy of the solution by itself, reduces the oscillations observed by the Galerkin approximation and makes it comparable to the Petrov-Galerkin approximation. In practice, in Galerkin FEM the mesh needs to be refined adaptively which means that only the elements with $Pe_h > 1$ need to be refined.

Error	Mesh 1	Mesh 2	Mesh 3	Mesh 4
Maximum	0.9707×10^{-1}	0.2850×10^{-1}	0.7992×10^{-2}	0.2224×10^{-2}
	–	3.41	3.57	3.59
Average	0.2769×10^{-1}	0.8543×10^{-2}	0.2322×10^{-2}	0.6103×10^{-3}
	–	3.24	3.68	3.80

Table 7.4: Maximum and average error and error ratio for the Galerkin approximation of the Poisson equation

7.3.2 Boundary Conditions

In order to validate the code used to generate the results presented in this section, tests were performed using different boundary conditions and compared to the results reported in the literature. For instance, the streamline and potential flow around a cylinder were calculated and the results are showed in Figure 7.9. The results obtained agree with the results reported by Segerlind [105, pp. 132]. The streamline and potential flow are described by the Laplace equations, or homogeneous Poisson equations,

$$\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = 0$$

and

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0$$

and specific boundary conditions, as described in Figure 7.9.

μ	Mesh 1	Mesh 2	Mesh 3	Mesh 4
1.0e+1	0.9684×10^{-1}	0.2844×10^{-1}	0.7974×10^{-2}	0.2215×10^{-2}
	–	3.41	3.57	3.60
1.0e+0	0.9609×10^{-1}	0.2828×10^{-1}	0.7996×10^{-2}	0.2226×10^{-2}
	–	3.40	3.54	3.59
1.0e−1	$0.1011\times10^{+0}$	0.2290×10^{-1}	0.6080×10^{-2}	0.1594×10^{-2}
	–	4.41	3.77	3.81
1.0e−2	$0.1294\times10^{+0}$	0.2938×10^{-1}	0.6104×10^{-2}	0.1346×10^{-2}
	–	4.40	4.81	4.53
1.0e−3	$0.4098\times10^{+0}$	0.4899×10^{-1}	0.1012×10^{-1}	0.2038×10^{-2}
	–	8.36	4.84	4.97
1.0e−4	$0.3883\times10^{+1}$	$0.1835\times10^{+0}$	0.1818×10^{-1}	0.3156×10^{-2}
	–	21.16	10.09	5.76
1.0e−5	$0.3870\times10^{+2}$	$0.8101\times10^{+0}$	0.4229×10^{-1}	0.9882×10^{-2}
	–	47.77	19.16	4.28
1.0e−6	$0.3869\times10^{+3}$	$0.8338\times10^{+1}$	$0.2195\times10^{+0}$	0.2053×10^{-1}
	–	46.40	37.99	10.69
1.0e−7	$0.3854\times10^{+4}$	$0.8359\times10^{+2}$	$0.2203\times10^{+1}$	0.8564×10^{-1}
	–	46.11	37.94	25.72

Table 7.5: Maximum error and error ratio for the Galerkin approximation of the convection-diffusion equation with $v = 1$

μ	Mesh 1	Mesh 2	Mesh 3	Mesh 4
1.0e+1	0.2762×10^{-1}	0.8533×10^{-2}	0.2320×10^{-2}	0.6095×10^{-3}
	–	3.24	3.68	3.81
1.0e+0	0.2721×10^{-1}	0.8404×10^{-2}	0.2285×10^{-2}	0.6004×10^{-3}
	–	3.24	3.68	3.81
1.0e−1	0.2629×10^{-1}	0.7603×10^{-2}	0.2025×10^{-2}	0.5267×10^{-3}
	–	3.46	3.75	3.84
1.0e−2	0.2469×10^{-1}	0.5889×10^{-2}	0.1411×10^{-2}	0.3558×10^{-3}
	–	4.19	4.17	3.97
1.0e−3	0.6433×10^{-1}	0.1120×10^{-1}	0.1867×10^{-2}	0.3796×10^{-3}
	–	5.74	6.00	4.92
1.0e−4	$0.5810\times10^{+0}$	0.3284×10^{-1}	0.3799×10^{-2}	0.6569×10^{-3}
	–	17.69	8.64	5.78
1.0e−5	$0.5764\times10^{+1}$	$0.1647\times10^{+0}$	0.4992×10^{-2}	0.1581×10^{-2}
	–	35.00	32.99	3.16
1.0e−6	$0.5761\times10^{+2}$	$0.1621\times10^{+1}$	0.1225×10^{-1}	0.3451×10^{-2}
	–	35.54	132.33	3.55
1.0e−7	$0.5737\times10^{+3}$	$0.1620\times10^{+2}$	0.3284×10^{-1}	0.5001×10^{-2}
	–	35.41	493.30	6.57

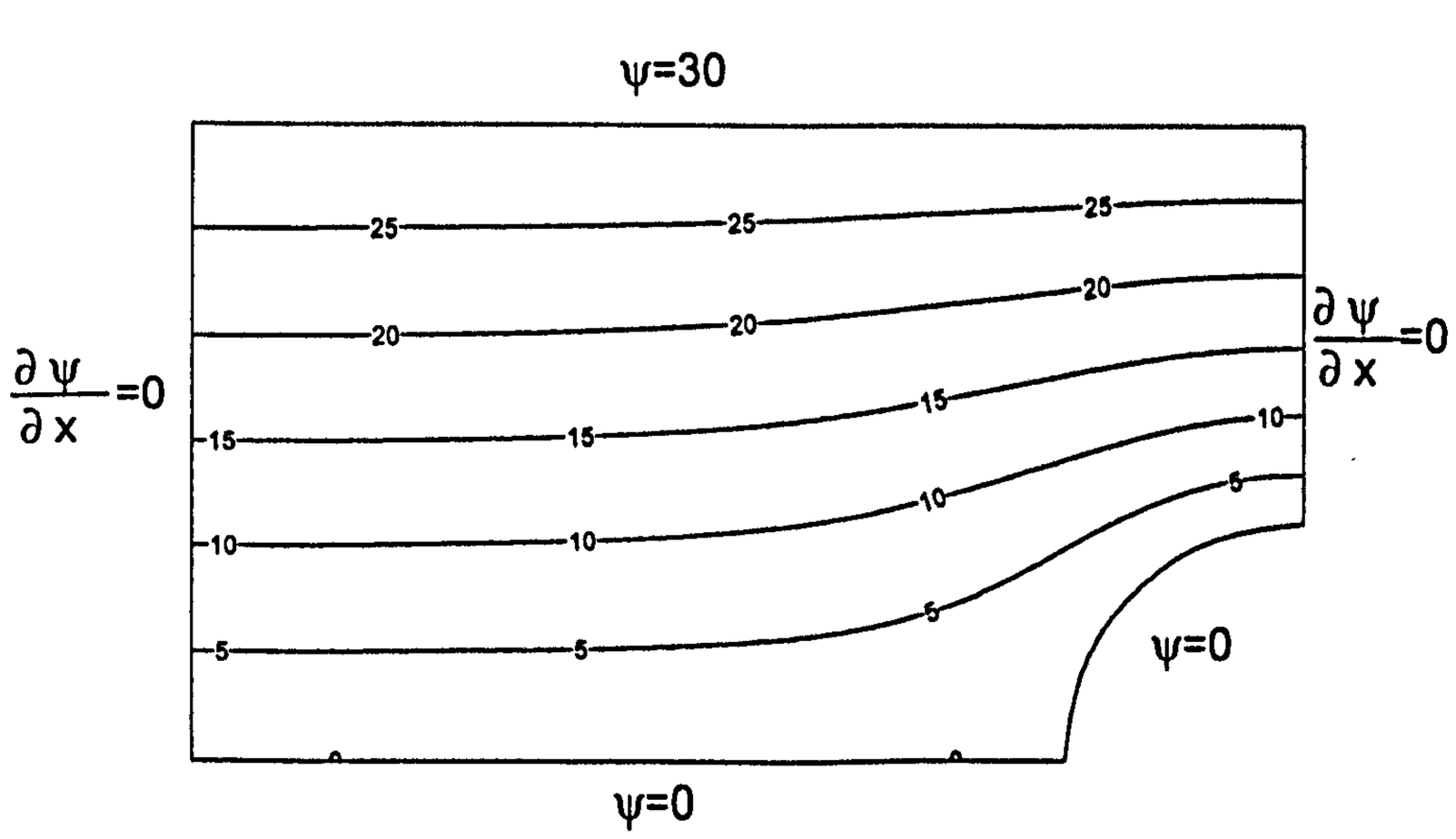
Table 7.6: Average error and error ratio for the Galerkin approximation of the convection-diffusion equation with $v = 1$

μ	Mesh 1	Mesh 2	Mesh 3	Mesh 4
1.0e+1	0.9665×10^{-1}	0.2839×10^{-1}	0.7963×10^{-2}	0.2211×10^{-2}
	–	3.40	3.57	3.60
1.0e+0	0.9488×10^{-1}	0.2760×10^{-1}	0.7816×10^{-2}	0.2178×10^{-2}
	–	3.44	3.53	3.59
1.0e−1	$0.1281\times10^{+0}$	0.3508×10^{-1}	0.1005×10^{-1}	0.2751×10^{-2}
	–	3.65	3.49	3.65
1.0e−2	$0.2530\times10^{+0}$	0.8960×10^{-1}	0.3122×10^{-1}	0.1106×10^{-1}
	–	2.82	2.87	2.82
1.0e−3	$0.2767\times10^{+0}$	$0.1046\times10^{+0}$	0.4510×10^{-1}	0.2044×10^{-1}
	–	2.65	2.32	2.21
1.0e−4	$0.2790\times10^{+0}$	$0.1062\times10^{+0}$	0.4686×10^{-1}	0.2210×10^{-1}
	–	2.63	2.27	2.12
1.0e−5	$0.2793\times10^{+0}$	$0.1063\times10^{+0}$	0.4705×10^{-1}	0.2228×10^{-1}
	–	2.63	2.26	2.11
1.0e−6	$0.2793\times10^{+0}$	$0.1064\times10^{+0}$	0.4707×10^{-1}	0.2230×10^{-1}
	–	2.63	2.26	2.11
1.0e−7	$0.2793\times10^{+0}$	$0.1064\times10^{+0}$	0.4707×10^{-1}	0.2230×10^{-1}
	–	2.63	2.26	2.11

Table 7.7: Maximum error and error ratio for the Petrov-Galerkin approximation of the convection-diffusion equation with $v = 1$

μ	Mesh 1	Mesh 2	Mesh 3	Mesh 4
1.0e+1	0.2761×10^{-1}	0.8525×10^{-2}	0.2317×10^{-2}	0.6086×10^{-3}
	–	3.24	3.68	3.81
1.0e+0	0.2733×10^{-1}	0.8764×10^{-2}	0.2295×10^{-2}	0.6030×10^{-3}
	–	3.12	3.82	3.81
1.0e−1	0.3973×10^{-1}	0.1223×10^{-1}	0.3328×10^{-2}	0.8680×10^{-3}
	–	3.25	3.67	3.83
1.0e−2	0.6284×10^{-1}	0.2516×10^{-1}	0.1024×10^{-1}	0.3712×10^{-2}
	–	2.50	2.46	2.76
1.0e−3	0.6635×10^{-1}	0.2735×10^{-1}	0.1279×10^{-1}	0.6240×10^{-2}
	–	2.43	2.14	2.05
1.0e−4	0.6671×10^{-1}	0.2756×10^{-1}	0.1306×10^{-1}	0.6523×10^{-2}
	–	2.42	2.11	2.00
1.0e−5	0.6675×10^{-1}	0.2758×10^{-1}	0.1308×10^{-1}	0.6552×10^{-2}
	–	2.42	2.11	2.00
1.0e−6	0.6675×10^{-1}	0.2759×10^{-1}	0.1309×10^{-1}	0.6555×10^{-2}
	–	2.42	2.11	2.00
1.0e−7	0.6675×10^{-1}	0.2759×10^{-1}	0.1309×10^{-1}	0.6555×10^{-2}
	–	2.42	2.11	2.00

Table 7.8: Average error and error ratio for the Petrov-Galerkin approximation of the convection-diffusion equation with $v = 1$



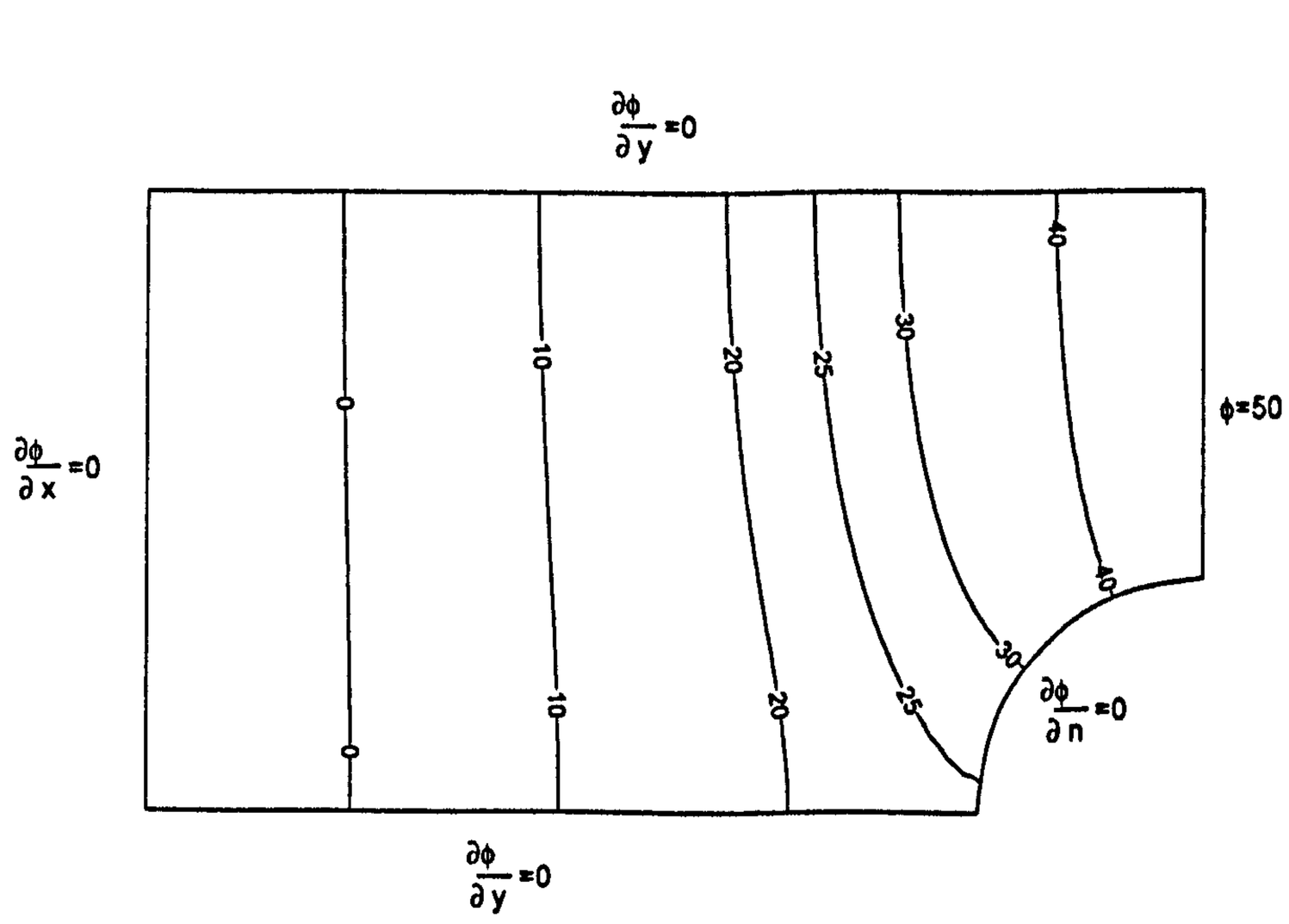


Figure 7.9: Streamline (top) and potential (bottom) flow around a cylinder

7.4 Algebraic Solver - DCG

In this section the parallel performance of DCG is addressed. A series of tests were run to measure DCG performance and some of the most relevant results are reported.

7.4.1 Convergence Rate

Figure 7.10 shows the convergence rate of Problem 2D solved by CG (1 processor), DCG (2 and 4 processors) and DSD (2 processors), which is basically DCG, except that p (search direction) is equal to r (residual) at each iteration, i.e. a steepest descent iteration is performed instead of a CG iteration. It shows that DCG converges much faster than DSD. Figure 7.11 shows the 2-norm of the perturbation on the right-hand side at each iteration for the same problem in 2 subdomains. Note that it is almost monotonic and similar on both subdomains. Similar behaviour has been observed with more then two subdomains.

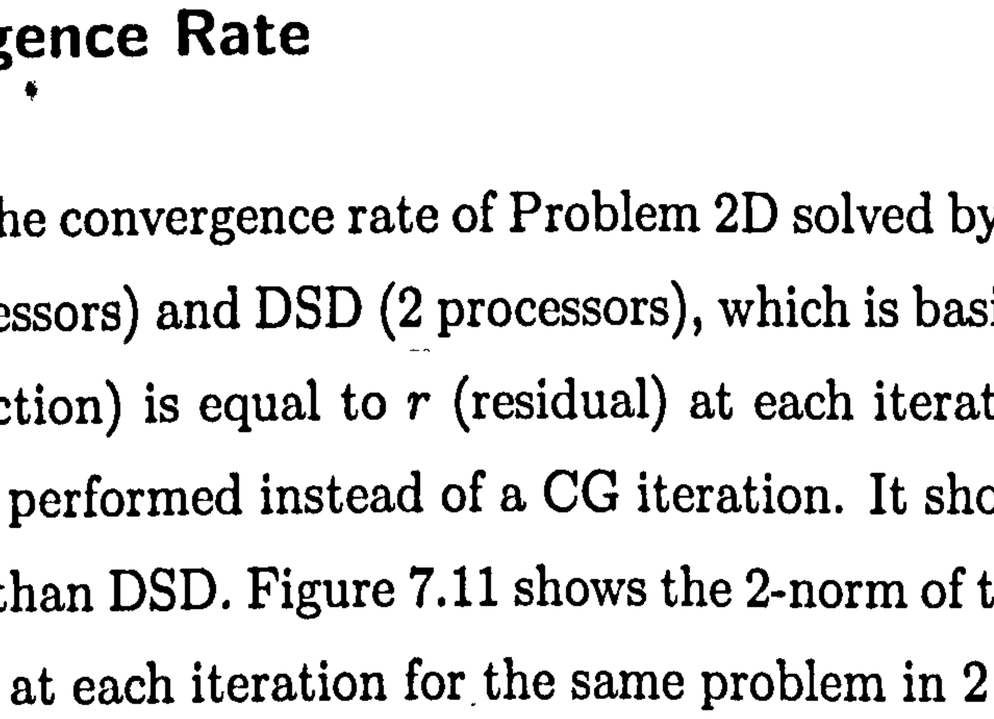


Figure 7.10: Problem 2D convergence rate (mesh square4)

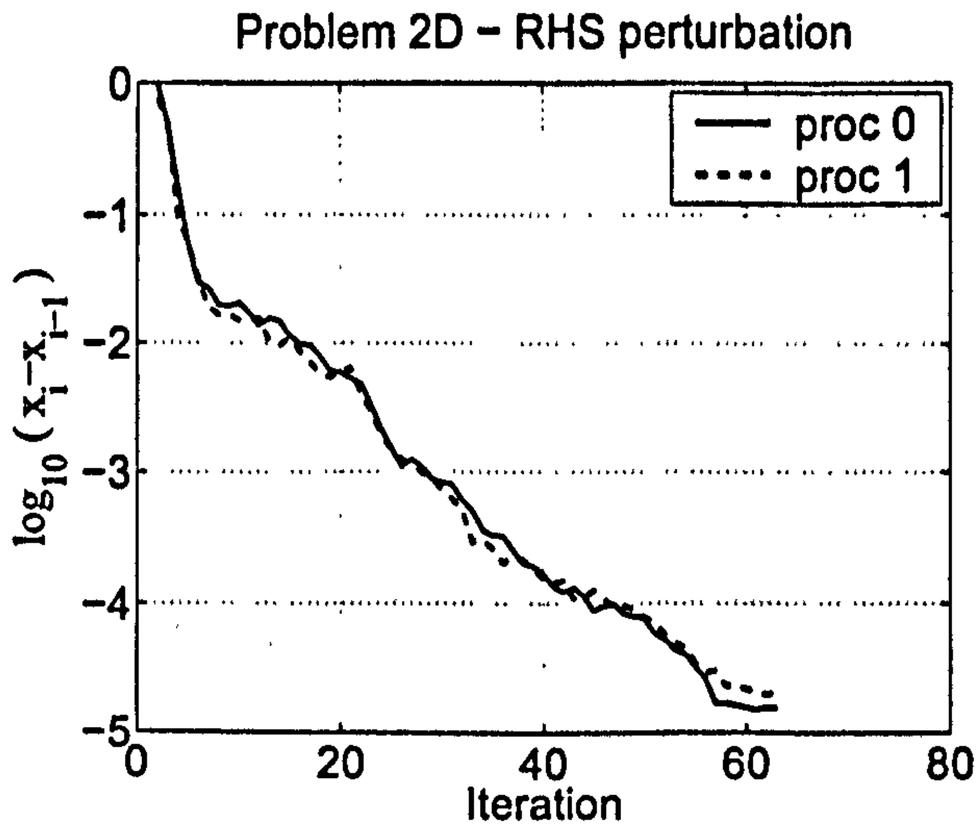


Figure 7.11: Problem 2D RHS perturbation (mesh square4)

7.4.2 Speedup and Efficiency

Table 7.9 shows the number of iterations, the run time (in seconds), the speedup and the efficiency of Problem 3Da obtained by our sequential and parallel implementations. The speedup and efficiency are also plotted on Figures 7.12 and 7.13. The results presented show that the parallelization of the method led to a good overall performance. One can also notice that the speedup is even superlinear in some cases. This is mainly due to the minimized amount of communication, the data transfer scheme adopted and cache memory.

The execution time per iteration of two given problems are shown in Figure 7.14. For this case, the mesh roughly doubles the number of nodes as the number of processors is doubled, i.e. the size of the mesh of each processor is approximately the same. There is a slightly increase on the time per iteration as the number of processors is increased.

n	Procs	Iters	Run-time	Speed-up	Efficiency
294,518	1	380	146.0	-	-
	2	561	118.4	1.23	0.62
	4	575	59.6	2.45	0.61
	8	514	19.2	7.60	0.95
	16	454	6.0	24.33	1.52
	32	752	3.9	37.44	1.17
581,027	1	534	501.4	-	-
	2	734	357.1	1.40	0.70
	4	881	241.4	2.08	0.52
	8	709	116.5	4.30	0.54
	16	691	48.2	10.40	0.65
	32	659	9.4	53.34	1.67

Table 7.9: Number of iterations, execution time, speedup and efficiency of problem 3Db for n=294,518 and n=581,027

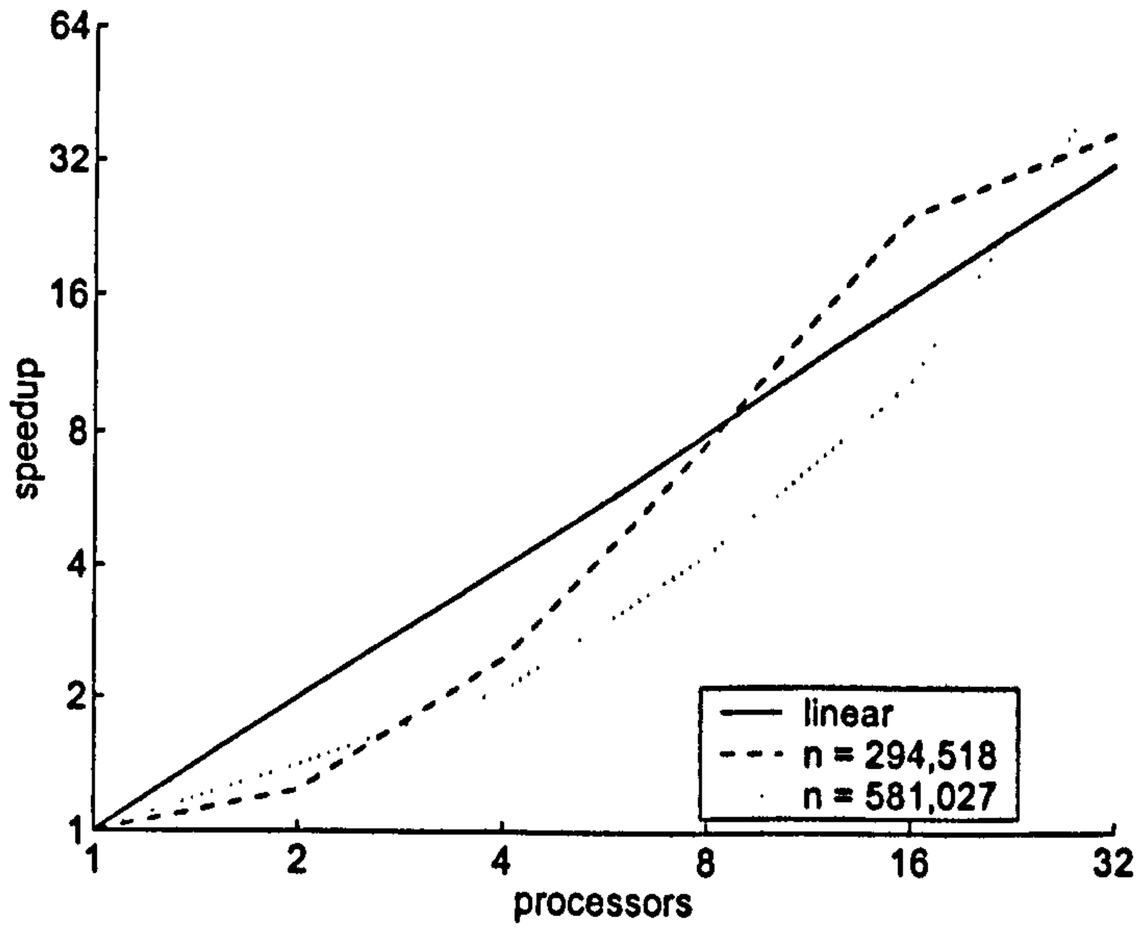


Figure 7.12: Speedup of problem 3Db for n=294,518 and n=581,027

Figure 7.13: Efficiency of problem 3Db for n=294,518 and n=581,027

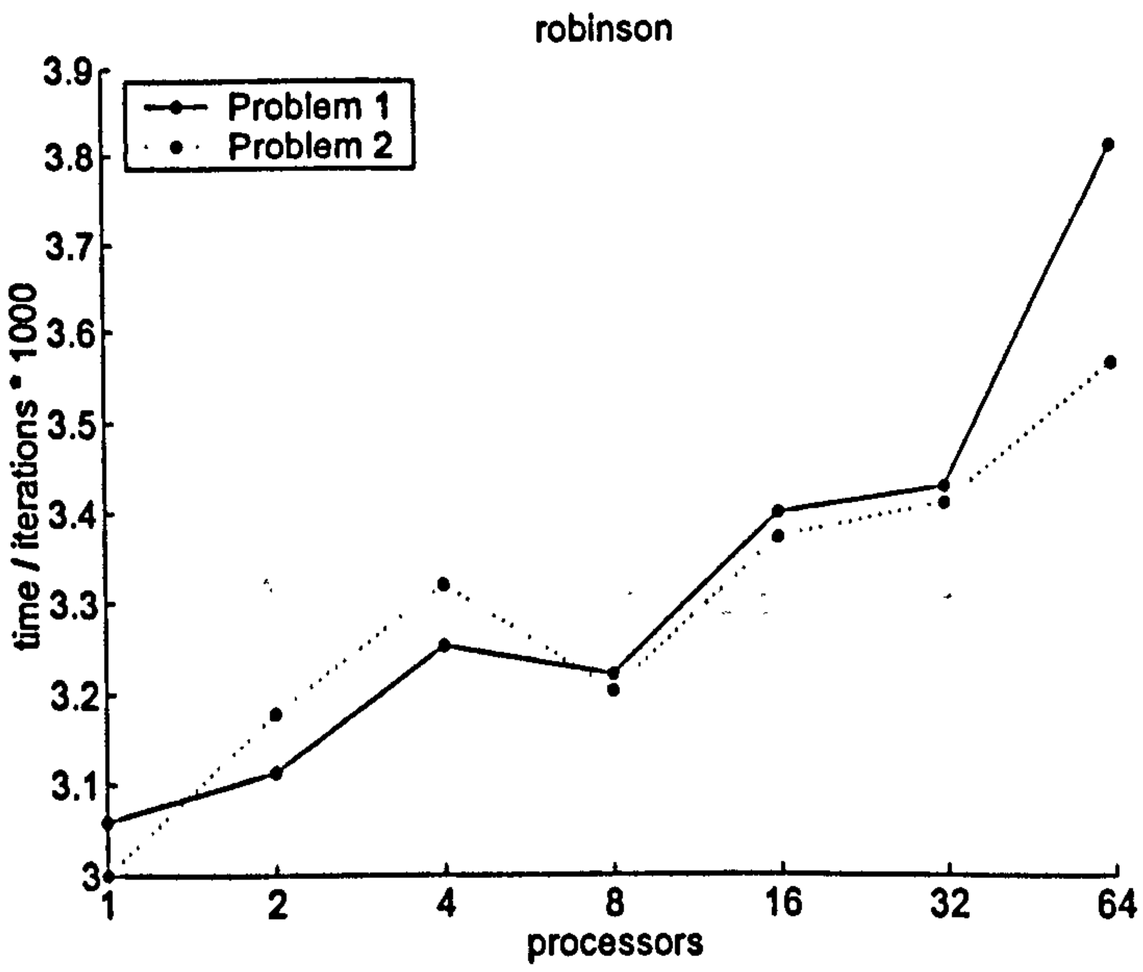


Figure 7.14: Execution time per iteration of two two-dimensional problems in a square domain where the number of nodes of the mesh roughly doubles as the number of processors doubles, i.e. the problem size of each subdomain is roughly the same for any number of processors.

7.4.3 Overlapping and Interface Update

A important factor on the performance of DCG is the overlapping. However, increasing the overlapping does not necessarily reduce the number if iterations and mainly the execution time. Table 7.10 shows the number of iterations and the execution time of problem 3Db with two different meshes. In most of the cases the execution time increases as the overlap increases since the amount of computation per subdomain is increased. The number of iterations usually decreases for 8 or more processors but, as shown on the table, it is not always the case. For most of the tests run an overlap of two layers of elements achieves a given accuracy with least iterations and many times with least time as well. However, in comparison to one layer of overlapping the reduction is not relevant in most cases.

Table 7.11 lists the number of iterations to achieve a given accuracy when $\hat{x}$ is only updated at each h iterations, i.e. there is exchange of data among the processors only if the modulus of iteration number to h is zero – mod $(i, h) = 0$. In this case, the amount of overlapping is actually relevant to obtain the solution. If h is increased and the overlap remains minimal, the residual norm oscillates greatly, decreasing at the iterations with data exchange and increasing otherwise. The increase on overlapping when h increases might even reduce the number of iterations to achieve the solution though it will in most cases increase the execution time. In architectures where the network speed is considerable slow, the use of more overlapping combined to a greater h may result in a reduction to the execution time. In other words, the size of h and the amount of overlapping to achieve a minimal execution time depends on the ratio of communication to computation time of the parallel computer.

	n = 294,518		n = 581,027	
p	ov = 1	ov = 2	ov = 1	ov = 2
2	550	599	734	807
	150.2	162.5	481.3	540.8
4	597	630	847	863
	74.5	88.1	273.1	282.1
8	514	501	885	710
	20.5	28.9	109.6	108.6
16	517	462	613	657
	7.8	9.7	28.3	39.3

Table 7.10: Number of iterations and execution time for overlap equals to 1 and 2.

7.4.4 Preconditioning

In order to validate the preconditioned DCG algorithm (Section 6.3.2, page 160), some tests were performed using the Jacobi and polynomial preconditioners. Note that this tests were not performed in order to evaluate the preconditioners but to numerically validate Algorithm 6.3.3. The Jacobi preconditioner consists simply by the inverse of the diagonal of A, i.e. $M = [diag(A)]^{-1}$. The polynomial preconditioners used can be expressed by

$$\left(\sum_{i=0}^{m} \gamma_{m,i} \left(I - [diag(A)]^{-1}A\right)^{i}\right) [diag(A)]^{-1} \tag{7.3}$$

which can be computed as a sequence of vector updates and matrix-vector products [34] and where m is the degree of the polynomial. The coefficients $\gamma_{m,i}$ define the kind of polynomial preconditioner being used. The Neumann preconditioner is obtained by setting $\gamma_{m,i} = 1$ for all i. The weighted and unweighted least-squares polynomial preconditioners are described in [59]. The coefficients used in the tests reported in this work were obtained directly from PIM [34].

Overlap	p	h=1	h=2	h=5	h=10
1	2	940	1454	6373	4887
	4	2287	-	-	6158
	8	1694	-	-	7046
	16	1960	-	-	-
	32	1395	-	-	-
	64	1064	-	-	-
2	2	953	940	999	1019
	4	1256	1268	1220	1286
	8	1097	1221	1314	1539
	16	1002	1124	1624	2176
	32	1088	1323	1824	2608
	64	962	1365	1879	2758
4	2	1066	1031	1022	1024
	4	1977	1248	1203	1379
	8	1568	1180	1212	1369
	16	1555	1219	1429	1748
	32	-	1169	1529	2039
	64	1140	1146	1514	2197

Table 7.11: Number of iterations for $\hat{x}$ being updated at each h iterations with overlapping equals to 1, 2 and 4.

Table 7.12 lists the number of iterations needed to solve a given problem using the PCG and PDCG methods. The number of iterations of PCG is independent of the number of processors while the performance of PDCG depends strongly on the number of processors used. The results obtained were satisfactory although the number of iterations of both preconditioned and non-preconditioned cases increases as the number of processors increases. The same tests were run on a different problem and the results were similar. For this problem, the Jacobi preconditioner was successfully applied, reducing the number of iterations from 281 (non-preconditioned) to 87 when used in combination to the PDCG method on 8 processors. Using CG, the number of iterations was reduced from 243 (non-preconditioned case) to 94.

Figure 7.15 shows the residual of DCG without preconditioning and preconditioned by the Neumann polynomial preconditioner of degree 4, in 2 and 16 processors. The rate of convergence is considerably improved in both cases. However, the performance deteriorates as the number of processors increases. The same behaviour is observed on DCG and is characteristic of the method.

7.4.5 Boundary Conditions

DCG as been tested primarily to solve problems with Dirichlet boundary conditions. A set of tests using a combination of Dirichlet and Neumann boundary conditions has shown that the method does not perform well for problems dominated by Neumann boundary conditions - Table 7.13. This is due to the fact that the only means of communication among subdomains is the overlapping on the interface. The use of multiple meshes might correct or reduce this deficiency.

Preconditioner	Degree	PCG	PDCG			
			p=2	p=4	p=8	p=16
None	-	38	39	38	40	39
Jacobi	-	38	37	35	35	36
Neumann	1	180	-	-	-	-
	2	28	26	24	25	29
	4	28	25	24	26	30
Unweighted	1	19	24	23	25	29
Least-squares	2	20	19	19	23	29
	4	18	19	21	27	31
Weighted	1	19	25	23	25	29
Least-squares	2	18	18	18	23	28
	4	17	20	22	28	31

Table 7.12: Number of iterations of the preconditioned CG and DCG methods; problem 3Db

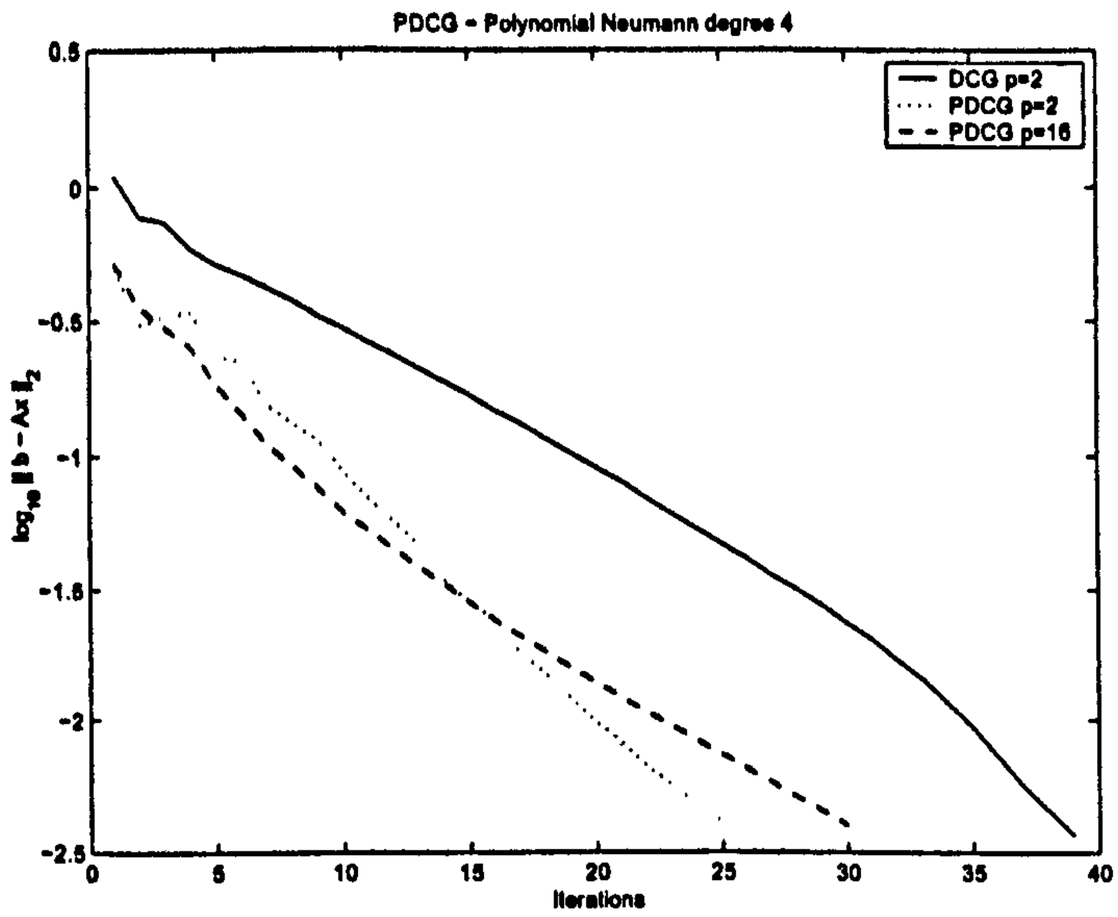


Figure 7.15: Residual of DCG and PDCG with Neumann polynomial preconditioner of degree 4; problem 3Db, 2 and 16 processors

	$\nabla^2\phi = g$			$\nabla^2\phi = 0$		
p	Case 1	Case 2	Case 3	Case 1	Case 2	Case 3
1	77	292	91	74	285	90
2	86	2013	104	80	1713	103
3	82	2721	99	77	2258	99
4	84	4296	98	77	3453	98
5	80	6233	96	76	4984	96
6	80	6593	98	83	5268	108
7	83	8121	100	78	6504	99
8	82	9020	100	76	7144	99

Table 7.13: Number of iterations for Dirichlet and Neumann boundary conditions. Case 1: only Dirichlet; case 2: predominantly Neumann; case 3: predominantly Dirichlet. $g = 3\pi^2 \cos(\pi x/20)\cos(\pi y/120)\cos(\pi z/20)$.

7.5 Poisson Solver

To measure the parallel efficiency, the Poisson solver was split into four main procedures, namely

- Loading: one processor loads the mesh;
- Initialization: the mesh is partitioned, the nodal graph is created and arrays/variables are initialized;
- Discretization: the Poisson equation is discretized and the boundary conditions are applied;
- Algebraic solver: the system of equations is solved by CG for a single processor and DCG otherwise.

The procedure loading is executed only by the root processor. Hence, it is independent of the number of processors and does not have any communication. The procedure initialization has two main parts: partitioning the mesh and creating the nodal graph. The nodal graph routine is totally local and therefore perfectly parallel. The mesh partition is initially done by the root processor and then each part is scattered from the root processor to the appropriate processor. The root keeps only its own part. This approach has been chosen for simplicity albeit compromising the overall speedup. The parallel efficiency could be improved by replacing Metis by ParMetis [64] and hence turning this procedure parallel.

The discretization is perfectly parallel, i.e. does not involve any communication. Its efficiency depends mainly on memory access. The algebraic solver exchanges data at each iteration and is the most expensive part of the solver. DCG achieves almost linear speedup for Problem 3Db, as is shown on Figure 7.16 and 7.17.

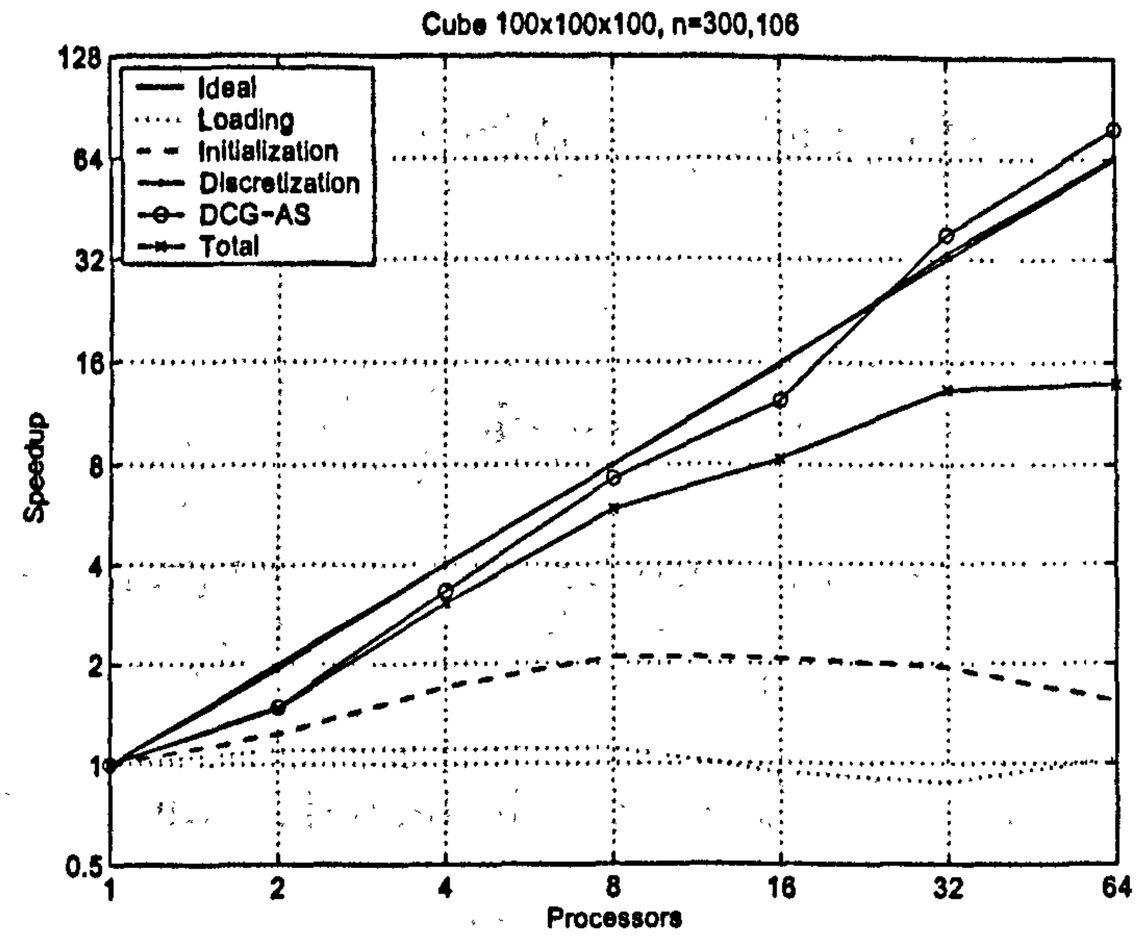


Figure 7.16: Problem 3Db speedup

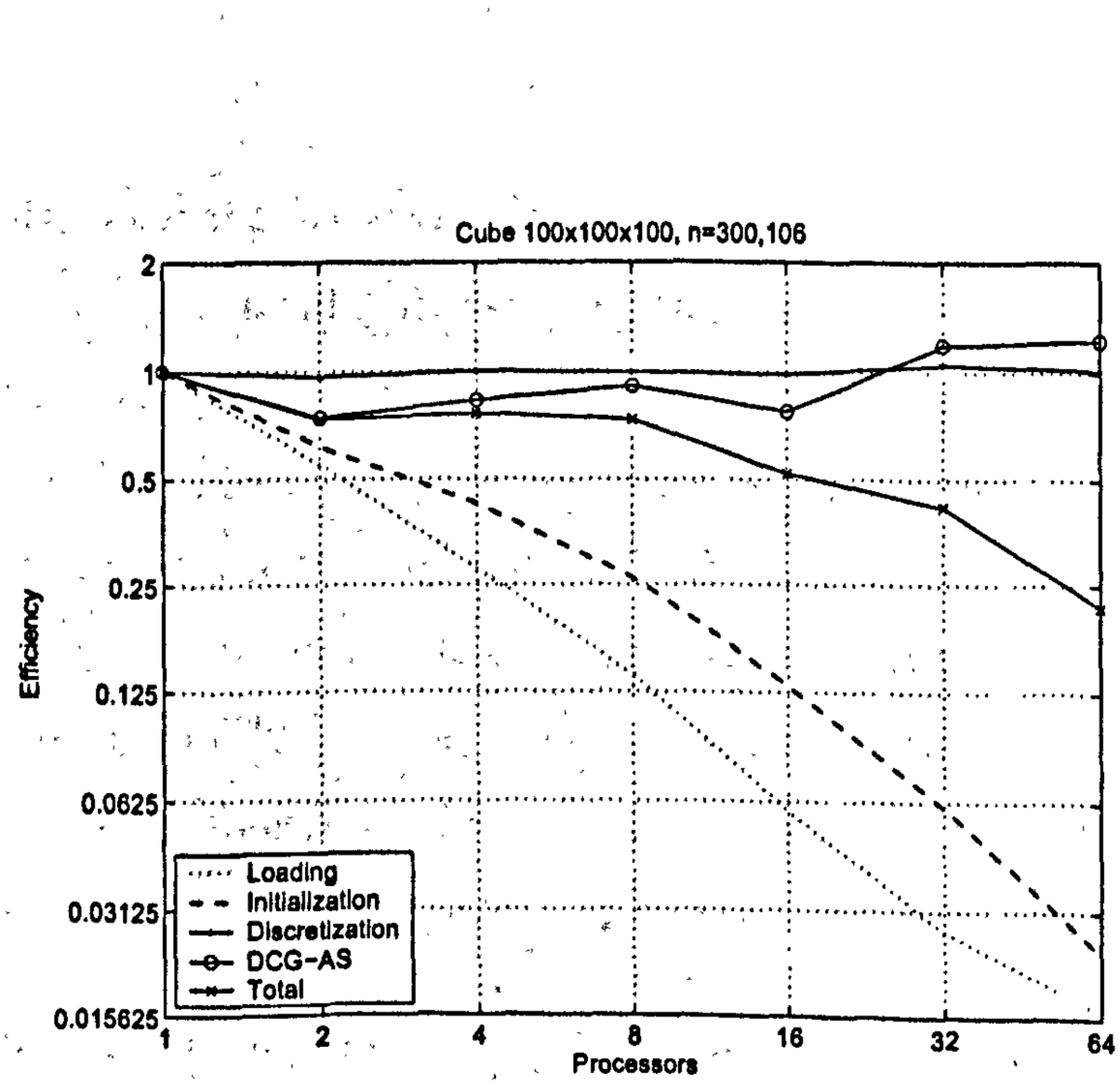


Figure 7.17: Problem 3Db efficiency

7.6 Final Remarks

Numerical results to measure the performance of DCG have been reported. The results show that DCG achieves high parallelism for most of the cases presented. However, the method is still not suitable for any type of boundary condition. This problem might be addressed in the future but is not part of the scope of this work. It has been shown through the numerical results that the method is efficient for many cases, both mathematically and computationally .

Chapter 8

Conclusion and Future Developments

In this thesis, a study on the solution of systems of linear equations on parallel computers using domain decomposition techniques and Krylov subspace methods has been performed. The main achievement of this study is the so-called distributive conjugate gradient (DCG) algorithm. DCG has been shown that simple domain decomposition methods can achieve high scalability and efficiency through the use of adequate solvers at subdomain level. A comparison between the Galerkin and Petrov-Galerkin finite element methods for solving the convection-diffusion equation was also carried out.

Most of the numerical results presented in this work were obtained using the parallel PDE solver implemented as part of this project, called PUPS – Parallel Unstructured PDE Solver. Although the conceptual design of PUPS proposes a parallel solver capable of solving a variety of partial differential equations, in n spatial dimensions, using finite element methods with a choice of several methods to solve systems of linear equations, the actual implementation to date only comprises the methods needed for the development of DCG. The solver has successfully been used in a geometry optimization project to solve the Poisson equation using

linear triangular elements to discretize the equation and the conjugate gradient method to solve the system of linear equations.

DCG presents a novel method for approximating the solution at subdomain level. This method, which consists roughly of a modification of the conjugate gradient (CG) algorithm, uses a technique to update the information from a previous step according to the perturbation applied to the sub-system. This modification produces great improvement in the convergence rate. When using CG without the modification as the local solver, an unacceptably high amount of computations is required, since each perturbed problem is solved purely as a new problem.

The combination of a simple domain decomposition method and an efficient subdomain solver led to an algorithm that is relatively easy to parallelize. Communication among subdomains is comprised of the exchange of data of the interface nodes at every iteration and of the calculation of the norm of the residual over the entire domain which is only performed at selected iterations. The iterations where the norm needs to be calculated are determined by an algorithm that roughly predicts the number of iterations needed to achieve a certain accuracy. This prediction is dynamically corrected as the computation proceeds. It has been shown that the number of times that the norm has to be calculated is considerably reduced by the adoption of the prediction algorithm.

DCG has achieved high parallelism for most of the cases tested. The parallel speed-up is roughly linear for up to 100 processors for many of the problems tested. It has been shown through the numerical results that the method is efficient both mathematically and computationally. However, the method is still not suitable for all types of boundary condition. Problems with Dirichlet, Neumann and mixed boundary conditions were solved but DCG requires an unacceptably large number of iterations when the boundary conditions are predominately Neumann or mixed.

Further work is needed to improve DCG for solving problems with any boundary conditions. The only means of communicating information among subdomains of the current algorithm is the overlap between subdomains. A technique that allows information to travel quicker over the entire domain is needed. Increasing the overlap between subdomains or the use of multi-level domain decomposition methods are two possible options.

The emphasis of this thesis was on the development of a solver based on single-level domain decomposition methods and iterative methods, such as the conjugate gradient method, as subdomain solvers. Based on DCG, further development can be done in order to develop a method for nonsymmetric systems. A brief study took place using BiCG and BiCGSTAB as the initial subdomain solver. This study has shown that the technique used on DCG cannot be used straightforwardly for the nonsymmetric case.

DCG could also be improved by the use of a multi-level domain decomposition method instead of a single-level method. In this work, multi-level methods were not considered because of the use of unstructured meshes and the difficulties of generating multiple meshes in this case. Algebraic multi-level methods can be considered as well as geometric multi-level methods on structured grids.

A preconditioned version of DCG (PDCG) was also presented. Some numerical tests were performed using the Jacobi and polynomial preconditioners in order to validate the algorithm. Although the results obtained were satisfactory, further research is needed on the use of preconditioners with DCG and the use of DCG as a preconditioner. There is also a need for substantial advancement in the theoretical analyses of DCG to provide general conditions under which the method will converge.

The Galerkin and Petrov-Galerkin finite element methods were used for solving the

convection-diffusion equation. The numerical results have shown that for problems dominated by convection the Galerkin method is outperformed by the Petrov-Galerkin method and, depending on the mesh, it fails. However, for problems dominated by convection being solved using a relatively fine mesh or for problems dominated by diffusion both methods are similar and the error is proportional to the square of the mesh size.

The parallel PDE solver proposed in this project has only its initial state implemented. Poisson and convection-diffusion equations in two- and three-dimensions, with Dirichlet, Neumann and mixed boundary conditions might be solved. The basic modules, that comprise geometric and algebraic basic operations both in serial and in parallel, are relatively well developed. Also, an interface to MPI has been developed to an advanced state. However, only linear triangular and tetrahedral elements were implemented and therefore many more may be added. A few methods to solve systems of linear equations and preconditioners are available. To increase the algebraic capabilities of the solver, integration to a package for solving system of linear equations is the main option which has been considered.

Appendix A

SunFire 15K Bandwidth and Communication Time

Words	p=2	p=4	p=8	p=16	p=32
1	8.22	6.78	5.19	3.94	3.50
2	16.49	13.58	10.36	7.98	7.16
4	32.29	26.38	20.03	15.41	13.98
8	62.26	50.40	37.53	28.13	25.89
16	119.91	94.89	68.09	49.10	47.64
32	236.62	187.41	110.95	99.47	93.89
64	328.02	233.80	116.80	73.11	42.77
128	622.81	430.81	243.56	136.86	29.19
256	1062.26	705.00	440.28	250.27	146.31
512	1605.32	1104.84	760.69	416.01	244.83
1024	1899.69	1292.17	1030.83	553.25	334.59
2048	2338.52	1552.96	1273.01	735.68	462.54
4096	2637.84	2106.63	1549.31	931.28	575.38
8192	2706.49	2175.61	1904.04	1082.27	625.34
16384	3231.15	2658.22	1982.62	1090.41	671.33
32768	3572.99	2820.97	2134.00	1108.56	701.47

Table A.1: Sun Fire 15K system measured bandwidth (megabits/second) of a broadcast collective communication.

Words	p=2	p=4	p=8	p=16	p=32
1	7.78e-006	9.45e-006	1.23e-005	1.62e-005	1.83e-005
2	7.76e-006	9.42e-006	1.24e-005	1.60e-005	1.79e-005
4	7.93e-006	9.71e-006	1.28e-005	1.66e-005	1.83e-005
8	8.22e-006	1.02e-005	1.36e-005	1.82e-005	1.98e-005
16	8.54e-006	1.08e-005	1.50e-005	2.08e-005	2.15e-005
32	8.66e-006	1.09e-005	1.85e-005	2.06e-005	2.18e-005
64	1.25e-005	1.75e-005	3.51e-005	5.60e-005	9.58e-005
128	1.31e-005	1.90e-005	3.36e-005	5.99e-005	2.81e-004
256	1.54e-005	2.32e-005	3.72e-005	6.55e-005	1.12e-004
512	2.04e-005	2.97e-005	4.31e-005	7.88e-005	1.34e-004
1024	3.45e-005	5.07e-005	6.36e-005	1.18e-004	1.96e-004
2048	5.61e-005	8.44e-005	1.03e-004	1.78e-004	2.83e-004
4096	9.94e-005	1.24e-004	1.69e-004	2.81e-004	4.56e-004
8192	1.94e-004	2.41e-004	2.75e-004	4.84e-004	8.38e-004
16384	3.25e-004	3.94e-004	5.29e-004	9.62e-004	1.56e-003
32768	5.87e-004	7.43e-004	9.83e-004	1.89e-003	2.99e-003

Table A.2: Sun Fire 15K system measured average (over 1000 repetitions) communication time (seconds) of a broadcast collective communication.

Words	p=2	p=4	p=8	p=16	p=32
1	9.82	7.65	8.78	9.39	9.15
2	19.57	15.38	17.40	18.80	18.28
4	37.81	31.58	32.69	35.83	35.08
8	70.84	60.41	63.01	65.96	65.60
16	131.36	125.69	113.91	119.26	121.98
32	245.88	243.82	213.89	229.38	241.51
64	437.71	374.27	308.16	382.60	410.53
128	817.67	756.10	567.03	714.72	772.61
256	1358.71	1169.68	869.15	1156.58	1193.09
512	1879.76	1803.62	1141.83	1508.03	1622.19
1024	2735.63	2637.55	1473.28	2363.49	2356.90
2048	3614.69	3366.34	1859.93	3140.29	3214.99
4096	4226.61	4018.02	2020.18	3631.15	3736.08
8192	4763.18	4501.73	2161.78	4096.04	4147.46
16384	5074.42	4708.64	2238.52	4472.72	4252.05
32768	5506.86	4840.34	2743.37	4603.77	4593.39

Table A.3: Sun Fire 15K system measured bandwidth (megabits/second) of a round-trip point-to-point communication.

Words	p=2	p=4	p=8	p=16	p=32
1	6.52e-006	8.37e-006	7.29e-006	6.81e-006	6.99e-006
2	6.54e-006	8.32e-006	7.36e-006	6.81e-006	7.00e-006
4	6.77e-006	8.11e-006	7.83e-006	7.15e-006	7.30e-006
8	7.23e-006	8.48e-006	8.13e-006	7.76e-006	7.80e-006
16	7.80e-006	8.15e-006	8.99e-006	8.59e-006	8.39e-006
32	8.33e-006	8.40e-006	9.58e-006	8.93e-006	8.48e-006
64	9.36e-006	1.09e-005	1.33e-005	1.07e-005	9.98e-006
128	1.00e-005	1.08e-005	1.45e-005	1.15e-005	1.06e-005
256	1.21e-005	1.40e-005	1.89e-005	1.42e-005	1.37e-005
512	1.74e-005	1.82e-005	2.87e-005	2.17e-005	2.02e-005
1024	2.40e-005	2.49e-005	4.45e-005	2.77e-005	2.78e-005
2048	3.63e-005	3.89e-005	7.05e-005	4.17e-005	4.08e-005
4096	6.20e-005	6.52e-005	1.30e-004	7.22e-005	7.02e-005
8192	1.10e-004	1.16e-004	2.43e-004	1.28e-004	1.26e-004
16384	2.07e-004	2.23e-004	4.68e-004	2.34e-004	2.47e-004
32768	3.81e-004	4.33e-004	7.64e-004	4.56e-004	4.57e-004

Table A.4: Sun Fire 15K system measured average (over 1000 repetitions) communication time (seconds) of a round-trip point-to-point communication.

References

[1] AKIN, J. E. *Application and Implementation of Finite Element Methods.* Academic Press, 1982.

[2] AKIN, J. E. *Finite Element for Analysis and Design.* Academic Press, 1994.

[3] Altair Engineering Inc. www.altair.com, Last accessed June 2005.

[4] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. *LAPACK User's Guide*, third ed. SIAM Publications, Philadelphia, 1999.

[5] ANSYS Inc. Corporate. www.ansys.com, Last accessed June 2005.

[6] ATLURI, S. N., AND SHEN, S. The basis of meshless domain discretization: the meshless local Petrov-Galerkin (MLPG) method. *Advances in Computational Mathematics 23* (2005), 73–93.

[7] AutoForm Engineering GmbH. www.autoform.ch, Last accessed June 2005.

[8] AXELSSON, O. Iteration number for the conjugate gradient method. *Mathematics and Computers in Simulation 61* (2003), 421–435.

[9] BABUŠKA, I., BANERJEE, U., AND OSBORN, J. E. Survey of meshless and generalized finite element methods: A unified approach. *Acta Numerica 12*, 0 (2003), 1–125.

[10] BANK, R. E., AND HOLST, M. A new paradigm for parallel adaptive meshing algorithms. *SIAM Review 45*, 2 (2003), 291–323.

[11] BANK, R. E., SHERMAN, A. H., AND WEISER, A. Refinement algorithms and data structures for regular local mesh refinement. In *Scientific Computing* (Amsterdan, 1983), R. S. *et al.* , Ed., IMACS, pp. 3–17.

[12] BARRET, R., BERRY, M., CHAN, T., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND VAN DER VORST, H. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM Publications, Philadelphia, 1993.

[13] BEAUWENS, R. Iterative solution methods. *Applied Numerical Mathematics 51* (2004), 437–450.

[14] BECKER, D. *O Método Iterativo GMRES em Blocos Dinâmicos para a Solução de Sistemas Lineares com Múltiplos Termos Independentes em Computadores Paralelos.* MSc Dissertation, Universidade Federal do Rio Grande do Sul, Programa de Pós Graduação em Matemática Aplicada, June 2002.

[15] BECKER, D., AND THOMPSON, C. P. A novel, parallel PDE solver for unstructured grids. In *5th International Conference on Large-Scale Scientific Computations* (2005). (to appear - accepted).

[16] BENZI, M. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics 182* (2002), 418–477.

[17] BRAUNL, T. Braunl's law. The Open Channel, 1991. University of Stuttgart.

[18] BREZIS, H., AND BROWDER, F. Partial differential equations in the 20th century. *Advances in Mathematics 135*, 1 (1998), 76–144.

[19] BRUNER, C. W. S. *Parallelization of the Euler Equations on Unstructured Grids.* PhD Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, May 1996.

[20] BÜCKER, M., AND BASERMANN, A. A comparison of QMR, CGS and TFQMR on a distributed memory machine. FORSCHUNGSZENTRUM JÜLICH GmbH - KFA, Zentralinstitut für Angewandte Mathematik, Interner Bericht, KFA-ZAM-IB-9412, May 1994.

[21] CASTAÑOS, J. G., AND SAVAGE, J. E. The dynamic adaptation of parallel mesh-based computation. In *SIAM 7th Symposium on Parallel and Scientific Computation* (1997):

[22] CASTAÑOS, J. G., AND SAVAGE, J. E. Parallel refinement of unstructured meshes. In *Proceedings of the IASTED International Conference in Parallel and Distributed Computing and Systems* (MIT, Boston, USA, Nov. 1999).

[23] CASTAÑOS, J. G., AND SAVAGE, J. E. PARED: a framework for the adaptive solution of PDEs. In *8th IEEE Symposium on High Performance Distributed Computing* (1999).

[24] CASTAÑOS, J. G., AND SAVAGE, J. E. Repartitioning unstructured adaptive meshes. In *Proc. Intl. Parallel and Distributed Processing Symposium* (2000).

[25] CHAN, T. F., AND MATHEW, T. P. Domain decomposition algorithms. *Acta Numerica* (1994), 61–143.

[26] CHAN, T. F., AND SMITH, B. F. Domain decomposition and multigrid algorithms for elliptic problems on unstructured meshes. *Electronic Transactions on Numerical Analysis 2* (December 1994), 171–182.

[27] CHAND, K. K. Component-based hybrid mesh generation. *International Journal for Numerical Methods in Engineering 62* (2005), 747–773.

[28] CHEN, X., AND SUN, J. Global convergence of a two-parameter family of conjugate gradient methods without line seach. *Journal of Computational and Applied Mathematics 146* (2002), 37–45.

[29] CHRISOCHOIDES, N. A survey of parallel mesh generation methods. To appear in Numerical Solution of Partial Differential Equations on Parallel Computers (eds. Are Magnus Bruaset, Petter Bjorstad, Aslak Tveito), 2005.

[30] CONCUS, P., GOLUB, G. H., AND P. O'LEAVY, D. A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. In *Matrix Computations* (1976), J. R. Bunch and D. J. Rose, Eds., Academic Press, pp. 309–332.

[31] DA CUNHA, R. D. *A study on iterative methods for the solution of systems of linear equations on transputer networks.* PhD Dissertation, University of Kent at Canterbury, Computing Laboratory, Kent, 1992.

[32] DA CUNHA, R. D. *Introdução à Linguagem de Programação Fortran 90.* Editora da UFRGS, Porto Alegre, 2005.

[33] DA CUNHA, R. D., BECKER, D., AND PATTERSON, J. C. New parallel (rank-revealing) QR factorization algorithms. In *Euro-Par* (2002), B. Monien and R. Feldmann, Eds., vol. 2400 of *Lecture Notes in Computer Science*, Springer-Verlag Inc., pp. 677–686.

[34] DA CUNHA, R. D., AND HOPKINS, T. PIM 2.2 The Parallel Iterative Methods Package for Systems of Linear Equations. User's Guide (Fortran 77 version).

[35] DA CUNHA, R. D., AND HOPKINS, T. The parallel iterative methods (PIM) package for the solution of systems of linear equations on parallel computers. *Applied Numerical Mathematics 19* (1995), 33-50.

[36] DAI, Y., LIAO, L., AND LI, D. On restart procedures for the conjugate gradient method. *Numerical Algorithms 35* (2004), 249-260.

[37] DOWD, K., AND SEVERANCE, C. *High Performance Computing.* O'Reilly & Associates Inc, Sebastopol, CA, USA, 1998.

[38] DUFF, I. S., AND VAN DER VORST, H. A. Developments and trends in the parallel solution of linear systems. *Parallel Computing 25*, 13-14 (1999), 1931–1970.

[39] EAGER, D. L., ZAHORJAN, J., AND LAZOWSKA, E. D. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers 38*, 3 (Mar. 1989), 408–423.

[40] EL-REWINI, H., AND ABD-EL-BARR, M. *Advanced Computer Architecture and Parallel Processing.* John Wiley & Sons, 2005. http://www3.interscience.wiley.com, Last accessed December 2005.

[41] FARRIS, C., AND MISRA, M. Distributed algebraic multigrid for finite element computations. *Mathl. Comput. Modeling 27*, 8 (1998), 41-67.

[42] FERZIGER, J. H., AND PERIĆ, M. *Computational Methods for Fluid Dynamics*, 3rd, rev. ed. Springer-Verlag Inc., Germany, 2002.

[43] FISCHER, B., RAMAGE, A., SILVESTER, D., AND WATHEN, A. On parameter choice and iterative convergence for stabilised discretisations of advection-diffusion problems. *Computer Methods in Applied Mechanics and Engineering*, 179 (1999), 179–195.

[44] FLETCHER, C. A. J. *Computational Techniques for Fluid Dynamics, Fundamental and General Techniques*, vol. 1. Springer Series in Computational Physics, 1988.

[45] FLETCHER, R., AND REEVES, C. M. Function minimization by conjugate gradients. *The Computer Journal 7* (1964), 149–154.

[46] Fluent Inc. www.fluent.com, Last accessed June 2005.

[47] FOKKEMA, D. R., SLEIJPEN, G. L. G., AND VAN DER VORST, H. A. Generalized Conjugate Gradient Squared. Tech. Rep. 851, Mathematical Institute, Utrecht University, 1994.

[48] GOLUB, G. H., AND VAN LOAN, C. E. *Matrix computations*, second ed. Academic Press, New York, 1989.

[49] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations*, third ed. The Johns Hopkins University Press, Baltimore, MD, 1996.

[50] GRESHO, P. M., AND SANI, R. L. *Incompressible Flow and the Finite Element Method, Volume 1: Advection-Diffusion.* John Wiley & Sons, 1998.

[51] GRESHO, P. M., AND SANI, R. L. *Incompressible Flow and the Finite Element Method Volume 2: Isothermal Laminar Flow.* John Wiley & Sons, 1998.

[52] Gridgen, Pointwise, Inc. www.pointwise.com, Last accessed June 2005.

[53] GROPP, W., LUSK, E., AND SKJELLUM, A. *Using MPI - Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series, USA, 1994.

[54] GROPP, W., AND SMITH, B. *Simplified Linear Equation Solvers Users Manual.* Argonne National Laboratory, June 1993. ANL-93/8-REV 1.

[55] HAYES, J. P. *Computer Architecture and Organization*, second ed. McGraw-Hill, USA, 1988.

[56] HENDRICKSON, B. CHACO home page. http://www.cs.sandia.gov/~bahendr/chaco.html, Last accessed July 2005.

[57] HENDRICKSON, B., AND LELAND, R. *The Chaco User's Guide: Version 2.0*, 1994. Sandia Tech Report SAND94-2692.

[58] HESTENES, M. R., AND STIEFEL, E. Methods of conjugate gradient for solving linear systems. *Journal of Research of the National Bureau of Standards 49* (1952), 409–436.

[59] HOLTER, W. H., NAVON, I. M., AND OPPE, T. C. Parallelizable preconditioned Conjugate Gradient methods for the Cray Y-MP and the TMC CM-2. Tech. rep., Supercomputer Computations Reserach Institute, Florida State University, Dec. 1991.

[60] HUANG, C. Y. AND ODEN, J. T. GAMMA2D: a multiregion/multiblock, structured/unstructured grid generator package for computational mechanics. *Computers & Structures 53*, 2 (1994), 375–410.

[61] HUANG, CHUNG-YUAN. Recent progress in multiblock hybrid structured and unstructured mesh generation. *Computer Methods in Applied Mechanics and Engineering 150* (1997), 1–24.

[62] JONES, M. T., AND PLASSMANN, P. E. Adaptive refinement of unstructured finite-element meshes. *Finite Elem. Anal. Des. 25*, 1-2 (1997), 41–60.

[63] JONES, M. T., AND PLASSMANN, P. E. Parallel algorithms for adaptive mesh refinement. *SIAM Journal on Scientific Computing 18*, 3 (1997), 686–708.

[64] KARYPIS, G. METIS home page. www-users.cs.umn.edu/~karypis/metis, Last accessed March 2004.

[65] KARYPIS, G., AND KUMAR, V. *METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* University of Minnesota, Sept. 1998. Version 4.0.

[66] KARYPIS, G., SCHLOEGEL, K., AND KUMAR, V. *PARMETIS Parallel Graph Partitioning and Sparse Matrix Ordering Library.* University of Minnesota, Mar. 2002. Version 3.0.

[67] KEYES, D. E. How scalable is domain decomposition in practice? In *Eleventh International Conference on Domain Decomposition Methods* (Bergen, 1999), C.-H. Lai, P. E. Bjørstad, M. Cross, and O. B. Widlund, Eds., Domain Decomposition Press, pp. 286–297.

[68] KNUPP, P., AND STEINBERG, S. *Fundamentals of grid generation.* CRC Press, 1994.

[69] KOELBEL, C. H., LOVEMAN, D. B., SCHREIBER, R. S., STEELE JR, G. L., AND ZOSEL, M. E. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[70] LANCZOS, C. Solution of sysems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards 49* (1952), 33–53.

[71] LISEĬKIN, V. D. *Grid Generation Methods.* Springer-Verlag Inc., Germany, 1999.

[72] LÖHNER, R. *Applied CFD Techniques An Introduction based on Finite Elements Methods.* John Wiley & Sons, England, 2001.

[73] LÖHNER, R., SHAROV, D., LUO, H., AND RAMAMURTI, R. Overlapping unstructured grids. AIAA Paper 01-0439, 2001.

[74] MATHIEU, J., AND SCOTT, J. *An Introduction to Turbulent Flow.* Cambridge University Press, 2000.

[75] The MathWorks. http://www.mathworks.com Last accessed March 2006.

[76] MAVRIPLIS, D. J. An assesment of linear versus nonlinear multigrid methods for unstructured mesh solvers. *Journal of Computational Physics 175*, 1 (Jan. 2002), 302–325.

[77] MITCHELL, W. *Unified Multilevel Adaptive Finite Element Methods for Elliptic Problems.* PhD Dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1988.

[78] MITCHELL, W. F. A comparison of adaptive refinement techniques for elliptic problems. *ACM Transactions on Mathematical Software 15*, 4 (1989), 326–347.

[79] MODI, A. Direct numerical simulation of turbulent flows, 1999.

[80] MODI, J. J. *Parallel Algorithms and Matrix Computation.* Oxford University Press, New York, 1988.

[81] MPI. The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi, Last accessed June 2006.

[82] MSC.Software Corporation. www.mscsoftware.com, Last accessed June 2005.

[83] MURTY, B. S. N., AND HUSAIN, A. Orthogonality correction in the conjugate-gradient method. *Journal of Computational and Applied Mathematics 9* (1983), 299–304.

[84] NOTAY, Y. Flexible conjugate gradient. *SIAM Journal on Scientific Computing 22*, 4 (2000), 1444–1460.

[85] OLIKER, L., AND BISWAS, R. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing 52* (1998), 150–177.

[86] OLIKER, L., BISWAS, R., AND GABOW, H. N. Parallel tetrahedral mesh adaptation with dynamic load balancing. *Parallel Computing* (2000), 1583–1608. Special Issue on Graph Partitioning.

[87] OpenMP. www.openmp.org, Last accessed January 2006.

[88] ORKISZ, J. Finite difference method. In *Handbook of Computational Solid Mechanics* (Berlin, 1998), Kleiber, Ed., Springer-Verlag Inc.

[89] OWEN, S. Meshing Research Corner. www.andrew.cmu.edu/user/sowen/mesh.html, Last accessed June 2005.

[90] OWEN, S. J. A survey of unstructured mesh generation technology. In *7th International Meshing Roundtable* (Dearborn, MI, Oct. 1998), Sandia National Laboratories.

[91] PÉBAY, P. P., AND THOMPSON, D. C. Parallel mesh refinement without communication. In *13th International Meshing Roundtable* (Williamsburg,

VA, Sept. 2004), Sandia National Laboratories, SAND2004-3765C, pp. 437–448.

[92] PEPPER, D., AND HEINRICH, J. C. *The Finite Element Method: Basic Concept and Applications.* Series in Computational and Physical Processes in Mechanics and Thermal Sciences, London, 1992.

[93] Posix. http://posixcertified.ieee.org/, Last accessed January 2006.

[94] PREIS, R., AND DIEKMANN, R. *The PARTY Partitioning-Library, User Guide Version 1.1.* University of Paderborn, Sept. 1996. Technical Report tr-rsfb-96-024.

[95] REDDY, J. N. *An Introduction to the Finite Element Method.* McGraw-Hill, 1984.

[96] REID, J. K. On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations. In *Large Sparse Sets of Linear Equations* (New York, 1971), J. K. Reid, Ed., Academic Press, pp. 231–254.

[97] REUTER, H. *Investigation into the scalability of a Sun Fire 15K.* MSc Dissertation, Cranfield University, Applied Mathematics and Computing Group, Cranfield, UK, 2003.

[98] RIVARA, M. C. Mesh refinement processes based on the generalized bisection simplices. *SIAM Journal on Numerical Analysis 21* (1984), 604–613.

[99] SAAD, Y. A flexible inner-outer preconditioned GMRES algorithm. Tech. Rep. umsi-91-279, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, Minnesota, 1991. Appeared in SISSC, vol.4, 1993 [100].

[100] SAAD, Y. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Stat. Compt. 14* (1993), 461–469.

[101] SAAD, Y. *Iterative methods for sparse linear systems.* PWS Publishing Company, Boston, 1995.

[102] SAAD, Y., AND SCHULTZ, M. H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific Computing 7*, 3 (July 1986), 856–869.

[103] SAAD, Y., AND VAN DER VORST, H. A. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123 (2000), 1–33.

[104] SCHNEIDERS, R. Mesh generation & grid generation on the web. http://www-users.informatik.rwth-aachen.de/ roberts/meshgeneration.html, Last accessed June 2006.

[105] SEGERLIND, L. J. *Applied Finite Element Analysis*, 2nd ed. John Wiley & Sons, 1984.

[106] SEWELL, E. G. A finite element program with automatic user-controlled mesh grading. In *Advances in Computer Methods for Partial Differential Equations III* (New Brunswick, 1979), R. Stepleman, Ed., IMACS, pp. 8–10.

[107] SHERWIN, S. J., AND KARNIADAKIS, G. E. A new triangular and tetrahedral basis for high-order (*hp*) finite element methods. *International Journal for Numerical Methods in Engineering 38* (1995), 3775–3802.

[108] SHERWIN, S. J., AND KARNIADAKIS, G. E. Tetrahedral (*hp*) finite elements: Algorithms and flow simulations. *Journal of Computational Physics 124* (1996), 14–45.

[109] SHEWCHUK, J. R. An introduction to the conjugate gradient method without the agonizing pain. www-2.cs.cmu.edu/~jrs/jrspapers.html, Last accessed March 2004, 1994.

[110] SHI, Y. Reevaluating Amdahl's Law and Gustafson's Law, Oct. 1996. http://www.cis.temple.edu/ shi/docs/amdahl/amdahl.html, Last accessed January 2006.

[111] SMITH, B. F., BJØRSTAD, P. E., AND GROPP, W. D. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, Cambridge, United Kingdom, 1996.

[112] SONNEVELD, P. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Stat. Compt. 10*, 1 (Jan. 1989), 36–52.

[113] STÜBEN, K. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics 128* (2001), 281–309.

[114] SUN MICROSYSTEMS, I. *Sun Fire 15K System Overview*, Nov. 2001. Sun Documentation, Revision A, Part No. 806-3509-10 (V2).

[115] THOMASSET, F. *Implementation of Finite Element Methods for Navier-Stokes Equations.* Springer-Verlag Inc., 1981.

[116] THOMPSON, J. F., SONI, B. K., AND WEATHERILL, N. P. *Handbook of Grid Generation.* CRC Press, 1999. www.mathnetbase.com/ejournals/books/book_km. asp?id= 615, Last accessed June 2005.

[117] VAN DE VORST, H. A. Efficient and reliable iterative methods for linear systems. *Journal of Computational and Applied Mathematics 149* (202), 251–265.

[118] VAN DER VORST, H. A. Iterative methods for large linear systems, lectures notes. www.math.uu.nl/people/vorst/lecture.html, Last accessed March 2004.

[119] VAN DER VORST, H. A. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Stat. Compt. 13*, 2 (Mar. 1992), 631–644.

[120] VARGA, R. S. *Matrix Iterative Analysis.* Prentice-Hal, Inc, 1962.

[121] VERSTEEG, H. K., AND MALALASEKERA, W. *An introduction to computational fluid dynamics: the finite volume method.* Harlow: Longman Scientific and Technical, 1995.

[122] VOYAGES, K. M., AND NIKITOPOULOS, D. E. An accurate, flexible, parallel Poisson solver with direct substructuring for complex geometries. AIAA 2000-0274, 2000.

[123] WALSHAW, C. JOSTLE home page. www.gre.ac.uk/~c.walshaw/jostle.

[124] WALTZ, J. Parallel adaptive unstructured finite element schemes for 3D compressible and incompressible flows. AIAA 2002-2978, 2002.

[125] WEISSINGER, J. *Development of a Discrete Adaptive Gridless Method for the Solution of Elliptic Partial Differential Equations.* PhD Dissertation, Cranfield University, School of Engineering, Applied Mathematics and Computing Group, Cranfield, UK, July 2003.

[126] WESSELING, P., AND OOSTERLEE, C. W. Geometric multigrid with applications to computational fluid dynamics. *Journal of Computational and Applied Mathematics 128* (2001), 311–334.

[127] Xu, B. Methods for PDE-constrained optimisation with equality geometric constrains. Personal communications - second review report.

[128] Young, D. M. *Iterative Solution of Large Linear Systems.* Academic Press, 1971.

[129] Young Jr., D. M. *Iterative Methods for Solving Partial Difference Equations of Elliptic Type.* PhD Dissertation, Harvard University, Cambridge, Mass, May 1950.

[130] Zienkiewicz, O. C., and Taylor, R. L. *The Finite Element Method,* 5th ed. Elsevier Butterworth-Heinemann, Oxford, 2000.

[131] Zienkiewicz, O. C., and Taylor, R. L. *The Finite Element Method: the Basis,* 5th ed., vol. 1. Elsevier Butterworth-Heinemann, Oxford, 2000.