Parallel Domain Decomposition Preconditioning for the Adaptive Finite Element Solution of Elliptic Problems in Three Dimensions

by

Sarfraz Ahmad Nadeem

M.Sc., M.Phil.

Submitted in accordance with the requirements for the degree of Doctor of Philosophy.

The University of Leeds
School of Computing

August 2001

The candidate confirms that the work submitted is his own and the appropriate credit has been given where reference has been made to the work of others.

Abstract

A novel *weakly overlapping* two level additive Schwarz domain decomposition preconditioning algorithm is presented which is appropriate for the parallel finite element solution of elliptic partial differential equations in three dimensions. This algorithm allows each processor to be assigned one or more subdomains and, as with most parallel domain decomposition solvers, each processor is able to solve its own subproblem(s) concurrently.

The novel feature of the technique proposed here is that it requires just a single layer of overlap in the elements which make up each subdomain at each level of refinement, and it is shown that this amount of overlap is sufficient to yield an optimal preconditioner. The number of elements in this overlap region between subdomains is $O(h^{-2})$ as the mesh size $h \to 0$. This is an improvement over the $O(h^{-3})$ overlapping elements required to obtain optimality for a conventional two level additive Schwarz algorithm. The quality and effectiveness of this new algorithm is examined using both global uniform and local non–uniform refinement with two representative partitions of the domain Ω.

This preconditioning algorithm is then generalized such that the resulting preconditioner is not only suitable for symmetric problems but also for nonsymmetric and convection–dominated elliptic problems. This generalization, in the absence of theoretical or mathematical background, is based on empirical observations. Moreover, it turns out to be more effective and robust than the original symmetric preconditioning algorithm when applied to symmetric positive definite problems. This generalized algorithm is tested on symmetric, nonsymmetric and convection–dominated partial differential equations, where the number of iterations observed suggests that the preconditioner may in fact be optimal, i.e. the condition number of the preconditioned systems is bounded as the mesh is refined or the number of subdomains is increased. Due to non–physical oscillations in the solution of convection–dominated problems when discretized by the Galerkin finite element method, unless the size of elements is sufficiently small, we have extended our implementation of the generalized preconditioning algorithm to be applicable to systems arising from a more stable finite element discretization technique based upon streamline diffusion. Numerical experiments for a wide variety of problems are included to demonstrate the optimal or near–optimal behaviour and quality of this generalized algorithm.

Parallel performance of the generalized preconditioning algorithm is also evaluated and analyzed. All the timings quoted are for a SG Origin 2000 computer and all software implementations described in this thesis have been coded and tested using ANSI C and the MPI communication library.

Acknowledgements

First of all, I would like to thank The Almighty for granting me the much needed strength, energy, courage and will for this study.

My deepest gratitude to my supervisor, Dr. Peter K. Jimack, for the continuous support, ideas and encouragement he has provided throughout my work on this thesis. His unlimited availability and guidance, along with his wisdom, will be warmly remembered forever. It is also a great pleasure and opportunity to thank him for correcting my poorly drafted chapters of this thesis.

Thanks to my family and friends, both in Pakistan and UK, in particular my mother, *Naseem Akhtar* who kept on asking me *when you are going to finish* and praying for my success. I'm also in debt to my wife *Naheed* and daughter *Alveenah* (who was born in the middle of this study) for allowing me the time towards this study which I was supposed to spare for them. I thank all of them for their love and support.

My gratitude goes to former and present members of the Computational PDE Unit for providing an excellent research environment, helpful discussions and caring assistance.

My thanks go to the Support and General Office staff of the School of Computing who were always happy and ready to help me.

This thesis would not have been possible without the financial support of the Government of Pakistan in the form of a Quaid-e-Azam scholarship.

Declarations

Some parts of the work presented in this thesis have been published or submitted for publication in the following articles:

[105] **Jimack, P. K. and Nadeem, S. A.** A weakly overlapping parallel domain decomposition preconditioner for the finite element solution of elliptic problems in three dimensions. In Arabnia, H. R., editor, *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume III, pages 1517–1523. CSREA Press, USA, 2000.

[106] **Jimack, P. K. and Nadeem, S. A.** Parallel application of a novel domain decomposition preconditioner for the stable finite element solution of three–dimensional convection–dominated PDEs. In Sakellariou, R. *et al.*, editor, *Euro–Par 2001 Parallel Processing*, Lecture Notes in Computer Science 2150, pages 592–601. Springer, 2001.

[107] **Jimack, P. K. and Nadeem, S. A.** A weakly overlapping parallel domain decomposition preconditioner for the finite element solution of convection–dominated problems in three dimensions. In *Proceedings of International Parallel CFD 2001 Conference*, Egmond ann Zee, The Netherlands, 21–23 May 2001. To appear.

[10] **Bank, R. E., Jimack, P. K., Nadeem, S. A., and Nepomnyaschikh, S. V.** A weakly overlapping domain decomposition preconditioner for the finite element solution of elliptic partial differential equations. *SIAM Journal of Scientific Computing*, 2001. To appear.

[130] **Nadeem, S. A. and Jimack, P. K.** Parallel implementation of an optimal two level additive Schwarz preconditioner for the 3-d finite element solution of elliptic partial differential equations. *Int. J. Num. Meth. Fluids*, 2001. Submitted.

Contents

1 Introduction **1**

1.1 Finite Element Method 1

1.1.1 Galerkin Finite Element Method 2

1.1.1.1 Piecewise Linear Finite Elements 4

1.1.1.2 Higher Order Finite Elements 6

1.1.2 Petrov–Galerkin Finite Element Method 8

1.2 Mesh Adaptivity 10

1.2.1 h–Adaptivity 11

1.2.2 r–Adaptivity 12

1.2.3 p–Adaptivity 12

1.3 TETRAD 13

1.3.1 Description 13

1.3.2 Data structure 14

1.3.2.1 Data Objects 14

1.3.2.2 Further Details 15

1.3.3 Adaption Algorithm 16

1.3.4 Adaption Control 19

1.4 Parallel Computer Architecture 20

1.4.1 Background 20

1.4.2 The Classical von Neumann Machine 21

1.4.3 Pipeline and Vector Architecture 22

1.4.4 SIMD Systems 22

1.4.5 General MIMD Systems 22

1.4.5.1 Shared Memory MIMD 23
1.4.5.2 Distributed Memory MIMD 23
1.5 Message Passing Interface 25
1.5.1 Startup 25
1.5.2 Point to Point Communication 26
1.5.3 Collective Communication 27
1.5.4 Some Miscellaneous Functions 28
1.6 Parallel Finite Element Method 29
1.6.1 Parallel Sparse Matrix Assembly 29
1.6.2 Direct Solvers 31
1.6.2.1 General Methods 32
1.6.2.2 Frontal Methods 32
1.6.2.3 Multifrontal Methods 33
1.6.3 Iterative Solvers 34
1.6.3.1 Stationary Methods 34
1.6.3.2 Nonstationary Methods 35
1.6.4 Parallel Solution 37
1.6.4.1 Matrix–Vector Product 38
1.6.4.2 Inner Product 39
1.7 Preconditioning 40
1.8 Contents of Thesis 46

2 Domain Decomposition **49**

2.1 Domain Decomposition Solution Methods 49
2.2 Schwarz Alternating Methods 51
2.2.1 Multiplicative Schwarz Procedure 52
2.2.2 Additive Schwarz Procedure 54
2.3 Schur Complement Methods 56
2.3.1 Direct Substructuring Methods 57
2.3.2 Iterative Substructuring Methods 60
2.3.3 Finite Element Tearing and Interconnecting Method 61

2.4 Domain Decomposition: A Brief Review 65
2.5 Mesh Partitioning 68
2.6 Graph Partitioning 70
2.6.1 Geometric Techniques 71
2.6.2 Spectral Techniques 72
2.6.3 Graphical Techniques 73
2.6.4 Some Miscellaneous Techniques 74
2.6.5 Graph Partitioning Software 75

3 A Symmetric Weakly Overlapping Additive Schwarz Preconditioner 78

3.1 Introduction 78
3.2 Theory 80
3.3 The Preconditioner 84
3.4 Implementation 89
3.5 Restriction 95
3.5.1 Setup Phase 96
3.5.1.1 Data Collection 96
3.5.1.2 Matching of Coarse and Fine Meshes 99
3.5.2 Iteration Phase 103
3.5.2.1 Coarsening 104
3.5.2.2 How to Treat Green Nodes 106
3.6 Preconditioning Solve 107
3.6.1 Choice of Best Sequential Solver 108
3.7 Interpolation 110
3.7.1 Setup Phase 110
3.7.2 Iteration Phase 111
3.7.2.1 Prolongation 112
3.8 Computational Results 114
3.8.1 Uniform Refinement 115
3.8.2 Non–Uniform Refinement 117
3.9 Discussion 119

4 A Generalized Nonsymmetric Additive Schwarz Preconditioner 121

4.1 Introduction 122

4.2 The Preconditioner 124

4.3 Implementation 125

4.4 Application to Symmetric Problems 128

4.4.1 Uniform Refinement 128

4.4.2 Non–Uniform Refinement 131

4.4.3 Generalized Preconditioner: Pros and Cons 132

4.5 Application to Nonsymmetric Problems 133

4.6 Convection–Dominated Problems 136

4.6.1 Computational Results 138

4.7 Streamline–Diffusion Method 142

4.7.1 Computational Results 144

4.8 Local Adaptivity 147

4.9 Discussion 149

5 Parallel Performance 152

5.1 Assessment of Parallel Performance 153

5.1.1 Decomposition 154

5.1.2 Communication 155

5.1.3 Load Balancing 157

5.1.4 Parallel Overhead 160

5.2 Sample Execution 161

5.3 Discussion 170

6 Conclusion and Future Directions 173

6.1 Summary 173

6.2 Future Directions 174

List of Figures

1.1 Triangular elements (in 2–d) of various degrees. 7

1.2 Tetrahedral elements (in 3–d) of various degrees. 7

1.3 Orphan edges (dashed lines) produced by refinement. 16

1.4 Regular refinement of one tetrahedron into 8 tetrahedra. 17

1.5 Green refinement of one tetrahedron into 6 tetrahedra. 18

1.6 Knock–on effect of green refinement (two dimensional example). . . . 19

1.7 Partition of a computational domain into 16 subdomains. 30

2.1 Decomposition into overlapping and non–overlapping subdomains. . . 50

2.2 Schwarz's original figure. 66

3.1 Uniform (top) and Hierarchical (bottom) refinement in the overlap region. 85

3.2 Refinement without (top) and with (bottom) transition *green* elements. 91

3.3 Uniform refinement of subdomain Ω_i owned by process i. 94

3.4 Data sets sent by process $i = 0$ to every other process j. 97

3.5 Data sets received by process $i = 0$ from every other process j. 99

3.6 Refinement tree: Refinement levels between 0 and 3. 102

3.7 Coarsening of a residual vector. 106

3.8 Prolongation of a solution vector. 113

3.9 Domain partitioning strategies for 2, 4, 8 and 16 subdomains: recursive coordinate bisection (RCB) partitions (left) and anisotropic partitions (right). 114

4.1 Solution plot for u given by (4.17) and $\varepsilon = 0.01$. 138
4.2 Analogous solution plot for a two–dimensional generalization of Figure 4.1. 139
4.3 Solution plot when u is a function of x only, convection is dominating along x–axis and $\varepsilon = 0.01$. 148

5.1 Communication required by a typical processor for our implementation of the two level additive Schwarz preconditioner (left), and the communication required by the same processor when the coarse grid solve is undertaken on a single processor (right). 156
5.2 Global uniform refinement of three representative subdomains in an isotropic partition. 157
5.3 Two representative subdomains and their neighbouring subdomains for the RCB partition of $\Omega = (0, 2) \times (0, 1) \times (0, 1)$ into sixteen subdomains. 158

List of Tables

3.1 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 1 using the RCB partitioning strategy. 116
3.2 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 1 using the anisotropic partitioning strategy. 116
3.3 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 2 using the RCB partitioning strategy. 117
3.4 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 2 using the anisotropic partitioning strategy. 117
3.5 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 3 using the RCB partitioning strategy. 118

3.6 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 3 using the anisotropic partitioning strategy. 118

4.1 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 1 using the RCB partitioning strategy. 129

4.2 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 1 using the anisotropic partitioning strategy. 129

4.3 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 2 using the RCB partitioning strategy. 130

4.4 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 2 using the anisotropic partitioning strategy. 130

4.5 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 3 using the RCB partitioning strategy. 131

4.6 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 3 using the anisotropic partitioning strategy. 132

4.7 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 4 using the RCB partitioning strategy. 135

4.8 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 4 using the anisotropic partitioning strategy. 135

4.9 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 5 using the RCB partitioning strategy. 136

4.10 The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 5 using the anisotropic partitioning strategy. 136

4.11 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$. 139

4.12 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$. 140

4.13 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$. 140

4.14 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$. 141

4.15 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$. 141

4.16 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$. 142

4.17 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$. 144

4.18 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$. 145

4.19 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$. 145

4.20 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$. 146

4.21 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$. 146

4.22 The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$. 147

4.23 The number of GMRES iterations at different levels of local refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 7 (when $\varepsilon = 0.01$) using the anisotropic partitioning strategy. 149

5.1 The performance of the parallel solver for the Galerkin FE discretization when solving Test Problem 1 using the RCB partition. The times are quoted in seconds and the speedups are relative to the best sequential solution time. 164

5.2 The performance of the parallel solver for the Galerkin FE discretization when solving Test Problem 4 using the RCB partition. The times are quoted in seconds and the speedups are relative to the best sequential solution time. 165

5.3 The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the RCB partition and $\varepsilon = 1.0 \times 10^{-2}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time. 166

5.4 The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the anisotropic partition and $\varepsilon = 1.0 \times 10^{-2}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time. 166

5.5 The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the RCB partition and $\varepsilon = 1.0 \times 10^{-3}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time. 167

5.6 The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the anisotropic partition and $\varepsilon = 1.0 \times 10^{-3}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time. 167

5.7 The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 7 using the anisotropic partition and $\varepsilon = 1.0 \times 10^{-2}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time. 169

Chapter 1

Introduction

The development of high speed computers during the second half of the last century has enormously increased the efficient utilization of numerical techniques for simulations and analysis of complicated systems in every field of science. As a result a new scientific field of *scientific computing* has emerged which is a complement to theoretical and experimental science. The exact analytical solution to mathematical models is possible in limited and simple cases only, whereas approximate solutions for most of the models of practical importance are determined using numerical techniques.

1.1 Finite Element Method

The Finite Element Method is one of the modern and well known techniques for the numerical solution of partial differential equations which model a variety of problems in almost all fields of science and engineering and is one of the most widely used techniques to approximate the numerical solution of partial differential equations. The mathematical study of Finite Element Methods (for example, [108]) establishes the roots of this technique in variational methods introduced at the beginning of last century. Today finite element methods are extensively used for a large variety of problems in most of the disciplines of science and engineering.

The basic idea in any numerical technique is to discretize the given continuous problem, such as partial differential equations or integral equations, to get the corre-

sponding discrete problem or system of equations with a finite number of unknowns that may be solved using a computer. In the finite element technique we may start the discretization procedure by reformulation of the given partial differential equation to get an equivalent variational problem. The solution of this variational problem is identical to that of the original problem in the form a of partial differential equation [46, 108, 136, 144, 165, 181].

To solve a given partial differential equation approximately using a finite element technique, the four basic steps suggested in [108] are:

1. obtain a variational, or weak, formulation of the given problem,
2. produce a finite element discretization,
3. solve the discrete problem,
4. combine steps 1–3 in a computer implementation of the finite element method.

In the following sections we discuss the finite element discretization of a general partial differential equation representing the elliptic class of problems.

1.1.1 Galerkin Finite Element Method

The finite element method is a piecewise application of a variational method, the study of which involves the basic steps described above. The term *variational formulation* means the weak formulation, which is a recast of the given partial differential equation to an equivalent integral form by trading the differentiation between the *test function* and the *dependent variables*. In this thesis we consider a general class of elliptic problems represented by the following partial differential equation (PDE) of the form

$$-\varepsilon \underline{\nabla} \cdot (A \underline{\nabla} u) + \underline{b} \cdot \underline{\nabla} u + c\, u = f \qquad \text{on} \qquad \Omega \subset \mathcal{R}^d \tag{1.1}$$

(where A is symmetric and strictly positive-definite, $\underline{b}$ may be $\underline{0}$ and $c \geq 0$) subject to well-posed boundary conditions: $u = u_E$ on Γ_1 , the Dirichlet (essential) boundary

condition, and $\hat{\underline{n}} \cdot (A\underline{\nabla} u) = g$ on Γ_2 , the Neumann (natural) boundary condition, such that $\partial\Omega = \Gamma_1 \cup \Gamma_2$, $\Gamma_1 \cap \Gamma_2 = \emptyset$ and $\hat{\underline{n}}$ is the outward unit vector normal to Γ_2.

The above equation (1.1) is a linear second order PDE which represents an elliptic class of PDE arising in large number in the fields of engineering and the physical sciences. The matrix A and the vector $\underline{b}$ will be referred to as the diffusion matrix and the convection vector respectively, and ε as the diffusion coefficient (here we assume $\|A\| \approx 1$). In order to solve this elliptic problem, we need its weak form which is obtained by multiplying this equation by a suitable *test function*, v say, and integrating over the entire domain Ω as follows:

$$-\varepsilon \int_\Omega \underline{\nabla} \cdot (A\underline{\nabla} u) v \; d\Omega + \int_\Omega (\underline{b} \cdot \underline{\nabla} u) v \; d\Omega + c \int_\Omega u \; v \; d\Omega = \int_\Omega f \; v \; d\Omega. \tag{1.2}$$

Application of the Divergence Theorem to the above equation (1.2) gives

$$\varepsilon \int_\Omega (\underline{\nabla} v \cdot (A\underline{\nabla} u)) \; d\Omega - \varepsilon \int_{\partial\Omega} ((A\underline{\nabla} u) \cdot \hat{\underline{n}}) v \; dS + \int_\Omega (\underline{b} \cdot \underline{\nabla} u) v \; d\Omega + c \int_\Omega u \; v \; d\Omega = \int_\Omega f \; v \; d\Omega. \tag{1.3}$$

If we take $v \in \mathcal{H}_0^1(\Omega)$, the space of all functions whose first derivatives are square integrable over Ω and zero everywhere on Γ_1, then the required weak solution of equation (1.1), u, must satisfy

$$\varepsilon \int_\Omega (\underline{\nabla} v \cdot (A\underline{\nabla} u)) \; d\Omega - \varepsilon \int_{\Gamma_2} g \; v \; dS + \int_\Omega (\underline{b} \cdot \underline{\nabla} u) v \; d\Omega + c \int_\Omega u \; v \; d\Omega = \int_\Omega f \; v \; d\Omega, \tag{1.4}$$

where

$$g = \hat{\underline{n}} \cdot (A\underline{\nabla} u) \quad \text{on} \quad \Gamma_2. \tag{1.5}$$

Let $\mathcal{H}_E^1(\Omega)$ be the space of all functions whose first derivatives are square integrable in the domain Ω and satisfy Dirichlet boundary condition $u = u_E$ on Γ_1. Then the resulting weak formulation of equation (1.1) may be written as follows:

Find $u \in \mathcal{H}_E^1$ such that

$$\varepsilon \int_\Omega (\underline{\nabla} v \cdot (A\underline{\nabla} u)) \; d\Omega - \varepsilon \int_{\Gamma_2} g \; v \; dS + \int_\Omega (\underline{b} \cdot \underline{\nabla} u) v \; d\Omega + c \int_\Omega u \; v \; d\Omega = \int_\Omega f \; v \; d\Omega \tag{1.6}$$

for all $v \in \mathcal{H}_0^1$.

1.1.1.1 Piecewise Linear Finite Elements

For the finite element approximation of the solution u in equation (1.6), the domain Ω of the problem must be decomposed into a finite number of small non–overlapping subdomains which are known as *elements*. Such a decomposition is always possible provided the boundary of the domain Ω is not curved, that is, it is a polygon (in 2–d) or polyhedron (in 3–d). However, it is possible to handle domains with curved boundaries [46, 108, 165]. A collection of such *elements*, which should cover the computational domain of the problem, is called a *mesh*. Assume that the nodes in the mesh (i.e. the vertices of the elements) are numbered from 0 to $N-1$ (note that C style numbering, starting from 0, will be used throughout this thesis). Thus there is a total of N nodes in the mesh with $N = N_I + N_E$ where N_I is the number of nodes in the interior of the domain Ω or on the Neumann boundary Γ_2 and N_E is the number of vertices on the Dirichlet boundary Γ_1. The node numbering in the interior of the domain Ω and at the Neumann boundary Γ_2 is assumed to be first, followed by the numbering of nodes at the Dirichlet boundary Γ_1. On each element the desired solution u of equation (1.6) will be approximated by a linear polynomial. This choice of linear polynomial is the most simple but higher degree polynomials can also be used in a similar way (see §1.1.1.2 for further details). Now we define piecewise linear basis functions $\Phi_j(\underline{x})$ at all the node points $j = 0, \ldots, N-1$ such that

$$\Phi_j(\underline{x}) = \begin{cases} 1 & \text{at node } j \quad \text{(i.e. when } \underline{x} \text{ is the position vector of node } j\text{)}, \\ 0 & \text{at node } k \quad \text{(i.e. when } \underline{x} \text{ is the position vector of node } k \neq j\text{)}. \end{cases} \tag{1.7}$$

In terms of these linear basis functions Φ_j we can approximate u by u_h given in the form of a linear combination

$$u_h = \sum_{j=0}^{N-1} u_j \Phi_j(\underline{x}), \tag{1.8}$$

where u_j are unknowns to be determined for $j = 0, \ldots, N_I - 1$ and are given by the Dirichlet boundary condition $u = u_E$ for $j = N_I, \ldots, N_I + N_E - 1$. By substituting

equation (1.8) and $v = \Phi_i(\underline{x})$ for $i = 0, \ldots, N_I - 1$ into equation (1.6) an algebraic system of N_I equations, corresponding to a discretization of the original elliptic problem, is obtained. This system of equations is given by

$$\varepsilon \sum_{j=0}^{N-1} u_j \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j))\ d\Omega - \varepsilon \int_{\Gamma_2} g\ \Phi_i\ dS + \sum_{j=0}^{N-1} u_j \int_\Omega (\underline{b} \cdot \underline{\nabla}\Phi_j)\Phi_i\ d\Omega + c \sum_{j=0}^{N-1} u_j \int_\Omega \Phi_j \Phi_i\ d\Omega = \int_\Omega f\Phi_i\ d\Omega. \tag{1.9}$$

for $i = 0, \ldots, N_I - 1$.

As an alternative, this system of equations can be written as

$$\begin{aligned}\sum_{j=0}^{N_I-1} u_j \left(\varepsilon \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j))\ d\Omega + \int_\Omega \Phi_i(\underline{b} \cdot \underline{\nabla}\Phi_j)\ d\Omega + \ c \int_\Omega \Phi_i\Phi_j\ d\Omega\right) = \\ - \sum_{j=N_I}^{N-1} u_j \left(\varepsilon \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j))\ d\Omega + \int_\Omega \Phi_i(\underline{b} \cdot \underline{\nabla}\Phi_j)\ d\Omega + \ c \int_\Omega \Phi_i\Phi_j\ d\Omega\right) \\ + \int_\Omega f\Phi_i\ d\Omega\ + \varepsilon \int_{\Gamma_2} g\ \Phi_i\ dS.\end{aligned} \tag{1.10}$$

for $i = 0, \ldots, N_I - 1$.

In matrix form this system of equations can be written as

$$\mathcal{A}\underline{u} = \underline{b} \tag{1.11}$$

where $\mathcal{A}$ is the global stiffness matrix of the form

$$\mathcal{A} = \varepsilon\mathcal{K} + \mathcal{C} + c\mathcal{S} \tag{1.12}$$

and $\underline{u} = (u_0, \ldots, u_{N_I-1})^T$.

Here $\mathcal{K}$, with entries

$$\mathcal{K}_{ij} = \varepsilon \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j))\ d\Omega\ , \tag{1.13}$$

and $\mathcal{S}$ with entries

$$\mathcal{S}_{ij} = \ c \int_\Omega \Phi_i\Phi_j\ d\Omega\ , \tag{1.14}$$

are symmetric positive definite and $\mathcal{C}$ with entries

$$\mathcal{C}_{ij} = \int_\Omega \Phi_i(\underline{b} \cdot \underline{\nabla}\Phi_j)\ d\Omega\ , \tag{1.15}$$

is skew–symmetric. Finally, entries of the right–hand side vector $\underline{b}$ are given by

$$b_i \;=\; -\sum_{j=N_I}^{N-1} u_j \left(\varepsilon \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j))\; d\Omega + \int_\Omega \Phi_i(\underline{b}\cdot\underline{\nabla}\Phi_j)\; d\Omega + \; c\int_\Omega \Phi_i\Phi_j\; d\Omega \right) + \int_\Omega f\Phi_i\; d\Omega \; + \varepsilon \int_{\Gamma_2} g\;\Phi_i\; dS\;, \quad (1.16)$$

for $i = 0, \ldots, N-1$. From the definition of our basis function, we note that the entries $\mathcal{A}_{ij}$ of the matrix $\mathcal{A}$ are always zero if the nodes i and j are not mutually connected through an edge in the mesh. This leads to a matrix with most of its entries equal to zero. Such a matrix is generally referred as *sparse matrix.*

1.1.1.2 Higher Order Finite Elements

In the previous subsection piecewise linear finite elements are used to derive the system of linear equations (1.11). For the purpose of completeness, in this subsection we introduce some common higher order finite elements. Once we have at our disposal a variety of elements of different shapes and orders, it is possible to choose appropriate elements for a given problem.

The linear (four–node) tetrahedral element was used in §1.1.1.1. Higher order tetrahedral elements, that is, with interpolation functions of higher degree, can be used in a similar way. To make the development of higher order elements systematic and easy to understand, we begin by considering triangular elements (in 2–d). Here, Pascal's triangle may be used to describe the development of triangular elements corresponding to polynomials of various degrees in two coordinates x and y. A few such elements for polynomials of degree between zero and three are shown in Figure 1.1. Each marked point (node) defines the position of a degree of freedom and so in each case a local basis function may be defined by using the unique polynomial Φ_j which is one at node j and zero at all other nodes.

In general, a pth–order triangular element has n nodes with

$$n = \frac{1}{2}(p+1)(p+2) \quad (1.17)$$

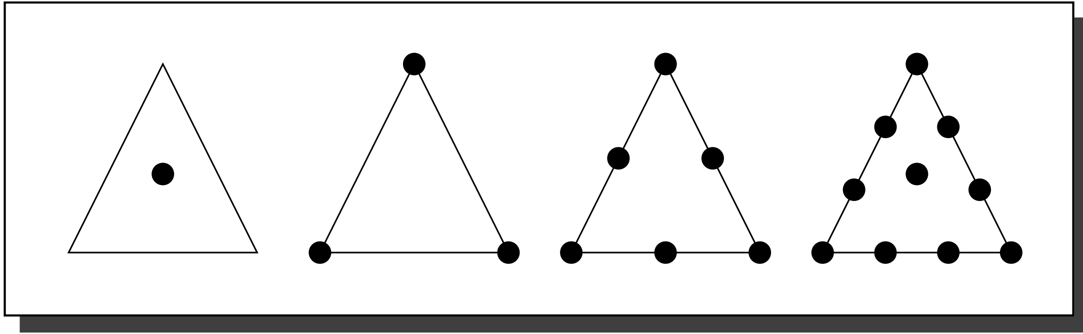

Figure 1.1: Triangular elements (in 2–d) of various degrees.

and a complete polynomial of degree p may be written as

$$u(x,y) = \sum_{i=0}^{n} a_i x^r y^s = \sum_{j=0}^{n-1} u_j \Phi_j, \qquad r+s \leq p \tag{1.18}$$

Analogous to these triangular elements, rectangular and convex quadrilateral elements can also be developed from Pascal's triangle. Since a linear rectangular element has four corners a general pth–order rectangular element has n nodes given by

$$n = (p+1)^2 \qquad (p = 0, 1, \ldots). \tag{1.19}$$

The above ideas can easily be extended to three dimensions where Pascal's triangle takes the form of a Christmas tree [144] and the elements are tetrahedra. Such tetrahedral elements for polynomials of degree between zero and three are given in Figure 1.2

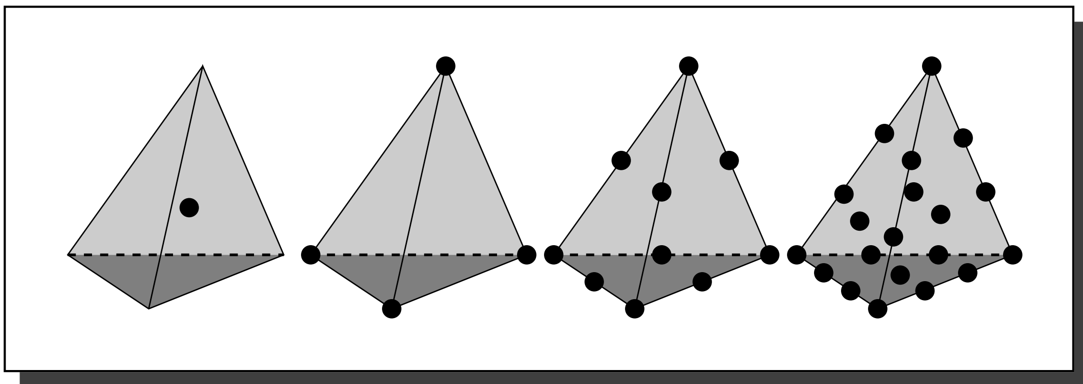

Figure 1.2: Tetrahedral elements (in 3–d) of various degrees.

A general pth–order tetrahedral element has n nodes given by

$$n = \frac{1}{6}(p+1)(p+2)(p+3) \tag{1.20}$$

and the corresponding polynomial of degree p may be written as

$$u(x,y,z) = \sum_{i=0}^{n} a_i x^r y^s z^t = \sum_{j=0}^{n-1} u_j \Phi_j, \quad r+s+t \leq p. \tag{1.21}$$

A similar extension to the corresponding prism (or brick) elements in three dimensions is also possible.

1.1.2 Petrov–Galerkin Finite Element Method

An important topic of interest for researchers of numerical analysis is to be able to solve convection–dominated problems efficiently and accurately. It is also understood that applications of standard finite element methods to such problems typically lead to non–physical oscillations of the numerical solution when the mesh is not sufficiently refined in regions of rapid change in the solution [88]. Such solution regions usually occur because of shock waves or boundary layers [164]. A number of stabilization techniques are available in the literature [65, 81, 88, 100, 108, 109, 129, 142, 143] which result in a modified system of linear algebraic equations that may be solved efficiently to yield non–oscillatory solutions on all meshes. The streamline–diffusion method is one of these existing techniques.

Recall that the general elliptic problem (1.1) is a convection–diffusion problem for the bounded domain $\Omega \subset \mathcal{R}^3$ with ε representing the magnitude of diffusivity and the vector $\underline{b} = (b_x, b_y, b_z)^T$ representing convection. When $0 < \varepsilon << \|\underline{b}\|$ the dominance of the convection term poses difficulties to the standard Galerkin finite element method [136, 144, 165, 181], making the approximation (1.10) unstable: typically oscillations are observed in solution boundary layers in the absence of a sufficiently fine mesh. Amongst the other possible solutions, such as introducing a highly refined mesh in specific regions (e.g. [163, 164]) to capture such features more accurately, it is possible to stabilize the discretization by introducing some diffusion in accordance with the direction of streamlines [65, 81, 88, 100, 108, 109, 129]. This

is achieved by using a Petrov–Galerkin approach, where the *test function* and *trial function* are no longer identical. For example we may replace the *test function* Φ_i in equation (1.9) with a *new test function*

$$\Psi_i = \Phi_i + \alpha \underline{b} \cdot \underline{\nabla} \Phi_i \tag{1.22}$$

where α is stabilization parameter. The introduction of the *new test function* makes the finite element method non–conforming. However evaluation in an element–wise manner makes the non–conforming term disappear when, as in this example, Φ_i is piecewise linear. This yields the following discrete form:

$$\begin{aligned}
\varepsilon \sum_{j=0}^{N-1} u_j \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j))\ d\Omega - \varepsilon \int_{\Gamma_2} g\ \Phi_i\ dS - \varepsilon\alpha \int_{\Gamma_2} g(\underline{b}\cdot\underline{\nabla}\Phi_i) dS \\
+ \sum_{j=0}^{N-1} u_j \int_\Omega \Phi_i(\underline{b}\cdot\underline{\nabla}\Phi_j)\ d\Omega + \alpha \sum_{j=0}^{N-1} u_j \int_\Omega (\underline{b}\cdot\underline{\nabla}\Phi_i)(\underline{b}\cdot\underline{\nabla}\Phi_j) d\Omega \\
+ c \sum_{j=0}^{N-1} u_j \int_\Omega \Phi_i\ \Phi_j\ d\Omega + c\alpha \sum_{j=0}^{N-1} u_j \int_\Omega (\underline{b}\cdot\underline{\nabla}\Phi_i)\Phi_j\ d\Omega \\
= \int_\Omega f\ \Phi_i\ d\Omega + \alpha \int_\Omega f(\underline{b}\cdot\underline{\nabla}\Phi_i) d\Omega, \\
\text{for } i = 0, \ldots, N_I - 1.
\end{aligned} \tag{1.23}$$

Ultimately this system of equations in matrix representation takes the same form as given by equation (1.11) with a modified global stiffness matrix $\mathcal{A}$ of the form

$$\mathcal{A} = \varepsilon\mathcal{K} + \mathcal{C} + c\ \mathcal{S} + \alpha\mathcal{U}\ , \tag{1.24}$$

where $\mathcal{K}$, $\mathcal{C}$ and $\mathcal{S}$ are exactly same as in equation (1.12) and $\mathcal{U}$ has entries

$$\mathcal{U}_{ij} = \int_\Omega (\underline{b}\cdot\underline{\nabla}\Phi_i)(\underline{b}\cdot\underline{\nabla}\Phi_j) d\Omega + c \int_\Omega (\underline{b}\cdot\underline{\nabla}\Phi_i)\Phi_j\ d\Omega. \tag{1.25}$$

Similarly, with the introduction of new terms, entries of the right–hand side vector can be expressed as

$$\begin{aligned}
b_i = \int_\Omega f\ \Phi_i\ d\Omega + \alpha \int_\Omega f(\underline{b}\cdot\underline{\nabla}\Phi_i) d\Omega + \varepsilon \int_{\Gamma_2} g\ \Phi_i\ dS + \varepsilon\alpha \int_{\Gamma_2} g(\underline{b}\cdot\underline{\nabla}\Phi_i) dS \\
- \sum_{j=N_I}^{N-1} u_j \left(\varepsilon \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j))\ d\Omega + \int_\Omega \Phi_i(\underline{b}\cdot\underline{\nabla}\Phi_j)\ d\Omega + c \int_\Omega \Phi_i\ \Phi_j\ d\Omega \right) \\
- \alpha \sum_{j=N_I}^{N-1} u_j \left(\int_\Omega (\underline{b}\cdot\underline{\nabla}\Phi_i)(\underline{b}\cdot\underline{\nabla}\Phi_j) d\Omega + c \int_\Omega (\underline{b}\cdot\underline{\nabla}\Phi_i)\Phi_j\ d\Omega \right).
\end{aligned} \tag{1.26}$$

It is to be noted that for $\alpha = 0$ we get the original system of equations (1.10). It can be shown, [142, 143], that a solution of equations (1.23) can be obtained efficiently with an iterative solver under the assumption that

$$\alpha = \frac{\delta\ h}{||b||} \tag{1.27}$$

where $\delta > 0$ is a tuning parameter. Description of how best to chose this tuning parameter is postponed until Chapter 4 where the implementation of this technique is discussed.

1.2 Mesh Adaptivity

As the number of unknowns in a domain Ω grows the finite element solution over Ω should become more accurate. A dense mesh therefore may provide a sufficiently accurate solution throughout the domain Ω. However, this is not only expensive computationally but inefficient too as it involves solving the problem for all unknowns, many of which may not necessarily be required. Therefore, it is often desirable to construct a mesh where the number and size of the elements vary considerably in different regions of the domain Ω. Generally smaller elements are desired in the regions where the exact solution varies rapidly or certain derivatives of the exact solution are large. The process of refinement increases the number of unknowns by increasing the number of elements in such regions, and hence also the accuracy of the finite element solution. However, this refinement aims not to increase the total number of unknowns too significantly whilst trying to significantly increase the accuracy. Hence the use of refinement seeks to optimize the increase in the degrees of freedom rather than increase them uniformly. In principal general mesh adaption strategies involve some kind of error estimator for each element, for example, a line segment in 1–d, a triangle or quadrilateral in 2–d and a tetrahedron or hexahedron in 3–d. Elements indicated by the error estimator as being insufficiently accurate may then be adapted either by subdivision of the element (*h–adaptivity*), relocating mesh points (*r–adaptivity*), enrichment of the degree of polynomials used as the

trial functions (*p–adaptivity*) or some combination of these. The first two of the above mentioned techniques cause a change in the mesh geometry whereas the third technique introduces different trial functions for different elements. We now briefly review these commonly used mesh adaptation techniques.

1.2.1 h–Adaptivity

This is the most widely used mesh refinement technique [7, 120, 131, 164] which takes a relatively coarse mesh as the base level mesh. Assuming that some sort of error estimator is available, the elements with larger errors than some suitable tolerance are subdivided into a number of smaller elements of the same type. The same procedure is continually repeated for the elements with the largest errors in the newly constructed mesh until either the error for each element in the newest mesh is no larger than the tolerance, or the highest allowable level of mesh refinement is reached. In this way it is possible to start with a reasonably small mesh, solve the finite element equations and estimate the error, then refine the particular elements at the base level and repeat recursively until we have a converged solution on a fixed mesh but hopefully avoiding an excessive number of elements. Usually this entire process is subject to some restrictions on its effectiveness, such as:

- The error indicator must be reliable.
- The error tolerance selected must be realistic for the problem being solved.
- The refinement tree must be restricted to some highest level to avoid the excessive creation of elements.
- The choice of base level mesh is important and may have significant bearing on the final fully refined mesh [10].
- Different choices of tolerance for different refinement levels may save unnecessary computational cost. This is dependent on the refinement strategy used but usually a fixed fraction strategy is used to define the tolerance (i.e. *tolerance*= 20% of maximum error on any element).

1.2.2 r–Adaptivity

In this technique the position of the nodes and the shape of the elements is altered but the number of elements in the original mesh is not. This sort of mesh vertex relocation is proposed in a number of algorithms such as is [40, 104, 127, 128]. We do not intend to describe any particular *r–adaptivity* algorithm here but only some general observations to highlight a few points. Typically an initial mesh is chosen which is not as coarse as in the case of *h–adaptivity* since it must already contain enough degrees of freedom to represent the solution. An error estimate or indicator, such as a residual norm or stored energy functional is calculated. Node relocation is then done according to this indicator. The final mesh resulting from *r–adaptivity* is often capable of representing well sharp solution features like shock waves, boundary layers etc. However this type of mesh adaption is not as robust as *h–adaptivity* since the reduction in the error (indicator) to some pre–defined tolerance may not be possible for a given initial mesh. Other problems like falling into local minima traps or mesh tangling may also be encountered. A common way of overcoming such problems is to make use of edge swapping.

The technique of edge swapping changes the mesh topology by swapping the edges in some predetermined manner [84, 145]. There are also some limitations associated with such changes in the mesh: the number of nodes in the original mesh remains unchanged and the position of the nodes depends on the geometry of the starting mesh. Edge swapping is beneficial however to improve the worst and/or bad angles in the mesh, which affects the aspect ratio of mesh elements.

1.2.3 p–Adaptivity

Like other mesh adaptivity techniques, *p–adaptivity* [7, 182] is usually based on some error estimate for each element in the mesh. The elements with an error exceeding some tolerance are adapted by increasing the degree of the trial polynomial, such as from linear to quadratic, or a higher order polynomial. This phenomena is also known as *p–enrichment*. The number of new degrees of freedom obtained through

p–refinement is the same as that obtained by *h–refinement* where the order of the polynomial in *p–refinement* corresponds to the level of refinement in *h–refinement*. However, it is to be noted that as *p–refinement* does not require subdivision of mesh elements, it helps to avoid the burden of element–subdivision and its convergence rate for smooth solutions is exponential. A down–side of this approach however is that the matrices in (1.12) become much less sparse as p is increased.

1.3 TETRAD

In this work we will only consider the use of *h–refinement* on tetrahedral elements in three dimension. TETRAD is a general purpose tetrahedral mesh adaption tool, based upon *h–adaptivity* of the tetrahedra, which can be used to support a variety of numerical solution schemes [163, 164]. To handle the complexity of unstructured meshes and to support a range of solvers, a large set of *data structures* is implemented. This is the tool that has been used to construct mesh hierarchies for the work in this thesis.

1.3.1 Description

The software package TETRAD (TETRahedral ADaptivity) is written in C and may be used to adapt (*refinement and derefinement*) meshes consisting of elements of tetrahedra in 3–D. The processes of *refinement* and *derefinement* are activated according to a scheme involving the marking of elements and/or the edges of elements. The algorithm is hierarchical in nature as is described in [163, 164]. The main data objects of *edges*, *elements*, *nodes* and *faces* form the computational domain and naturally map onto the TETRAD adaption algorithm and data structure. These objects contain the connectivity information required to adapt the mesh structure by a local mesh *refinement* or *derefinement* procedure. TETRAD assumes that there exists an initial unstructured tetrahedral mesh, taken as the base mesh of the computational domain. The refinement process adds new nodes to the initial base mesh through *edge*, *element* and *face* subdivision. Each such change being taken

into account within the data structure of the software by constructing a data hierarchy. The procedure of *derefinement* is an inverse process to *refinement*, where the local mesh of the previous level, before its *refinement*, is recovered by removing the *edges*, *elements*, *nodes* and *faces* that had been inserted during the *refinement*. This process of local coarsening can only be undertaken until the base level mesh is reached. In a refined mesh neighbouring elements can differ by at most one level.

This adaptation procedure is initiated whenever some predefined criteria for the *refinement* or *derefinement* of the mesh is satisfied. In this work we do not focus on the specific criteria available, such as *a posterior* error estimates [1, 17, 24, 118, 131] for example. In the following sections we briefly describe the data structure of TETRAD, the algorithm used for adaption and how the adaptivity is controlled.

1.3.2 Data structure

The TETRAD data structure, based on *h–adaptivity* algorithms [19, 111, 120], is implemented in C in the form of *structures* and *pointers* with the following set of *data objects* at the core of the software package. The difference between a *pointer* and an *object* can be defined as that a pointer is a means of accessing an object whereas an object is a collection of information to be accessed through pointers.

1.3.2.1 Data Objects

- **An Edge Object**

  This object is defined by two *node objects*, vertices of the mesh, mutually connected to form an edge in the mesh of the computational domain. There also exists a *parent* and two *child* pointers for each edge. At the base level mesh the parent pointers all point to null and at the highest level mesh, where no child edges are present, the child pointers all point to null. These pointers support mesh *refinement* and *derefinement*. Moreover *orphan* edges are introduced to refine the interior of elements and faces and to make the mesh conformal (details given below in §1.3.2.2).

- **An Element Object**

  Each of the tetrahedral element objects in the mesh consists of four *node objects*, six *edge objects* and four *face objects*. Each *element object* has pointers to its *parent* and *child* elements, with the exception of base level elements where no parent element exists and elements at the highest level where no child elements exist, to support the *refinement* and *derefinement* processes.

- **A Node Object**

  Each *node object* corresponds to a vertex of the computational mesh and consists of the coordinates (x,y,z) of the vertex and a pointer to a linked list of element objects which surround the vertex in the mesh: known as the node's family. This family (of elements) for each *node object* supports the connectivity of the data structure and is of fundamental importance since all other mesh connectivities may be deduced from it, as explained in §1.3.2.2.

- **A Face Object**

  All the *face objects* of a tetrahedral element are defined by three *node objects*. These face objects also have a pointer to a single *element object* as only the boundary faces are stored and so have limited functionality. Nevertheless, these objects are useful for defining boundary conditions of different types.

1.3.2.2 Further Details

In addition to these four basic/fundamental *data objects* there also exists a *link object* which provides the link (as obvious from the nomenclature) between the basic data objects, e.g., the node's family (as described above) provides a link between *node objects* and *element objects*. Also the elements which share an edge may be determined by finding the common elements in the families of those two nodes which define the edge. This sort of node–element connectivity support is one amongst a number of possible natural choices [19].

For the adaption hierarchy the parent and child pointers in the case of edges and elements are stored together with node–element connectivity. Two new child edges

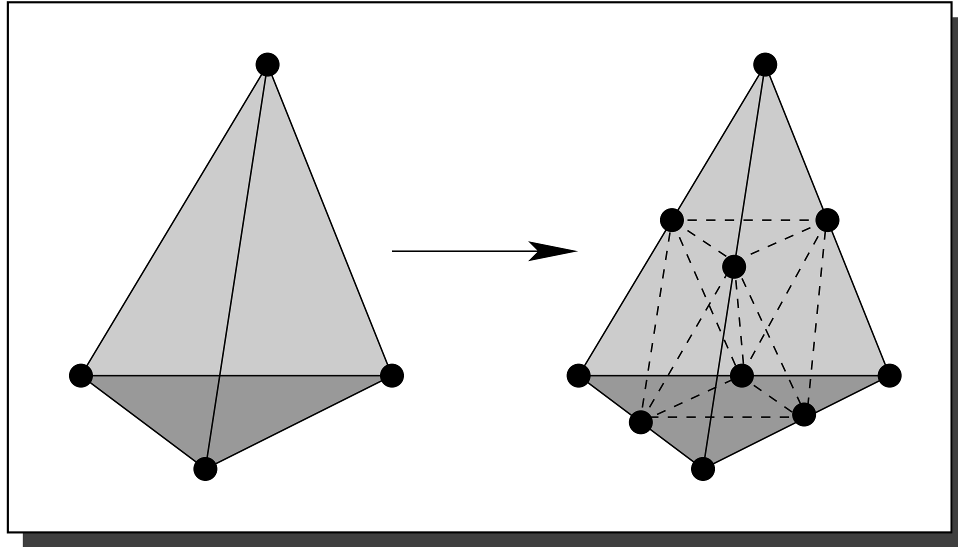

Figure 1.3: Orphan edges (dashed lines) produced by refinement.

are obtained by the bisection of each parent edge by introducing a node to its middle. However in the case of an element, the number of child elements varies depending on the type of dissection of the parent element by the refinement process (see §1.3.3). The resulting element tree contains all of the information and element refinement history required for both *derefinement* and further *refinement* to take place. In contrast to the element tree structure, we get *orphan* edges (as mentioned above) as a result of element and face dissection. These orphan edges are incorporated into the overall data structure by introducing a separate linked list of edges for each level of the refined mesh. The nodes and faces at each level are also organized in the form of linked lists in order to completely specify the mesh adaption hierarchy. On top of all this, a *mesh* data structure is also defined with pointers to the linked lists corresponding to each of the data objects. Some other fundamental figures are also provided, e.g., the number of elements in the base level mesh and the final mesh, the number of nodes, edges, faces, etc. in the final mesh.

1.3.3 Adaption Algorithm

The adaption of the mesh is carried out by *refinement* and *derefinement* of elements, edges and faces as in [19], which requires that all the elements sharing an edge must

also be refined since the edge is refined by introducing a node to that edge. Similarly, if a node is removed all the elements sharing that node must be derefined. Based on some predefined criteria, an element, and consequently each edge of that element, is marked for refinement, derefinement or left unchanged. However each edge marked for *refinement* or *derefinement* must have to pass some additional tests (again for details see [19]) before their adaption. For the purpose of refinement, only two types of element subdivision are permitted so as to maintain the quality of the tetrahedra and the simplicity of the algorithm. The first type of subdivision, called *regular* subdivision, is the popular subdivision by *eight* illustrated in Figure 1.4. The

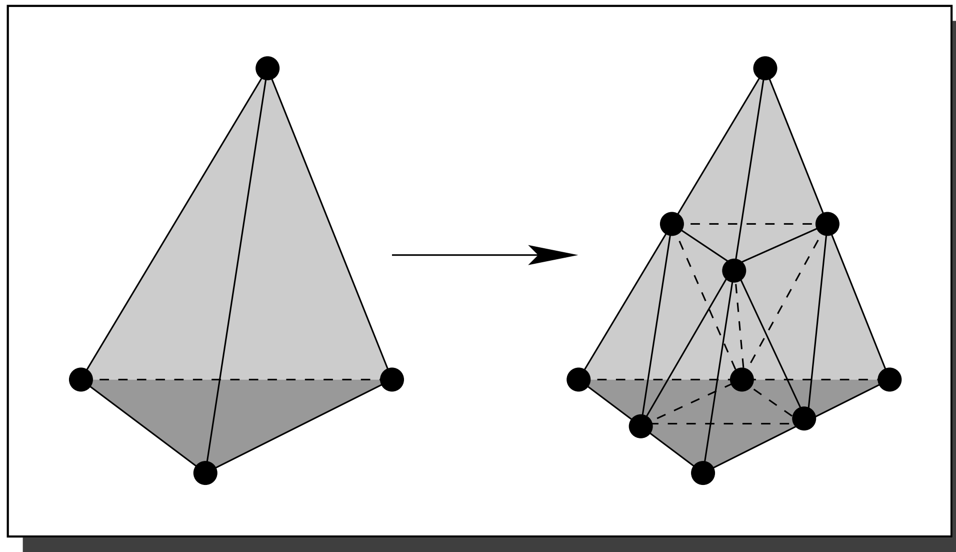

Figure 1.4: Regular refinement of one tetrahedron into 8 tetrahedra.

second type of refinement is called *green* subdivision [164] which is used when an element only has some of its edges marked for *refinement*: typically on the interface between two different levels of refinement. Green *refinement* creates between 6 and 14 green child elements depending on the number of edges of the parent element marked for refinement. This type of *refinement*, when only one edge is marked, is illustrated in Figure 1.5. For further details interested readers are referred to [111, 120, 131, 164]. Due to the poor quality of green child elements relative to regular child elements, in terms of their aspect ratio for example, green child elements are prohibited from any further refinement. Instead, when additional refinement is

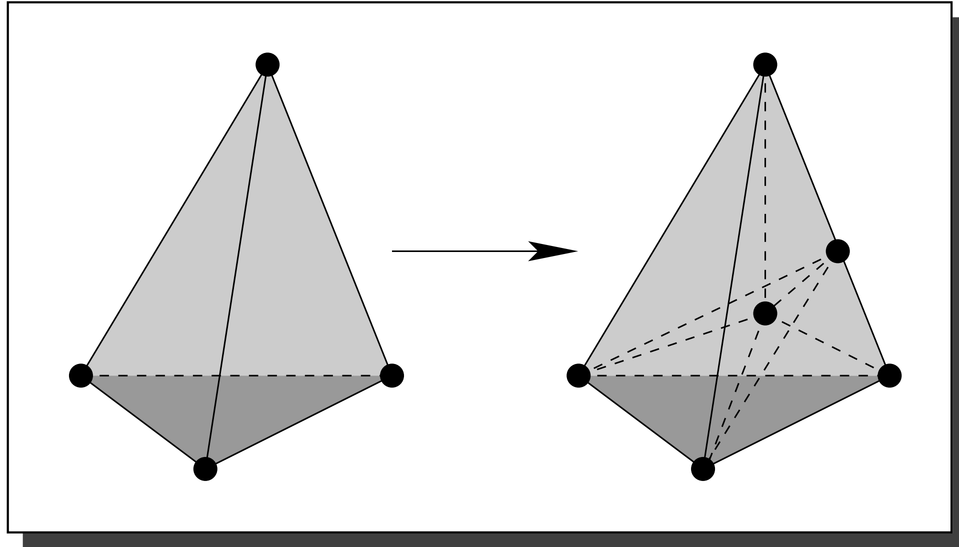

Figure 1.5: Green refinement of one tetrahedron into 6 tetrahedra.

desired, they are first derefined and then refined regularly. Since green elements generally indicate a change in the mesh levels they tend to act as interface/transit elements between changes in the grid resolution. Following the analysis presented in [131], any degradation in mesh quality as a result of this refinement procedure is kept bounded and this is verified by numerical evidence in [164] for the implementation in TETRAD.

The elements with less than or equal to 5 edges marked for refinement are refined in a manner referred as *green refinement* by catering a new node at the centroid of the element. All the nodes in the element, including the nodes resulting from edge refinement, are then connected to this new node. These new *green elements* are geometrically more distorted [131] than *regular elements* and are not further refined. However if required these green elements are first derefined to get the parent elements which are then refined in *regular manner*. This may have a knock–on effect as illustrated in two dimensions Figure 1.6. If edge A is marked for refinement then tetrahedron T_A must also be refined. Since tetrahedron T_A is green and its further refinement is ruled out, therefore, it must be first derefined along with neighbouring green tetrahedra to get the parent which is then refined in regular manner. This requires the refinement of edge B and consequently the refinement of tetrahedron

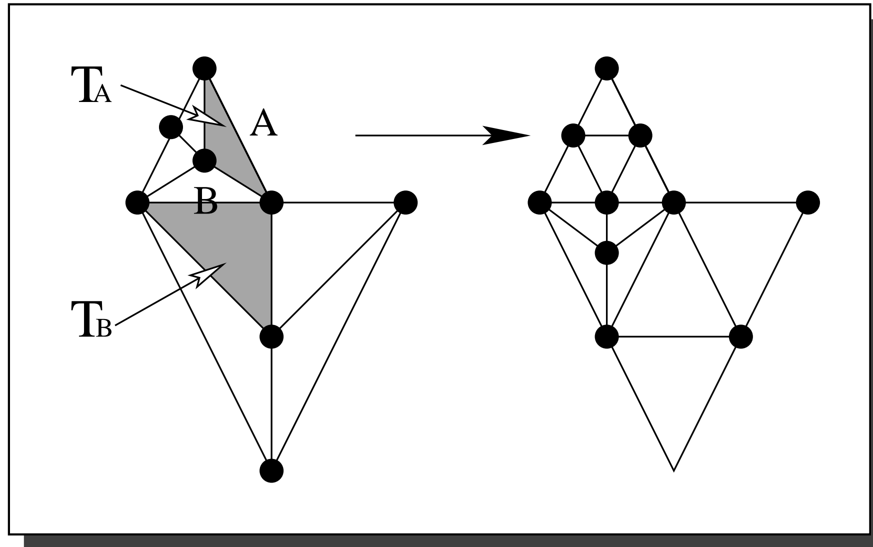

Figure 1.6: Knock–on effect of green refinement (two dimensional example).

T_B. Since tetrahedron T_B is green, the same procedure of derefinement and then refinement needs to be repeated as in the case of tetrahedron T_A. We note that all of this bookkeeping is necessary because of the inserted *new node* at the center of each tetrahedron when refined in green manner. The green refinement approach adopted here raises a number of general issues concerning mesh quality which involve not only the discretization error but also the norm [17], in which the error is to be controlled. These issues will be explained in more detail in Chapters 3 and 4, where special coding effort is required to handle these new green nodes in order to implement the domain decomposition preconditioning algorithms which are the subject of this thesis. Alternative forms to this green refinement are possible without the insertion of a new node at the centre of tetrahedra, but this work only had access to TETRAD, and so the additional node must be dealt with.

1.3.4 Adaption Control

The adaption algorithm is combined with a flagging mechanism for the mesh regions to be refined or derefined according to some user–defined criteria. This could be to refine some particular region of the computational domain, defined through the mesh geometry, or possibly to adapt based upon a local solution error indicator.

These approaches can be used independently (e.g. the first approach for global refinement and second for local refinement) or they can be combined together (for local refinement). In addition to this a *safety layer* of refinement flagging is applied [164] and a maximum level for mesh refinement is specified. The issue of mesh derefinement is dealt in a similar manner. In this work our main concern is the application of an efficient finite element solver so we will mainly use simple geometric rules to drive the adaption.

1.4 Parallel Computer Architecture

In this thesis we are concerned with the efficient application of the finite element method on parallel computer architectures. In this section we present a brief outline of some of the main issues associated with achieving this.

1.4.1 Background

Today in almost all areas of science and engineering experiment, observation and design prototyping are being replaced (or at least compensated) by computations. Also, phenomena which would be almost impossible to study through experiments, such as the evolution of the universe, are now being simulated. Undertaking such calculations places a continued pressure for the development of computers with both greater speed and greater storage in the form of memory. Ultimately this will lead to fundamental physical limitations affecting the performance of such great computers.

The development of parallel computers is led, along with other reasons, by these physical limitations of sequential computers. For example, suppose we wish to build a computer with the capability of performing one trillion arithmetic operations per second. Assume that we have built more or less a conventional von Neumann computer with extremely fast computational power. A simple arithmetic operation, addition say, inside a loop of length of one trillion would have to successively fetch the contents of two memory locations into the registers, perform the operation and store the result in to the third memory location (modified form of

an example from [50]). This would require 3×10^{12} copies between memory locations and registers to be undertaken each second. Now suppose that data travels between CPU and memory at the speed of light (3×10^8 meters/second), then the distance between the CPU and a word of memory should average 10^{-4} meters $[(3 \times 10^8 \text{meters/second} \times 1\text{second})/(3 \times 10^{12}\text{meters})]$. If our computer has 3 trillion *memory words* forming a square grid of side 10^{-4} meters we would clearly need to be able to fit a single word of memory into a square a side length equal to $(2 \times 10^{-4}\text{meters}/\sqrt{3} \times 10^6) \approx 10^{-10}$ meters, the size of a relatively small atom. This suggests that building such a computer is impossible, thus motivating the development of parallel computers which have multiple CPUs capable of performing tasks simultaneously.

We now briefly describe a few standard types of parallel architecture. The original classification of parallel computers, popularly known as Flynn's taxonomy [82], is based upon the number of instruction streams and the number of data streams. The classical von Neumann machine is *single–instruction single–data* (SISD) and the opposite extreme is a *multiple–instruction multiple–data* (MIMD) computer. The other possible systems are *single–instruction multiple–data* (SIMD) and *multiple–instruction single–data* (MISD), although the latter rarely occurs in practice.

1.4.2 The Classical von Neumann Machine

The classical von Neumann machine is a combination of a CPU and main memory wherein the CPU is a combination of a control unit and an algorithmic–logic unit (ALU). The memory stores the instructions and data, the control unit directs execution of the program and the ALU perform the calculations. During the execution, instructions and data are stored in very fast memory locations called registers which, being expensive, are relatively few in number in any machine. The route for instructions and data between memory and registers in the CPU is via the bus – a collection of parallel wires together with some hardware. The additional devices required for the von Neumann machine to be useful are input/output devices and an extended storage device such as a hard disk.

1.4.3 Pipeline and Vector Architecture

Pipelining was the first widely used extension to the basic von Neumann model. By splitting various circuits in a CPU into functional units and setting them up in a pipeline, then theoretically (and once it is full) a result is produced by this pipeline during each instruction cycle. A further improvement can be obtained by adding vector instructions to the basic machine instructions set. These instructions are similar to those in Fortran 90 which replace the loops of Fortran 77. The CRAY C90 and NEC SX4 are examples of vector processors.

1.4.4 SIMD Systems

In contrast to vector processors, a SIMD system consists of a single CPU devoted to control and a large number of subordinate ALUs – each with its own small amount of memory. The control processor broadcasts instructions to subordinate processors, each of which than either execute that instruction or remain idle. It means that a completely synchronous execution of instructions is undertaken. The most famous SIMD machines are the CM–1 and CM–2 from Thinking Machines, and the MP–2 from Maspar. It should be noted however that, for the most part, SIMD systems are now obsolete.

1.4.5 General MIMD Systems

The key difference of MIMD machines from SIMD machines is that with the former each processor is a complete CPU with its own control unit and ALU making it independent in executing its program at its own pace (unless otherwise specifically programmed to synchronize with other processors at some particular point). Thus, even if the same instruction set is being executed, there may not be any correspondence between the specific instructions being executed on different processors at a given instant. The MIMD systems are generally divided into two main categories: ***shared–memory*** and ***distributed–memory*** systems.

1.4.5.1 Shared Memory MIMD

A generic shared memory machine is defined as a collection of processors (CPUs) and memory modules interconnected through a network such that each processor can access any of the memory modules. In other words, any set of data stored in the shared memory is common to all processors and can be accessed by each of them. The advantage of this is that it is very simple to program and, in principle, very fast. However if more than one processor tries to access the same memory location at the same time, there may be a long delay while this contention is resolved. In this category of system, the simplest network for interconnection between the processors and the memory modules is bus based. Delays are also caused therefore if the bus becomes saturated due to too many processors simultaneous attempting to access the memory. Although each processor has access to a large amount of memory in this model, the limited bandwidth of the bus does not usually allow these architectures to have a large number of processors. Some other shared memory architectures, instead of a bus–based interconnection network, rely on some form of switch–based interconnection network. For example, the basic unit of the Convex SPP1200 is a 5×5 crossbar switch. Hence any processor can access any memory module and any other processor can access any other memory module without interference with each other. This last described system provides an example of non–uniform memory access (NUMA), where non–uniform access times occur depending on the location of data relative to the CPU that requires it. In recent years the use of NUMA architecture has become very popular amongst the designers of shared memory systems, including the SG Origin, for example, on which much of the work in this thesis was undertaken.

1.4.5.2 Distributed Memory MIMD

In distributed memory systems, each processor has its own private memory module to which other processors have no access. Distributed memory systems can be visualized as a graph with edges as the communications wires. We can imagine two major types of such systems: those in which each vertex of the graph correspond to

a processor and memory module pair, conventionally known as node, and those in which some of the vertices of the graph correspond to nodes and others to switches. These two types are referred to as static networks and dynamic networks respectively. The ideal interconnection network is the fully connected network, i.e. each node is statically connected to every other node to allow direct communication between any two nodes. Obviously the cost of such a network makes it extremely difficult to build such a machine with more than a few nodes. To overcome this, use is made of fast switches such as the crossbar switch, however it is unusual to have such a switch with more than 16 processors. The Fujitsu VPP500 with 224 nodes and a 224×224 crossbar is one of a number of exceptions however. A related approach is the use of multistage switching networks such as the *omega* network, which requires $p \log_2(p)/2$ switches for p nodes in contrast to p^2 switches in the case of a crossbar. Every node can still communicate directly to every other node but with an increased probability of a delay in transmitting a message due to interference at a switch. IBM, in its SP series of computers, have used a combination of crossbar and omega networks with the largest machines consisting of 512 nodes.

Other distributed memory topologies include a linear array in which all the nodes have two immediate adjacent neighbour nodes except the extreme end nodes, which have a single neighbour node. A ring network is a slightly more powerful variant of this with each end node being a neighbour to the others. These networks only cost $p-1$ or p wires respectively and can easily be extended to include an arbitrary number of nodes with very little expenditure. However, multiple pairs of nodes are unable to communicate at the same time because of the limited number of wires and in the worst case a message has to pass through $p-1$ or $p/2$ wires respectively to reach its destination. A better, practical and closer to fully connected, network of this static type is known as the hypercube. This topology is defined inductively, whereby a dimension 0 hypercube consists of a single node and any hypercube with dimension $d > 0$ consists of two hypercubes, each of dimension $d-1$ and joined by 2^{d-1} communications wires connecting corresponding pairs of nodes in the $(d-1)$–dimensional cubes. Hence, a dimension d hypercube consists of 2^d nodes

and each node is connected to d other nodes. In the worst case any message has to pass through $\log_2(p)$ wires. Although the first *massively parallel* MIMD system was a hypercube (an nCUBE 10 with 1024 nodes), it still lacks in scalability to a certain extent. Meshes and tori are alternative topologies which are simply higher dimensional analogues of the linear array and ring architecture respectively. An important feature of these systems is that, with the increase in dimension, the probability of interfere between any two pair of communicating nodes decreases. Examples include the Intel Paragon, a two dimensional mesh, and the CRAY T3E, a three dimensional torus, both of which scale to thousands of nodes.

1.5 Message Passing Interface

Message Passing Interface (MPI) is the outcome of a community effort. A forum, known as the MPI forum, was established comprising of over 80 experts from more than 40 organizations in order to define a standard for a portable message passing system to support parallel applications and libraries [133]. This forum defined the syntax and semantics of the functions which make up what is today called the MPI library. This library is based upon a MIMD distributed memory programming model, however MPI codes still run efficiently on shared memory architectures. Indeed, MPI is today one of the most widely used and most powerful libraries for programming parallel systems. In this section we explore a small subset of MPI functions, such as one–to–one communications functions, broadcast and reduction operations, and a few other miscellaneous functions which are relevant to the work undertaken in the rest of this thesis.

1.5.1 Startup

We begin by describing how to start a simple parallel C program using MPI. The very first function which indicates the start of a parallel program is `MPI_Init` which has parameters which are pointers to the parameters of the main function, that is, *argc* and *argv*. Any other MPI function can be called only after calling this function.

Conversely, no MPI function can be called after a call to the MPI function MPI_Finalize. This function, which requires no parameters, indicates the end of the parallel part of the program and cleans up any left over or unfinished jobs such as memory deallocation. It is not essential for MPI_Init to be the first or MPI_Finalize to be the last executable statements in the program. The other two basic MPI functions, MPI_Comm_rank and MPI_Comm_size, return the *rank* of each process (an integer between 0 and $p-1$), and *size* (the total number of processes involved). The flow of control in any program typically depends very heavily on knowledge of these quantities.

1.5.2 Point to Point Communication

The current implementation of MPI assumes a static allocation of processes, that is, the number of processes, size, is set at the beginning of program execution. The rank of all the processes and the size of the *communicator* is known to each process (a communicator is simply a name for a subset of the processes). The communication of messages to particular processes is controlled by targeting their ranks. The pair of basic functions provided by MPI for sending and receiving messages in this way are MPI_Send and MPI_Recv respectively.

Suppose that, in a simple two process example, we wish the process with rank 0 to send a message to the process with rank 1. Each process can execute a copy of the same program, with the different effect on each process being gained through the use of conditional statements (e.g. if $rank = 0$ then send else if $rank = 1$ then receive). This single program paradigm is the most commonly used approach for parallel programming in MPI. It has the advantage of minimizing the likelihood of programmer errors due to one process calling the send function and the other process not calling the corresponding receive function. When such an error occurs the behaviour of the sending process in undefined by MPI. In general however, it cannot be assumed that the execution will continue beyond the unsuccessful *blocking* send since, if the system does not provide adequate buffering, the sending process will hang on forever.

Now we consider another sequence of calls of the send–receive communication pair. Suppose that one process calls the receiving function but the other process does not call the corresponding send function until some later time. Since `MPI_Recv` is also a blocking function, if the message is not available for it to receive it the process will remain waiting and idle until the message becomes available. As an alternative to this scenario, MPI provides another form of communication, known as *non–blocking* communication.

When a non–blocking receive is called, instead of waiting for the message, the system merely allows the receiving process to receive a message from the sending process, but the program execution itself continues. At some time before it is necessary to use the message to be received, it is possible to check on its arrival through the call to another function.

The non–blocking send function can be used in a similar way. Thus most non–blocking operations require at least two calls, one to start the operation and the other to check that it has been successfully completed. The basic functions in MPI for non–blocking communication are named `MPI_Isend` and `MPI_Irecv`, where the 'I' stands for immediate, since these functions return immediately after they are called. There are two main MPI functions for checking the completion of non–blocking communications. `MPI_Wait` checks to see if the operation is complete and blocks until it is. `MPI_Test` always returns immediately but with a value which indicates if the operation has completed or not.

It should be noted that the matching of a blocking send/receive with a non–blocking receive/send communication operation is not illegal, and some times it proves to be most appropriate. For example, if we are going to use data as soon as it is received then `MPI_Recv` should be used even if the message is sent using `MPI_Isend` at some earlier time.

1.5.3 Collective Communication

A communication involving all of the processes in a communicator is said to be a *collective communication*. The broadcast communication pattern is an example of

a collective communication in which a single process sends the same message using the function MPI_Bcast to every other process in the communicator. This type of communication can be a lot more efficient than a corresponding sequence of sends to all of the processes from the originating process. The converse situation where we need to send messages from all processes in the communicator to a single process is known as reduction. MPI provides a general class of *reduction operations* through the function MPI_Reduce. As with MPI_Bcast, this function must be called by all processes in the communicator however the result of this operation is available to the *root* process only. For example, if we perform the reduction operation to find the maximum of a set of values distributed across the processes, only the root process will return this maximum value. Another function, MPI_Allreduce, performs reduction operations in the same way as MPI_Reduce except that the result of the operation is made available to all processes.

1.5.4 Some Miscellaneous Functions

Assume that in a parallel program we need to synchronize all of the processes before performing a particular task. To block all of the processes in the communicator at a particular point, until the last one reaches that point, MPI provides the function MPI_Barrier. This function is useful for the timing of parallel programs; especially for timing particular subsections of a program. To find the time taken by any parallel program, or any subsection of a parallel program, MPI provides a double precision function MPI_Wtime. When called this function returns the elapsed time in seconds from some particular point in the past. It should be noted that this function returns the *wall–clock* time which makes it unreliable for performance analysis. The last function to be considered here is related to aborting from a parallel program. If a parallel program crashes during the execution because of some bug in the program, it is possible that the processes which did not encounter the bug continue to execute or wait upon the crashed process(s). To ensure that no processes continue to run after the program has behaved in an unexpected manner the MPI_Abort function is provided.

1.6 Parallel Finite Element Method

A system of linear equations of the form (1.11) with a few thousand degrees of freedom can be solved easily and efficiently using a conventional computer. However as the size of this system of linear equations grows, due to mesh refinement for example, then conventional computers face the twin problems of lack of speed and memory. Also, in some cases, even when a conventional computer is able to solve a very large problem, the time taken to do so may be such that the solution becomes useless [116], e.g. with forecast problems which require a solution to be delivered by a specific time.

Parallel computers/architectures provide a means of overcoming these speed and memory problems and can lead to massive savings in solution time. For any problem to be solved in parallel we need to partition the data, which in the case of the finite element method means dividing the computational mesh into a number of parts. There are a number of ways of partitioning the computational mesh and these are discussed in the next chapter. Nevertheless, a consequence of such a partition is the requirement that the assembly of the coefficient matrix (which is sparse) and the computation of the right–hand side vector in (1.11) have to be undertaken in a distributed manner (preferably in parallel).

1.6.1 Parallel Sparse Matrix Assembly

Consider a system of linear equations (1.11) with the sparse coefficient matrix $\mathcal{A}$ and a corresponding mesh of the computational domain Ω, whose elements are partitioned into p subdomains Ω_i for $i = 0, \ldots, p-1$, where each subdomain Ω_i is assigned to the process i. Due to the partition of the elements of the mesh, the unknowns in the solution vector $\underline{u}$ may be distributed across the p subdomains such that part of the solution vector, $\underline{u}_i$ say, is owned by process i. Additionally, each process i shares some of the unknowns at the interface of the subdomains with one or more other processes. The same distribution may be associated with the entries of the right–hand side vector $\underline{b}$, so that the system of linear equations may be written

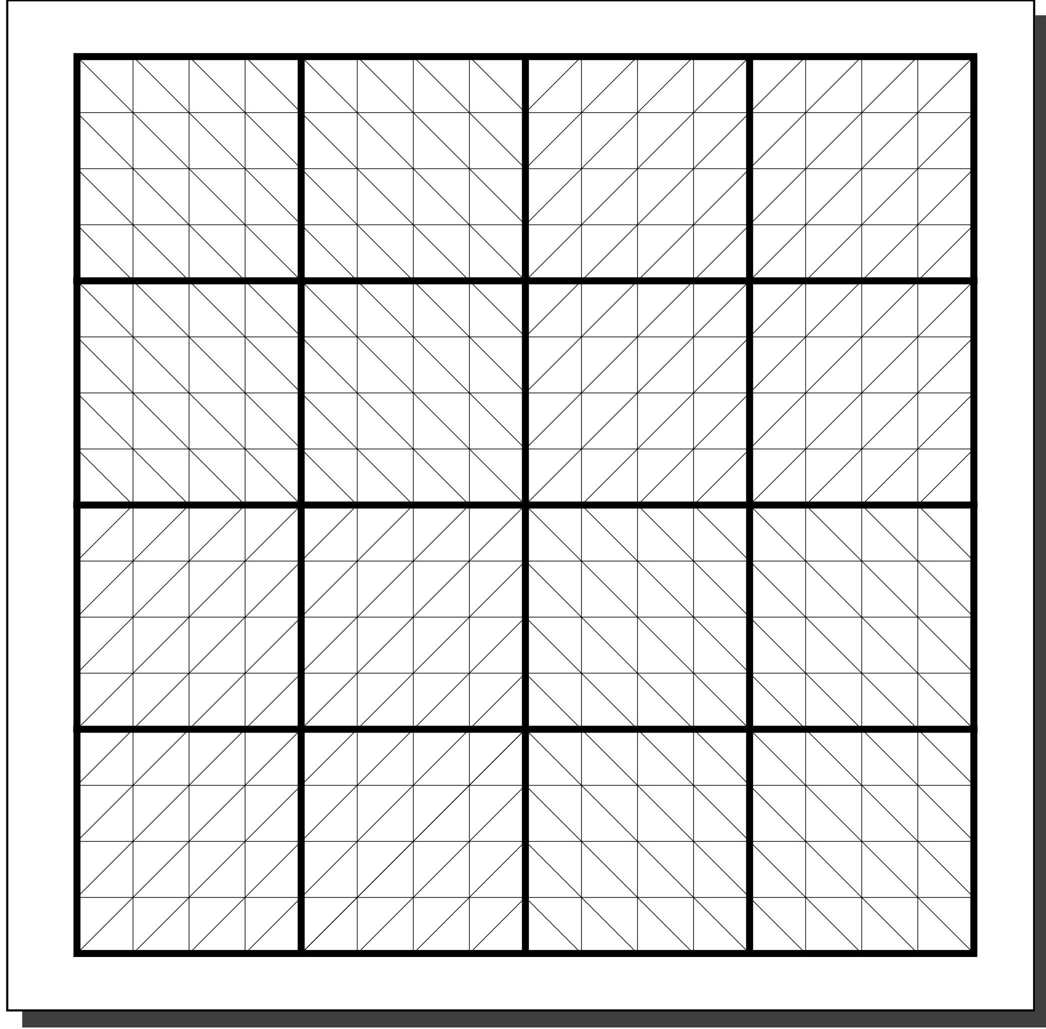

Figure 1.7: Partition of a computational domain into 16 subdomains.

in block matrix form as:

$$\begin{bmatrix} A_0 & & & & B_0 \\ & A_1 & & & B_1 \\ & & \ddots & & \vdots \\ & & & A_{p-1} & B_{p-1} \\ C_0 & C_1 & \dots & C_{p-1} & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_0 \\ \underline{u}_1 \\ \vdots \\ \underline{u}_{p-1} \\ \underline{u}_s \end{bmatrix} = \begin{bmatrix} \underline{b}_0 \\ \underline{b}_1 \\ \vdots \\ \underline{b}_{p-1} \\ \underline{b}_s \end{bmatrix}. \tag{1.28}$$

Here the blocks A_i, B_i, C_i and A_s are themselves sparse. Also $\underline{u}_i$ is the vector of unknown nodal values strictly inside the subdomain Ω_i $(i = 0, \dots, p-1)$ and $\underline{u}_s$ is the vector of unknown nodal values for nodes on the interface between subdomains.

From the definition of the basis function Φ_i it is clear that the integrals required to calculate A_i, B_i, C_i and $\underline{b}_i$ are interior to the subdomain i and so these matrix blocks can be assembled independently by each process i. However, due to the distribution of A_s and $\underline{b}_s$ across the subdomains, each process is required to assemble its contribution independently so that we have

$$A_s = \sum_{i=0}^{p-1} A_{s(i)}, \tag{1.29}$$

$$\underline{b}_s = \sum_{i=0}^{p-1} \underline{b}_{s(i)} \tag{1.30}$$

where $A_{s(i)}$ and $\underline{b}_{s(i)}$ are the contributions computed by process i on Ω_i. Now it is straightforward to implement a simple solver in parallel. For example, if an iterative method is implemented then this typically requires the calculation of matrix–vector products and inner products. The distributed matrix–vector product and the distributed inner product require only a small amount of communication amongst the processes, which is now almost a standard in parallel computing and is examined in §1.6.4.

1.6.2 Direct Solvers

The use of direct solvers is one of the basic approaches used for solving a linear system of equations such as (1.11). Direct solution methods typically use variations on Gaussian elimination and generally consist of the following steps:

1. Factorization of the coefficient matrix of the linear system – the most expensive part.

2. Use of this factorization to solve the system through forward and backward substitution.

The *mathematical structure* and the *storage structure* of the coefficient matrix play important roles in choosing any algorithm to solve the corresponding system.

The mathematical structure defines the mathematical features of the coefficient matrix such as Hermitian, non–Hermitian, definite, indefinite, etc., whereas the storage structure usually represents the storage format such as dense, sparse, banded, structured, etc. The appropriate type of factorization of a coefficient matrix is determined by the mathematical structure (e.g. Choleski factorization for Hermitian, *LU* factorization with pivoting for general Non–Hermitian, etc.).

The direct solution methods can further be categorized with respect to the properties of the coefficient matrix such as direct methods for *Dense Matrices, Band Matrices, Sparse Matrices, Structured Matrices* etc. We are interested in direct solution methods for the sparse matrices which may fall in any of the following group [53, 62]:

- General techniques,
- Frontal methods,
- Multifrontal methods.

1.6.2.1 General Methods

First we outline a completely *general approach* [60] where the numerical and sparsity pivoting can be performed at the same time (by making use of dynamic and sparse structures). This approach gives very satisfactory performance over a wide range of mathematical structures and further gains and simplification can be achieved for symmetric or banded coefficient matrices. If the coefficient matrix is symmetric and the diagonal pivots produce a stable factorization than there is no need to check numerical values for stability and the search for a pivot is simplified.

1.6.2.2 Frontal Methods

Frontal schemes [99] can be viewed as an extension to band or variable–band schemes. These methods perform well for systems with small bandwidth however for grid–based problems, such as discretization of partial differential equations, the efficiency is geometry dependent. The origin of frontal methods lies in the solution of finite

element problems in structural analysis. One of the earliest computer applications of frontal method was considered for a symmetric positive definite system [103] and extended to unsymmetric systems [98] and systems without restrictions on the finite element application [61].

1.6.2.3 Multifrontal Methods

As the name suggests, the multifrontal approach, [63], is an extension of frontal methods. These methods can handle symmetric or nearly symmetric coefficient matrices efficiently and any type of sparse ordering for symmetric systems is allowed. If the bandwidth of the coefficient matrix is not small, a large amount of storage and arithmetic operations are required. In such situations multifrontal methods [62] attempt to reduce the required number of arithmetic operations while keeping some of the benefits of frontal schemes.

The main issue in the implementation of the above described and most other direct solution methods for solving sparse linear equations lies in preserving the sparsity in the matrix factors. The LU decomposition is favoured for this reason and strategies are developed to retain much of the sparsity of the original matrix in the LU factors. On the other hand this leads to a need to compromise the numerical pivoting strategy in order to choose pivots to limit the fill–in. The common threshold strategy and the discussion of other similar possibilities to limit the fill–in may be found in [2, 49, 62]. For all the methods described, there are three levels at which parallelism can be exploited in the solution of sparse linear systems. The finest grain parallelism lies in the elimination operations. At the coarsest level, matrix partitioning techniques are often designed for parallel computing and are particularly appropriate for distributed memory computers. Indeed, matrix partitioning methods are often only competitive in the context of parallelism. At an intermediate level, the sparsity of the matrix can sometimes be used to assist with the parallelism if there is a substantial independence between pivot steps in the sparse elimination. For example, in the best possible case, if the matrix is a permutation of a diagonal matrix then all of the operations may be performed in parallel.

1.6.3 Iterative Solvers

The phenomena of obtaining a solution to a linear system of algebraic equations such as given by (1.11) through successive approximations is known as an *iterative solution method.* There are two main types of iterative solution methods, [13], known as *stationary* and *nonstationary* methods. The former type of method represents older techniques which are simple, easy to understand and implement, but have the disadvantage of being less effective. The latter type consists of relatively recent developments, which are harder to understand and implement but can be very effective. A measure of the effectiveness of an iterative method is given by the number of iterations taken to converge: the fewer the number of iterations the better the method. In the following sections, a few representative iterative methods of each type are described. It should be noted that the performance of most of these methods may be improved through the use of preconditioning. Since parallel domain decomposition preconditioning is the subject of this thesis we postpone a discussion of the role of preconditioners to a separate section of its own.

1.6.3.1 Stationary Methods

In this section, we present stationary methods and summarize their convergence behaviour and effectiveness.

- **The Jacobi Method**

  In this method, the approximation to each variable in the linear system of equations is updated with respect to the other variables at each iteration. The method is also known as the method of *simultaneous displacements* since the order of equations in the system is irrelevant as this method treats them independently. This makes the method converge slowly but suitable for parallel implementation.

- **The Gauss–Seidel Method**

  This method is similar to the Jacobi method except that the most recent values of other variables are used in the update of each variable. Whenever

the Jacobi method converges, the Gauss–Seidel method converges at least as fast for the same problem. This method is also known as the method of *successive approximations*.

- **The Successive Overrelaxation Method**

  The successive overrelaxation (SOR) method is an extrapolation of the Gauss–Seidel method by introducing a weighted average between any two consecutive iterations. The value of this extrapolation factor should always be in the interval (0, 2) otherwise this method may fail [110]. By setting the value of the extrapolating factor equal to 1 this method simplifies to the Gauss–Seidel method.

- **The Symmetric Successive Overrelaxation Method**

  If the coefficient matrix of the given system of linear equations is symmetric then this method combines a forward and a backward sweep of the SOR method such that the resulting iteration matrix is similar to the symmetric coefficient matrix. Because of this double sweep, for any optimal choice of the extrapolating factor, this method is slower than the SOR method [179]. Hence there is no advantage of this method over the SOR method however it can be used as a preconditioner with other methods where maintaining symmetry and/or definiteness is important.

1.6.3.2 Nonstationary Methods

The two main subclasses of nonstationary type (i.e. Conjugate Gradient and its variants and Minimum Residual and its variants) are described. These methods are examples of Krylov subspace methods.

- **The Conjugate Gradient Method**

  The conjugate gradient (CG) method [4] is one of oldest and most well known iterative methods of nonstationary type and is very effective for symmetric positive definite (SPD) systems of equations. This method generates successive approximations to both the solution and the residual based upon a two–term

recurrence relation. Hence, only a small amount of memory, one matrix–vector product and only two inner products per iteration are required, and the method performs well for sparse matrices.

- **The Biconjugate Gradient Method**

  This method is a variant of the CG method and generates two sequences of vectors similar to those in the CG method. One sequence is built on the system with the original coefficient matrix and the other sequence on its transpose. These two sequences are made orthogonal to each other, hence the name of this method. With limited memory requirements this method is suitable for a nonsymmetric (and non–singular) coefficient matrix [67] but with the possibility of irregular convergence or breakdown in the worst case. This method requires two matrix–vector multiplications at each iteration, one with the original coefficient matrix and the other with its transpose.

- **The Conjugate Gradient Squared Method**

  This method is a variant of the biconjugate gradient method and updates the sequences of vectors corresponding to the original coefficient matrix and its transpose to the same vector. This makes the convergence rate of the conjugate gradient squared [162] method almost double but with many more irregularities. It may also lead to unreliable results. Unlike the biconjugate gradient method, this method requires only one matrix vector multiplication (for the original coefficient matrix) per iteration. If sequences of vectors are updated to different vectors instead of one vector, the above method becomes the biconjugate gradient stabilized [167] method, which converges smoothly.

- **Minimum Residual and Symmetric LQ**

  MINRES and SYMMLQ both generate a sequence of mutually orthogonal vectors and can be used as an alternative to the CG method for symmetric but indefinite linear systems of equations [134]. The Symmetric LQ method [135] is based on the LQ factorization of the coefficient matrix and generates the same solution iterations as the CG method for SPD systems.

- **The Generalized Minimal Residual Method**

  The generalized minimal residual (GMRES) method [154], like the minimum residual method, builds a sequence of orthogonal vectors which are than combined through a least square solve and update. In contrast to the minimum residual and the conjugate gradient methods, this method stores all vectors generated at previous iterations and thus needs a fairly large amount of memory. In order to control the memory requirement, a restarted variant of this method is used by specifying a suitable maximum number of orthogonal vectors to be generated before restarting. This method is very useful for systems with a general nonsymmetric coefficient matrix. The orthogonalization may be done by using either the classical Gram–Schmidt procedure, or a modified Gram–Schmidt procedure for extra stability. The parallel implementation is straightforward with the main effort being to implement the inner products and the matrix–vector product in parallel. It is a preconditioned version of this parallel GMRES algorithm that is used throughout this thesis.

1.6.4 Parallel Solution

Like many other iterative methods, when GMRES is implemented in parallel a relatively small number of computational kernels dominate. These are listed below.

- Matrix–vector product,
- Inner product,
- Preconditioning solves (if preconditioner is applied).

For a distributed memory programming paradigm the computation of the sparse matrix–vector product and the accumulation of each inner product require some communication amongst the processes. The amount of this communication depends on the non–zero structure of the matrix, how these non–zero entries are distributed across the processes, and the number of vector entries shared by more than one process (i.e. on an interface). The topic of preconditioning, which is the subject

of this thesis, is introduced in §1.7. Here we describe the parallel aspects of the matrix–vector product and the inner product in turn.

1.6.4.1 Matrix–Vector Product

The matrix–vector product is relatively easy to implement on high performance computers. For a sparse linear system of equations to be solved on a parallel computer, it is natural to map the equations in the linear system and the corresponding unknowns to the same process. Recall from §1.6.1 that due to the partition of the mesh elements, the unknowns in the solution vector may be distributed across the p processes (subdomains) such that process i owns its own solution vector, $\underline{u}_i$ say, and the unknowns at the interface of subdomain Ω_i with other subdomains. Hence the sparse matrix should be distributed across the processes accordingly (see (1.28)).

Following this same block structure, the distributed matrix–vector product of the form

$$\mathcal{A}\underline{x} = \begin{bmatrix} A_0 & & & & B_0 \\ & A_1 & & & B_1 \\ & & \ddots & & \vdots \\ & & & A_{p-1} & B_{p-1} \\ C_0 & C_1 & \dots & C_{p-1} & A_s \end{bmatrix} \begin{bmatrix} \underline{x}_0 \\ \underline{x}_1 \\ \vdots \\ \underline{x}_{p-1} \\ \underline{x}_s \end{bmatrix} = \begin{bmatrix} \underline{y}_0 \\ \underline{y}_1 \\ \vdots \\ \underline{y}_{p-1} \\ \underline{y}_s \end{bmatrix} = \underline{y} \tag{1.31}$$

can be calculated as

$$\underline{y}_i = A_i\underline{x}_i + B_i\underline{x}_s \tag{1.32}$$

on process i, for $i = 0, \dots, p-1$, and

$$\underline{y}_s = \sum_{i=0}^{p-1} C_i\underline{x}_i + A_{s(i)}\underline{x}_s, \tag{1.33}$$

where $A_{s(i)}$ is defined by (1.29). It is to be noted that the blocks A_i, B_i, C_i are themselves sparse, the entries of vectors $\underline{x}_i$ and $\underline{y}_i$ are strictly inside the subdomain Ω_i $(i = 0, \dots, p-1)$ whereas A_s, $\underline{x}_s$ and $\underline{y}_s$ are distributed across the processes, and hence to form the sum in (1.33) communication is required amongst neighbouring processes. It is worth mentioning here however that, for the work presented in this

thesis, the distributed data structure is designed in such a way as to eliminate this requirement to communicate across the processes for the matrix–vector product. That is, $\underline{y}_i$ is stored on process i and

$$\underline{y}_{s(i)} = C_i \underline{x}_i + A_{s(i)} \underline{x}_s \tag{1.34}$$

is stored on process i too (i.e. $\underline{y}_s$ is not fully assembled).

1.6.4.2 Inner Product

It may be argued that the calculation of sparse matrix–vector products need not necessarily cause a significant performance degradation due to communication costs on modern parallel computers. However, the case of inner products is not the same. Inner products use all of the components of the given vectors to compute a single floating–point result which is required by all of the processes. Therefore the calculation of an inner product requires a global communication, which also provides a necessary synchronization point in the parallel code. To elaborate upon the global communication required for distributed inner products, let us consider two distributed vectors $\underline{x}$ and $\underline{y}$ given by

$$\underline{x} = \begin{bmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{p-1} \\ \underline{x}_s \end{bmatrix} \tag{1.35}$$

and

$$\underline{y} = \begin{bmatrix} \underline{y}_0 \\ \vdots \\ \underline{y}_{p-1} \\ \underline{y}_s \end{bmatrix}, \tag{1.36}$$

where

$$\underline{x}_s = \sum_{i=0}^{p-1} \underline{x}_{s(i)} \tag{1.37}$$

and

$$\underline{y}_s = \sum_{i=0}^{p-1} \underline{y}_{s(i)} \tag{1.38}$$

are explained in §1.6.4.1. The inner product of these vectors may be expressed as

$$\underline{x} \cdot \underline{y} = \begin{bmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{p-1} \\ \underline{x}_s \end{bmatrix} \cdot \begin{bmatrix} \underline{y}_0 \\ \vdots \\ \underline{y}_{p-1} \\ \underline{y}_s \end{bmatrix}. \tag{1.39}$$

The vectors $\underline{x}_i$ and $\underline{y}_i$ are owned by process i and this process also stores $\underline{x}_{s(i)}$ and $\underline{y}_{s(i)}$. Equation (1.39) can be expressed as

$$\underline{x} \cdot \underline{y} = \sum_{i=0}^{p-1} \left(\underline{x}_i \cdot \underline{y}_i + \underline{x}_{s(i)} \cdot \underline{y}_{s(i)} \right). \tag{1.40}$$

To form the sum in (1.40) a global reduction operation is required. This global communication can be implemented using MPI in either of two ways.

1. A global accumulation on one process followed by a broadcast of the accumulated result to each process by using functions `MPI_Reduce` and `MPI_Bcast` respectively.

2. A global accumulation on all processes by using function `MPI_Allreduce`.

The operations for global accumulation of a result on one process are referred as *reduction operations* and those for global accumulation on all processes are referred as *global reduction operations* (see §1.5 for details). The *global reduction operations* provide the functionality for consecutive implementation of *reduction operations* and *broadcast operations* and should yield the faster result.

1.7 Preconditioning

Preconditioning is simply a means of transforming a system of linear equations to an equivalent new system of linear equations such that both systems have the

same solution but the new system is easier to solve in some sense by an iterative method. There are applications and occasions where iterative methods may fail to converge or suffer from a slow rate of convergence (see details in §1.6.3, for example). Preconditioning of the corresponding systems may help to overcome such difficulties and improve the efficiency and robustness of the iterative methods. The convergence of any linear system is associated with the spectral properties of the corresponding coefficient matrix and a preconditioner is a matrix that attempts to transform the spectrum to a more favourable form for convergence. Therefore, instead of solving a linear system such as (1.11), a new left–preconditioned system

$$\mathcal{M}^{-1}\mathcal{A}\underline{u} = \mathcal{M}^{-1}\underline{b} \tag{1.41}$$

or the equivalent right–preconditioned system

$$\mathcal{A}\mathcal{M}^{-1}\underline{v} = \underline{b}\,, \qquad \underline{u} = \mathcal{M}^{-1}\underline{v} \tag{1.42}$$

could be solved. The problem of finding an efficient preconditioner lies in identifying a matrix $\mathcal{M}$ (the preconditioner) that should satisfy the following properties:

1. $\mathcal{M}$ is a good approximation of $\mathcal{A}$ in some sense.
2. The system $\mathcal{M}\underline{u} = \underline{v}$ is much easier and cheaper to solve than the original system.

By an efficient preconditioner we mean that it is possible to obtain a much faster convergence of the iterative method for the preconditioned system, in terms of the overall solution time, than for the original system. The preconditioner $\mathcal{M}$ may be chosen from algebraic techniques which can be applied to general matrices or may be problem dependent so as to exploit special features of the particular class of problem being solved. The problem–dependent preconditioners often prove to be very powerful, but there is still room for additional efficient techniques to be developed for large classes of problems.

Algebraic preconditioners tend to be based on direct solution methods where a part of the computation is skipped, for example an incomplete factorization of the

matrix $\mathcal{A}$ of the form $\mathcal{A} = LU - R$. Here L and U have the same (or similar) sparse structure as the *lower* and *upper* parts of $\mathcal{A}$ respectively and R is the *residual* or *error* of the factorization. This leads to the notion of incomplete LU (or ILU) factorization [5, 152], where the incomplete factors form the preconditioner $\mathcal{M} = LU$. In the context of iterative methods this requires the evaluation of the expression $\underline{v} = (LU)^{-1}\underline{u}$ for a given vector $\underline{u}$ at each iteration. The following two steps are required for this computation.

1. Obtain $\underline{w}$ by solving $L\underline{w} = \underline{u}$,

2. Compute $\underline{v}$ from $U\underline{v} = \underline{w}$.

A drawback of the ILU approach is that it has sequential steps which tend to increase the computational complexity in parallel either in terms of iteration count or, more usually, cost per iteration. A standard trick for exploiting parallelism lies in selecting and numbering all those unknowns first which are not directly connected with each other. This approach is known as red–black ordering and does allow parallel preconditioners. The simplest approach in parallel is just to make use of diagonal scaling.

There are a number of different forms in which a given preconditioner can be implemented. These different implementations do not affect the eigenvalues of the preconditioned matrix for a given preconditioner. However, the convergence behaviour also depends upon the eigenvectors which may vary considerably for different implementations, and hence the convergence behaviour could be different for different implementations. These implementations are briefly described below.

1. **Left Preconditioning**

   The left preconditioning algorithm is defined by the solution of the system (1.41). If $\mathcal{A}$ and $\mathcal{M}$ are symmetric then $\mathcal{M}^{-1}\mathcal{A}$ is not necessarily symmetric. However, if $\mathcal{M}$ is SPD then $< x, y >= (x, \mathcal{M}y)$ defines an inner product and $\mathcal{M}^{-1}\mathcal{A}$ is symmetric with respect to this new inner product. For such systems MINRES [134] or CG [4] (if $\mathcal{A}$ is SPD) can be used. The well known preconditioned CG is based on this observation. In the case of minimal residual

methods such as MINRES or GMRES [154] (see §1.6.3 for details), minimization of the preconditioned residual $\mathcal{M}^{-1}(\underline{b} - \mathcal{A}\underline{u})$ may be quite different from minimization of the original residual $(\underline{b} - \mathcal{A}\underline{u})$. This could have consequences on stopping criteria that are based on the norm of residual.

2. **Right Preconditioning**

   Solution of the system (1.42) defines the right preconditioning algorithm. Here $\mathcal{A}\mathcal{M}^{-1}$ may not be symmetric even if both $\mathcal{A}$ and $\mathcal{M}$ are symmetric. The stopping criteria for right preconditioned algorithms is based upon the error norm $\|\underline{v} - \underline{v}_k\|_2$ rather than the error norm $\|\underline{u} - \underline{u}_k\|_2$ (less than or equal to $\|\mathcal{M}^{-1}\|_2\|\underline{v} - \underline{v}_k\|_2$). Since right preconditioning affects the operator only, not the right–hand side, this may be more useful as compared to left preconditioning, for example, in software designed or for problems with multiple right–hand side.

3. **Split or Two–Sided preconditioning**

   In many cases it is possible that $\mathcal{M}$ is available only in the form of some factorization such as $\mathcal{M} = LU$. Here L and U are *lower* and *upper* triangular matrices. An iterative method applied to the system

   $$L^{-1}\mathcal{A}U^{-1}\underline{v} = L^{-1}\underline{b} \quad \text{with} \quad \underline{u} = U^{-1}\underline{v} \tag{1.43}$$

   defines the split or two–sided preconditioner. This form of preconditioning may be more useful for preconditioners which are available only in the form of a factorization and may be seen as a compromise between left and right preconditioning.

4. **Flexible Preconditioning**

   This form of preconditioning is a variant of left or right preconditioning which provides the flexibility for the preconditioning matrix $\mathcal{M}$ not to be necessarily a constant operator at each iteration of the solver. A number of variants of iterative procedures have been developed in the literature that accommodate

variations in the preconditioning operator at each iteration [150, 152]. Such iterative procedures are known as *flexible* iterations or preconditioners. The additional cost associated with flexible variants over standard algorithms is an extra amount of memory to save an additional vector (of the size of the problem) and, more importantly, the possibility of breakdown.

There exist many successful techniques for preconditioning a sparse linear system, and there is virtually no limit on the available options for designing a good preconditioner. The cost of most preconditioners is such that the work per iteration is multiplied by a constant factor only. Similarly, with the exception of a few preconditioners (such as preconditioners based on modified incomplete factorization and/or multigrid techniques [13]) the reduction in the number of iterations to reach convergence is also only usually by a constant factor. There is a further trade–off between the efficiency of a preconditioner in a classical sense and its parallel efficiency. Many popular preconditioners (such as ILU, discussed above) have large sequential components. In the following summary therefore, we outline only a few additional preconditioners which show the most potential for efficient parallel implementation.

1. **Element–by–Element Preconditioning**

   For finite element problems, the assembly of the full coefficient matrix in not always possible or necessary and the product of this matrix with vectors is as simple and easy to compute if the coefficient matrix is not assembled. This also helps with parallelism by distributing the corresponding matrix–vector products across the processes. In such a situation, preconditioners should also be developed at the element level. Some of the first element–by–element preconditioners are reported in [102]. For a symmetric positive definite coefficient matrix $\mathcal{A}$, the corresponding element matrices $\mathcal{A}_e$ are decomposed as $\mathcal{A}_e = L_e L_e^T$ to construct the preconditioner $\mathcal{M} = LL^T$ where $L = \sum_{e=0}^{E-1} L_e$ and E is the total number of elements in the mesh. The parallelism in this approach is exploited in the simultaneous treatment of non–adjacent elements [91].

2. **Polynomial Preconditioning**

   Since the matrix–vector products generally offer more parallelism than other parts of an iterative solver, polynomial preconditioning seeks to exploit this in order to improve the parallel performance of the solver. The approach is based upon solution of $p_k(\mathcal{A})\mathcal{A}\underline{u} = p_k(\mathcal{A})\underline{b}$, where $p_k(\mathcal{A})$ is a polynomial that needs to be defined (the choice of an effective low degree polynomial may be problematic). Here m steps of a Krylov subspace iterative solver lead to a solution from a higher dimensional space (but with *holes*) with an overhead cost of m iteration steps only. For a suitable choice of polynomial this high dimensional space with holes may approximate the solution almost as well as the full Krylov subspace. In this case the overhead associated with a large number of iteration steps has been saved. On the other hand, the holes may cause the iteration to miss important search directions, ultimately leading to a higher iteration count. A number of general polynomial preconditioners are discussed in [3, 152].

3. **Sparse Approximate Inverse Preconditioning**

   The explicit inverse of a sparse matrix is always dense even if some of its entries are actually zero, that is, it is structurally dense. However, if we are able to compute a good sparse approximation to the inverse it is possible to eliminate this problem of it being dense. Perhaps the most simple technique to approximate the inverse of a general sparse matrix is to solve the problem

   $$\min_{\mathcal{M}} \|\mathcal{I} - \mathcal{A}\mathcal{M}\|_F^2 \tag{1.44}$$

   where the Frobenius norm of the matrix $\mathcal{A}$ is defined by $\|\mathcal{A}\|_F = \sqrt{\sum_{ij} a_{ij}^2}$. The resulting matrix $\mathcal{M}$ corresponds to the right approximate inverse of $\mathcal{A}$ and the left approximate inverse can similarly be computed from

   $$\min_{\mathcal{M}} \|\mathcal{I} - \mathcal{M}\mathcal{A}\|_F^2. \tag{1.45}$$

   For preconditioning purposes the structure of $\mathcal{M}$ is constrained to be fully, or at least partially, sparse. These minimization problems can in turn be expressed as a number of independent least–square subproblems each involving

only a few variables. As these subproblems are independent of each other, they can be solved in parallel [45]. It is also possible to successively increase the density of the approximation $\mathcal{M}$ to reduce the values of (1.44) and (1.45) in order to ensure the faster convergence of the preconditioned iterative method [47]. A number of different parallel implementations of sparse approximate inverse preconditioners are described in [12, 16, 15, 90]. These sparse approximate inverse preconditioners may require large numbers of iterations to converge, however, they are reasonably robust and can perform well when standard methods may fail.

4. **Domain Decomposition Preconditioning**

   There is a long history behind the construction of preconditioners and it is difficult to construct efficient general purpose preconditioners for wide classes of problems. This becomes even more difficult when preconditioners have to be in parallel. Recently there has been a renewed focus on coarse–grain parallel preconditioners: the sparse approximate inverse techniques outlined above but also particularly on domain decomposition methods. Domain decomposition techniques have been in use for many years, going as far back as 1870 [156]. These techniques have been successfully applied to large classes of problems, especially those governed by elliptic partial differential equations [27, 26, 28, 29, 30, 36, 44, 57, 59, 160]. As domain decomposition preconditioning is the main subject of this thesis, its introduction is postponed to the following chapter where the key ideas are described in detail.

1.8 Contents of Thesis

This thesis is concerned with the development of new efficient domain decomposition preconditioning techniques of additive Schwarz type which are appropriate for the parallel adaptive finite element solution of elliptic partial differential equations in three dimensions. The novel feature of the presented preconditioning techniques is that the refinement of a single layer of elements in the overlap region is sufficient to

yield optimal preconditioners.

The presentation of this thesis is organized in the following manner. Having introduced the basic concepts of finite element methods, mesh adaptivity and hierarchies, parallel computer architectures, direct and iterative solvers, and some general preconditioning methods in Chapter 1, we will discuss some of the most well–known domain decomposition solution methods in Chapter 2. These include multiplicative and additive Schwarz methods, direct and iterative substructuring methods and the Finite Element Tearing and Interconnecting (FETI) method. Some mesh and graph partitioning techniques, together with a few related public domain software packages, will be discussed.

Chapter 3 consists of our first symmetric weakly overlapping two level additive Schwarz preconditioner. In the early part of this chapter we introduce a model problem, describe the related background theory and then the development of the preconditioning algorithm itself. This is followed by various implementation details, particularly the restriction operation, the preconditioning solve and the interpolation operation. Then we present numerical evidence in support of the theoretical optimality, showing that theory works in practice. For this purpose, test problems with global uniform as well as local non–uniform refinement are considered.

We extend these ideas in Chapter 4 and develop a generalized nonsymmetric preconditioning algorithm. This new algorithm outperforms the original symmetric algorithm and has a number of advantages. It can also be applied to nonsymmetric and convection–dominated problems. For the convection–dominated problems, a stabilized finite element method, the streamline–diffusion method, is implemented and the results obtained show its effectiveness over the Galerkin method. The issue of local refinement for convection–dominated problems is also addressed.

Parallel performance of the generalized nonsymmetric preconditioning algorithm, considered due to its superiority in all respects over the original symmetric algorithm, is assessed in Chapter 5. A few issues of concern such as subdomain shape, inter–process communication, load imbalance and parallel overhead are also addressed. The numerical results show the obtained timings and speedups for a variety of test

problems.

Finally, Chapter 6 concludes the thesis with suggestions on how to improve the parallel performance and a number of possible extensions to the existing research work.

Chapter 2

Domain Decomposition

In this chapter we provide an overview of domain decomposition (DD) solvers and explain their connection with the finite element method and the use of parallel computers. Techniques for decomposing domains based upon mesh and graph partitioning, along with some related software tools, are also discussed.

2.1 Domain Decomposition Solution Methods

The DD approach provides a powerful strategy for obtaining efficient solution methods for partial differential equations. Historically, particular attention has been given to the application of DD methods for linear elliptic problems, based on the partitioning of the physical domain of the problem. In [160], the phenomena of DD is defined as:

> ***Domain decomposition refers to the process of subdividing the solution of a large linear system into smaller problems whose solution can be used to produce a preconditioner (or solver) for the system of equations that results from discretizing the partial differential equation on the entire domain.***

Since the subdomains (smaller problems) can be handled independently, such methods are very attractive for parallel computing platforms. The development of parallel

computers is one of the main reasons for the increased popularity and importance of DD methods in recent years. Generally there are two main classes of DD strategy which depend upon the partition of the initial domain into subdomains, that is, whether the subdomains are overlapping or non–overlapping. In the case of overlap-

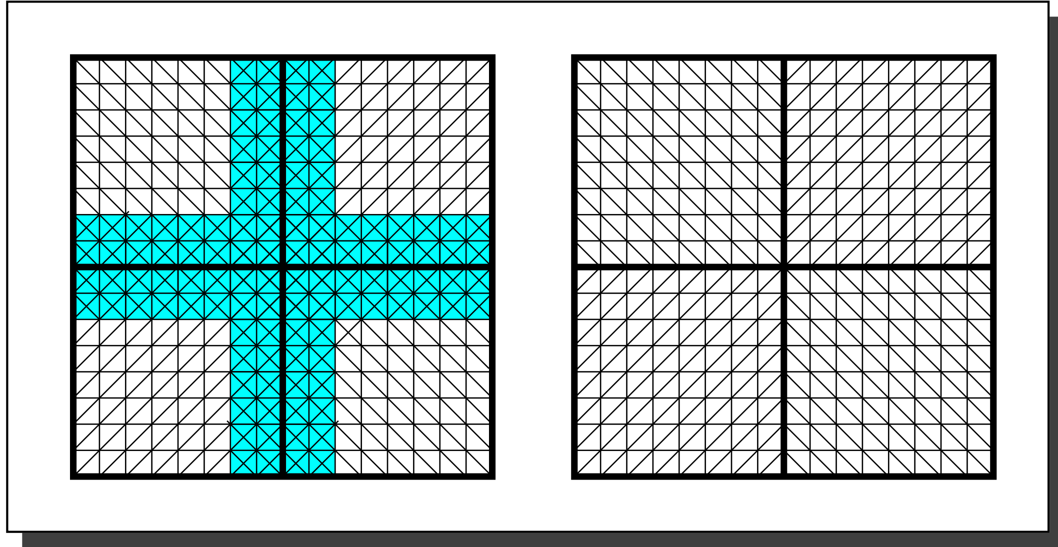

Figure 2.1: Decomposition into overlapping and non–overlapping subdomains.

ping subdomains, variables in the overlap regions are associated with more than one subproblem whereas, when non–overlapping subdomains are used, only variables on the subdomain interfaces are associated with more than one subproblem. The main advantage gained from using overlapping regions is that it can lead to a better rate of convergence, however the size of subdomain problems is clearly greater. In [22] it is shown that the distinction between these two classes of DD methods is not actually that fundamental and that the same technical tools can be used for the analysis of both of these methods. Hence, we consider an alternative categorization of DD solution techniques as follows.

1. **Schwarz Alternating Methods**, in which subdomains are generally overlapping (also known as Schwarz Alternating Procedures). The original alternating procedure was introduced in 1870 by H. A. Schwarz [156]. Today the DD literature is full of variants of this procedure: two of the most important (described in more detail below) being

   (a) **Multiplicative Schwarz Procedures**, which are an analogue of the block Gauss–Seidel Method.

   (b) **Additive Schwarz Procedures**, which are analogue of the block Jacobi Method.

2. **Schur Complement Methods**, in which subdomains are non–overlapping (also known as substructuring methods). The Schur complement methods may also be further divided into two types of solver (also described in more detail below):

   (a) **Direct Substructuring Methods**, in which a direct solution method is employed.

   (b) **Iterative Substructuring Methods**, in which a preconditioned Krylov subspace iterative method is employed.

Each of the above mentioned Schwarz and Schur complement methods is used to derive a parallel preconditioner or solver based on the DD for the system of linear equations (1.11). In the following two sections we outline the general principles for Schwarz alternating procedures for overlapping subdomains and Substructuring methods for non–overlapping subdomains respectively. A short survey of DD methods is then presented in §2.4, whereas in §2.5 we briefly describe mesh partitioning techniques followed by some well known graph partitioning techniques in §2.6, where some public domain software tools for graph partitioning are also discussed.

2.2 Schwarz Alternating Methods

The original idea of an alternating procedure, due to Schwarz [156], is for two subdomains only and comprises of the following three components:

1. Alternating between two subdomains.

2. Solving a Dirichlet problem at each iteration on one subdomain only.

3. Using the most recent solution from one subdomain to obtain boundary conditions to solve a Dirichlet problem on the other subdomain.

This procedure resembles the block Gauss–Seidel iterative solution method in that the most recent solution is always used, and is known as a multiplicative Schwarz procedure. A modification to this original procedure that is analogous to the block Jacobi iterative solution method, known as the additive Schwarz procedure, is suitable for parallel implementation. In this version steps 2 and 3 above are solved independently and may therefore be solved concurrently. It turns out that in practice both of these procedures are best used as preconditioners to other iterative solution methods. Furthermore, in the last twenty years or so, people from disciplines including the physical sciences, engineering and particularly from scientific computing have introduced many valuable variants of these procedures. The basic description of these two procedures is presented in the following subsections.

2.2.1 Multiplicative Schwarz Procedure

Let the computational domain Ω of the problem (1.1) be divided into an arbitrary number of subdomains, say p, Ω_i such that $\Omega = \cup_{i=0}^{p-1}\Omega_i$ and each subdomain overlaps with its neighbouring subdomains (i.e. has at least one shared interior point). Let N_i be the set of subdomains Ω_j which are neighbours to the subdomain Ω_i, then the multiplicative Schwarz procedure can be described as a sequence of the following steps:

1. Loop over the subdomains Ω_i for $i = 0, \ldots, p-1$.

2. For subdomain Ω_i determine (update) boundary conditions for the local problem using the most recent solution values from subdomains $\Omega_j \in N_i$.

3. Solve local problem over the subdomain Ω_i.

It should be noted that in step 2 above the first subdomain Ω_i will use an initial guess to determine the boundary conditions instead of the most recent solution from the subdomains $\Omega_j \in N_i$. After the local problem over the first subdomain

Ω_i is solved, a combination of the initial guess and the most recent solution from the neighbouring subdomains will be used until the solve for the last subdomain where the most recent solution from all neighbouring subdomains is available. This situation takes place only during the first sweep through the subdomains and for any subsequent sweeps over the subdomains the most recent solution is available from all of the neighbouring subdomains. As with most iterative solution methods requiring an initial guess to start with, a good choice of this initial guess will solve the problem in fewer iterations. Iterative solution methods typically obey the principal of correction by adding a *correction* to the approximated solution at each iteration. Mathematically this can be expressed as

$$\underline{u}_i := \underline{u}_i + \underline{\delta}_i \tag{2.1}$$

where $\underline{u}_i$ is the approximate solution vector of the local discrete problem on subdomain Ω_i and $\underline{\delta}_i$ is a correction vector. This may be obtained by solving a local system of the form

$$A_i \, \underline{\delta}_i \; = \; \underline{r}_i \tag{2.2}$$

on subdomain Ω_i, with $\underline{r}_i$ the local part of the global residual vector ($\underline{b} - \mathcal{A}\underline{u}$), where $\underline{b}$ is defined by (1.16), corresponding to the linear system of equations (1.11). A point to be noted here is that after solving the local problem on subdomain Ω_i the parts of the global residual vector which belong to the neighbouring subdomains $\Omega_j \in N_i$ need to be updated. Hence the above sequence for the multiplicative Schwarz procedure can be rephrased as:

1. Loop over the subdomains Ω_i for $i = 0, \ldots, p-1$.
2. Determine (update) boundary conditions for the local problem using the most recent values from subdomains $\Omega_j \in N_i$.
3. Solve the local system $A_i \underline{\delta}_i = \underline{r}_i$ for $\underline{\delta}_i$.
4. Update the local approximate solution $\underline{u}_i := \underline{u}_i + \underline{\delta}_i$.
5. Update the local residual $\underline{r}_j$ for $\Omega_j \in N_i$.

An alternative to this procedure is obtained by eliminating all variables which correspond to the interior (non–overlapping part) of the subdomains. Such a multiplicative Schwarz procedure, which would directly compute the solution in the overlap regions between subdomains only, would resemble a form of the block Gauss–Seidel iteration on the Schur complement problem for a consistent choice of the initial guess [41]. The Schur complement problem is described in §2.3.

Finally we make the following observations on the applications of multiplicative Schwarz procedures:

- The multiplicative Schwarz procedure can be used as a preconditioner for other iterative solvers.
- When used as a preconditioner for a symmetric problem, the symmetry is lost in the preconditioned system. A sweep over the subdomains Ω_i for $i = 0, \ldots, p-1$ followed by another sweep in the reverse direction recovers the symmetry however.
- The multicolouring of subdomains for the multiplicative Schwarz procedure can be exploited in a similar way to the block symmetric successive overrelaxation (SSOR) method. By assuming that any two adjacent subdomains have different colours then subdomains of the same colour have no coupling. In this situation the standard SSOR preconditioner can be used to solve the block system derived from the original problem (1.11). Similarly, a Schwarz multiplicative procedure can be applied for the preconditioning of such a system. The colour based decoupling of subdomains is one of the necessary features for efficient parallelism.

2.2.2 Additive Schwarz Procedure

The additive Schwarz procedure is a variant of the original multiplicative Schwarz procedure and is equivalent to a block Jacobi solution method. The additive Schwarz procedure updates the boundary conditions for all subdomains together once the

previous sweep over all subdomains is complete. This makes it slower to converge but ideally suited for implementation on a parallel computer since each subdomain solve is independent. An iteration of the additive Schwarz procedure can be expressed as:

1. Determine (update) boundary conditions for all subproblems.
2. Loop over the subdomains Ω_i for $i = 0, \ldots, p-1$.
3. Solve local problem over subdomain Ω_i.

The boundary conditions for all of the local problems in step 1 use the initial guess for the first iteration and for any subsequent iterations the solution from the previous iteration is used. The boundary conditions are updated for all subproblems from the local approximate solution of neighbouring subdomains obtained in the previous iteration. As with the multiplicative Schwarz method it is possible to use the principle of correction whereby, after completing a sweep over the subdomains, the solution from each local problem is used to update the global solution as follows:

$$\underline{u} := \underline{u} + \sum_{i=0}^{p-1} \underline{\delta}_i, \tag{2.3}$$

where $\underline{u}$ is the approximate solution vector of the global problem on the whole domain Ω and $\underline{\delta}_i$ is the correction vector contributed by each of the local problems. This is obtained by solving a local linear system

$$A_i \underline{\delta}_i = \underline{r}_i \tag{2.4}$$

on subdomain Ω_i with $\underline{r}_i$ representing the part of the global residual vector $(\underline{b} - \mathcal{A}\underline{u})$, where $\underline{b}$ is defined by (1.16), of the global system of equations (1.11) corresponding to the local problem on subdomain Ω_i. Unlike the multiplicative Schwarz procedure, the residual vector for the global system only has to be updated once after the solution for all subproblems is obtained, hence it does not matter whether the local problems are solved sequentially (in any order) or concurrently. The additive Schwarz procedure in this revised form, therefore, can be described as:

1. Determine (update) boundary conditions for all subproblems using available values from the previous iteration.

2. Loop over the subdomains Ω_i for $i = 0, \ldots, p-1$.

3. Solve the local problem $A_i \underline{\delta}_i = \underline{r}_i$ for $\underline{\delta}_i$.

4. Loop over the subdomains Ω_i for $i = 0, \ldots, p-1$.

5. Update the global approximate solution $\underline{u} = \underline{u} + \underline{\delta}_i$.

As noted earlier, due to the absence of data dependency amongst the local problems, this procedure can be implemented in parallel as follows:

1. Determine (update) boundary conditions for all subproblems using available values from the previous iteration.

2. Solve the local problems $A_i \underline{\delta}_i = \underline{r}_i$ for $\underline{\delta}_i$ in parallel.

3. Update the global approximate solution $\underline{u} = \underline{u} + \sum_{i=0}^{p-1} \underline{\delta}_i$.

Obviously there is no need of a loop over subdomains as the local problems are solved in parallel in step 2 but a global reduction operation, as described in §1.5.3 and implemented in §1.6.4, is required in step 3 to sum the contributions from each local problem in order to update the global approximate solution vector.

2.3 Schur Complement Methods

In this section we discuss the use of Schur complement methods to solve a linear system of equations (1.11) on a parallel computer. Let us begin with the assumption that a finite element mesh is defined on the domain Ω and a partition of this domain into p subdomains Ω_i, for $i = 0, \ldots, p-1$, is defined through the partition of the mesh such that $\Omega = \cup_{i=0}^{p-1} \Omega_i$, that is, each subdomain has a complete mesh suitable for solving finite element problems. Although this notation is valid for both cases of overlapping and non–overlapping (disjoint) subdomains, Schur complement methods

are more appropriate for non–overlapping subdomains and we shall discuss them in this context.

As explained above DD methods attempt to solve the system of linear equations (1.11) on the entire domain Ω by using the solution of the subproblems on the subdomains Ω_i, for $i = 0, \ldots, p-1$. For a suitable partition of the domain Ω (consistent with a partition of a global mesh) into p subdomains, the corresponding system of linear equations in block matrix notation is given by (1.28) and may also be written as:

$$\begin{bmatrix} A_I & B_I \\ C_I & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_I \\ \underline{u}_s \end{bmatrix} = \begin{bmatrix} \underline{b}_I \\ \underline{b}_s \end{bmatrix}, \tag{2.5}$$

where

$$A_I = \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & \ddots & \\ & & & A_{p-1} \end{bmatrix} \tag{2.6}$$

is a block matrix corresponding to the interior nodes of all subdomains Ω_i for $i = 0, \ldots, p-1$, A_s is the block matrix corresponding to all nodes at the interfaces between subdomains,

$$B_I = \begin{bmatrix} B_0 \\ B_1 \\ \ldots \\ B_{p-1} \end{bmatrix} \tag{2.7}$$

represents the coupling of all subdomains to the interfaces and

$$C_I = \begin{bmatrix} C_0, C_1, \ldots, C_{p-1} \end{bmatrix} \tag{2.8}$$

represents the coupling of interfaces to all subdomains.

2.3.1 Direct Substructuring Methods

The early applications of *substructuring* or *Schur complement* methods exploited a direct solution framework from where the name of this method is borrowed. Consider

the system of linear equations (2.5) which can further be expressed in the form

$$A_I \underline{u}_I + B_I \underline{u}_s = \underline{b}_I \tag{2.9}$$

$$C_I \underline{u}_I + A_s \underline{u}_s = \underline{b}_s \tag{2.10}$$

The substitution of equation (2.9) into equation (2.10) yields

$$C_I \left(A_I^{-1} (\underline{b}_I - B_I \underline{u}_s) \right) + A_s \underline{u}_s = \underline{b}_s. \tag{2.11}$$

On further simplification this equation takes the form

$$S \underline{u}_s = \underline{g} \tag{2.12}$$

where

$$S = A_s - C_I A_I^{-1} B_I \tag{2.13}$$

and

$$\underline{g} = \underline{b}_s - C_I A_I^{-1} \underline{b}_I. \tag{2.14}$$

The matrix S defined by equation (2.13) is known as the *Schur complement matrix* associated with the variables at the interface between subdomains and the corresponding system (2.12) is known as *Schur complement system* or *reduced system.* The matrix A_I associated with this Schur complement system is a block diagonal matrix given by (2.6), where each block A_i is associated with the interior variables of subdomain Ω_i for $i = 0, \ldots, p-1$. This phenomena naturally decouples the global problem into p independent subproblems which can be solved concurrently. In order to solve the interface problem, we need to assemble the Schur complement matrix S given by (2.13) which involve the inversion of the block diagonal matrix A_I and is given by

$$A_I^{-1} = \begin{bmatrix} A_0^{-1} & & & \\ & A_1^{-1} & & \\ & & \ddots & \\ & & & A_{p-1}^{-1} \end{bmatrix}. \tag{2.15}$$

Now substitution of equations (2.7, 2.8, 2.15) into (2.13) gives

$$\begin{aligned} S &= A_s - \left[\begin{array}{c} C_0, C_1, \ldots, C_{p-1} \end{array} \right] \left[\begin{array}{cccc} A_0^{-1} & & & \\ & A_1^{-1} & & \\ & & \ddots & \\ & & & A_{p-1}^{-1} \end{array} \right] \left[\begin{array}{c} B_0 \\ B_1 \\ \ldots \\ B_{p-1} \end{array} \right] \\ &= A_s - \left[\begin{array}{c} C_0, C_1, \ldots, C_{p-1} \end{array} \right] \left[\begin{array}{c} A_0^{-1} B_0 \\ A_1^{-1} B_1 \\ \ldots \\ A_{p-1}^{-1} B_{p-1} \end{array} \right] \\ &= A_s - \sum_{i=0}^{p-1} C_i A_i^{-1} B_i. \end{aligned} \tag{2.16}$$

From this equation the *local Schur complement matrix* for the subdomain Ω_i can be defined as

$$S_i = A_{s(i)} - C_i A_i^{-1} B_i \tag{2.17}$$

where $A_{s(i)}$ is the contribution to A_s from subdomain Ω_i, for $i = 0, \ldots, p-1$. Therefore the global Schur complement matrix S for the linear system (1.11) can be expressed in terms of local Schur complement matrices S_i, corresponding to linear systems on subdomains Ω_i, for $i = 0, \ldots, p-1$, as follow:

$$S = A_s - \sum_{i=0}^{p-1} C_i A_i^{-1} B_i = \sum_{i=0}^{p-1} \left(A_{s(i)} - C_i A_i^{-1} B_i \right) = \sum_{i=0}^{p-1} S_i. \tag{2.18}$$

This leads to the observation that the Schur complement matrix can be formed from the smaller Schur complements matrices from the subdomains. However this is an expensive operation due to the need to calculate $A_i^{-1} B_i$ on each subdomain. Nevertheless the system of linear equations expressed by the equations (2.12) can be solved, as suggested in [152], in the following five steps:

1. Factorize A_i on each subdomain Ω_i for $i = 0, \ldots, p-1$.

2. Compute $A_i^{-1} B_i$ and $A_i^{-1} \underline{b}_i$ on subdomain Ω_i for $i = 0, \ldots, p-1$.

3. Form $\sum_{i=0}^{p-1} S_i$ to get the *global Schur complement matrix* and the right–hand side $\underline{b}_s = \sum_{i=0}^{p-1} C_i A_i^{-1} \underline{b}_i$ where a *global reduction operation* is required.

4. Solve the *global Schur complement system* given by equation (2.12).

5. Compute remaining unknowns on each subdomain using backward substitution (as A_i has been factorized).

Note that on a parallel machine the solution of the *Schur complement problem* in step 4, may become a serial bottleneck. Many different methods could be used for this dense system of equations (including a parallel direct solver, e.g. [53, 62]). Note that by using a direct solution method as described here the solution of the linear subsystem corresponding to each subdomain must be exact. This is not necessarily the case when iterative solution methods are used, as discussed in the following section.

2.3.2 Iterative Substructuring Methods

As we can see from the previous section, direct substructuring methods have the drawback in both time and memory of requiring exact factrorizations of the subdomain matrices A_I. The use of an iterative scheme can avoid this cost. Furthermore, the use of a preconditioner with iterative solution techniques can make the convergence process fast and increase the algorithm's efficiency. Possible iterative techniques and parallel preconditioners exploiting the Schur complement system of linear equations for the interface variables between subdomains are considered here (see also [64, 117, 153, 161]). Recall that in the previous section the Schur complement system, also called the reduced system, has been derived. This system (2.12) is generally dense with the coefficient matrix given by (2.13) which, in terms of local Schur complement matrices, is also given by (2.18). Similarly, the right–hand side (2.14) of this system can be expressed as

$$\underline{g} = \sum_{i=0}^{p-1} \left(\underline{b}_{s(i)} - C_i A_i^{-1} \underline{b}_i \right) . \tag{2.19}$$

Application of any Krylov subspace method such as CG or GMRES to solve the Schur complement system (2.12) does not necessarily require the explicit assembly of the Schur complement matrix S. This is very important from the computational point of view as the explicit construction of this matrix, as described in the previous section, is very expensive. Instead these methods only need the action of multiplication of the matrix with a given direction vector, v_s say. Thus from equation (2.18), we have

$$Sv_s = \sum_{i=0}^{p-1} \left(A_{s(i)} - C_i A_i^{-1} B_i \right) v_s. \tag{2.20}$$

This product, in more appropriate form, can be written as:

$$Sv_s = \sum_{i=0}^{p-1} A_{s(i)} v_s - \sum_{i=0}^{p-1} C_i \left(A_i^{-1} \left(B_i v_s \right) \right) \tag{2.21}$$

and may therefore be obtained using only local matrix–vector products and subdomain solves. Each of these operations, on each subdomain, may be performed in parallel.

The benefit of Schur complement methods is that they require the solution of a smaller system than the original system. In fact, for a small number of subdomains, the interface between subdomains is small and therefore the size of interface problem is small. However, with an increase in the number of subdomains the interface between the subdomains usually becomes large and so does the size of the corresponding interface problem. Due to this increase in the number of subdomains and size of interface problem, an iterative solver may require more iterations to converge. Therefore a good preconditioner may be required to solve the system (2.18) on a finite element mesh with very large number of degrees of freedom and large number of subdomains in order to obtain a reasonable rate of convergence [57, 97, 159].

2.3.3 Finite Element Tearing and Interconnecting Method

In this section we briefly explain the Finite Element Tearing and Interconnecting (FETI) Method which belongs to the class of methods referred as non–overlapping DD methods [160] and also referred as conjugate gradient based substructuring

methods [77]. These methods solve the interface problem (2.12), usually by an iterative method such as CG, as this avoids the explicit formation of the Schur complement matrix (2.13). The substructuring approach combines both direct and iterative solvers (for the subdomain and outer solves respectively) and can have a better parallel scalability [18, 75] than the CG method applied directly to the global system (since the computational work on each process is higher for the same amount of communication).

The FETI method, or DD based iterative method with Lagrange multipliers, can be considered as a two step preconditioned CG algorithm where problems with Dirichlet boundary conditions are solved in the preconditioning step and related problems with Neumann boundary conditions are solved in a second step. An auxiliary problem based on the subdomain rigid body modes acts as a coarse problem to propagate the error globally during the preconditioned CG iteration, thus accelerating the convergence. Recall that in §2.2.1 N_i is defined as the set of subdomains Ω_j which are neighbours of subdomain Ω_i. Similarly, let N_j for $j \neq i$ be the set of subdomains Ω_k which are neighbours to the subdomain Ω_j. A variational form of the problem (1.1) can be expressed in terms of the following algebraic system of equations [70, 77]:

$$A_i \underline{u}_i = \underline{b}_i + \sum_{\Omega_j \in N_i} B_j^T \underline{\lambda} \qquad \text{for } i = 0, \ldots, p-1 \tag{2.22}$$

$$B_i \underline{u}_i = B_j \underline{u}_j. \tag{2.23}$$

Here the matrices B_j are signed boolean matrices which map a local vector $\underline{u}_j$ to the corresponding entries in the global subdomain interface vector. Also, A_i, $\underline{u}_i$ and $\underline{b}_i$ (for $i = 0, \ldots, p-1$) are the stiffness matrix, the vector of unknowns and the right–hand side vector respectively associated with finite element discretization of the subdomain Ω_i. The vector of Lagrange multipliers $\underline{\lambda}$ is the interaction between neighbouring subdomains along their common boundary and the number of Lagrange multipliers is equal to the total number of unknowns at the global subdomain interface. Equation (2.23) ensures equal solution values at nodes shared between neighbouring subdomains and provides an additional equation to help in

solving equations (2.22).

We first consider the simple case where A_i is non–singular for $i = 0, \ldots, p-1$, then equation (2.22) can be rewritten as

$$\underline{u}_i = A_i^{-1}\left(\underline{b}_i + \sum_{\Omega_j \in N_i} B_j^T \underline{\lambda}\right) \qquad \text{for } i = 0, \ldots, p-1. \tag{2.24}$$

Substitution of this equation in (2.23) yields

$$B_i A_i^{-1}\left(\underline{b}_i + \sum_{\Omega_j \in N_i} B_j^T \underline{\lambda}\right) = B_j A_j^{-1}\left(\underline{b}_j + \sum_{\Omega_k \in N_j} B_k^T \underline{\lambda}\right) \tag{2.25}$$

which may be rearranged,

$$\left(B_i A_i^{-1} \sum_{\Omega_j \in N_i} B_j^T - B_j A_j^{-1} \sum_{\Omega_k \in N_j} B_k^T\right) \underline{\lambda} = B_j A_j^{-1} \underline{b}_j - B_i A_i^{-1} \underline{b}_i, \tag{2.26}$$

and solved for $\underline{\lambda}$. Once $\underline{\lambda}$ is obtained, the vector of unknown variables can be obtained by substituting for $\underline{\lambda}$ in equations (2.22).

Unfortunately, for some problems the stiffness matrix for a subdomain may not always be non–singular. In this situation the solution process described above will break down. To overcome this situation, let us assume that subdomains Ω^D include a Dirichlet condition and subdomains Ω^N are pure Neumann (also known as floating subdomains). In this case the A_i for $\Omega_i \in \Omega^D$ are positive definite and the A_k for $\Omega_k \in \Omega^N$ are positive semi–definite. Thus any subsystem of equations in (2.22) corresponding to subdomains $\Omega_k \in \Omega^N$ is singular. Assuming that this system is consistent then a general solution can be written as:

$$\underline{u}_k = A_k^+\left(\underline{b}_k + \sum_{\Omega_i \in N_k} B_i^T \underline{\lambda}\right) + R_k \underline{\alpha} \tag{2.27}$$

where A_k^+ is a pseudo–inverse of A_k [14], the columns of R_k are the singular vectors of A_k and the vector $\underline{\alpha}$ indicates a linear combination of these vectors. The replacement of equation (2.22) by the equation (2.27) rather than (2.24) introduces a set of additional unknowns $\underline{\alpha}$, so the overall system of equations becomes underdetermined. Since A_k is symmetric, there exists at least one solution of the singular equation

(2.22) if and only if the right–hand side $\left(\underline{b}_k + \sum_{\Omega_j \in N_k} B_j^T \underline{\lambda}\right)$ has no component in the null space of A_k. This can be written as

$$R_k^T \left(\underline{b}_k + \sum_{\Omega_j \in N_k} B_j^T \underline{\lambda}\right) = 0 \tag{2.28}$$

which is an additional equation that is helpful in obtaining $\underline{\lambda}$, $\underline{\alpha}$ and the vectors of unknown variables $\underline{u}_i$. Combining together equations (2.22), (2.23), (2.27) and (2.28), the interface system can be written as:

$$\begin{bmatrix} F_I & -G_I \\ -G_I^T & 0 \end{bmatrix} \begin{bmatrix} \underline{\lambda} \\ \underline{\alpha} \end{bmatrix} = \begin{bmatrix} \underline{d} \\ \underline{e} \end{bmatrix} \tag{2.29}$$

where

$$F_I = \sum_{i=0}^{p-1} B_i A_i^+ B_i^T, \tag{2.30}$$

$$G_I = \left[B_0 R_0 \ldots\ B_{N_f-1} R_{N_f-1}\right], \tag{2.31}$$

$$\underline{d} = \sum_{i=0}^{p-1} B_i A_i^+ \underline{b}_i, \tag{2.32}$$

$$\underline{e}_i = \left[\underline{b}_i^T R_i\right]^T \qquad (i = 0, \ldots, N_f - 1), \tag{2.33}$$

and N_f is the number of floating subdomains. For full derivation details of this further reading, such as [69, 70, 76, 77, 79], is suggested. This system can be solved for $(\underline{\lambda}, \underline{\alpha})$ and the unknown variables in the interior of subdomains may then be obtained from equations (2.24) and (2.27). It is expensive to explicitly assemble the matrix F_I which makes the use of a direct methods to solve this system impractical. However, an iterative method such as CG may be efficient, although the indefiniteness of the Lagrangian precludes a straightforward application of CG. Alternatively, due to the symmetry of F_I, (2.29) can be formulated as a constrained minimization problem:

$$\min_{\underline{\lambda}} \Phi(\underline{\lambda}) = \frac{1}{2}\underline{\lambda}^T F_I \underline{\lambda} - \underline{\lambda}^T (G_I \underline{\alpha} + \underline{d}) \tag{2.34}$$

subject to

$$G_I^T \underline{\lambda} = \underline{e}. \tag{2.35}$$

As F_I is positive semi–definite, this problem can be solved by CG provided the value of $\underline{\lambda}$ at each iteration satisfies the constraint (2.35). Furthermore, suitable preconditioning of this problem [75, 114] can make the convergence faster, although this is not considered here.

2.4 Domain Decomposition: A Brief Review

Having outlined some of the major DD solution techniques we now put these into an historical context and provide details of further reading by giving a brief review of some of the literature in this (very large) field. We concentrate on the domain decomposition methods covered in §2.2 and §2.3 only, and consider them in the same order, that is, Schwarz methods, Schur complement methods and FETI methods. This review is not intended to be exhaustive by any means and the reader is referred to survey and review articles and recent monographs such as [42, 119, 141, 160, 177, 178] for still further details.

It is believed that the first domain decomposition method was introduced by Schwarz [156] and was used to show the existence of analytical solutions of elliptic boundary value problems on domains which consist of the union of simple overlapping subregions. It was assumed that the shape of these subregions was such that the elliptic problem in each subregion could be solved analytically. The case of two subregions is illustrated in [160], as shown in Figure 2.2.

The multiplicative Schwarz algorithm is a straightforward extension of the classical alternating Schwarz algorithm to an arbitrary number of subdomains once the order in which each subproblem is to be considered has been selected. However, this extension of the theory had some difficulties at the early stage of its development [31, 39, 177], since for large numbers of subdomains the convergence rate deteriorates rapidly (the condition number grows like $1/H^2$ where H is the diameter of the smallest subdomain). The additional solution of a global coarse grid problem is the most common mechanism for improving this convergence rate since it provides global communication of information across the subdomains at each it-

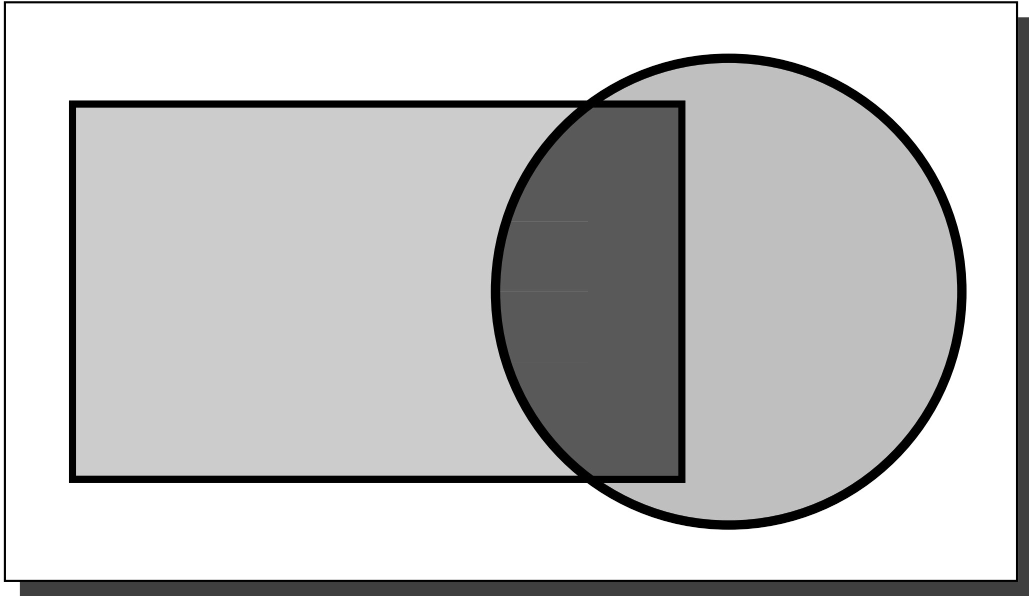

Figure 2.2: Schwarz's original figure.

eration. The multiplicative Schwarz algorithm is inherently a sequential algorithm although parallelism is possible by introducing a subdomain colouring such that no two subdomains of same colour are neighbours. The introduction of additive Schwarz methods [56, 124] removed completely the inherent sequential behaviour of multiplicative Schwarz methods. However, despite less scope for parallelism, multiplicative Schwarz methods are often superior to additive Schwarz methods because their algebraic convergence rate tends to be higher. A hybrid Schwarz preconditioner is also possible [35]: this aims to combine the advantages of the two methods, that is, faster convergence of multiplicative Schwarz and inherent parallelism of additive Schwarz methods.

The two level additive Schwarz methods (i.e. making use of a global coarse grid problem) are introduced in [57] and a few variants of these can be found in [35, 121]. In two level Schwarz methods, the local preconditioner may be either multiplicative or additive. Multilevel Schwarz methods also exist in the literature, [58] for example. Here, the coarse problem is itself solved by a two level method (repeating recursively to as many levels as required). A comprehensive study of multilevel algorithms and their convergence theory is carried out in [180]. BPX algorithms [32] are a special

case of the multilevel additive Schwarz method but are derived in a different way. An optimality proof for multilevel additive Schwarz algorithms is given in [132] and a generalization of both multilevel multiplicative and additive Schwarz methods appears in [177]. We refer to [87] for an abstract theory of both multiplicative and additive Schwarz algorithms.

Non–overlapping DD methods are defined on a decomposition of the domain which consists of mutually disjoint subdomains. We discuss here these methods in the context of preconditioning algorithms for Schur complement systems. These techniques, often known as substructuring methods, were first studied in [26, 29]. However, a similar adoption of DD for organizing large structural analysis problems has also been reported [140]. A key feature of substructuring methods that is missing in [140] however, is that they facilitate the design of effective parallel algorithms [160]. The original idea behind substructuring methods was to provide direct solution algorithms based on explicit computation and factorization of a sequence of Schur complement matrices. We refer to [48] for a survey article on Schur complement methods and the parallel implementation of direct substructuring methods is discussed in [20].

Since the original introduction of these direct substructuring methods a number of promising iterative alternatives have been introduced. For these methods, even without preconditioning, the condition number of discrete, second order elliptic systems is generally improved from $O(h^{-2})$ to $O(h^{-1})$ [21]. The method proposed in [43] shows that the convergence rate is independent of aspect ratios for several model problems and it is further shown in [22] that the these methods are identical to certain classical alternating Schwarz iterations. Similarly, an algebraic proof of the equivalence between certain overlapping Schwarz methods and the Schur complement iteration is given in [41]. Iterative substructuring algorithms may also be used with p–version finite element methods [6]. Furthermore, a hierarchical basis may be used along the interface [160] which introduces the possible use of multilevel preconditioners on the interface [166]. For a complete discussion of many iterative substructuring algorithms, including problems in three dimensions, we refer to [55]

whereas in [178] a unified investigation of non–overlapping DD methods is presented.

A more recent DD algorithm for the iterative solution of equations arising from the finite element discretization of self–adjoint elliptic PDEs, known as the FETI method, was introduced in [69, 77, 78]. A detailed introduction to this method can be found in [79], with significant theoretical work in [122]. For preconditioning within a unified framework for accelerating convergence of the FETI method see [70]. Here we briefly review the literature related to the advancement of DD methods which belong to the FETI class.

Extensive numerical tests are reported in [18, 148] for example, and the concept of scaling used in the FETI preconditioner is introduced in [147]. Similar scaling can also be found in [115]. More recently a dual–primal FETI algorithm [72] is introduced which is suitable for second order elliptic problems in the plane and for plate problems. However, numerical experiments show poor performance for this algorithm in three dimensions. Some more recent experiments, [73], with an alternative algorithm are however encouraging. The condition number bound established in [123] for the FETI method equipped with a Dirichlet preconditioner is of the form $C\,(1+\log(H/h))^2$, and a corresponding result for fourth order elliptic problems is also established. A proof for this condition number bound is provided in [123]. The same preconditioner is also defined for three dimensions but does not perform so well. This may be related to the poor performance of many vertex–based iterative substructuring methods [55], however the addition of a few constraints to this basic algorithm [73] can improve the performance.

2.5 Mesh Partitioning

A number of computational techniques, including the FEM discussed in §1.1 and §1.6, use unstructured meshes to solve the large–scale scientific and engineering problems such as those arising in computational fluid dynamics or computational mechanics for example. As seen in §1.6, due to their size, these problems are often solved in parallel and therefore require the partitioning of the mesh on the problem

domain. So far in this chapter we have considered DD solution techniques which assume that the problem domain has already been partitioned by elements. This partition takes place on a mesh which covers the domain and so in this section we discuss some of the main issues and heuristics associated with mesh partitioning.

To ensure the quality of a partition, so that the problem can be solved efficiently in parallel, it is desirable for a partition of the mesh M into p submeshes M_i (for $i = 0, \ldots, p-1$) to satisfy the following two main properties.

- The submeshes M_i should be of equal size so as to ensure the computational load is balanced.

- The inter–process communication should be minimized by keeping the interface between subdomains to a minimum.

Other features of a partition, such as good subdomain aspect ratios or subdomain inter–connectivity bandwidths, may also be required by some parallel solution algorithms (see, for example, [71]) but these will not be discussed here.

Assume that the domain of the problem has already been meshed into a number of coarse elements (for example, triangles in 2-d or tetrahedra in 3-d), where each element is a combination of edges and vertices. Generally this meshed domain is partitioned by using a graph representation of the mesh in one way or another and the resulting subdomains are assigned to distinct processes. There are therefore a number of choices over the type of partition and these are considered below.

- **Edge Based partition**

  This partition does not allow splitting of edges between subdomains and therefore each edge of the mesh should be assigned to a unique process. This type of partition is particularly suitable for finite volume techniques.

- **Vertex Based Partition**

  In this partition each vertex of the mesh is allocated to a unique process. Consequently, some elements and edges at the interface may be assigned to more than one process.

- **Element Based Partition**

  This type of partition requires the mapping of each element to a unique process, that is, elements should not be sliced between subdomains. All the information related to an element should therefore be mapped to the same process. Some edges and nodes will lie on the subdomain boundaries however.

For the parallel finite element solution described in §1.6 an element based partition is used. This is the type of partition that we will consider throughout this work. By defining a dual graph relating elements of the mesh with graph vertices and the adjacency of mesh elements with graph edges, the problem of element based mesh partitioning can be expressed as a conventional graph partitioning problem. In the following section, this issue of finite element mesh partitioning through graph partitioning is addressed.

2.6 Graph Partitioning

A graph G consists of a non–empty set of elements $V(G)$ and a subset $E(G)$ of the set of pairs of elements of $V(G) \times V(G)$. The elements $V(G)$ and $E(G)$ are known as vertices and edges of the graph G respectively, where any two vertices connected through an edge are said to be adjacent. If the edges are ordered pairs of vertices, the graph is known as a *directed graph* and if these edges are unordered pairs of vertices it is an *undirected graph*.

As discussed above, when we want to solve a PDE, such as (1.1), on a parallel computer we need to partition the computational mesh of the problem domain. The following two issues may be associated with such a partition.

1. Obtaining a suitable partition of the mesh.

2. Distributing the submeshes appropriately across the processes.

Graph partitioning techniques are related to the first issue only, whereas the second issue is architecture dependent. The task of graph partitioning is to subdivide the graph into a number of small subgraphs (which should be of approximately equal size

for the reasons discussed in §2.5). The problem of finding the partition of a graph with equal–sized subgraphs and the minimum possible number of edges crossing from one subgraph to another is NP–hard. For large, or even moderate sized, finite element grids finding the optimal solution is not practical therefore. Hence we tend to use heuristics: many of which appear in the literature [93, 112, 139, 170]. These techniques involve a variety of algorithms such as those based on geometric techniques, spectral techniques or graphical techniques. Some other heuristics which are harder to categorize are also summarized in this section. We also provide a short survey of some of the available graph partitioning software.

2.6.1 Geometric Techniques

1. **Recursive Coordinate Bisection**

   For recursive coordinate bisection (RCB) of a given undirected graph it is necessary that the coordinates of the vertices are available. A simple strategy to determine the coordinate direction of the longest expansion of the domain is introduced in [175] and generalized in [158]. Without loss of generality, it is assumed that this coordinate direction is along the x–axis. The nodes are then partitioned into two sets by assigning the half of the nodes with smallest x–coordinates to one subdomain and rest of the nodes to other subdomain. This process is repeated recursively on each subdomain. A straight observation is that this technique does not exploited the adjacency information of the graph and so it is unlikely that the resulting subdomains will have a low cut-weight (the cut–weight, also called edge–cut, is the term used for the number of edges crossing from one subgraph to another).

2. **Recursive Inertial Bisection**

   This is a generalization of the RCB technique whereby a point mass, as well as a position, is associated with each vertex of the graph and the resulting principal axis of inertial (PA) is calculated. The graph is then bisected by taking an orthogonal cut to the PA such that the total mass on either side

of the cut is approximately equal. This procedure is also repeated recursively for each subdomain. Although this technique is fast, the partitions produced generally still have a relatively high cut–weight [52].

3. **Stripe–wise Method**

   The sorting of the nodes in this method is similar to that in RCB however there is no restriction to a total of 2^n partitions. Instead, an arbitrary number of orthogonal cuts along a suitably chosen axis produces an arbitrary number of partitions of almost equal size. However, this stripe like decomposition of the mesh is not advised [95] because of its large cut–weight and its adverse affect on the scalability of typical parallel solvers, due to deteriorating aspect ratios. (In [95] isotropic problems are considered however for some anisotropic problems this partition may be preferable provided it is properly aligned.)

2.6.2 Spectral Techniques

1. **Recursive Spectral Bisection**

   This technique exploits the properties of eigenvectors of the Laplacian of a graph. For any given graph, the Laplacian matrix L is defined to have the following entry in row i, column j:

$$L_{ij} = \begin{cases} -1 & \text{if vertices } i \ \& \ j \text{ are connected,} \\ \text{degree of vertex } i & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

   Note that the degree of a vertex is equal to the number of graph edges radiating from it. Furthermore, when the graph is undirected the matrix is symmetric, and it may be shown that it is negative semidefinite with a single eigenvalue of zero (provided the graph is connected: there are more zero eigenvalues for disconnected graphs). When the graph is connected, the eigenvector associated with the second smallest (in magnitude) eigenvalue has some useful properties [96]. This eigenvector is called the *Fiedler vector* and the signs of its entries may be used to divide the domain into two roughly equal parts. The Recursive

Spectral Bisection (RSB) technique sorts the components of the Fiedler vector and then assigns the first half of the sorted vertices to one subdomain and the rest to the other subdomain. This procedure is repeated recursively until a required number of subdomains is produced.

2. **Multidimensional Spectral Graph Partitioning**

   By making a novel use of multiple eigenvectors [92], this generalization of RSB divides a graph into 4 (spectral quadrisection) or 8 (octasection) parts at once. It is shown in [94] that, for some problems, this multidimensional approach significantly out–performs standard RSB.

2.6.3 Graphical Techniques

1. **Recursive Graph Bisection**

   This technique is similar to RCB however the idea of graph distance is used instead of Euclidean distance. The graph distance between any two vertices v_i and v_j is defined as the number of edges in the shortest path connecting vertices v_i and v_j. The recursive graph bisection (RGB) algorithm begins by finding the diameter of the graph (or alternatively the pseudo-diameter, as the true diameter is expensive to find [85]). The nodes are sorted according to their graph distance from one of the extreme nodes of this diameter. One half of these sorted nodes are assigned to one subdomain and the remaining half to the second subdomain. This approach is then repeated recursively on each subdomain. If the original graph is connected then this algorithm guarantees that at least one of the two subdomains will be connected, however the other subdomain may not be. Thus the situation can arise where not all of the subdomains are connected.

2. **Double Stripping Technique**

   This technique uses two parameters, p_1 and p_2 say, such that the product of these two parameters is equal to the required number of subdomains ($p = p_1 \times p_2$). The original graph is first partitioned into p_1 subgraphs using a one

way partition. This consists of performing a level set traversal from a (pseudo) peripheral vertex and assigning an average number of consecutive nodes to each different subgraph in a graphical analogy of the strip–wise decomposition. Then each of these p_1 subgraphs are further subdivided into p_2 parts in a similar manner. For certain problems however, the subgraphs obtained using this technique may be quite long and twisted in shape, giving a higher number of edge–cuts than desired [152].

3. **Greedy Algorithm**

   The Greedy algorithm [68] builds its first subdomain without looking ahead. Once this is achieved it discards this subdomain and forms the second subdomain (again without looking ahead) and so on. Ultimately the quality of last subdomain is not generally very good in comparison with the early subdomains. This graph–based algorithm uses the level set principle as in the RGB technique where vertices are claimed by walking through the graph one level at a time, beginning with a (pseudo–) peripheral vertex.

2.6.4 Some Miscellaneous Techniques

1. **Recursive Node Cluster Bisection**

   This is an attempt to combine the features of RSB and graph coarsening techniques [96], based on the idea of node clustering [170]. Here, some of the connected vertices in a graph are clustered together to create a single, highly–weighted, vertex whose degree is equal to the number of vertices adjacent to the vertices forming the cluster. A generalized spectral bisection approach is then used on this weighted graph. This method can lead to some imbalance between subdomains and so the balance must be restored by a suggested recovery scheme [96].

2. **Kernighan–Lin Type Algorithms**

   An iterative graph partitioning algorithm is introduced in [113] which first divides a graph into an arbitrary number of equally–sized subgraphs. During

each subsequent iteration subsets of vertices are determined from each subgraph such that interchange of these subsets leads to an improved cut–weight. If such subsets exist then their interchange is carried out and this becomes the new partition, otherwise the algorithm terminates. The complexity of the algorithm is nonlinear in nature and each iteration takes $O(|E|^2 \log |E|)$ time, however a variant of this algorithm introduced in [80], reduces the complexity to $O(|E|)$.

2.6.5 Graph Partitioning Software

As indicated by the examples above, a large number of graph partitioning heuristics have been developed and the performance of these methods is typically influenced by various choices of parameters and details of their implementation. For this reason many public domain graph partitioning software packages (specializing in one or more particular aspect of graph partitioning) are now available. A few of these are briefly described below.

1. **Chaco**

   Chaco is a software package [93] designed for graph partitioning which allows the recursive application of several methods to find small edge separators in weighted graphs. It also provides the facility to improve the quality of an existing partition. The main issues addressed are:

   (i) Partition of graphs using a variety of methods with different properties.

   (ii) Embedding the generated partitions intelligently into different topologies such as hypercubes, meshes, etc.

   (iii) The use of spectral methods to sequence graphs such that locality is preserved.

   Finally, the software may be used either as a library or in stand–alone form.

2. **PARTY**

   The PARTY partitioning library [139] consists of a variety of different parti-

tioning methods that can be used either as a stand–alone program or included within an application code in the form of a library. In addition to these resources, PARTY, provides an interface to the Chaco software which can therefore be invoked from the PARTY environment.

3. **Jostle**

   The software package Jostle [172, 173] is build upon two main types of algorithm. The first is multilevel graph partitioning. The main idea is to take a large graph and construct a sequence of smaller and simpler graphs that in some sense approximate the original graph. When the graph is sufficiently small it is partitioned using some other method. This smallest graph and the corresponding partition is then propagated back through all the levels to the original graph. A local refinement strategy, such as Kernighan–Lin [113], is also employed at some or every level of this back–propagation. The second main strategy used is diffusion. This method assumes that an initial partition (balance) is given, and load balance is achieved by repeatedly moving objects (nodes) from partitions (processes) that have too heavy a load to neighbouring partitions (processes) with too small a load.

4. **METIS**

   Metis is another powerful software package [112] for the partition of large irregular graphs, large meshes and for computing fill reducing orderings of sparse matrices. The algorithms implemented in Metis are also based on multilevel graph partitioning and like Jostle can be described by the following three phases:

   (i) **Coarsening Phase** Numerous vertices and edges of the graph are collapsed in order to reduce the size of the graph.

   (ii) **Partitioning Phase** The smaller graph obtained after the coarsening phase is partitioned into the desired number of subgraphs.

   (iii) **Prolongation Phase** After coarsening and partitioning into subgraphs each of these subgraphs is interpolated onto a finer graph and the parti-

tion is locally improved. This is repeated until a partition of the original graph is recovered.

The quality of subgraphs is maximized by partitioning the graph at the coarsest level and then concentrating effort on the interface boundaries of the partitions during the prolongation process.

Chapter 3

A Symmetric Weakly Overlapping Additive Schwarz Preconditioner

In this chapter we discuss a two level additive Schwarz method for preconditioning elliptic problems in three dimensions. The basic techniques of finite element methods, mesh adaption, message passing and preconditioning, discussed in Chapter 1, and some DD methods, mesh partitioning and graph partitioning techniques, discussed in Chapter 2, are used. Together these lead to a new parallel DD preconditioner which is applicable for the adaptive finite element solution of partial differential equations representing a variety of elliptic problems defined on a bounded Lipschitz domain $\Omega \subset \mathcal{R}^3$. In §3.1 we introduce a model problem and this is followed by the theory and development of the preconditioner in §3.2 and §3.3 respectively. An overview of the implementation is then provided in §3.4. The following three sections, 3.5 to 3.7, focus on algorithmic and implementation details and the computational results for some test problems using global uniform and local non–uniform mesh refinement are presented in §3.8.

3.1 Introduction

Domain decomposition methods assume that the computational domain Ω is partitioned into p subdomains, Ω_i, for $i = 0, \ldots, p-1$, which may or may not overlap.

We are interested in a two level additive Schwarz algorithm which is suitable for parallel implementation. For such an algorithm the overlap between subdomains is generally some fraction of the subdomain size and the convergence rate depends on this amount of overlap between subdomains. A DD algorithm is considered to be optimal if the condition number of the corresponding algebraic system is independent of the number and size of the subdomains and the mesh size. Without the use of a coarse grid correction this condition number would increase quickly as the finite element mesh is refined, and so the convergence rate would start to deteriorate. With two level additive Schwarz methods the introduction of a relatively coarse mesh on the global domain, [54, 174], allows the global flow of error information across the domain so that the quality of the algorithm is preserved as the number and size of subdomains is increased or the mesh is refined. In this case the condition number of the preconditioned system grows very slowly, if at all (depending upon the amount of overlap), and therefore a faster convergence rate is obtained.

We describe a two level additive Schwarz algorithm which acts as an optimal DD preconditioner for the finite element solution of a class of second–order self–adjoint elliptic problems in three dimensions. For detailed background reading on the subject we refer to some of the recent publications on DD [119, 141, 160, 176, 178]. It is possible to develop a unified theory [59] for a variety of DD algorithms, particularly in the context of iterative techniques in terms of subspace corrections.

For some theoretical background and a discussion of the preconditioning algorithm that we propose here we consider the following model problem.

Find $u \in \mathcal{H}_0^1(\Omega)$ such that

$$\int_\Omega (\underline{\nabla} v \cdot (A \underline{\nabla} u)) \, d\Omega - \int_{\Gamma_2} g \; v \; dS = \int_\Omega f \; v \; d\Omega \qquad \forall \, v \in \mathcal{H}_0^1(\Omega) \tag{3.1}$$

where $\Omega \subset \mathcal{R}^3$ is the domain of the problem and

$$\mathcal{H}_0^1(\Omega) = \{ u \in \mathcal{H}^1(\Omega) : u_{|\Gamma_1} = 0 \}. \tag{3.2}$$

We note that equations (3.1) and (3.2) represent only the symmetric part of equation (1.6) and the corresponding Dirichlet boundary conditions respectively. Here we

consider this symmetric problem only and therefore the corresponding bilinear and linear forms can be expressed as

$$\mathbb{A}(u,v) = \int_\Omega \underline{\nabla} v \cdot (A \underline{\nabla} u) d\Omega \tag{3.3}$$

and

$$\mathbb{F}(v) = \int_\Omega fv \; d\Omega + \int_{\Gamma_2} gv \; d\Omega \tag{3.4}$$

respectively. Here A is bounded, symmetric and strictly positive definite and Γ_2 is the Neumann part of the boundary, subject to the conditions: $\underline{\hat{n}} \cdot (A\underline{\nabla} u) = g$. For the finite element solution of (3.1), the domain Ω of the problem must be meshed by tetrahedral elements, $\mathcal{T}^h$ say, where h represents the length of the largest diagonal of the largest tetrahedron in the mesh. Although polynomials of any order can be used to construct the piecewise polynomial space of trial functions $\mathcal{V}^h$ (see §1.1.1.2 for details), here we consider only a piecewise linear space of trial functions $\mathcal{V}^h$ on $\mathcal{T}^h$. The construction of this space and further details are described in the following section.

3.2 Theory

For the finite element approximation to the solution of (3.1) from the finite dimensional space, $\mathcal{V}^h$, of continuous piecewise linear functions on $\mathcal{T}^h$, we need to solve a corresponding discrete problem, which can be defined as follows.

Find $u^h \in \mathcal{V}^h \cap \mathcal{H}_0^1(\Omega)$ such that

$$\mathbb{A}(u^h, v^h) = \mathbb{F}(v^h) \qquad \forall \; v^h \in \mathcal{V}^h \cap \mathcal{H}_0^1(\Omega). \tag{3.5}$$

This problem can be solved using a suitable choice of basis functions for $\mathcal{V}^h$ and hence transformed to a system of linear equations of the form (1.11). For the usual local choice of basis functions, Φ_i, the stiffness matrix $\mathcal{A}$ is sparse, symmetric and strictly positive definite, and has entries given by

$$\mathcal{A}_{ij} = \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j)) \; d\Omega. \tag{3.6}$$

Hence a conjugate gradient (CG) like algorithm [4, 86] for the iterative solution of this problem is most appropriate. Since the growth of the condition number of $\mathcal{A}$ as the mesh is refined is like $O(h^{-3})$ as $h \to 0$ (for details see [108]), the application of preconditioned CG methods for most practical choices of h is essential. In this chapter, although we consider only the symmetric version of the additive Schwarz preconditioner for symmetric problems, we actually use a more general iterative method, GMRES [154], which is equally suitable for symmetric and nonsymmetric problems. Although the cost of GMRES is slightly higher than CG for symmetric problems, this approach will allow us to generalize the codes developed to non-symmetric problems (see Chapter 4) more easily. Furthermore, when an effective preconditioner is used the total number of iterations required is small (see §3.8) and so the overhead of using GMRES rather than CG remains low.

Assume that the tetrahedral mesh $\mathcal{T}^h$ over the domain Ω is obtained by uniform refinement of some coarse tetrahedral mesh $\mathcal{T}^H$ of the same domain. We also define $\mathcal{V} = \mathcal{V}^h \cap \mathcal{H}_0^1(\Omega)$ as the trial and test space in (3.5) and $\mathcal{V}_c$ to be the piecewise linear finite element space $\mathcal{V}^H \cap \mathcal{H}_0^1(\Omega)$ defined on $\mathcal{T}^H$. Introduction of this coarse mesh $\mathcal{T}^H$ makes it possible to decompose the domain Ω into p subdomains, $\Omega_0, \ldots, \Omega_{p-1}$, where each of these subdomains is the union of tetrahedral elements in $\mathcal{T}^H$, and these subdomains are possibly overlapping. For $i = 0, \ldots, p-1$ we define the space $\mathcal{H}_0^1(\Omega_i) \in \mathcal{L}^2(\Omega)$ to be the extension of $\mathcal{H}^1(\Omega_i)$ such that

$$u(x, y, z) = 0 \qquad \forall\, (x, y, z) \in (\Omega - \bar{\Omega}_i) \cup \Gamma_1. \tag{3.7}$$

We also define corresponding finite dimensional spaces by

$$\mathcal{V}_i = \mathcal{V}^h \cap \mathcal{H}_0^1(\Omega_i), \tag{3.8}$$

so that these local spaces $\mathcal{V}_i$ form a decomposition of the global finite element space $\mathcal{V}$:

$$\mathcal{V} = \sum_{i=0}^{p-1} \mathcal{V}_i. \tag{3.9}$$

This means that, for each $v \in \mathcal{V}$, there exists a combination of $v_i \in \mathcal{V}_i$, for $i = 0, \ldots, p-1$, such that $v = \sum_{i=0}^{p-1} v_i$. This combination is not necessarily unique. For

the space decomposition given by (3.9), the additive Schwarz algorithm defines a preconditioner, $\mathcal{M}$ say, for matrix $\mathcal{A}$ in (1.11) as follows.

Let M_i be the projection from $\mathcal{V}$ to $\mathcal{V}_i$ for $i = 0, \ldots, p-1$ given by

$$\int_\Omega (M_i u) v_i \, d\Omega = \int_\Omega u \; v_i \, d\Omega \qquad \forall\, u \in \mathcal{V}, \;\; v_i \in \mathcal{V}_i, \tag{3.10}$$

and define $\mathbb{A}_i$ to be the restriction of $\mathbb{A}$ to $\mathcal{V}_i \times \mathcal{V}_i$ given by

$$\mathbb{A}_i(u_i, v_i) = \mathbb{A}(u_i, v_i) \qquad \forall\, u_i, v_i \in \mathcal{V}_i. \tag{3.11}$$

Now for the usual linear finite element basis of $\mathcal{V}$ and $\mathcal{V}_i$, we can express M_i as rectangular matrix, $\bar{\mathcal{M}}_i$ say, and a local stiffness matrix $\mathcal{A}_i$ can be derived from $\mathbb{A}_i$. This is done in the same way that $\mathcal{A}$ is derived from $\mathbb{A}$ above. The basic additive Schwarz preconditioner is then given by

$$\mathcal{M} = \sum_{i=0}^{p-1} \bar{\mathcal{M}}_i^T \mathcal{A}_i^{-1} \bar{\mathcal{M}}_i. \tag{3.12}$$

Here we note that the subdomain solves $(\mathcal{A}_i^{-1} \underline{y}_i)$ may be performed concurrently. These subdomain solves are required at each preconditioned GMRES iteration for the solution of the system $\mathcal{M}^{-1}\underline{z} = \underline{y}$. For the time being, we assume that these subdomain solves are exact but in practice, as we will see later on, it is not generally necessary to solve subdomain problems exactly.

A theoretical justification for preconditioners of the form (3.12) is evident from the following theorem.

Theorem 3.1 *The matrix $\mathcal{M}$ defined by (3.12) is symmetric and positive definite. Furthermore, if we assume that there is some constant $C > 0$ such that: $\forall\, v \in \mathcal{V}$ there are $v_i \in \mathcal{V}_i$ such that $v = \sum_{i=0}^{p-1} v_i$ and*

$$\sum_{i=0}^{p-1} \mathbb{A}_i(v_i, v_i) \leq \; C\mathbb{A}(v, v) \tag{3.13}$$

then the spectral condition number of $\mathcal{M}\mathcal{A}$ is given by

$$\kappa(\mathcal{M}\mathcal{A}) \leq \; N_c C \tag{3.14}$$

where N_c is the minimum number of colours of the subdomains Ω_i such that all neighbours are of different colours.

The proof of this theorem may be found in [177]. This theorem shows that the quality of a general additive Schwarz preconditioner only depends upon the stability of the splitting of $\mathcal{V}$ into subspaces $\mathcal{V}_i$. Therefore the preconditioner given by (3.12) is an optimal preconditioner provided that the splitting (3.9) is such that (3.13) holds with C independent of h, H and p. In order to obtain such a splitting it is in fact necessary to modify (3.9) to include the coarse grid space $\mathcal{V}_c$:

$$\mathcal{V} = \mathcal{V}_c + \sum_{i=0}^{p-1} \mathcal{V}_i. \tag{3.15}$$

The preconditioner (3.12) then becomes

$$\mathcal{M} = \bar{\mathcal{M}}_c^T \mathcal{A}_c^{-1} \bar{\mathcal{M}}_c + \sum_{i=0}^{p-1} \bar{\mathcal{M}}_i^T \mathcal{A}_i^{-1} \bar{\mathcal{M}}_i. \tag{3.16}$$

Here $\bar{\mathcal{M}}_c$ represents the rectangular matrix corresponding to the $\mathcal{L}^2$ projection M_c from $\mathcal{V}$ to $\mathcal{V}_c$ that is given by

$$\int_\Omega (M_c u) v_c \, d\Omega = \int_\Omega u \; v_c \, d\Omega \qquad \forall\, u \in \mathcal{V}, \;\; v_c \in \mathcal{V}_c. \tag{3.17}$$

Also the stiffness matrix $\mathcal{A}_c$ is derived from $\mathbb{A}_c$, the restriction of $\mathbb{A}$ to $\mathcal{V}_c \times \mathcal{V}_c$ that is given by

$$\mathbb{A}_c(u_c, v_c) = \mathbb{A}(u_c, v_c) \qquad \forall\, u_c, v_c \in \mathcal{V}_c. \tag{3.18}$$

Now we state (without proof, which can also be found in [177] for example) another theorem to reflect this modification to the preconditioner (3.12).

Theorem 3.2 *Provided the overlap between the subdomains Ω_i is of size $O(H)$, where H represents the mesh size of $\mathcal{T}^H$, then there exists $C > 0$ which is independent of h, H and p, such that for any $v \in \mathcal{V}$ there are $v_c \in \mathcal{V}_c$ and $v_i \in \mathcal{V}_i$ such that $v = v_c + \sum_{i=0}^{p-1} v_i$ and*

$$\mathbb{A}_c(v_c, v_c) + \sum_{i=0}^{p-1} \mathbb{A}_i(v_i, v_i) \leq \; C\mathbb{A}(v, v) \tag{3.19}$$

Hence, with the inclusion of a *coarse grid* solve and a *generous* overlap between subdomains, it is possible to use the additive Schwarz method to get an optimal preconditioner. However, there are a couple of issues to be considered in the context of the practical efficiency of such a preconditioner. These issues are:

1. The parallel solution of the global coarse grid problem.

2. The amount of overlap between subdomains.

Since it is hard to solve the global coarse grid problem in parallel, its implementation demands great care in order to avoid a potential bottleneck in the parallel performance. It may be best to solve this problem sequentially on a single process or on all processes (which introduces redundancy but may save some communication). Alternatively, a multilevel solution technique could be used, however a detailed discussion of this is outside the scope of this thesis. The issue of overlap between subdomains is also important. The theoretical results of Theorem 3.2 suggest that a fixed fraction of the subdomain size should be used. However, if we assume the uniform global refinement of the mesh $\mathcal{T}^h$, the number of elements of $\mathcal{T}^h$ in the overlap regions between subdomains is $O(h^{-3})$ as $h \to 0$ and this will cause a significant computational overhead for small h. This issue is usually addressed (e.g. [160]) by dropping the requirement of optimality and permitting only a small amount of overlap between subdomains: typically a fixed number of fine element layers. This reduces the total number of elements in the overlap significantly (to $O(h^{-2})$ in 3–d) but at the expense of increasing the iteration count to some degree. Both of these issues are addressed in a different manner in the following section, where we describe an optimal two level additive Schwarz preconditioner of the form (3.16) with substantially fewer elements in the overlap regions and a modified coarse grid solve.

3.3 The Preconditioner

After the introduction of the model problem representing a class of elliptic partial differential equation in §3.1 and the background theory of two level additive Schwarz DD preconditioners in §3.2, Now we describe the above mentioned optimal two level additive Schwarz preconditioner of the form (3.16), where the number of elements in the overlap region between subdomains is $O(h^{-2})$ as $h \to 0$ in contrast to $O(h^{-3})$ in the case of a generous overlap. This is achieved through the existence of a hierarchy

of meshes between $\mathcal{T}^H$ and $\mathcal{T}^h$, each defined by a single level of refinement of its predecessor. At each level of the hierarchy the overlap is defined to be just a single layer of elements. This idea of using hierarchical refinement with just one element in the overlap region at each level is contrasted against the uniform refinement of an $O(H)$ overlap region in Figure 3.1. In order to describe the DD preconditioner

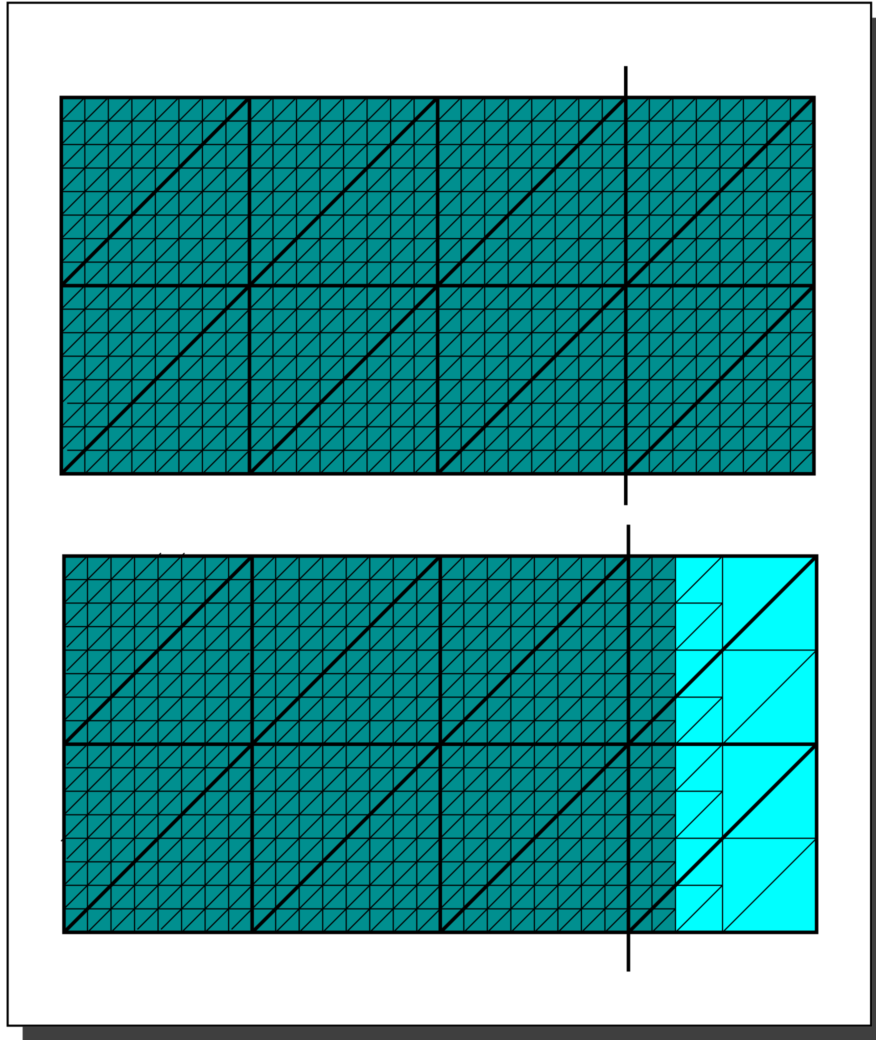

Figure 3.1: Uniform (top) and Hierarchical (bottom) refinement in the overlap region.

under consideration, we begin by introducing some notation. Let the domain Ω of the problem given by equations (3.1–3.2) be covered by $\mathcal{T}_0$, a set of N_0 tetrahedral elements τ_j^0 for $j = 1, \ldots, N_0$. These tetrahedral elements may be refined using the

techniques described in §1.3, where each tetrahedron is divided to produce 8 children using *regular refinement.* Here it is to be noted that although the child elements are not necessarily geometrically similar to the parent elements, their aspect ratios are always bounded independently of h [164]. The set $\mathcal{T}_0$ of N_0 tetrahedral elements τ_j^0 is such that $\tau_j^0 = \bar{\tau}_j^0$ and

$$\Omega = \bigcup_{j=1}^{N_0} \tau_j^0 \tag{3.20}$$

or

$$\mathcal{T}_0 = \{\tau_j^0\}_{j=1}^{N_0}. \tag{3.21}$$

Let the diameter of these base mesh elements be $diameter(\tau_j^0) = O(H)$, hence this tetrahedral mesh can be referred to as $\mathcal{T}^H$ by the notation used in the previous section. Let the domain Ω be divided into p non–overlapping subdomains Ω_i for $i = 0, \ldots, p-1$ such that

$$\bar{\Omega} = \bigcup_{i=0}^{p-1} \Omega_i, \tag{3.22}$$

$$\Omega_i \cap \Omega_j = \emptyset \quad \text{for } i \neq j, \tag{3.23}$$

$$\bar{\Omega}_i = \bigcup_{j \in I_i} \tau_j^0 \qquad \text{where} \quad I_i \subset \{1, \ldots, N_0\}\ ,\ I_i \neq \emptyset. \tag{3.24}$$

The refinement of $\mathcal{T}_0$ up to some specified level, J say, produces a family of tetrahedral meshes $\mathcal{T}_0, \ldots, \mathcal{T}_J$ such that mesh $\mathcal{T}_k$ at level k consists of N_k elements which are represented by τ_j^k such that

$$\Omega = \bigcup_{j=1}^{N_k} \tau_j^k \tag{3.25}$$

or

$$\mathcal{T}_k = \{\tau_j^k\}_{j=1}^{N_k}. \tag{3.26}$$

There is no compulsion for the tetrahedral meshes in this sequence to be either a global refinement of their predecessor or conforming. On the other hand they must satisfy some, reasonably standard (e.g. [25]) conditions.

1. If $\tau \in \mathcal{T}_{k+1}$ then either

   (a) $\tau \in \mathcal{T}_k$, or

(b) τ is child of an element of $\mathcal{T}_k$ produced by refinement.

2. Any two tetrahedral elements sharing a common point may differ by one level of refinement at the most.

3. In the transition from level k to $k+1$, only tetrahedral elements at level k may be refined.

4. The elements created in the transition from level k to $k+1$ are said to be at level $k+1$.

5. All tetrahedral elements with a common edge which lies on the interface between subdomains must be at the same level.

Once the domain Ω has been decomposed into subdomains Ω_i for $i = 0, \ldots, p-1$, and refined successively to achieve a nested sequence of tetrahedral meshes, the restriction of these tetrahedral meshes onto each subdomain Ω_i for $i = 0, \ldots, p-1$ may be defined by

$$\Omega_{ik} = \{\bigcup_j \tau_j^{(k)} : \tau_j^{(k)} \subset \bar{\Omega}_i\}. \tag{3.27}$$

That is, Ω_{ik} is the union of all tetrahedral mesh elements at level k within the boundary of subdomain Ω_i. In this way we get the same non–overlapping subdomains at each level of the mesh hierarchy, each consisting of a unique subset of elements from the global mesh $\mathcal{T}_k$ for $k = 0, \ldots, J$. A certain amount of overlap between neighbouring subdomains is now permitted through the definition:

$$\tilde{\Omega}_{ik} = \{\bigcup_j \tau_j^{(k)} : \tau_j^{(k)} \text{ has a common point with } \bar{\Omega}_i\}. \tag{3.28}$$

Having defined the restriction of each mesh $\mathcal{T}_k$ to each subdomain Ω_i, we now define the finite element spaces associated with tetrahedral elements in these subsets. Let G be some tetrahedral mesh and let $\mathcal{S}(G)$ be the space of continuous piecewise linear functions on G. Then the finite element spaces associated with the local tetrahedral meshes given above may be defined as:

$$\mathcal{W} \;=\; \mathcal{S}(\mathcal{T}_J), \tag{3.29}$$

$$\mathcal{W}_c = \mathcal{S}(\mathcal{T}_0), \quad (3.30)$$

$$\mathcal{W}_{ik} = \mathcal{S}(\Omega_{ik}), \quad (3.31)$$

$$\tilde{\mathcal{W}}_{ik} = \mathcal{S}(\tilde{\Omega}_{ik}), \quad (3.32)$$

$$\tilde{\mathcal{W}}_i = \tilde{\mathcal{W}}_{i0} + \ldots + \tilde{\mathcal{W}}_{iJ}. \quad (3.33)$$

A decomposition of the form (3.15), necessary to define a two level additive Schwarz preconditioner, can therefore be defined as

$$\mathcal{W} = \mathcal{W}_c + \tilde{\mathcal{W}}_0 + \ldots + \tilde{\mathcal{W}}_{p-1}. \quad (3.34)$$

For this preconditioner to be optimal, Theorem 3.1 implies that for any given $u^h \in \mathcal{W}$, it is sufficient to construct $u_c^h \in \mathcal{W}_c$ and $u_i^h \in \mathcal{W}_i$, for $i = 0, \ldots, p-1$, such that

$$u^h = u_c^h + \sum_{i=0}^{p-1} u_i^h \quad (3.35)$$

and

$$\mathbb{A}_c(u_c^h, u_c^h) + \sum_{i=0}^{p-1} \mathbb{A}_i(u_i^h, u_i^h) \leq C\,\mathbb{A}(u^h, u^h) \quad (3.36)$$

for some constant $C > 0$ and independent of h, H and p. We now state the last theorem involved in this discussion, which demonstrates that the splitting (3.34) of the finite element space $\mathcal{W}$ is stable. A proof of this theorem may be found in [11] or [10].

Theorem 3.3 *There exists $C > 0$ which is independent of h, H and p such that for any $u^h \in \mathcal{W}$ there are $u^H \in \mathcal{W}_c$ and $u^h \in \tilde{\mathcal{W}}_i$ for $i = 0, \ldots, p-1$, such that*

$$u^h = u^H + u_0^h + \ldots + u_{p-1}^h \quad (3.37)$$

and

$$\|u^H\|^2_{\mathcal{H}^1(\Omega)} + \|u_0^h\|^2_{\mathcal{H}^1(\Omega)} + \ldots + \|u_{p-1}^h\|^2_{\mathcal{H}^1(\Omega)} \leq C\|u^h\|^2_{\mathcal{H}^1(\Omega)}. \quad (3.38)$$

At this stage Theorem 3.1, with N_c replaced by $N_c + 1$ in order to account for the coarse grid space $\mathcal{W}_c$, and the equivalence of the norm $\mathbb{A}(\cdot,\cdot)^{1/2}$ with the norm $\mathcal{H}^1$ imply that our two level additive Schwarz preconditioner of the form (3.16) is optimal.

The above presented theoretical results, and the appropriate use of a mesh hierarchy, offer an optimal two level additive Schwarz preconditioner with an overlap of $O(h^{-2})$ elements in three dimensions whereas the standard theoretical results require an overlap of $O(h^{-3})$ for an optimal two level preconditioner in three dimensions. It should be noted however that practical implementations of the two level additive Schwarz method do not use a generous overlap in practice [160]. Typically, two to four fine mesh layers give a reasonable trade–off between a good sub–optimal and an economical preconditioner. In contrast to this, our weakly overlapping approach offers the guarantee of optimality but at this reduced cost per iteration.

3.4 Implementation

In this section we describe some minor modifications to the preconditioner described in the previous section in order to generate the partitioned hierarchical meshes required for practical implementation in parallel. A number of other important issues concerned with implementation and algorithmic details will also be discussed in this and the following sections. Some numerical results for both global uniform refinement and local non–uniform refinement will then be presented to demonstrate the quality of the proposed preconditioner.

Let the computational domain Ω be covered by tetrahedra which form the coarse mesh $\mathcal{T}_0$. Once this coarse mesh $\mathcal{T}_0$ is partitioned into p non–overlapping subdomains Ω_i, for $i = 0, \ldots, p-1$, a copy of this mesh is allocated to each process. The copy of the coarse mesh on process i is then refined only in subdomain $\tilde{\Omega}_{ik}$ up to level k in the refinement procedure. On the meshes so obtained, the corresponding continuous piecewise linear finite element spaces are defined by

$$\mathcal{U}_i = \mathcal{W}_c \cup \tilde{\mathcal{W}}_i, \tag{3.39}$$

for $i = 0, \ldots, p-1$. Hence from Theorem 3.3 the following corollary immediately follows.

Corollary 3.4 *Let $\mathcal{U}_i$, for $i = 0, \ldots, p-1$, be the space given by (3.39). Then*

$$\mathcal{W} = \mathcal{U}_0 + \ldots + \mathcal{U}_{p-1} \tag{3.40}$$

is a stable decomposition.

We refer to [8] for some of the advantages of this approach where it is shown that partitioned adaptive meshes may be generated in parallel in a well load balanced manner. In this approach, instead of a separate global coarse mesh solve per iteration, we are completing a coarse mesh solve as part of each subspace correction or local inner solve, that is, p times per iteration which is clearly a disadvantage. However, in many practical codes this coarse mesh solve is completed sequentially on a single process, [97] for example. Therefore this disadvantage is not necessarily of serious concern (but will be discussed in more detail in Chapter 5).

A second minor modification to the described preconditioner relates to our use of transition or *green* elements in the mesh. These are inserted between elements at any two consecutive refinement levels to make the mesh conforming [146, 164], and are described in §1.3 in detail. Figure 3.2 illustrates the use of green elements for a simple problem in two dimensions (this illustration represents the xy–plane cross–section of the 3–d mesh produced by TETRAD). In three dimensions they are more complex (again see [164]) and they have a significant effect on the practical implementation of our preconditioner. As such, they are discussed in detail later in this chapter at the appropriate stage.

We now illustrate the use of this weakly overlapping additive Schwarz domain decomposition preconditioner algebraically when solving the algebraic equations obtained from (3.1–3.2). The Galerkin finite element discretization of this problem generates a linear system of the form (1.11) with a sparse coefficient matrix $\mathcal{A}$. In order to solve this system in parallel, the sparse coefficient matrix $\mathcal{A}$, the right–hand side vector $\underline{b}$ and the solution vector $\underline{u}$ (to be determined) are required to be distributed across the subdomains Ω_i for $i = 0, \ldots, p-1$. Such a distributed system can be expressed in block matrix notation as follows:

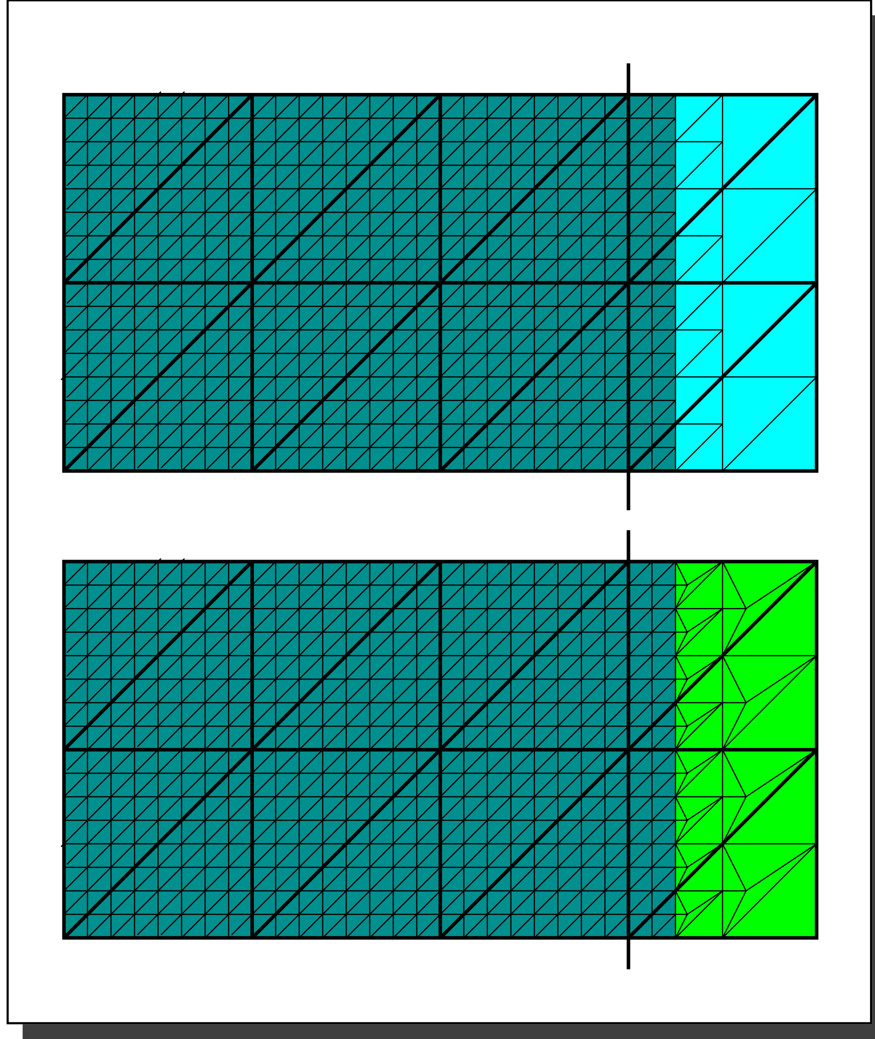

Figure 3.2: Refinement without (top) and with (bottom) transition *green* elements.

$$\begin{bmatrix} A_0 & & & & B_0 \\ & A_1 & & & B_1 \\ & & \ddots & & \vdots \\ & & & A_{p-1} & B_{p-1} \\ B_0^T & B_1^T & \dots & B_{p-1}^T & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_0 \\ \underline{u}_1 \\ \vdots \\ \underline{u}_{p-1} \\ \underline{u}_s \end{bmatrix} = \begin{bmatrix} \underline{b}_0 \\ \underline{b}_1 \\ \vdots \\ \underline{b}_{p-1} \\ \underline{b}_s \end{bmatrix}. \tag{3.41}$$

Here $\underline{u}_i$ is the vector of unknown values for nodes strictly inside the subdomain Ω_i $(i = 0, \dots, p-1)$ and $\underline{u}_s$ is the vector of unknown values for nodes on the interface between subdomains. Also, each block A_i, B_i and $\underline{b}_i$ is sparse and can be assembled by each process independently. The distribution of A_s and $\underline{b}_s$ across the

subdomains requires a partial assembly to be calculated and stored independently on each process, such as given by (1.29) and (1.30) respectively. Here an iterative solver such as CG [4] or GMRES [154] needs to be implemented in parallel. It should be recalled from §1.6.4 that the distributed matrix–vector products do not require any communication due to the distributed nature of the implemented data structure in this thesis. However, distributed inner products do require one global reduction operation, which is now a well established standard in parallel computing [160]. On each process i, for $i = 0, \ldots, p-1$, note that (3.41) may, after a suitable permutation, be expressed as

$$\begin{bmatrix} A_i & & B_i \\ & \bar{A}_i & \bar{B}_i \\ B_i^T & \bar{B}_i^T & A_{i,s} \end{bmatrix} \begin{bmatrix} \underline{u}_i \\ \underline{\bar{u}}_i \\ \underline{u}_{i,s} \end{bmatrix} = \begin{bmatrix} \underline{b}_i \\ \underline{\bar{b}}_i \\ \underline{b}_{i,s} \end{bmatrix}. \tag{3.42}$$

Here $\underline{u}_i$ still represents the unknowns inside Ω_i but $\underline{u}_{i,s}$ represents the unknowns on the interface of Ω_i with other subdomains and $\underline{\bar{u}}_i$ represents all remaining unknowns. The other blocks are formed similarly. Parallel application of the weakly overlapping additive Schwarz preconditioner $\mathcal{M}$ given by (3.16), using the decomposition (3.40), may now be described by considering the action of

$$\underline{z} = \mathcal{M}^{-1}\underline{y} \tag{3.43}$$

in the block matrix notation (3.42) as follows. On each process i, for $i = 0, \ldots, p-1$, solve the system

$$\begin{bmatrix} A_i & & B_i \\ & \bar{A}_i & \bar{B}_i \\ B_i^T & \bar{B}_i^T & A_{i,s} \end{bmatrix} \begin{bmatrix} \underline{z}_i \\ \underline{\bar{z}}_i \\ \underline{z}_{i,s} \end{bmatrix} = \begin{bmatrix} \underline{y}_i \\ M_i \underline{\bar{y}}_i \\ \underline{y}_{i,s} \end{bmatrix} \tag{3.44}$$

where

$$\begin{aligned} \underline{y}_i &= P_i \underline{y}, \\ \underline{\bar{y}}_i &= \bar{P}_i \underline{y}, \\ \underline{y}_{i,s} &= P_{i,s} \underline{y}. \end{aligned} \tag{3.45}$$

Here we note that

- P_i is a rectangular Boolean matrix that picks out the entries of $\underline{y}$ corresponding to nodes inside Ω_i.

- $P_{i,s}$ is a rectangular Boolean matrix that picks out the entries of $\underline{y}$ corresponding to nodes on interface of Ω_i.

- $\bar{P}_i$ is a rectangular Boolean matrix that picks out the entries of $\underline{y}$ corresponding to nodes outside $\bar{\Omega}_i$.

The above subproblem (3.44) is solved on process i, without any dependence on subproblems solved on any of the other processes. Hence these subproblems may be solved concurrently. Once the solution of each subproblem is available, its contribution towards overall global solution is then obtained as follows:

$$\underline{z} = \sum_{i=0}^{p-1} \left(P_i^T \underline{z}_i + \bar{P}_i^T M_i^T \underline{\bar{z}}_i + P_{i,s}^T \underline{z}_{i,s} \right) . \tag{3.46}$$

In equation (3.44) the blocks A_i and B_i are the components of the global stiffness matrix assembled for elements of the mesh on process i which cover the subdomain Ω_i, the blocks $\bar{A}_i$, $\bar{B}_i$ are the components of the global stiffness matrix assembled for elements of the mesh on process i which cover the region $\Omega \setminus \bar{\Omega}_i$, and the blocks A_s form the remaining components of the global stiffness matrix for the elements of the mesh on process i which have at least one vertex at the interface of subdomain Ω_i with the neighbouring subdomains.

The rectangular matrices M_i in (3.44) are the hierarchical restriction operators from the fine mesh covering subdomain Ω_j on process j (for all $j \neq i$) to the coarse mesh covering subdomain Ω_j on process i. These hierarchical restriction operators are similar to those generally used in multigrid algorithms [160]. The operators M_i^T in (3.46) are the corresponding hierarchical prolongation operators whose action is the inverse to that of the hierarchical restriction operators. Application of these operators require some communication of data across the processes. Further details of these hierarchical operators and the algorithms describing the operations of restriction and prolongation follows shortly. From equations (3.44–3.45) we note

that these subproblems are solved over the entire domain Ω such that on the mesh owned by process i subdomain Ω_i is refined up to some desired level of refinement. On this process all other subdomains contain just base level mesh elements except

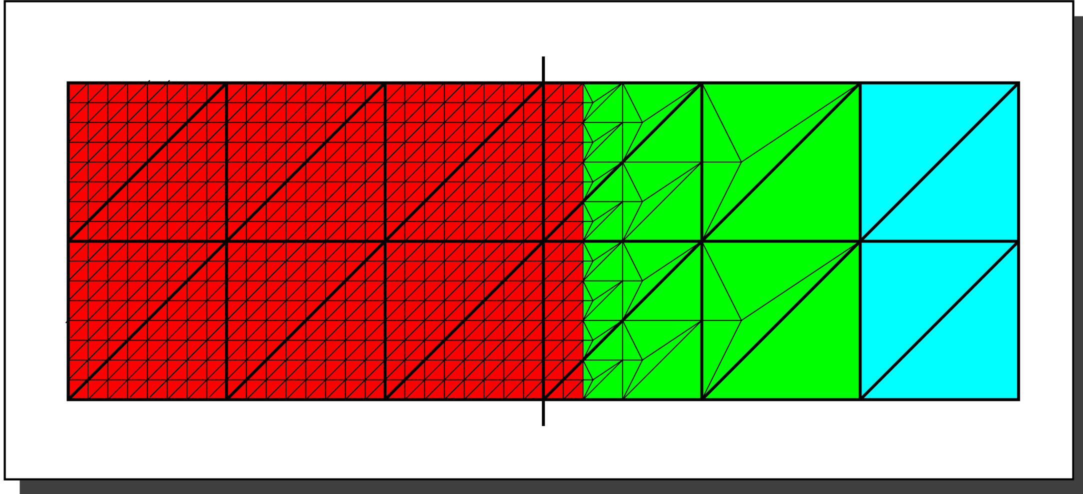

Figure 3.3: Uniform refinement of subdomain Ω_i owned by process i.

for a small portion of neighbouring subdomains to Ω_i, which contain a single layer of elements at each level of refinement around subdomain Ω_i. In other words, each of the subproblems is a combination of the weakly overlapping subdomain solve with the coarse mesh solve (see (3.39)), hence we are effectively solving a coarse mesh problem on each process at each iteration. Although this is an extra amount of work being undertaken it need not cause a significant overhead as this coarse mesh is generally very small compared to the fully refined mesh (see Figure 3.3). Also, each of the local problems approaches $\frac{1}{p}$ times the size of full problem as $h \to 0$, even when this coarse mesh is included with these subproblems.

The major implementation issues to be discussed here are those of computing the action of each of the restriction operations $M_i\underline{\bar{y}}_i$ in (3.44) at each iteration before each subproblem solve, and the corresponding prolongation operation $M_i^T\underline{\bar{z}}_i$ at the end of each preconditioning step. Evaluation of both of these actions is computed in two phases: a *setup phase* and an *iteration phase*. The setup phase occurs only once, before the first iteration, whereas the iteration phase occurs at each iteration. For a parallel programming paradigm it is understood that a communication task is an

additional overhead on top of the computational tasks. Hence we have attempted to implement both the setup phase and the iteration phase in such a way that computations and communication overlap each other. Additionally, the introduction of green elements and nodes to prevent non–conforming meshes create an additional complexity in both of these phases. All of the above issues concerning the setup phase and the iteration phase, which involve both coarsening and prolongation, the update of green nodes and some other related issues (such as accuracy of the subdomain solve, choice of best sequential solver, etc.) are discussed fully in the following sections.

3.5 Restriction

The implementation strategy for our weakly overlapping two level additive Schwarz DD preconditioner has been described in general in the previous section without specific detail of the construction of the components which build this preconditioner. In this section we consider some of these components, particularly the setup phase and the iteration phase of the hierarchical restriction operation. We also consider the computation of different values at the green nodes which arise due to the refinement strategy adopted in TETRAD. Recall the description of mesh adaption using TETRAD in §1.3 where we discuss the nature and effect of green refinement. Some of the computational work at each iteration is increased due to the use of green refinement and this is discussed below.

On each process i, we have a residual vector with entries corresponding to the interior and interface boundary nodes of the subdomain Ω_i. The current values for the nodes exterior to subdomain Ω_i on process i are computed by coarsening from fine mesh node values on other processes. In order to implement this coarsening, some knowledge of coarse mesh nodes on process i subdomain Ω_j ($j \neq i$) and the corresponding fine mesh nodes on process j is required. It is then necessary to match these coarse mesh nodes with the corresponding fine mesh nodes on process j. This requires some transfer of data between process i and every other process.

In the following section we discuss the collection of data to be transferred and the matching of this transferred data with the local data on each process.

3.5.1 Setup Phase

Let us consider the tasks to be undertaken by process i. We need to collect a set of data corresponding to each subdomain Ω_j for $j \neq i$. This data consists of an integer number of nodes and the (x, y, z) coordinate triples of each mesh point in the interior of the respective subdomain. Each set of such data (for subdomain Ω_j on process i) is then communicated to the corresponding process j. After the data has been received, process j matches the (x, y, z) coordinates triples of the received data with the (x, y, z) coordinates triples of the local mesh data for subdomain Ω_j. The subdomain Ω_j is owned by the process j and is fully refined. In this way each process sends a list of coordinates triples to every other process. Similarly each process receives a list of coordinates triples from every other process. Process i, for example, receives lists from each of the other processes j $(j \neq i)$ and then matches coordinates triples from the received lists with the local coordinates triples for the fine mesh in Ω_i. It is mandatory that each of the coordinate triples in the received lists matches one of the local coordinate triples since all meshes are refinements of the same coarse mesh. We now discuss the practical implementation of these procedures in turn.

3.5.1.1 Data Collection

Let us continue to assume that the problem domain Ω is decomposed into p subdomains Ω_i for $i = 0, \ldots, p-1$. Recall from §3.3 that process i refines the subdomain Ω_i only and rest of the subdomains (Ω_j, for $j \neq i$) have the base level coarse mesh except in the neighbouring subdomains, $\Omega_j \in N_i$ say (recall the definition of a neighbouring subdomain from §2.2.1), where a single layer of elements at each refinement level is required next to subdomain Ω_i. This refinement of subdomain Ω_i introduces transition (also known as green) elements in the neighbouring subdomains $\Omega_j \in N_i$ when the mesh inside subdomain Ω_i is refined uniformly. In case of local

non–uniform refinement, green elements may also exist in subdomain Ω_i as well (in which case the existence of green elements in the neighbouring subdomains is no longer compulsory). These green elements and the corresponding green nodes (every fourth node of a green element is referred to as a green node, which is introduced at the centre of the parent element being green refined (whereas the first three nodes of a green child element are the same as nodes of the parent element)) require a special treatment which is clearly distinguishable in the following discussion.

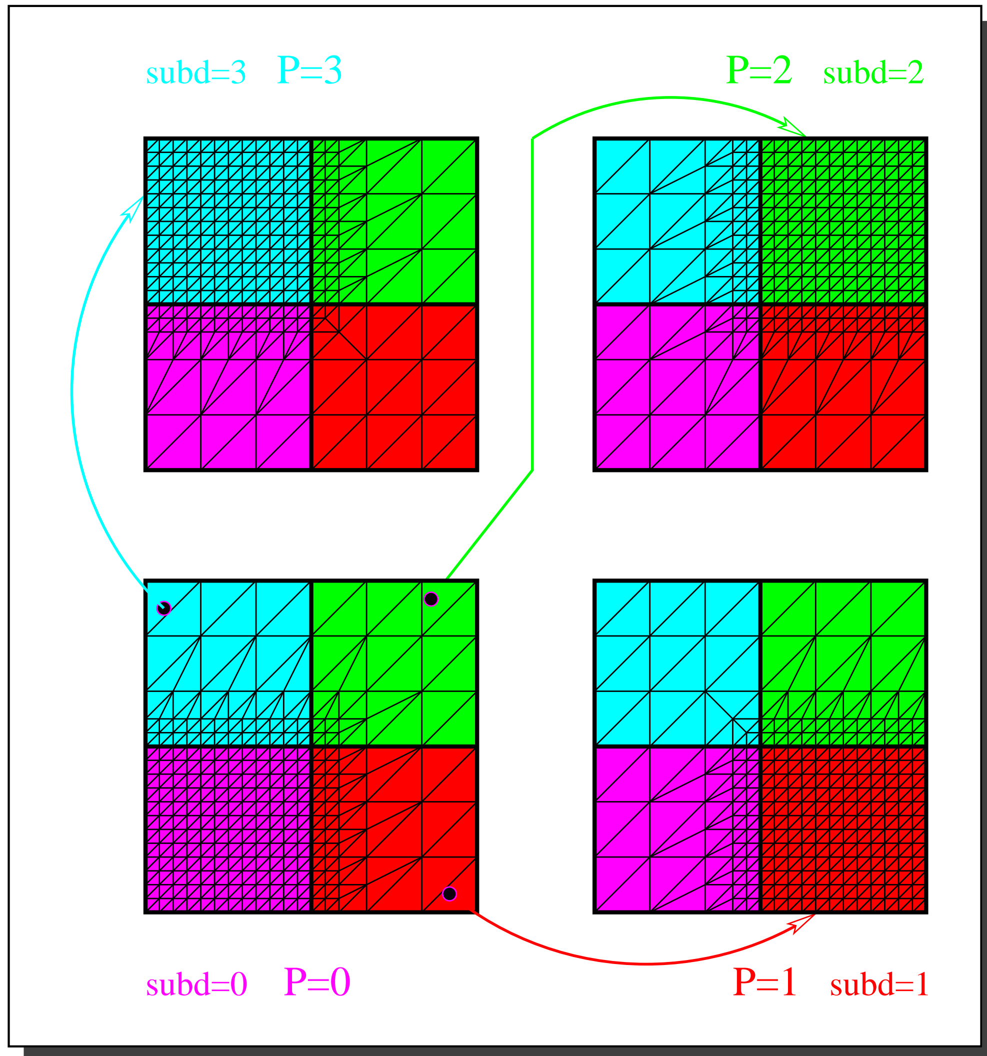

Figure 3.4: Data sets sent by process $i = 0$ to every other process j.

We now describe how to determine the amount of data in each subdomain Ω_j (for $j \neq i$), on process i, to be communicated to the corresponding process j. For

subdomain Ω_j on process i, the type of each element is examined and classified as being either regular or green. In the case where the element type is green, only the first three local nodes of the element are considered and the fourth node (which is green and therefore does not match with any node on process j) is ignored. There is a separate treatment for the green nodes. If the element type is regular, all four local nodes of the element are considered. Recall that for any finite element mesh there exists a mesh connectivity relating the local node numbering of the elements with the global node numbering of the mesh. For a local node being considered here, the corresponding global node is tested to ensure that it is neither on the Dirichlet boundary, the subdomain interface boundary and that it has not already been visited as part of the list for subdomain Ω_j. On passing this test the global node is marked as *visited[j]* and the *counter[j]* is iterated (which must be initialized for each subdomain Ω_j before it is iterated for first time). These global nodes are marked to avoid listing multiple copies of the same node as every local node in the interior of a subdomain is always shared by more than one element. Once this loop over elements is complete, the *counter[j]* gives the length (size) of the list of (x, y, z) coordinate triples to be constructed, consisting of all nodes which have just been marked for subdomain Ω_j. In order to construct this list of coordinate triples, every global node in the mesh is checked and the values of the (x, y, z) coordinate triples are added to the *list[j]* for the nodes previously marked *visited[j]* for subdomain Ω_j. This procedure is repeated for every subdomain Ω_j $(j \neq i)$ on process i, so that at the completion of this procedure we know the number of marked nodes and have a list of coordinate triples corresponding to the marked nodes for each subdomain Ω_j $(j \neq i)$. This number of marked nodes and the corresponding list of coordinate triples for subdomain Ω_j is then communicated to process j (for each $j \neq i$), as illustrated in 2–d for $p = 4$ in Figure 3.4.

With the completion of this communication, the sending part of the setup phase for restriction on process i is complete. On the other hand, each of the processes must now receive $p-1$ lists of coordinate triples, one from every other process. Each of these coordinate triples must then be matched with the corresponding node of

the fine mesh that belongs to the subdomain owned by that process. A record of these matching fine mesh nodes corresponding to the received coarse mesh nodes is necessary in order to implement the coarsening that is undertaken as part of the preconditioner (3.44). This matching procedure is explained next.

3.5.1.2 Matching of Coarse and Fine Meshes

Again we consider the action of process i. This process receives data consisting of an integer number representing the length of the list of (x, y, z) coordinate triples, and the list of these coordinate triples from every other process j. This is illustrated in 2–d for $p = 4$ in Figure 3.5. It is guaranteed that each of the received coarse

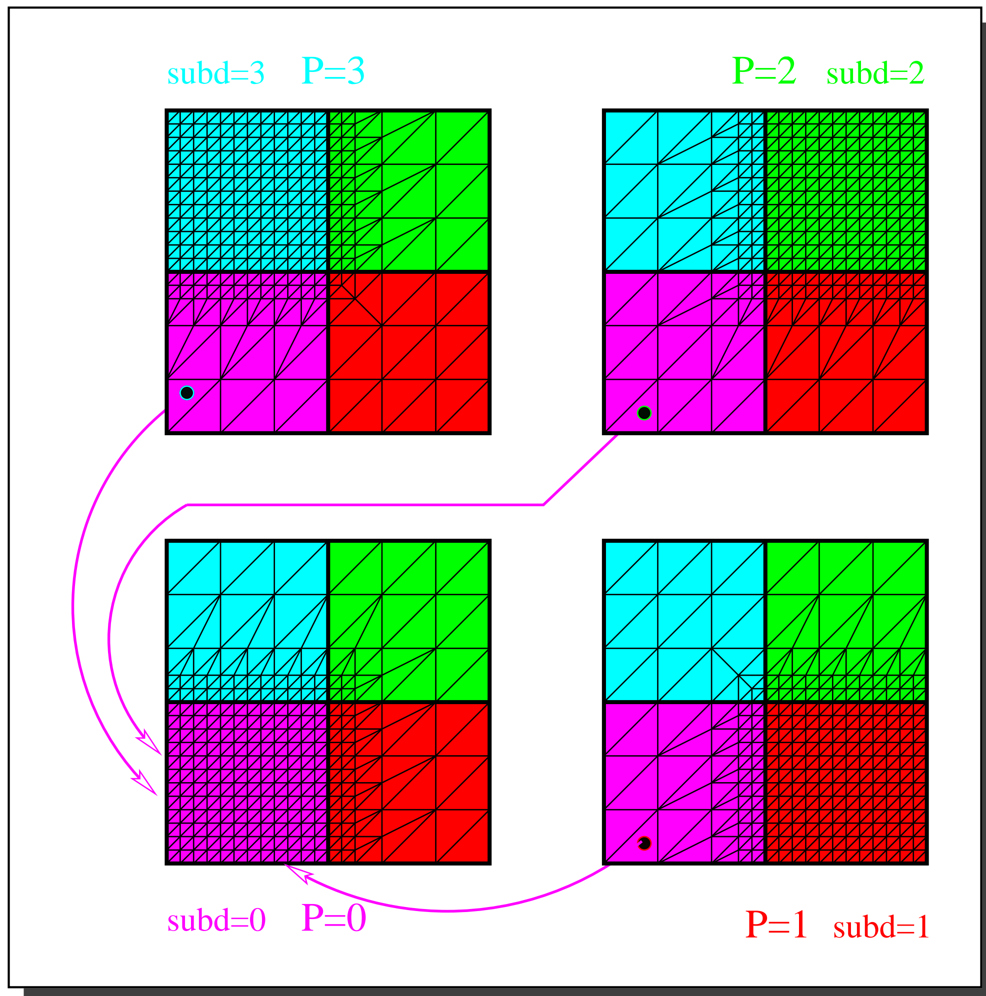

Figure 3.5: Data sets received by process $i = 0$ from every other process j.

mesh coordinate triples in the received list matches one of the fine mesh coordinate triples in the fine mesh of subdomain Ω_i on process i. To undertake this matching efficiently a linked list of base level elements in the interior of subdomain Ω_i is used. These base level elements are not part of the final mesh covering subdomain Ω_i but exist in the mesh hierarchy only. The hierarchical mesh data structures of TETRAD are then exploited along with the fact that, for each of the base level elements in the mesh hierarchy, all elements in the refinement tree of that element are bounded by the base level element. Similarly the coordinate triples for any element in this refinement tree are also bounded by the coordinate triples of the vertices of the base level element. Therefore, once a coordinate triple from a received list is found to be bounded by an element in the linked list of base level elements, the matching fine mesh coordinate triples can be easily and efficiently reached by using the hierarchy of the refinement tree. This search and match procedure is further optimized by using the breadth first search algorithm. Furthermore, the use of a recursive search algorithm ensures that the above mentioned guarantee for matching any coordinate triples in the received list is fulfilled. We now look at the matching procedure as implemented.

Assume that process i receives a list of coordinates triplets from process j, then it immediately starts matching these coordinate triples one by one with those of the fine mesh for subdomain Ω_i. For each of the received coordinate triples, it is examined and found to be bounded by the vertices of one of the base level elements in the subdomain Ω_i, which have already been arranged in the form of a linked list. The received coordinate triple is then tested to match with any of the vertices of the base level element. If a matching vertex is found, the corresponding global node is marked *match[j]* for process j and the matching procedure for this coordinate triple is terminated. If no matching vertex found, all of the child elements of the current base level element are examined to see which one bounds this coordinate triple. The vertices of that particular bounding element are then tested to match with the received coordinate triple. If a matching vertex found, the procedure is terminated as described above. If no matching vertex found, all of the child elements of the

current element are examined in the same way as before and this procedure continues until a match is found. The matches for all of the received coordinate triples are found in this way. It is important to note that most of the received coordinate triples match with vertices of base level elements, so this matching algorithm is particularly efficient.

The setup phase for the restriction process is now complete except for the following description of how to construct the linked list of base level elements for subdomain Ω_i on process i. It is assumed that this subdomain has been refined and the fine mesh consists of leaf elements only. These are all at the same level of refinement in the case of uniform refinement but may be anywhere between the base level and the maximum level of refinement in the case of local refinement. In general, all of the base level elements exist in the mesh hierarchy however, for adaptive or local refinement, some of the leaf elements may also be base level elements in regions where base mesh in not refined at all.

The TETRAD [164] data structure is such that in the mesh hierarchy on process i each element has a unique parent at the next lower level, except for the elements at the base level. These base level elements are the target to construct the required linked list. This phenomena is illustrated in 2–d in Figure 3.6. In order to construct a linked list of base level elements, the parent element of each leaf element in the mesh hierarchy is checked. For any element, if there does not exist a parent element it is a base level element and is stored in the linked list of base elements. Otherwise, if a parent element exists it is checked whether this has already been visited or not. If it has already been visited there is nothing further to do since the base level element in this refinement sub–tree has already been placed in the linked list. However, if the element has not been visited, the same procedure is repeated with this element which is now a leaf element at the next lower level in the mesh hierarchy. This process continues until a previously visited element or a base level element is found and added to the linked list. The same procedure is repeated for each leaf element in the fine mesh in Ω_i until all of the base level elements in the mesh hierarchy inside Ω_i have been found and added to the linked list.

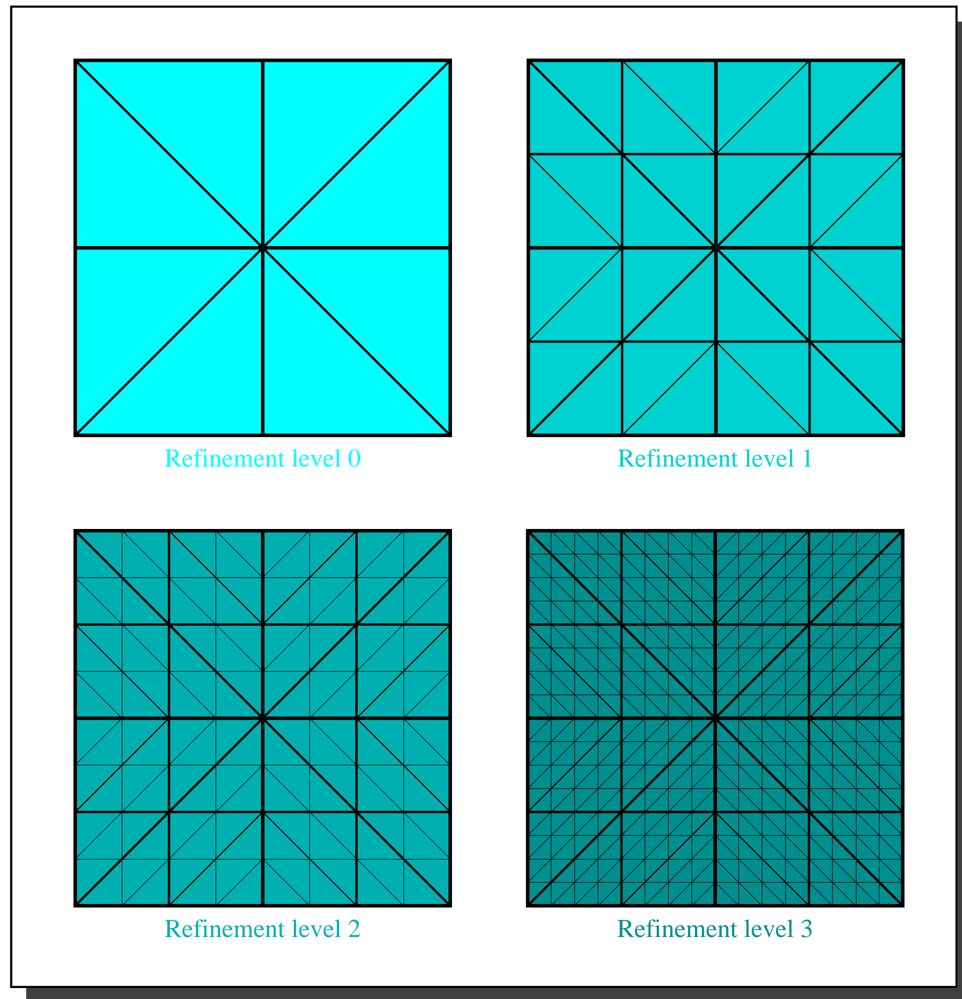

Figure 3.6: Refinement tree: Refinement levels between 0 and 3.

It is to be noted that for each base level element in the mesh hierarchy its own refinement tree is completely traversed only once. For example, if a single tetrahedral element is refined in a regular manner up to a fourth level of refinement, the final mesh consists of 4096 leaf elements. To construct a linked list of base level elements for this final mesh of 4096 leaf elements as described above, the refinement tree will be completely traversed only once. The process of traversing this refinement tree terminates 7 times at the first level, 56 times at the second level, 498 times at the third level and 3584 times at the fourth level of refinement as the parent element for all such elements in the mesh hierarchy has been visited previously or added to the linked list of base level elements.

3.5.2 Iteration Phase

In the setup phase, the relation of correspondence between nodes in the mesh covering subdomain Ω_i on process j (for $j \neq i$) has been established with the nodes in the mesh covering subdomain Ω_i on process i. We recall that on process i, only subdomain Ω_i is refined and all other subdomains Ω_j for $j \neq i$ have the base level coarse mesh except the neighbouring subdomains $\Omega_j \in N_i$, which have a single layer of fine elements at each level that surrounds the subdomain Ω_i. This relation between coarse mesh nodes and fine mesh nodes is exploited in the process of coarsening from the fine mesh covering a particular subdomain on one process to the corresponding coarse meshes that cover the same subdomain on the other processes. This process of coarsening is made simple and efficient by creating a linked list of child edges that exist at each level of refinement within each subdomain Ω_i. Every child edge included in this linked list must satisfy the following conditions.

1. Each edge must be a leaf edge at the level of refinement for which it is added to the linked list.
2. Each edge must not be an orphan edge or a base level edge.
3. Each edge must be in the interior of subdomain Ω_i.
4. Neither of the two edge nodes may be a Dirichlet boundary node.
5. Neither of the two edge nodes may be at the interface boundary of subdomain Ω_i with another subdomain.

Every edge in the mesh hierarchy that satisfies the above conditions is added to the linked list of child edges. The one–time construction of such a linked list may take place any time after the full refinement of the base level coarse mesh and before the first iteration. We now discuss the construction of such a linked list of child edges.

Let us consider that on process i subdomain Ω_i is fully refined. This refinement may be uniform or adaptive. Here we consider the part of the mesh that corresponds to subdomain Ω_i only. Recall from §1.3.2 that edges in the mesh for each level of

refinement are stored separately (i.e. there is separate data structure for each level). To construct the desired linked list of child edges we begin at the maximum level of refinement. Note that any edge with no child edge in the mesh hierarchy is a leaf edge. Similarly an edge with no parent in the mesh hierarchy is either an orphan edge or a base level edge (base level edges are always orphan edges whereas orphan edges are not always base level edges). In either case the inclusion of such edges in the linked list of child edges is not appropriate as they have no contribution to make towards the coarsening process. Further, the TETRAD data structure relating the end nodes of each edge with the corresponding global nodes in the mesh is used to ensure that neither of the nodes at the end of an edge being added to the linked list is on the Dirichlet boundary or the interface boundary of subdomain Ω_i. Any edge that satisfies the above conditions is marked *visited* and a corresponding edge counter is iterated. This process is repeated for the edges at each level of refinement in the mesh hierarchy except for the base level mesh. Thus all child edges at each level in the mesh hierarchy are marked *visited* and counted. During a second pass through the edge lists for each level (except for base level), starting from the maximum level of refinement, the edges previously marked *visited* are added to the required linked list of child edges. Once the construction of this linked list of child edges that exist at each level of mesh hierarchy is complete, we are ready to perform coarsening which is described below.

3.5.2.1 Coarsening

We now consider the coarsening procedure which is performed at each iteration after the computation of a residual vector and before the solve of subdomain problem (3.44) on each process. It should be noted that only the part of the residual vector that corresponds to the mesh covering subdomain Ω_i and its boundary is available on process i. The remaining part of the residual vector corresponding to the mesh covering all other subdomains Ω_j (for $j \neq i$) is obtained as a result of coarsening from the fine mesh on process j to the corresponding coarse mesh of subdomain Ω_j on process i.

For the coarsening of fine mesh values from subdomain Ω_j on process j to the coarse mesh values for the corresponding subdomain Ω_j on process i, the vector of fine mesh values is available on process j. It is also the case that the construction of the linked list of child edges is such that there always exists a unique parent edge to every child edge in this list. Furthermore, for each child edge in the linked list and the corresponding parent edge, it is necessary that any one end node of the child edge is the same as one of the end nodes of the parent edge (whereas the second node of the child edge lies at the mid point of the parent edge). It is also mandatory that both end nodes of the parent edge are in the interior of subdomain Ω_j on process j. Without any loss of generality, we assume that the first end node of the child edge is the first end node of the parent edge. Then, the second end node of the child edge may or may not correspond to one of the global fine mesh nodes that are marked *match[j]* with the corresponding coarse mesh nodes from the subdomain Ω_j on process i. If it is one of these matching nodes, there is nothing further to do for this child edge. Otherwise, one half of the residual value corresponding to the second end node of the child edge is added to the residual value corresponding to the first end node of the parent edge. On the other hand, if second node of the child edge is the second end node of the parent edge, then the first node of the child edge is tested as above. If it is one of the matching nodes, as explained above, then there is nothing to do. Otherwise, one half of the residual value corresponding to the first node of the child edge is added to the residual value corresponding to the second node of the parent edge. This is repeated for each child edge in the linked list and is illustrated for one such pair of child edges in Figure 3.7.

Once the coarsening of fine mesh values from subdomain Ω_j on process j to the coarse mesh values of subdomain Ω_j on process i is completed by process j, the values corresponding to the nodes marked *match[j]* are communicated to process i and then copied to the appropriate node positions in subdomain Ω_j on process i. Note that this process of coarsening is performed on a copy of the residual vector so that the original residual vector remains unchanged. This procedure of coarsening is repeated $p-1$ times on each process so that on each process the residual vector is

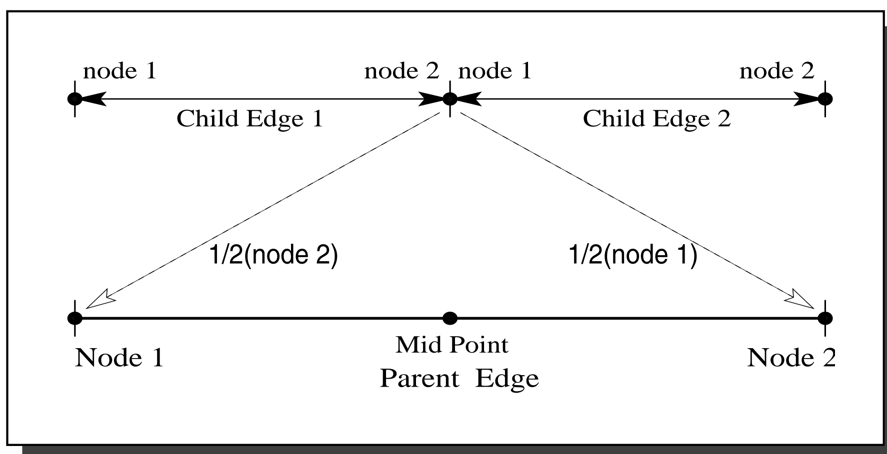

Figure 3.7: Coarsening of a residual vector.

available for all subdomains and all nodes in the fine mesh except the green nodes. The handling of these green nodes is the last milestone before we can solve the subdomain problems. Such a treatment of green nodes follows.

3.5.2.2 How to Treat Green Nodes

Let us consider process i after it has just received the necessary parts of its residual vector from all of the other processes j and copied these values to the appropriate node positions, which have been previously marked accordingly. The leftover green nodes in these subdomains are then treated separately at each iteration. We recall from §1.3.3 that green elements exist in the mesh only as the transition elements between changes in the refinement levels. Therefore it is obvious that green elements are fewer in number than regular elements in any mesh, especially in case of uniform refinement. Furthermore, the only simple access to the green nodes permitted by TETRAD is through the green elements (recall from §3.5.1 that every fourth node of a green element is a green node). For these reasons, we first extract all green elements from the final mesh and arrange them in the form of a linked list. This is another one–time task that can be completed any time after the final mesh is obtained and before the first iteration. The construction of the linked list of green elements is simpler than that of any other linked list that we have constructed so

far. Here we use the fact that TETRAD distinguishes between regular and green elements and our knowledge that any green element in the final mesh is always a leaf element, since further refinement of green elements is prohibited (see §1.3.3 for details). Thus a loop over leaf elements in the mesh and a simple test of each leaf element to judge whether it is a regular or green extracts all green elements in the mesh. The data structure that we use for these extracted green elements just requires them to be placed in a linked list. We now discuss how to treat green nodes in order to assign them appropriate values in the restricted residual vector.

For a green element in the linked list the residual corresponding to the fourth local node, which is a green node, is initialized to zero. This is important as there must always be some value at this position. Next, the parent element of this green element (which must exist) is examined. If this is found to be *visited* already, there is nothing to do for the green node corresponding to the green element currently under consideration. This parent element may be found to be *visited* between 5 and 13 times depending upon the type of green refinement of the parent element. If the parent element has not been *visited* yet, a weighted average of the residual values corresponding to local nodes of the parent element that are not on the Dirichlet boundary is added to the green node and then this parent element is marked as *visited*. Similarly for all green elements in the linked list, the residual values for the corresponding green nodes are interpolated from the values at the non–Dirichlet nodes of the parent element. With the completion of this procedure, the computation of all of the restricted residual vector entries on each process i is complete, and therefore we may proceed to solve the subdomain problems (3.44) on each process.

3.6 Preconditioning Solve

In this section we consider the sub tasks of solving the subdomain problems (3.44) concurrently on each process $i = 0, \ldots, p-1$. For our implementation of the weakly overlapping domain decomposition preconditioner under consideration, these sub-problems actually cover the entire domain of the problem, however most of the mesh

refinement is confined within a single subdomain, owned by the process. The mesh for the rest of the subdomains mostly consists of base level elements and therefore the subproblems combine the subdomain solve and a coarse mesh solve (thus replacing a separate coarse mesh solve) at each iteration. The effectiveness of this approach greatly depends on the efficiency with which we solve these subproblems at each iteration. This in turn depends upon the quality of the partition of the domain and the quality of the algorithm used to solve the subproblems.

3.6.1 Choice of Best Sequential Solver

The sequential solver that we have used here for the equations (3.44) on each process is the incomplete LU preconditioned GMRES method, taken from the SPARSKIT [151]. This has a number of parameters, such as a drop tolerance and level of fill–in, to control and tune the preconditioner. Here we discuss the incomplete LU factorization used to define the preconditioner very briefly.

An incomplete factorization of a sparse matrix A entails a decomposition of the form $A = LU - R$, where L and U have the same non–zero structure as that of the lower and upper parts of A respectively and R is the residual or error of this factorization. This factorization is known as $ILU(0)$, which is very simple and easy to compute but may lead to a crude approximation. When such a factorization is used as a preconditioner for a Krylov subspace solver it may require many iterations to converge. To remedy this drawback, several variants of incomplete factorization have been developed to allow more flexibility. Generally, a more accurate factorization requires less iterations to converge but the computational cost for such a factorization is certainly higher and so is the cost of each iteration (hence there is a trade–off which must be balanced). Some variants of the ILU factorization rely only on the level of fill–in and, as such, are blind to the numerical values in the matrix since the elements to be dropped depend only on the structure of the matrix A. However, there are alternative methods with a more sophisticated dropping strategy based on the magnitude of the elements rather than their location. The $ILUT$ preconditioner used in SPARSKIT is one of such technique which is now

briefly discussed.

An incomplete LU factorization with threshold ($ILUT$) can be derived from Gaussian elimination by adding certain conditions for dropping elements of small magnitude. There are two parameters, $ILUT(p, \tau)$, which control the dropping strategy: p is known as the level of fill–in and τ as the drop tolerance. The following two rules [152] are used in the factorization.

1. An element in row i with magnitude less than the relative tolerance τ_i is dropped where τ_i is obtained by multiplying the drop tolerance τ with the 2–norm of row i.

2. Again drop an element in row i when the magnitude is less than the relative tolerance τ_i but then only keep the p largest elements in the lower part of the row, the p largest elements in the upper part of the row and the diagonal element. The diagonal element is always kept.

The purpose of the second step is to control the number of elements in each row, and thus it helps to control the usage of memory, whereas the drop tolerance helps to control the computational cost. We are not concerned here with the implementation difficulties which may arise: we only comment on the choice of parameters p and τ in order to get the most out of the factorization that is used to precondition the sequential solver for the subproblem on each process. These parameter choices are based on our numerical experiments for different problems with uniform and local mesh refinement and in the light of recommendations in [151]. A non–zero drop tolerance of 0.005 or 0.001 usually gives reasonably good results, even for strong non–symmetries in the coefficient matrix. The recommended level of fill–in is 5 to 10 elements in each of the lower and upper part of each row plus a diagonal element. However, our experiments in three dimensions suggest that a higher amount of fill–in, for example from 20 to 25 elements in each of the lower and upper part for reasonably large problems, gives overall better results for the factorization time when compared against the iteration count.

3.7 Interpolation

The implementation of the hierarchical interpolation operation, given by equation (3.46), on the solution vectors obtained from the subproblem solves (3.44) on each process is described in this section. Since there are many similarities between components of the hierarchical interpolation operation and those of hierarchical restriction operation, we discuss only the main differences here.

3.7.1 Setup Phase

The solution vector obtained from the subproblem solve on each process i corresponds to the mesh on process i covering the whole domain (this contains nodes at the finest level in the interior and on the interface boundary of subdomain Ω_i). The following steps are prerequisites in order to interpolate the solution from the part of the mesh on process j which covers Ω_i to the fine mesh that covers Ω_i on process i.

1. Construct a list of coordinate triples for the nodes of the mesh on process i (and mark them accordingly) which are in the interior and on the interface boundary of each subdomain Ω_j for $j \neq i$ (except the green nodes which are treated separately).

2. Communicate this list of coordinate triples from process i to process j for each subdomain Ω_j on process i ($j \neq i$). Similarly process i receives $p-1$ such lists of coordinate triples; one from every other process $j \neq i$.

3. Construct a linked list of base level elements for subdomain Ω_i on process i to be used in matching the coordinate triples in step 4 below.

4. Match the coordinate triples for each of the received lists with those of the fine mesh for subdomain Ω_i on process i, and mark the matching nodes accordingly.

We note that the lists of coordinate triples for all subdomains Ω_j on process i are slightly bigger than those lists used in the setup phase for restriction due to the inclusion of the nodes on the subdomain interface with Ω_i. The computational

time is therefore expected to increase marginally. This affects the implementation very slightly for each of the above steps except for step 3 which is exactly the same as required in the setup phase for restriction. The same linked list of base level elements is used here as in the restriction operation.

3.7.2 Iteration Phase

In contrast to the setup phase for the hierarchical restriction operation and hierarchical interpolation operation, which are very similar to each other (and therefore share some of the components in their implementation), the iteration phase for these operations is the inverse of each other. However, a linked list of child edges that exist at each level in the mesh hierarchy that is not fundamentally different from the one used in the coarsening process is also used here. Hence, we introduce here only a couple of modifications in the construction of the linked list of child edges to be used in the prolongation process.

Let us again consider process i where, as usual, subdomain Ω_i is refined and the rest of the subdomains, Ω_j for $j \neq i$, have the base level coarse mesh except a single layer of refined elements at each mesh level in the neighbouring subdomains $\Omega_j \in N_i$ that surround the subdomain Ω_i. We note from step 1 in the setup phase for interpolation (see §3.7.1) that nodes at the interface boundary of subdomain Ω_j on process i are included in the list of coarse mesh coordinate triples that are matched with corresponding fine mesh coordinate triples at the interface boundary of subdomain Ω_j on process j. In order to interpolate these interface boundary nodes in the prolongation process, every child edge in the linked list of child edges (to be constructed) must satisfy only the first four of the conditions given in §3.5.2. Therefore, we get a slightly bigger linked list of child edges as compared to the one constructed previously for use when coarsening the residual vector. Note that the processes of coarsening and prolongation are the inverse of each other: typically coarsening starts from the finest level of mesh refinement and terminates at the base level mesh whilst the prolongation starts at the base level mesh and terminates at the finest level of mesh. Hence, the construction of the linked list of child edges should

reflect the operation for which it is being constructed. In the case of prolongation, the construction of such a linked list starts from the first level of the mesh hierarchy and terminates at the finest level. We now use this newly constructed linked list of child edges, that exist at each level in the mesh hierarchy for subdomain Ω_i on process i, to describe the prolongation process.

3.7.2.1 Prolongation

As the final stage of each preconditioning step, that is, after the subproblem solve on each process, the solution from each subdomain Ω_j $(j \neq i)$ on process i is prolongated to the fine mesh covering the same subdomain on process j. To achieve this, each process i sends a set of solution values, corresponding to the part of the mesh on process i that covers Ω_j $(j \neq i)$ to every other process j. Such a set consists of solution values for nodes in the interior or on the interface boundary of subdomain Ω_j, which have been previously marked in the setup phase as described in §3.7.1. Similarly, each process i receives $p-1$ sets of solution values, one from each process j for $j \neq i$. These phenomena are illustrated in Figure 3.4 and Figure 3.5 where data sets now consist of solution values. Each of solution set received by process i is then prolongated to the fine mesh of subdomain Ω_i and added to the solution values computed by process i during the subproblem solve. We now discuss this prolongation.

Let us consider the process i that has just received a set of solution values for the mesh on process j that covers Ω_i. Note that for each node in the set of received solution values, there always exists a corresponding node in subdomain Ω_i on process i (which has already been marked accordingly). Also, each child edge in the pre–formed linked list, with its corresponding parent edge, satisfies each of the properties discussed in §3.5.2.1, with the exception that any of the end nodes of an edge may lie on the interface boundary of Ω_i.

Suppose that the first end node of a given child edge is a node of the parent edge, then the second end node of the child edge may or may not correspond to one of the coarse mesh nodes on process j. If it is one of these matching nodes, there

is nothing to do for this child edge. Otherwise one half of the solution value of the first end node of the parent edge is added to the solution value of the second end node of the child edge. Similarly, if the second end node of the child edge is a node of the parent edge, then the first end node of the child edge is tested for a match as before. There is nothing further to do for this child edge if the first end node matches a node on process j as explained above. However, if the first end node is not marked as a match then one half of the solution value of the second end node of the parent edge is added to the solution value of the first end node of the child edge. This is illustrated in Figure 3.8. This procedure is repeated for each child

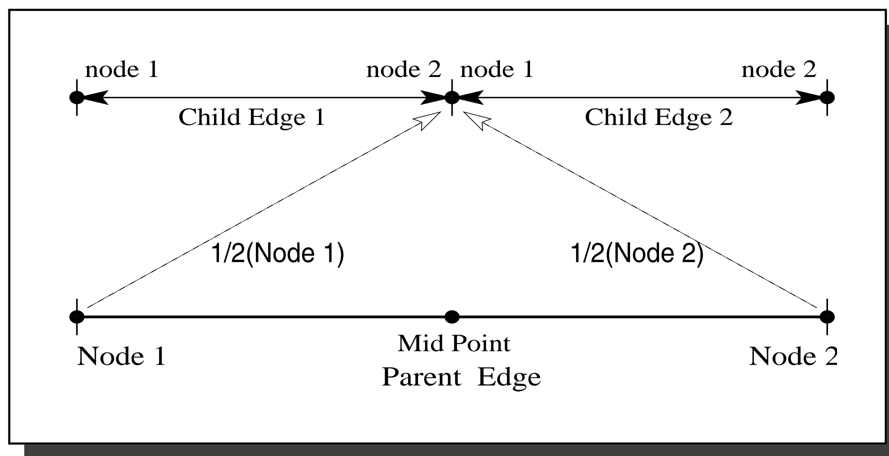

Figure 3.8: Prolongation of a solution vector.

edge in the linked list. In this way, each process i performs $p-1$ prolongations, one for each set of solution values received from every other process j.

We note that there is no prolongation associated with the green nodes. Instead the solution value for each green node is manipulated in a similar manner to that explained in §3.5.2.2 for the residual vector (and this is required only in case of local refinement of subdomain Ω_i on process i). This means that the solution of subdomain solves at green nodes never gets used when assembling the preconditioned solution in the case of uniform refinement of the subdomains. However, in case of local refinement, the solution at those green nodes in the interior of subdomain Ω_i does form part of the overall solution on that subdomain.

3.8 Computational Results

We now present some numerical results to demonstrate the rapid convergence and algorithmic efficiency of the weakly overlapping additive Schwarz DD preconditioner described in this chapter. Two sample sets of partitions into 2, 4, 8 and 16 subdomains are used for all results presented here. These are based upon two simple domain partitioning strategies shown in Figure 3.9.

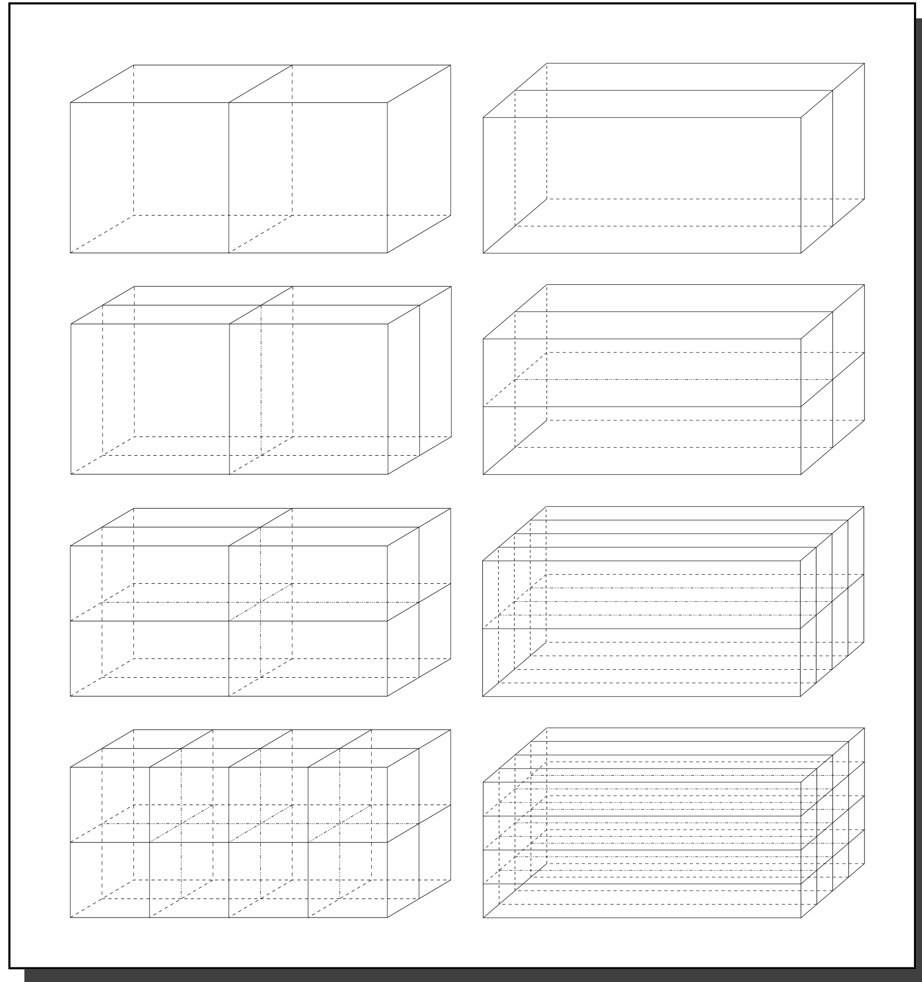

Figure 3.9: Domain partitioning strategies for 2, 4, 8 and 16 subdomains: recursive coordinate bisection (RCB) partitions (left) and anisotropic partitions (right).

The first is a simple variant of recursive coordinate bisection (RCB) [175], where cuts are made perpendicular to the longest coordinate direction of the domain. In the second case RCB is used but with the cuts constrained always to be parallel

to the x–axis. The GMRES algorithm with right preconditioning and an initial guess of $\underline{0}$ have been used for all of the presented results. A preconditioned GMRES solver with an algebraic preconditioner based on the incomplete LU factorization (as discussed §3.6) is used for the subproblem solves. It should also be noted that as the norm of the computed solution depends on p, the number of subdomains, all values of the L_2 norm and L_∞ norm quoted in this chapter are for the worst case of $p \in \{2, 4, 8, 16\}$ (which is usually when $p = 16$).

3.8.1 Uniform Refinement

For the test problems in this section, we use sequences of uniformly refined meshes $\mathcal{T}_k$. The base level coarse mesh consists of just 768 tetrahedral elements and is refined uniformly to between 1 and 4 levels. Two test problems are used but we consider then one at a time.

Test Problem 1

$$
\begin{aligned}
-\underline{\nabla} \cdot (\underline{\nabla} u) &= f \qquad on \;\; \Omega \equiv (0,2) \times (0,1) \times (0,1) \in \mathcal{R}^3 \\
u &= g \qquad on \;\; \partial\Omega.
\end{aligned}
$$

Here f is chosen to permit the exact solution $u = g = \sin(\pi x)\sin(\pi y)\sin(\pi z)$. The number of iterations required to decrease the 2–norm of the residual by a factor of 10^5 for the two representative partitioning strategies, the corresponding infinity norm of the exact error, and the L_2 norm are shown in Table 3.1 and Table 3.2. The results in Table 3.1 correspond to the RCB partitioning strategy and the results in Table 3.2 correspond to the anisotropic partitioning strategy.

Test Problem 2

$$
\begin{aligned}
-\underline{\nabla} \cdot \left(\begin{pmatrix} 10^2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \underline{\nabla} u \right) &= f \qquad on \;\; \Omega \equiv (0,2) \times (0,1) \times (0,1) \in \mathcal{R}^3 \\
u &= g \qquad on \;\; \partial\Omega.
\end{aligned}
$$

Refinement Level	Iterations				Error Norms	
(Elements/Vertices)	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	6	11	16	19	3.165×10^{-2}	2.898×10^{-3}
2(49152/9537)	7	12	17	19	9.968×10^{-3}	4.845×10^{-4}
3(393216/70785)	8	13	19	20	3.068×10^{-3}	9.070×10^{-5}
4(3145728/545025)	8	13	19	20	8.833×10^{-4}	1.565×10^{-5}

Table 3.1: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 1 using the RCB partitioning strategy.

Refinement Level	Iterations				Error Norms	
(Elements/Vertices)	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	11	11	10	3.165×10^{-2}	2.119×10^{-3}
2(49152/9537)	7	12	15	17	9.968×10^{-3}	4.967×10^{-4}
3(393216/70785)	8	13	15	16	3.068×10^{-3}	6.803×10^{-5}
4(3145728/545025)	8	14	14	15	8.832×10^{-4}	1.574×10^{-5}

Table 3.2: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 1 using the anisotropic partitioning strategy.

Again f is such that exact solution $u = g = \sin(\pi x)\sin(\pi y)\sin(\pi z)$ is satisfied. The results in Table 3.3 and Table 3.4 show the iterations required to decrease the 2–norm of the residual by a factor of 10^5, the infinity norm of the exact error, and the L_2 norm, for the RCB and anisotropic partitioning strategies respectively.

It may be observed that the number of iterations is indeed independent of h (with the possible exception of Table 3.3 where this independence is not yet evident). Furthermore, as p increases the number of iterations appears to have almost stopped growing by the time $p = 16$. We note that although it is theoretically necessary to solve the local subproblems on each process exactly, in practice a sufficiently accurate approximate solve will maintain the optimality of the preconditioner [44, 177]. Our observation, based on numerical experiments, is that reducing the residual by a

Refinement Level (Elements/Vertices)	Iterations				Error Norms	
	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	9	14	19	26	7.063×10^{-2}	2.901×10^{-3}
2(49152/9537)	11	16	22	27	3.108×10^{-3}	4.926×10^{-4}
3(393216/70785)	12	18	25	27	1.241×10^{-3}	9.153×10^{-5}
4(3145728/545025)	13	19	27	30	5.069×10^{-4}	1.565×10^{-5}

Table 3.3: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 2 using the RCB partitioning strategy.

Refinement Level (Elements/Vertices)	Iterations				Error Norms	
	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	8	12	11	11	7.063×10^{-2}	2.121×10^{-3}
2(49152/9537)	8	13	15	17	3.108×10^{-3}	4.996×10^{-4}
3(393216/70785)	9	14	15	17	1.241×10^{-3}	6.842×10^{-5}
4(3145728/545025)	9	14	15	15	5.069×10^{-4}	1.572×10^{-5}

Table 3.4: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 2 using the anisotropic partitioning strategy.

factor of 10^2 (and some times even 10^1) when solving local subproblems appears to be sufficient.

3.8.2 Non–Uniform Refinement

In this section, the effectiveness of the preconditioner is demonstrated for the local (as opposed to global) refinement of the base level coarse mesh.

Test Problem 3

$$\begin{aligned} -\underline{\nabla} \cdot (\underline{\nabla} u) &= f \qquad on\ \Omega \equiv (0,1) \times (0,1) \times (0,1) \in \mathcal{R}^3 \\ u &= g \qquad on\ \partial\Omega. \end{aligned}$$

Refinement Level (Elements/Vertices)	Iterations				Error Norms	
	p=2	p=4	p=8	p=16	L_∞	L_2
1(20832/4403)	8	12	15	21	4.594×10^{-1}	4.385×10^{-3}
2(198816/22237)	9	13	16	22	2.194×10^{-1}	1.107×10^{-4}
3(499123/100708)	9	15	17	23	1.365×10^{-1}	1.039×10^{-4}
4(2139159/429435)	10	16	19	25	7.271×10^{-2}	6.369×10^{-6}

Table 3.5: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 3 using the RCB partitioning strategy.

Refinement Level (Elements/Vertices)	Iterations				Error Norms	
	p=2	p=4	p=8	p=16	L_∞	L_2
1(20832/4403)	8	12	15	22	4.594×10^{-1}	4.232×10^{-3}
2(198816/22237)	9	14	17	23	2.194×10^{-1}	1.484×10^{-4}
3(499123/100708)	9	15	18	24	1.365×10^{-1}	8.061×10^{-6}
4(2139159/429435)	10	17	19	26	7.271×10^{-2}	6.371×10^{-6}

Table 3.6: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, the infinity norm of the error and the L_2 norm for Test Problem 3 using the anisotropic partitioning strategy.

Here f and g are chosen so that the analytic solution is given by

$$u = (1-(2x-1)^{70})(1-(2y-1)^{70})(1-(2z-1)^{70}) \quad \text{on } \Omega. \tag{3.47}$$

Note that this solution is unity in the interior of Ω but tends to zero very rapidly in a thin layer (of width ≈ 0.02) near the boundary; allowing the Dirichlet condition $u = 0$ to be satisfied throughout $\partial\Omega$.

A base level coarse mesh of 3072 tetrahedral elements is used with *a priori* local h–refinement undertaken in such a way that those mesh elements with an exact interpolation error greater than some tolerance are refined. For this problem a tolerance of 10^{-9} is used and most of the elements refined are close to the boundary of the domain Ω (i.e. in the boundary layer). The number of iterations required to

decrease the 2–norm of the residual by a factor of 10^5 are presented in the Table 3.5 and Table 3.6 (along with the infinity norm and two norm of the exact error) for the RCB and anisotropic partitioning strategies respectively. We observe that the number of iterations is almost independent of h. The independence from p is less evident however. We believe this to be due to the poor quality of both of the partitions into 16 subdomains for this particular problem: in each case the number of fine mesh elements in each mesh varies significantly for example. A full discussion of the significance of the partitioning is presented in Chapter 5.

3.9 Discussion

A symmetric version of our weakly overlapping additive Schwarz DD preconditioning algorithm for elliptic problems in three dimensions has been described in this chapter. Starting with a brief introduction to a symmetric model problem in §3.1, some theoretical background to two level additive Schwarz solution methods is given in §3.2 with details of our preconditioning algorithm in §3.3. Implementation issues are discussed from §3.4 to §3.7 and results for a small number of test problems are presented in §3.8. These include both an isotropic and an anisotropic diffusion problem, each with global uniform refinement, and an isotropic boundary layer problem with local non–uniform refinement. All of the results are presented for two different partitioning strategies. For the first, isotropic, test problem we see that the number of iterations ceased to grow after just the third level of mesh refinement for both partitioning strategies. Hence the number of iterations is independent of h and bounded above. The results for the anisotropic problem with global uniform refinement are somewhat different for the two different partitioning strategies. In both cases the number of iterations would seem to be independent of h however for the anisotropic partition this number is clearly much smaller than when using RCB. This indicates that the partitioning strategy is important for the solution of any particular problem and, although there is always a bound on the iteration count, this constant may depend significantly upon the quality of the partition. The results

for the third test problem with local mesh refinement, whilst very encouraging, also show the importance of the particular choice of partitioning strategy.

Chapter 4

A Generalized Nonsymmetric Additive Schwarz Preconditioner

The preconditioner developed in the previous chapter, for elliptic problems in three dimensions which are symmetric positive definite (SPD), can be extended to more general problems. A major motivation for this extension comes from [38] where an important observation is made in the context of what Cai and Sarkis call a "restricted additive Schwarz" method. We also note that this restricted additive Schwarz approach provides the default parallel preconditioner for nonsymmetric sparse linear systems in PETSc [160]. The generalization of this restricted additive Schwarz technique to our weakly overlapping approach in three dimensions is the main topic of this chapter. A similar generalization for elliptic problems in two dimensions can be found in [9].

In addition to symmetric problems, this chapter also considers nonsymmetric and convection–dominated elliptic problems. Since it is clear that many important and practical problems lead to nonsymmetric linear systems when discretized by the finite element method (FEM) [108], we consider a general nonsymmetric model problem which is introduced and discretized in §4.1. This is followed, in §4.2, by a description of the modifications required to generalize the existing weakly overlapping additive Schwarz preconditioner, with the implementation of the corresponding changes described in §4.3. The restricted preconditioner is then applied to the same

suit of symmetric test problems as in the previous chapter and the results are presented in §4.4, along with a discussion of the pros and cons of these two preconditioners. Application of this generalized preconditioner to nonsymmetric test problems is considered in §4.5. In §4.6 and §4.7 respectively we solve convection–dominated problems using both the standard Galerkin FEM and a more stable streamline–diffusion technique. The use of adaptivity through local refinement is considered in §4.8.

4.1 Introduction

The symmetric weakly overlapping additive Schwarz algorithm described in the previous chapter is an optimal DD preconditioner that is suitable for the parallel adaptive finite element solution of second order self–adjoint elliptic problems in three dimension. Here we generalize this algorithm is such a way that the theoretical optimality results of [10] are no longer valid but the practical performance and applicability is improved. For this purpose we consider second order elliptic problems in three dimension that are not necessarily self–adjoint (self–adjoint equations may also be considered however). One of the most important class of problems that comes into this category involves convection–diffusion equations. Such a model equation can be extracted from the general class of elliptic problems (1.1) and can be written as

$$-\varepsilon \underline{\nabla} \cdot (A \underline{\nabla} u) + \underline{b} \cdot \underline{\nabla} u = f \qquad \text{on } \ \Omega \subset \mathcal{R}^3. \tag{4.1}$$

Provided $\varepsilon > 0$ this is an elliptic problem (since A is SPD) but when $\underline{b} \neq \underline{0}$ it is not self–adjoint. Following the arguments given in §1.1.1, equation (4.1) may be discretized as follows (we again assume, for simplicity, that $u = 0$ on the Dirichlet boundary Γ_1).

First we express (4.1) as a weak formulation. Find $u \in \mathcal{H}_0^1$ such that

$$\varepsilon \int_\Omega (\underline{\nabla} v \cdot (A \underline{\nabla} u)) \ d\Omega - \varepsilon \int_{\Gamma_2} g \ v \ dS + \int_\Omega (\underline{b} \cdot \underline{\nabla} u) v \ d\Omega = \int_\Omega f \ v \ d\Omega \tag{4.2}$$

for all $v \in \mathcal{H}_0^1$. Here $\Omega \subset \mathcal{R}^3$ is the physical domain of the problem,

$$\mathcal{H}_0^1(\Omega) = \{u \in \mathcal{H}^1(\Omega) : u_{|\Gamma_1} = 0\}, \tag{4.3}$$

and Γ_2 is the Neumann boundary where $\underline{n} \cdot (A\underline{\nabla} u) = g$. For this nonsymmetric problem, the usual bilinear and linear forms are

$$\mathbb{A}(u,v) = \varepsilon \int_\Omega (\underline{\nabla} v \cdot (A\underline{\nabla} u)) d\Omega + \int_\Omega (\underline{b} \cdot \underline{\nabla} u) v \; d\Omega \tag{4.4}$$

and

$$\mathbb{F}(v) = \int_\Omega f v \; d\Omega + \varepsilon \int_{\Gamma_2} g v \; d\Omega \tag{4.5}$$

respectively. It is to be noted that the first integral in equation (4.4) is SPD whereas the second is skew–symmetric. For the finite element solution of equation (4.2) from the finite dimensional space, $\mathcal{V}^h$, of continuous piecewise linear functions on tetrahedral elements, $\mathcal{T}^h$, the following discrete problem must be solved.

Find $u^h \in \mathcal{V}^h \cap \mathcal{H}_0^1(\Omega)$ such that

$$\mathbb{A}(u^h, v^h) = \mathbb{F}(v^h) \qquad \forall \, v^h \in \mathcal{V}^h \cap \mathcal{H}_0^1(\Omega). \tag{4.6}$$

For a given choice of basis for $\mathcal{V}^h$, (4.6) leads to a system of linear equations that is similar to the one given by (1.11). For the usual choice of local finite element basis functions, Φ_i, the stiffness matrix $\mathcal{A}$ is sparse but not generally symmetric, since its entries are given by

$$\mathcal{A}_{ij} = \varepsilon \int_\Omega (\underline{\nabla}\Phi_i \cdot (A\underline{\nabla}\Phi_j)) \; d\Omega + \int_\Omega \Phi_i(\underline{b} \cdot \underline{\nabla}\Phi_j) d\Omega. \tag{4.7}$$

The nonsymmetric nature of the stiffness matrix requires a general Krylov subspace solver such as GMRES [154], as discussed in §1.6.3 and in §3.2, to be implemented. For different choices of ε, A and $\underline{b}$, we note that our model problem can represent symmetric, nonsymmetric and convection–dominated problems. We will return to these problems in §4.4 after our generalization of the weakly overlapping preconditioner in the following sections.

4.2 The Preconditioner

While the theoretical results of the previous chapter and [10] demonstrate that the preconditioner given by equations (3.44–3.46) is optimal for the class of linear self–adjoint PDEs (leading to SPD linear systems) considered, it is clear that a number of important practical problems cannot be realistically modelled by such equations. Here we consider one such class of problems, as described in §4.1. Convection–diffusion problems arise frequently in fluid mechanics, heat and mass transfer, environmental modelling, etc. and when discretized by the standard FEM [108], lead to nonsymmetric linear systems. In this section, we consider a generalization of the preconditioner from the previous chapter so that, in addition to symmetric problems, it is also suitable for nonsymmetric problems. In this regard we again refer to [38]. This paper makes the empirical observation that the regular additive Schwarz preconditioner (3.12) can be generalized to

$$\bar{\mathcal{M}} = \sum_{i=0}^{p-1} \bar{\mathcal{M}}_i^0 \mathcal{A}_i^{-1} \bar{\mathcal{M}}_i, \tag{4.8}$$

where $\bar{\mathcal{M}}_i^0$ has the action of mapping all terms corresponding to nodes outside $\bar{\Omega}_i$ to zero, preserving all terms corresponding to nodes inside Ω_i and scaling all terms on the interface of Ω_i (in our case this scaling is a simple average as described below). This modification to (3.12) not only has the effect of reducing the communication cost of each iteration but also leads to a reduction in the number of iterations required to converge provided an appropriate solver is used. In the case of our weakly overlapping preconditioner the same philosophy may be applied and so equation (3.46) is replaced by

$$\underline{z} = \sum_{i=0}^{p-1} \left(P_i^T \underline{z}_i + P_{i,s}^T D_{i,s} \underline{z}_{i,s} \right) \tag{4.9}$$

(where $D_{i,s}$ is a known diagonal scaling matrix), which means that the preconditioner is no longer SPD. Hence, even for a self adjoint differential operator, instead of the CG algorithm, a more general iterative solver such as the GMRES is required (for details of a public domain implementation see [4] or [154]). Although the cost per iteration is slightly greater for GMRES than CG, the reduced inter–process com-

munication during both the setup phase and the iteration phase for interpolation (see §3.7 for details), and the corresponding decrease in the number of iterations required to converge (see §4.4) always appears to ensure that the reduced preconditioner is more effective. Such a comparison for the weakly overlapping additive Schwarz preconditioner in two dimensions is given in [9].

Generalization of the entire weakly overlapping additive Schwarz DD algorithm to nonsymmetric preconditioning may therefore be expressed by (3.44–3.45) and (4.9). In the following section, we address the modifications required to implement this generalized nonsymmetric preconditioning algorithm. The corresponding changes in the linear system (3.41) due to the nonsymmetric nature of the problems to be considered will also be addressed. However this modification is not required in the case of symmetric problems.

4.3 Implementation

Having described our proposed amendments to the weakly overlapping AS preconditioning algorithm, in this section we discuss the corresponding implementation details. It should be noted that the following discussion is restricted to the implementation of (4.9) only as no other component of the original algorithm presented in Chapter 3 is altered. Recall from §1.1.1 and §4.1 that the standard Galerkin finite element discretization of (4.1) leads to a system of linear equations of the form (1.11) where the coefficient matrix $\mathcal{A}$ is sparse and (generally) nonsymmetric. The representation of this system of linear equations in block matrix notation can be expressed as

$$\begin{bmatrix} A_0 & & & & B_0 \\ & A_1 & & & B_1 \\ & & \ddots & & \vdots \\ & & & A_{p-1} & B_{p-1} \\ C_0 & C_1 & \dots & C_{p-1} & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_0 \\ \underline{u}_1 \\ \vdots \\ \underline{u}_{p-1} \\ \underline{u}_s \end{bmatrix} = \begin{bmatrix} \underline{b}_0 \\ \underline{b}_1 \\ \vdots \\ \underline{b}_{p-1} \\ \underline{b}_s \end{bmatrix}, \tag{4.10}$$

where $\underline{u}_i$, $\underline{b}_i$ (for $i = 0, \ldots, p-1$), $\underline{u}_s$ and $\underline{b}_s$ have their usual meanings as explained in the previous chapter. Also the blocks A_i, B_i, C_i and $\underline{b}_i$ are sparse and can be assembled on each process independently. The blocks A_s and $\underline{b}_s$ consist of contributions from all processes, given by (1.29) and (1.30) respectively, which again can be assembled and stored independently on each process. It is clear that an iterative solver suitable for nonsymmetric problems, such as GMRES [154], is required to solve this system of linear equations. The distributed matrix–vector product and the distributed inner product required for parallel implementation of GMRES are discussed in §1.6.4. On each process i, for $i = 0, \ldots, p-1$, note that (4.10) may, after a suitable permutation, be expressed as

$$\begin{bmatrix} A_i & & B_i \\ & \bar{A}_i & \bar{B}_i \\ C_i & \bar{C}_i & A_{i,s} \end{bmatrix} \begin{bmatrix} \underline{u}_i \\ \underline{\bar{u}}_i \\ \underline{u}_{i,s} \end{bmatrix} = \begin{bmatrix} \underline{b}_i \\ \underline{\bar{b}}_i \\ \underline{b}_{i,s} \end{bmatrix}. \tag{4.11}$$

Here $\underline{u}_i$ still represents the unknowns inside Ω_i but $\underline{u}_{i,s}$ represents the unknowns on the interface of Ω_i with other subdomains and $\underline{\bar{u}}_i$ represents all remaining unknowns. The other blocks are formed similarly. Application of the generalized nonsymmetric preconditioner, $\bar{\mathcal{M}}$ say, by considering the action of $\underline{z} = \bar{\mathcal{M}}^{-1}\underline{y}$ is then given below.

Following the notation of Chapter 3 (equations (3.44–3.45)), on each process i, for $i = 0, \ldots, p-1$, solve the system

$$\begin{bmatrix} A_i & & B_i \\ & \bar{A}_i & \bar{B}_i \\ C_i & \bar{C}_i & A_{i,s} \end{bmatrix} \begin{bmatrix} \underline{z}_i \\ \underline{\bar{z}}_i \\ \underline{z}_{i,s} \end{bmatrix} = \begin{bmatrix} \underline{y}_i \\ M_i\underline{\bar{y}}_i \\ \underline{y}_{i,s} \end{bmatrix}, \tag{4.12}$$

where

$$\begin{aligned} \underline{y}_i &= P_i\underline{y} \\ \underline{\bar{y}}_i &= \bar{P}_i\underline{y} \\ \underline{y}_{i,s} &= P_{i,s}\underline{y}. \end{aligned} \tag{4.13}$$

This is exactly as described in §3.4 and, as before, all subproblems are solved independently and concurrently. On completion of these subproblem solves at each

iteration, their contribution towards the overall parallel preconditioned solution is now determined by equation (4.9) instead of (3.46). This operation requires communication with the neighbouring processes only (as opposed to the global communication required by (3.46)), matching the interface boundary nodes with the same nodes from the neighbouring subdomains. Implementation details of these communications of interface node values, and their subsequent scaling, is described below.

After the preconditioning solve (4.12) is complete, each process i has subdomain solution values for three different types of node: nodes interior to subdomain Ω_i, nodes exterior to subdomain Ω_i (which are discarded for the rest of the computation) and nodes at the interface boundary of subdomain Ω_i. Only those values obtained from the preconditioning solve on process i which correspond to the interior and the interface boundary nodes of subdomain Ω_i contribute towards the overall parallel solution. Furthermore, the subproblem solution values corresponding to the interface boundary nodes need to be adjusted so that the contribution from each of the sharing processes can be averaged. For this purpose, a list of subproblem solution values for those interface boundary nodes of subdomain Ω_i which are common with each neighbouring subdomain $\Omega_j \in N_i$ are sent to process j. Similarly, process i receives such list of subproblem solution values from each neighbouring process j, for the interface boundary nodes of subdomain Ω_i which are common with the neighbouring subdomains $\Omega_j \in N_i$. The number of such lists to be sent from, and received by, each process depends upon the number of neighbouring processes. These received subdomain solution values from each neighbouring process j are then added to the corresponding subdomain solution values for the interface boundary nodes of subdomain Ω_i on process i. As a result, for each node on the interface boundary of subdomain Ω_i, process i holds a sum of solution values from all sharing processes. These solution values are then scaled by the number of sharing processes. Hence, for each node at the interface boundary, we get an average of the subproblem solution values from all sharing processes (and therefore the solution value for a particular node at the interface boundary is same on each of the sharing processes).

We note that our discussion of the generalized preconditioner described in §4.2, and its implementation, focused on the modification of equation (3.46) to equation (4.9). This means that the preconditioner is no longer SPD. However, it is still equally suitable for symmetric problems as for nonsymmetric problems. It is shown in the next section that one of the benefits of applying this nonsymmetric preconditioner to a symmetric problem is a decrease in the number of iterations required for convergence.

4.4 Application to Symmetric Problems

In order to demonstrate the quality and convergence behaviour of the generalized nonsymmetric preconditioner, we present here numerical results for the same set of test problems as considered in §3.8. The computational domain is again partitioned into 2, 4, 8 and 16 subdomains (see Figure 3.9) using the RCB and the anisotropic partitioning strategies. The results are obtained using GMRES with right preconditioning and an initial guess of $\underline{0}$. Here we remind the reader that as the exact error depends on p, the number of subdomains, so does the norm of this error. Therefore, in this and the following sections, all values of the L_∞ norm and the L_2 norm presented are the maximum of the values over all choices of $p \in \{2, 4, 8, 16\}$.

4.4.1 Uniform Refinement

In this section we present the results for two test problems: the first problem is a simple Poisson equation (Test Problem 1) and the second problem is a more demanding anisotropic diffusion equation (Test Problem 2).

In each case the domain $\Omega = (0, 2) \times (0, 1) \times (0, 1)$ and Dirichlet boundary conditions are applied throughout $\partial\Omega$. The results in Tables 4.1–4.4 show the quality and convergence behaviour of the generalized nonsymmetric preconditioning algorithm for a tolerance of 10^{-5} relative to the initial residual. Each of the subdomain problems is solved approximately but sufficiently accurately to ensure that the optimality of the preconditioner is maintained. A base level coarse mesh of just 768

tetrahedral elements is used and is refined uniformly between 1 and 4 levels.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	3	4	4	3.165×10^{-2}	2.898×10^{-3}
2(49152/9537)	12	3	5	5	6	9.968×10^{-3}	4.845×10^{-4}
3(393216/70785)	22	4	6	7	8	3.068×10^{-3}	9.070×10^{-5}
4(3145728/545025)	42	4	8	9	9	8.833×10^{-4}	1.565×10^{-5}

Table 4.1: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 1 using the RCB partitioning strategy.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	3	4	5	3.165×10^{-2}	2.119×10^{-3}
2(49152/9537)	12	3	4	6	6	9.968×10^{-3}	4.967×10^{-4}
3(393216/70785)	22	4	5	7	8	3.068×10^{-3}	6.804×10^{-5}
4(3145728/545025)	42	4	7	9	10	8.832×10^{-4}	1.574×10^{-5}

Table 4.2: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 1 using the anisotropic partitioning strategy.

We note that in all cases the increase in the number of iterations is very slow as the number of subdomains and the level of refinement is increased. The difference in the number of iterations shown in Table 4.3 and Table 4.4 is similar to that seen in §3.8.1 for the symmetric preconditioning algorithm. Also the number of iterations required in each case is significantly less than the number required by the original symmetric preconditioning algorithm to reduce the residual by the same factor (Tables 3.1–3.4) – in some cases the iteration count is less than one half of the number required by the original symmetric preconditioner. This clearly demonstrates the superiority of the generalized nonsymmetric preconditioner.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	6	7	8	8	12	7.063×10^{-2}	2.901×10^{-3}
2(49152/9537)	10	9	10	10	13	3.108×10^{-3}	4.926×10^{-4}
3(393216/70785)	18	10	11	12	15	1.241×10^{-3}	9.153×10^{-5}
4(3145728/545025)	34	11	13	14	18	5.069×10^{-4}	1.565×10^{-5}

Table 4.3: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 2 using the RCB partitioning strategy.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	6	3	3	3	3	7.063×10^{-2}	2.121×10^{-3}
2(49152/9537)	10	4	4	5	5	3.108×10^{-3}	4.996×10^{-4}
3(393216/70785)	18	5	5	6	6	1.241×10^{-3}	6.842×10^{-5}
4(3145728/545025)	34	5	6	6	7	5.069×10^{-4}	1.572×10^{-5}

Table 4.4: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 2 using the anisotropic partitioning strategy.

In Tables 4.1–4.4 an iteration count is also given (in the column labelled ILU) for when the corresponding test problem is solved using the *ILU* preconditioned GMRES iterative solver [151, 152]. We recall from §3.6.1 that this is the same sequential solver that is used for the solution of the subdomain problems at each preconditioning step. The nature of this preconditioner, like our generalized preconditioner, is nonsymmetric. The results quoted are obtained by using an amount of fill–in equal to 25 and a drop tolerance of 0.005 to achieve a relative tolerance of 10^{-5}. The benefit of the drop tolerance approach to the incomplete factorization of the sparse matrix, into lower and upper triangular matrices, is that it is not blind to the numerical values of the sparse matrix entries. In fact the parameters, p, the level of fill–in and τ, the drop tolerance, supplement each other in determining the best possible

factorization into the L and U parts, taking into account both the space available for fill–ins and the numerical values of the matrix entries to keep or drop. This is in contrast to the approach of using only the level of fill–in which depends only upon the structure of the sparse matrix but not its particular numerical values. It is clearly evident that the number of iterations required to converge almost doubles every time the mesh is refined by one level. This is in contrast to the near–optimal behaviour of our weakly overlapping generalized nonsymmetric two level additive Schwarz preconditioner, where the number of iterations grows only very slowly as the mesh is refined. Throughout the rest of this chapter an ILU column has been included in all of the tables of iteration counts so that the growth in the number of iterations of our additive Schwarz preconditioner may be contrasted with this for each of the test problems that we consider.

4.4.2 Non–Uniform Refinement

The effectiveness of the generalized nonsymmetric preconditioner when applied to a symmetric problem with non–uniform local h–refinement is demonstrated by the results presented in this section for Test Problem 3 (previously considered in §3.8.2 using the symmetric preconditioning algorithm). For this problem, as in §3.8.2, a slightly larger base level coarse mesh of 3072 tetrahedral elements for the domain $\Omega = (0,1) \times (0,1) \times (0,1)$ is used.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(20832/4403)	12	6	7	7	9	4.594×10^{-1}	4.384×10^{-3}
2(198816/22237)	15	9	9	10	12	2.194×10^{-1}	1.107×10^{-3}
3(499123/100708)	17	9	10	12	13	1.365×10^{-1}	1.039×10^{-4}
4(2139159/429435)	18	10	12	14	16	7.271×10^{-2}	3.369×10^{-6}

Table 4.5: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 3 using the RCB partitioning strategy.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(20832/4403)	12	7	7	10	10	4.594×10^{-1}	4.232×10^{-3}
2(198816/22237)	15	8	10	11	12	2.194×10^{-1}	1.484×10^{-3}
3(499123/100708)	17	9	10	12	14	1.365×10^{-1}	8.061×10^{-6}
4(2139159/429435)	18	11	14	16	17	7.271×10^{-2}	6.371×10^{-6}

Table 4.6: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 3 using the anisotropic partitioning strategy.

As for the uniform refinement case, from Table 4.5 and Table 4.6 we are able to observe that the number of iterations required to converge is again less than the number required by the original symmetric preconditioning algorithm (see Tables 3.5 and 3.6). Also the increase in the number of iterations is again quite slow as the number of subdomains and the refinement level are increased, despite the fact that the refinement is only local in nature. Again, the overall superiority of the generalized nonsymmetric preconditioning algorithm is apparent.

4.4.3 Generalized Preconditioner: Pros and Cons

In view of the results presented in §3.8 and this section, for the same set of test problems, using our weakly overlapping symmetric and generalized nonsymmetric additive Schwarz preconditioners respectively we may now consider the advantages and the disadvantages of these two preconditioners. For some of these observations we refer to [9, 38] whereas others reflect our numerical experiments during the code development and testing.

1. The symmetric preconditioner is based on a precise mathematical theory whereas the generalized nonsymmetric preconditioner is a modification of this based only on empirical observations (with no theoretical justification).

2. The symmetric preconditioner works well with CG, a specialized solver for symmetric problems, whereas the generalized nonsymmetric preconditioner requires a more general solver: GMRES for example.

3. The memory requirement of CG, used with the symmetric preconditioner, is lower than that of GMRES, required for the generalized nonsymmetric preconditioner, as it requires all previous Krylov vectors (search directions) to be stored.

4. The number of iterations required to converge (to a particular tolerance) using the symmetric preconditioner is always higher than that required by the generalized nonsymmetric preconditioner.

5. The inter–process communication cost for the setup phase and the iteration phase is higher (by about 100%) for the symmetric preconditioner than for the generalized nonsymmetric preconditioner. This is because the interpolation is replaced by a simple scaling step which requires communication with neighbouring processes only. This clearly reduces the cost of each iteration.

6. With the decreased communication cost and the decreased number of iterations, the generalized nonsymmetric preconditioner is the preferred preconditioner for parallel or distributed computing.

In rest of this chapter we provide evidence in support of a final observation, that the generalized nonsymmetric preconditioner may also be applied successfully to nonsymmetric and convection–dominated problems.

4.5 Application to Nonsymmetric Problems

The results presented in §4.4 demonstrate the superiority of the generalized nonsymmetric preconditioner over our original symmetric preconditioner for a variety of symmetric problems. In this section, we assess the quality of the generalized nonsymmetric preconditioning algorithm for nonsymmetric elliptic problems. We begin

by considering the following general nonsymmetric elliptic problem:

$$\begin{aligned} -\underline{\nabla} \cdot (\underline{\nabla} u) + \underline{b} \cdot \underline{\nabla} \quad u &= f \quad \text{on } \Omega \\ u &= g \quad \text{on } \partial\Omega. \end{aligned} \tag{4.14}$$

As before we consider the effectiveness of the generalized nonsymmetric preconditioning algorithm for nonsymmetric problems in terms of the iteration count for a given tolerance. The infinity norm and the two norm of the error are also provided. Two specific test problems are considered, see below, and the same two different partitioning strategies are applied as in §3.8.

Test Problem 4

$$\begin{aligned} \underline{b} &= \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \\ u &= (x-1)^2 \left(y - \frac{1}{2}\right)^2 \left(z - \frac{1}{2}\right)^2, \\ f &= 2(x-2) \left(y - \frac{1}{2}\right)^2 \left(z - \frac{1}{2}\right)^2 + 2(x-1)^2 \left(y - \frac{3}{2}\right) \left(z - \frac{1}{2}\right)^2 \\ &+ 2(x-1)^2 \left(y - \frac{1}{2}\right)^2 \left(z - \frac{3}{2}\right). \end{aligned}$$

Test Problem 5

$$\begin{aligned} \underline{b} &= \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \\ u &= \left(x - \frac{2(1-e^x)}{1-e^2}\right) y(1-y)z(1-z), \\ f &= \left(x - \frac{2(1-e^x)}{1-e^2}\right) \Big((3-2y)z(1-z) + y(1-y)(3-2z) \Big) + y(1-y)z(1-z). \end{aligned}$$

The two test problems are solved on the domain $\Omega = (0,2)\times(0,1)\times(0,1)$ subject to Dirichlet boundary conditions. Results for Test Problem 4 are given in Table 4.7

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	6	4	5	6	7	1.163×10^{-3}	1.060×10^{-4}
2(49152/9537)	12	3	5	6	6	5.977×10^{-4}	1.899×10^{-5}
3(393216/70785)	24	4	5	6	7	2.148×10^{-4}	3.452×10^{-6}
4(3145728/545025)	46	3	5	6	7	6.689×10^{-5}	5.767×10^{-7}

Table 4.7: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 4 using the RCB partitioning strategy.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	6	4	5	5	7	1.163×10^{-3}	1.846×10^{-4}
2(49152/9537)	12	5	6	12	18	5.977×10^{-4}	2.336×10^{-5}
3(393216/70785)	24	5	5	7	8	2.148×10^{-4}	4.970×10^{-6}
4(3145728/545025)	46	4	5	7	9	6.689×10^{-5}	7.601×10^{-7}

Table 4.8: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 4 using the anisotropic partitioning strategy.

and Table 4.8 for the RCB and the anisotropic partitioning strategies respectively. For Test Problem 5 the corresponding results are given in Table 4.9 and Table 4.10. In each case the GMRES iterative solver with right preconditioning and an initial guess of $\underline{0}$ was run until a relative residual of 10^{-5} was achieved. As can be seen, the generalized nonsymmetric preconditioner behaves in an excellent manner for these two nonsymmetric test problems using both partitioning strategies. It may be observed that the total number of GMRES iterations grows very slowly as the number of processes (subdomains) is increased or the mesh is refined (all calculations are performed with a base level mesh of just 768 tetrahedral elements). The strange jump in the number of iterations at level 2 in Table 4.8 is completely out of line however and is not yet understood by the author.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	4	4	4	6.358×10^{-4}	9.912×10^{-5}
2(49152/9537)	13	4	5	5	6	2.029×10^{-4}	1.261×10^{-5}
3(393216/70785)	26	4	6	7	7	5.908×10^{-5}	3.028×10^{-6}
4(3145728/545025)	51	4	8	9	10	1.913×10^{-5}	3.889×10^{-7}

Table 4.9: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 5 using the RCB partitioning strategy.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	4	4	5	6	6.358×10^{-4}	5.083×10^{-5}
2(49152/9537)	13	3	5	7	7	2.029×10^{-4}	1.156×10^{-5}
3(393216/70785)	26	4	6	8	9	5.908×10^{-5}	1.603×10^{-6}
4(3145728/545025)	51	5	8	10	11	1.913×10^{-5}	3.708×10^{-7}

Table 4.10: The number of GMRES iterations at different levels of refinement to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 5 using the anisotropic partitioning strategy.

4.6 Convection–Dominated Problems

There is a great deal of practical interest in producing numerical methods which can approximate the solutions of convection–diffusion equations such as (4.1), in the case where the convection term dominates. It is well known however that if there is a dominant convection term then standard numerical methods perform very poorly, or fail to work at all. For example, classical finite element schemes such as Galerkin methods frequently yield numerical solutions which suffer from very large, unrealistic and non–physical oscillations. A number of numerical methods have been devised to overcome these difficulties, usually through the introduction of a certain amount of artificial dissipation (often in a problem–specific manner). Of these, along with

others such as artificial diffusion, polynomial upwinding, crosswind diffusion, etc., the streamline–diffusion method [108] has been particularly successful (see §1.1.2). In this section we consider a model convection–dominated equation which is non self–adjoint due to the directionality introduced by the presence of an odd order derivative. It is first solved using the Galerkin method and the nonsymmetric preconditioner outlined above. In the following section the same preconditioner will also be shown to be effective for a more stable discretization. The model convection dominated equation is:

$$\begin{aligned} -\varepsilon\underline{\nabla}^2 u + \underline{b}\cdot\underline{\nabla}\quad u &= f \qquad \text{on } \Omega \\ u &= g \qquad \text{on } \partial\Omega, \end{aligned} \tag{4.15}$$

where $\varepsilon > 0$ is a constant diffusion coefficient (assumed to be such that $\varepsilon << \|\underline{b}\|$), $\underline{b}$ is a constant convection vector and, for simplicity, homogeneous Dirichlet boundary conditions are applied.

To gain insight into this class of convection–dominated equation consider the following analogous one dimensional problem [137]:

$$-\varepsilon\, u''(x) + u'(x) = 1 \quad \text{for} \quad 0 < x < 1, \tag{4.16}$$

with $u(0) = u(1) = 0$. This one dimensional equation yields the analytical solution

$$u(x) \quad = x + \tfrac{e^{-1/\varepsilon} - e^{-(1-x)/\varepsilon}}{1-e^{-1/\varepsilon}}. \tag{4.17}$$

The first term is this solution satisfies (4.16) and the boundary condition $u(0) = 0$. The second term ensures that the other boundary condition is satisfied. Thus, when $\varepsilon << 1$ the solution essentially satisfies a pure convection problem except in a region close to 1, where the solution exhibits an exponential boundary layer as shown in Figure 4.1. A solution plot for a similar problem in two dimensions with an exponential boundary layer increasing towards the corner at $x = 1, y = 1$, is shown in Figure 4.2. In higher dimensions it is difficult to visualize the solution but (4.15) can certainly yield a $u(x, y, z)$ with similar behaviour.

As indicated above, a number of different numerical schemes exist for the solution of convection–dominated equations. In the next section we will apply a stable

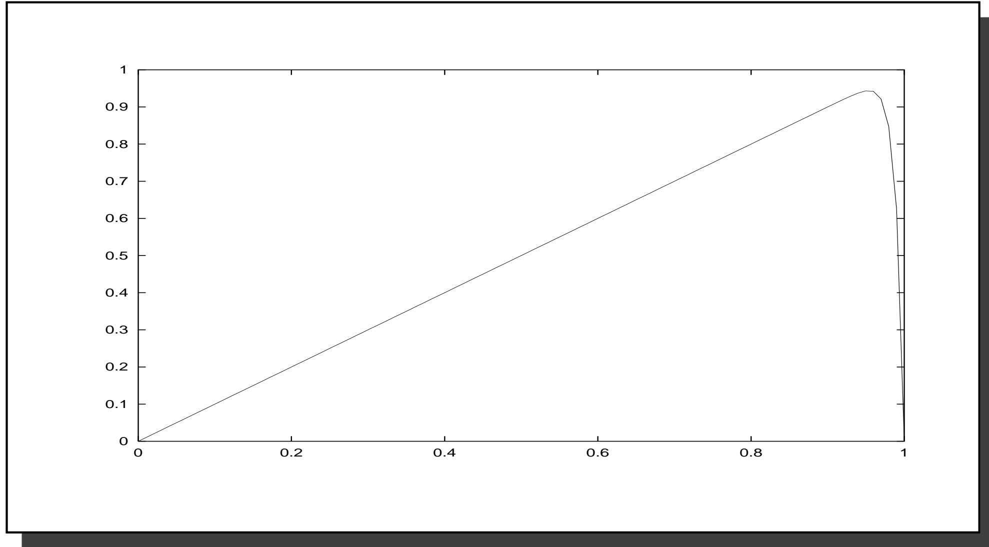

Figure 4.1: Solution plot for u given by (4.17) and $\varepsilon = 0.01$.

streamline–diffusion finite element scheme however for the rest of this section we present numerical results for the Galerkin finite element method. Although this is not stable for large mesh Peclet numbers (when ε/h is significant) we will see that our weakly overlapping preconditioner is still effective so long as the mesh is not too coarse.

4.6.1 Computational Results

In order to assess the quality of the generalized nonsymmetric preconditioner on a convection–dominated problem we consider the specific test problem, of the form (4.15), given below.

Test Problem 6

$$\begin{aligned} \underline{b} &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\ u &= \left(x - \frac{2(1-e^{x/\varepsilon})}{1-e^{2/\varepsilon}} \right) y(1-y)z(1-z) \\ f &= 2\varepsilon \left(x - \frac{2(1-e^{x/\varepsilon})}{1-e^{2/\varepsilon}} \right) \Big(\; y(1-y) + z(1-z) \; \Big) + y(1-y)z(1-z) \end{aligned}$$

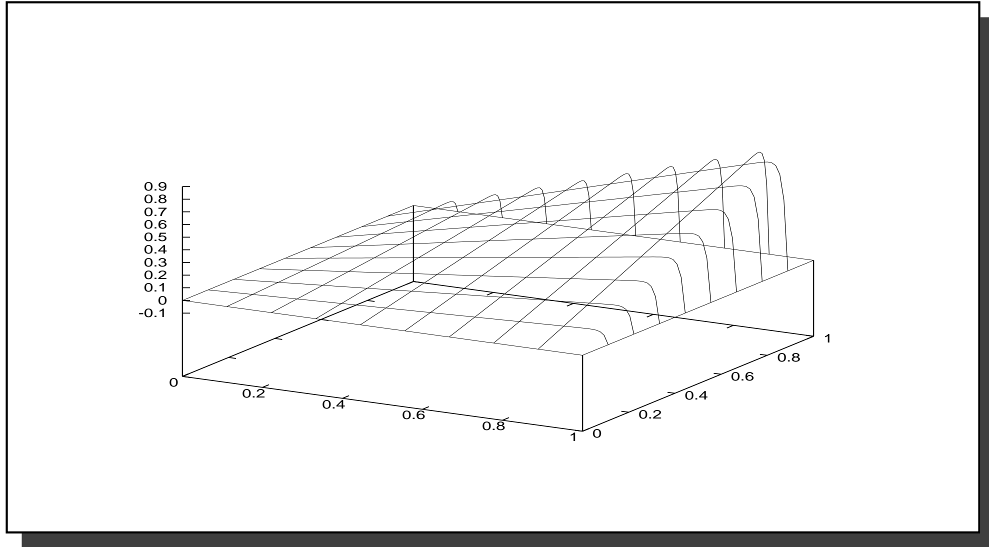

Figure 4.2: Analogous solution plot for a two–dimensional generalization of Figure 4.1.

This problem is solved on the domain $\Omega = (0,2) \times (0,1) \times (0,1)$ subject to the Dirichlet boundary conditions on $\partial\Omega$. The solution exhibits a steep layer of size $O(\varepsilon)$ near the boundary $x = 2$ when $0 < \varepsilon << \|\underline{b}\| = 1$. It should be noted that for $\varepsilon = 1.0$ and $\underline{b} = (1,1,1)^T$, this test problem reduces to Test Problem 5 considered in §4.5.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	4	4	5	3.534×10^{-3}	4.022×10^{-4}
2(49152/9537)	14	3	5	5	6	1.464×10^{-3}	7.068×10^{-5}
3(393216/70785)	27	4	6	7	8	5.497×10^{-4}	1.013×10^{-6}
4(3145728/545025)	54	4	7	9	10	1.903×10^{-4}	2.442×10^{-7}

Table 4.11: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	9	4	4	5	8	1.022×10^{-1}	8.208×10^{-4}
2(49152/9537)	11	4	4	5	6	1.005×10^{-1}	1.717×10^{-4}
3(393216/70785)	17	3	4	5	7	5.215×10^{-2}	1.266×10^{-5}
4(3145728/545025)	32	3	5	6	8	1.884×10^{-2}	1.650×10^{-6}

Table 4.12: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	45	11	15	16	32	2.300×10^{-1}	1.882×10^{-3}
2(49152/9537)	29	7	8	8	12	2.732×10^{-1}	3.372×10^{-4}
3(393216/70785)	19	6	6	6	9	2.641×10^{-1}	9.800×10^{-5}
4(3145728/545025)	23	5	5	5	8	2.105×10^{-1}	8.841×10^{-6}

Table 4.13: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$.

Tables 4.11–4.16 show the number of iterations required to solve the discrete finite element system to a moderately high level of accuracy using GMRES with our parallel implementation of the weakly overlapping generalized nonsymmetric DD preconditioner for two different partitioning strategies (see §3.8 for details). Results are presented for a sequence of meshes which represent between one and four levels of uniform refinement of a coarse tetrahedral mesh containing just 768 elements. It may again be observed that, as the mesh is refined or the number of processes (subdomains) is increased, the total number of GMRES iterations increases only very slowly. It is not clear however whether this increase will be bounded as the mesh

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	3	4	5	3.534×10^{-3}	4.432×10^{-4}
2(49152/9537)	14	3	5	6	7	1.464×10^{-3}	7.671×10^{-5}
3(393216/70785)	27	4	6	8	8	5.497×10^{-4}	1.226×10^{-5}
4(3145728/545025)	54	4	7	9	10	1.903×10^{-4}	2.592×10^{-6}

Table 4.14: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	9	3	3	4	4	1.022×10^{-1}	9.043×10^{-4}
2(49152/9537)	11	3	3	4	4	1.005×10^{-1}	1.717×10^{-4}
3(393216/70785)	17	3	4	4	5	5.215×10^{-2}	1.266×10^{-5}
4(3145728/545025)	32	4	5	6	6	1.884×10^{-2}	1.650×10^{-6}

Table 4.15: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$.

size tends to zero as is proved in [10] for the symmetric version of the preconditioner applied to pure diffusion problems.

It can be observed from Tables 4.11–4.13 that, for the RCB partition, the number of iterations required to converge increases as the value of ε decreases on the two coarsest meshes. For the finest mesh however, the number of iterations required to converge decreases with a decrease in the value of ε. This behaviour may be understood by looking at the infinity norm of the error in these three tables. On the coarse grid the error grows as ε is decreased due to the instability of the Galerkin scheme. The oscillations in the solution are proving hard for the preconditioned

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	45	8	10	12	13	2.230×10^{-1}	2.149×10^{-3}
2(49152/9537)	29	5	6	7	8	2.732×10^{-1}	3.372×10^{-4}
3(393216/70785)	19	4	5	5	5	2.641×10^{-1}	9.800×10^{-5}
4(3145728/545025)	23	4	4	4	4	2.105×10^{-1}	8.841×10^{-6}

Table 4.16: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$.

iterative solver to resolve. On the finest grid however h is just sufficiently small to be able to resolve the solution when $\varepsilon = 10^{-3}$ and so the performance of the preconditioner is not adversely affected.

Similar results are presented in Tables 4.14–4.16 for the anisotropic partition. Provided the mesh Peclet number is small enough we actually see a reduction in the number of iterations required as ε is decreased. A similar observation is made when solving the problem sequentially using the package described in [151]. In this case the preconditioner is based upon an incomplete LU (ILU) factorization of the finite element matrix, and this is able to exploit the weak coupling between neighbouring nodes with a common edge that is perpendicular to $\underline{b}$ when the problem is convection dominated. Hence, although the discrete problem is more non-symmetric in this case, it turns out to be less complex to solve in practice.

4.7 Streamline–Diffusion Method

The use of standard finite element discretization methods for convection–dominated problems generally leads to oscillatory numerical solution if the underlying mesh is not sufficiently refined [88]. Such behaviour is evident in the numerical results obtained using the Galerkin FEM in the previous section. In Table 4.13 and Ta-

ble 4.16 for example, the infinity norm of the error is hardly decreasing as the mesh is refined. For one dimensional problems the exact character of these non–physical oscillations is well understood, e.g., for linear elements for a problem with mesh size h and constant convection velocity $\underline{b}$, oscillations will occur if the ratio $|\underline{b}|h/2\varepsilon$ is greater than one [66]. The same is not true for problems in two or three dimensions however. For example, for the two–dimensional case useful, closed–form, solutions are hard to find and are often difficult to interpret when they are found [89]. A limited discussion of oscillations in the standard finite element solution can also be found in [157]. We are not aware of any work describing the precise nature of the oscillations observed when using the Galerkin method for convection–dominated equations in three dimensions with tetrahedral mesh elements.

A number of techniques have been proposed in an attempt to resolve this problem of non–physical oscillations. In particular, the use of stabilized finite element formulations [83, 101] has revolutionized the numerical analysis of convection–dominated problems in last decade or so. Nevertheless, this recent progress still lacks a completely methodological approach to fully resolving some of the key issues, such as the choice of stabilization parameters [33]. Typically the choice of a quasi–optimal parameter is based upon some problem specific empirical studies. For this reason the investigation of *parameter free* techniques for the solution of these problems has begun to receive some recent attention: see, for example, [33, 149] and references therein.

For the reminder of this section we restrict our attention to the streamline–diffusion approach introduced in §1.1.2. Recall from that section that this involves an upwinding parameter α defined by (1.27) and that a suitable choice of δ is obtained when α is of the form given by (1.27). If the problem is diffusion dominated then the best choice is $\alpha = 0$, in which case the streamline diffusion formulation (1.23) reduces to the standard Galerkin formulation (1.10). In practice, when using the streamline–diffusion method to compute a solution with shock waves or boundary layers, an appropriate choice of δ in (1.27) is crucial. In particular, the following two aspects should be considered.

1. It is possible to over–stabilize so as to get a smooth solution which is fundamentally inaccurate.

2. The choice of δ influences the performance of the iterative solver that is applied to the resulting linear algebraic system.

These two issues are related to each other in the sense that a good choice of δ to get a stable and accurate solution often also triggers relatively rapid convergence of the iterative solver [81]. In view of these observations, our choice of δ for the solution of the convection–dominated problems in this and the following sections is based upon the trial and error approach suggested in [33]. The value of $\delta = 0.11$ is used in the following computations.

4.7.1 Computational Results

The numerical results presented in §4.6.1 suggest that the generalized nonsymmetric preconditioner works well in practice for three dimensional convection dominated problems provided the mesh is sufficiently fine. However, when coarser meshes are used the standard Galerkin method becomes oscillatory and the performance of the preconditioner is affected.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	4	4	5	6.008×10^{-3}	4.181×10^{-4}
2(49152/9537)	14	3	5	5	6	2.779×10^{-3}	6.425×10^{-5}
3(393216/70785)	27	4	6	7	8	1.409×10^{-3}	1.071×10^{-5}
4(3145728/545025)	54	4	7	9	10	6.947×10^{-4}	2.194×10^{-6}

Table 4.17: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	4	4	6	6.803×10^{-2}	5.527×10^{-4}
2(49152/9537)	10	3	4	4	6	6.242×10^{-2}	1.138×10^{-4}
3(393216/70785)	17	3	4	5	7	2.954×10^{-2}	1.252×10^{-5}
4(3145728/545025)	32	3	5	6	8	8.104×10^{-3}	1.652×10^{-6}

Table 4.18: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$.

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	8	5	5	5	7	1.545×10^{-2}	68.78×10^{-4}
2(49152/9537)	9	4	5	5	7	2.390×10^{-2}	5.566×10^{-5}
3(393216/70785)	12	4	5	5	6	2.096×10^{-2}	1.283×10^{-5}
4(3145728/545025)	17	3	4	5	7	1.848×10^{-2}	1.651×10^{-6}

Table 4.19: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the RCB partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$.

In Tables 4.17–4.22 we now present numerical results for the solution of the same convection–dominated problems using a more stable finite element technique based upon the streamline–diffusion method. As expected, we see that the stabilized FEM provides a less oscillatory solution with a smaller error in terms of the infinity norm, and that the improvement over the standard Galerkin FEM is clearest when the mesh is coarse and the value of ε is small, that is, the mesh Peclet number is largest.

Given that the streamline–diffusion method works well for small values of ε, in Tables 4.17–4.22, the iteration counts for the preconditioned GMRES algorithm (implemented in parallel with right preconditioning) are seen to be very low in all

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	3	4	5	6.008×10^{-3}	4.583×10^{-4}
2(49152/9537)	14	3	5	6	7	2.779×10^{-3}	7.173×10^{-5}
3(393216/70785)	27	4	6	7	8	1.409×10^{-3}	1.303×10^{-5}
4(3145728/545025)	54	4	7	9	10	6.947×10^{-4}	2.399×10^{-6}

Table 4.20: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-1}$.

Refinement Level	Iterations					Error Norms	
(Elements/Vertices)	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	7	3	3	3	4	6.803×10^{-2}	5.527×10^{-4}
2(49152/9537)	10	3	3	4	4	6.242×10^{-2}	1.138×10^{-4}
3(393216/70785)	17	3	4	4	5	2.954×10^{-2}	1.252×10^{-5}
4(3145728/545025)	32	3	5	6	6	8.104×10^{-3}	1.652×10^{-6}

Table 4.21: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-2}$.

cases. The comparison with the solution of the discrete Galerkin equations given in Tables 4.11–4.16 demonstrates the effectiveness of the solver for all of the values of ε considered. It is also the case that for small values of ε and relatively coarse meshes, the streamline–diffusion solutions are obtained in fewer iterations. In each case the effectiveness and the quality of the preconditioner is demonstrated in terms of the small numbers of iterations required. Finally, we observe that for the most convection–dominated problems, the anisotropic partition (appropriately aligned with the anisotropy in the problem) provides converged results in fewer iterations than the isotropic RCB partition. This is consistent with the results obtained earlier

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(6144/1377)	8	2	3	3	3	1.545×10^{-2}	7.291×10^{-4}
2(49152/9537)	9	3	3	4	4	2.390×10^{-2}	5.566×10^{-5}
3(393216/70785)	12	3	3	4	4	2.096×10^{-2}	1.283×10^{-5}
4(3145728/545025)	17	3	4	4	4	1.848×10^{-2}	1.651×10^{-6}

Table 4.22: The number of GMRES iterations at different levels of refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 6 using the anisotropic partitioning strategy and $\varepsilon = 1.0 \times 10^{-3}$.

for the anisotropic diffusion problem (Test Problem 2).

4.8 Local Adaptivity

So far we have discussed the finite element solution of convection–dominated problems using uniform refinement of the subdomain meshes. It has been observed however that the solution of Test Problem 6, whilst smooth almost everywhere, varies very rapidly in a layer of size $O(\varepsilon)$ near the boundary $x = 2.0$ (where $0 < \varepsilon << \|\underline{b}\| = 1$). Whilst it is necessary for the mesh to be refined in this boundary layer in order to approximate the solution accurately, refinement is not necessarily required everywhere else in Ω. This is typical for convection–dominated problems, which frequently admit solutions containing steep boundary or internal layers. A further example of such a problem is given by Test Problem 7 below.

Test Problem 7

$$\begin{aligned} \underline{b} &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \\ u &= \frac{e^{-x/\varepsilon} - 1}{e^{-x/\varepsilon} - e^{(2-x)/\varepsilon}}, \\ f &= 0. \end{aligned}$$

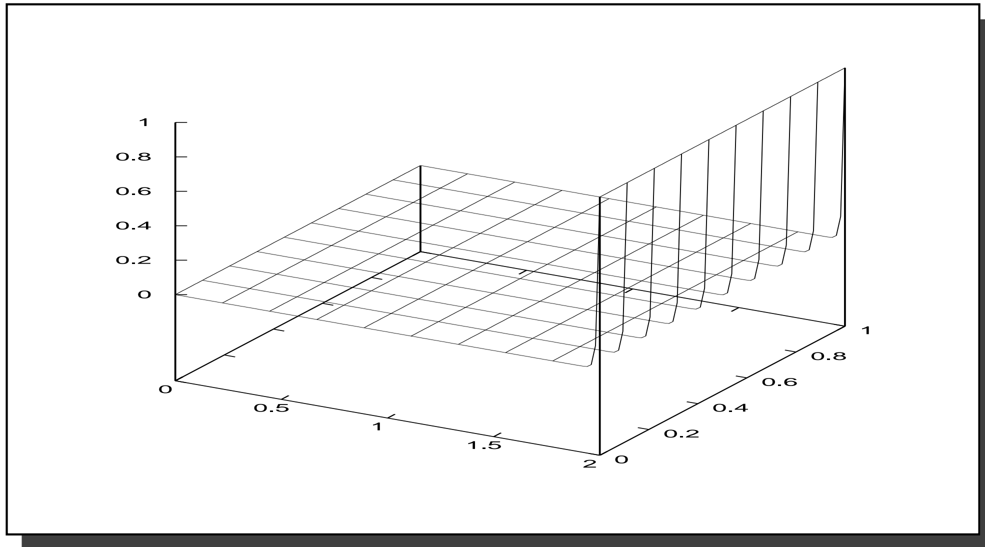

Figure 4.3: Solution plot when u is a function of x only, convection is dominating along x–axis and $\varepsilon = 0.01$.

The nature of the solution u is illustrated (in two dimensions for clarity) in Figure 4.3 for $\varepsilon = 0.01$. From this figure it is clear that mesh refinement only in the region near $x = 2.0$ should be sufficient in order to accurately approximate the solution. Moreover, Figure 4.3 also suggests that uniform refinement throughout the region in the immediate neighbourhood of $x = 2.0$ is required (i.e. no variation in y and z is necessary). Provided we use the anisotropic partitioning strategy introduced in §3.8 it should be possible to allow local adaptive refinement to take place in the boundary layer in a load–balanced manner. This is discussed further in Chapter 5.

This problem is also solved on the domain $\Omega = (0,2) \times (0,1) \times (0,1)$ using a coarse mesh that consists of just 768 elements and the Dirichlet boundary conditions applied throughout $\partial\Omega$. Table 4.23 shows some typical results for Test Problem 7 using local adaptive refinement around the region $x = 2.0$. In order to control this refinement, only the elements in a layer of size $O(\varepsilon)$ near the region $x = 2.0$ are refined. The results in Table 4.23 show the iteration count, the infinity norm and the two norm of the error for a sequence of locally refined grids. We again see only a very slow growth in the number of iterations as the mesh is refined or the

Refinement Level (Elements/Vertices)	Iterations					Error Norms	
	ILU	p=2	p=4	p=8	p=16	L_∞	L_2
1(2560/673)	8	4	5	5	6	6.628×10^{-1}	5.622×10^{-3}
2(9728/2166)	9	4	5	5	6	5.647×10^{-1}	9.127×10^{-4}
3(38400/7835)	10	4	5	5	6	2.567×10^{-1}	5.289×10^{-5}
4(153088/29920)	13	5	6	7	7	6.705×10^{-2}	9.628×10^{-6}

Table 4.23: The number of GMRES iterations at different levels of local refinement required to reduce the residual by a factor of 10^5, plus the infinity norm and the two norm of the error, for Test Problem 7 (when $\varepsilon = 0.01$) using the anisotropic partitioning strategy.

number of subdomains is increased. The number of iterations taken again appears to be bounded by a constant as the mesh is refined or the number of subdomains is increased.

4.9 Discussion

A nonsymmetric generalization of our 3–d symmetric weakly overlapping two level additive Schwarz preconditioner, proposed in Chapter 3, has been described. This generalization is based upon the empirical observations made in [38] for more conventional additive Schwarz preconditioners and in [9] for our weakly overlapping preconditioner in two dimensions. Results have been presented to show that even when applied to the solution of SPD problems this nonsymmetric generalization is superior to the original symmetric algorithm. Empirically we observe that the overall number of iterations required is cut by about 50%.

Furthermore, a nonsymmetric model problem is introduced in §4.1 and the modifications required for the generalization of the symmetric algorithm to the nonsymmetric algorithm are discussed in §4.2. The corresponding implementation details are described in §4.3. Following consideration of three SPD test problems in §4.4 a couple of nonsymmetric test problems are considered in §4.5. Here the excellent

performance of the generalized nonsymmetric preconditioning algorithm is shown to apply to a simple convection–diffusion problem, despite the lack of any theoretical justification for this.

At this point we note that the iteration counts achieved in this work compare favourably with those reported in other published work, e.g. [37, 38, 159]. In [38] the so called 'restricted additive Schwarz' (RAS) preconditioner shows a growth in the number of iterations from 13 to 18 and from 15 to 20 for the two dimensional Poisson and convection–diffusion equations respectively when the overlap is varied from 3 to 1 (on a 128 by 128 mesh). A symmetrized version of this preconditioner, incooprating harmonic overlaps (RASHO), is reported in [37]. In this case the number of iterations varies between 19 and 28 with a variation of overlap between 3 and 1 for the Poisson equation in two dimensions (also on a 128 by 128 mesh). Other examples against which we may compare include the optimal DD preconditioner for linear elasticity problems described in [159]. The number of iterations required to converge in this 2–d work varies between 10 and 12 for a variation in the overlap from 8 and 1 when using membrane elements. Similar iteration counts are obtained using shell elements.

In §4.6 we go on to consider convection–dominated problems and results are presented for a specific test case where convection dominates along the x–axis. Three different values of the diffusion coefficient ε and two different partitioning strategies are considered. For this anisotropic problem, the number of iterations required is lower for the (appropriately aligned) anisotropic partition. Furthermore, the expected oscillations in the solution using the standard Galerkin finite element method are observed. To overcome this oscillatory behaviour, we consider use of the streamline–diffusion method in §4.7 and results are presented in §4.7.1 for the same test problems considered in the previous section. Once more we see that our preconditioner works well for this class of linear system and again we find that the number of iterations required is lower for the anisotropic partition, particularly when the convection is more dominant. Also the non–physical oscillations in the solution have been removed. Finally, in §4.8, we consider solving a convection–dominated

problem using local, rather than global, refinement. In order to ensure a reasonable load balance for the parallel solver, results are presented for the anisotropic partition only. The number of iterations again appears to be almost independent of the level of local mesh refinement and the number of subdomains.

Chapter 5

Parallel Performance

Performance analysis of a parallel program is generally based upon the time taken by a number of processors to complete the execution of the program. Various quantities such as speedup, efficiency and scalability may be calculated by varying the number of processors and the size of the problem. There may be several issues which effect the performance of a parallel program and some of these are listed below.

- The quantity and patterns of the inter–processor communications.
- The imbalance in the computational load between different processors.
- The level of the parallel overhead (i.e. the additional computations that must be undertaken by the parallel algorithm compared to the best available sequential algorithm).

In addition to these it is also apparent that for DD methods such as those being considered in this thesis the geometric shape of subdomains may also have an effect on parallel performance.

The quality and effectiveness (in terms of iteration counts) of our symmetric weakly overlapping two level additive Schwarz preconditioning algorithm has been assessed in Chapter 3 whereas that of the generalized nonsymmetric algorithm has been assessed in Chapter 4 for a variety of test problems. The comparison of the two algorithms in §4.4.3 clearly suggests that the generalized nonsymmetric DD

preconditioning algorithm is superior since the number of iterations required to converge, for the same test problems, are significantly fewer. Furthermore, lack of the full interpolation step in the generalized preconditioner ensures that the inter–processor communication cost (which consists of the *setup phase* and the *iteration phase*) is almost half that of the symmetric algorithm at each iteration. Thus the generalized nonsymmetric preconditioning algorithm is the obvious choice for the assessment of parallel performance. In the following section we describe one of the simplest ways of assessing the performance of a parallel program and discuss in more detail some of the issues which can effect the parallel performance. Parallel results for a typical set of test problems are then presented in §5.2 followed by a discussion in §5.3. Throughout this chapter, unless explicitly stated to the contrary, it will be assumed that the parallel implementation is such that each process runs on a different processor and that each processor is identical.

5.1 Assessment of Parallel Performance

Performance of a parallel program may be assessed in terms of speedup, which is defined as the ratio of the time required to solve a problem using the best available algorithm (and implementation) on a single processor to the time required to solve the same problem using the parallel algorithm on p processors. Another measure is the parallel efficiency which is defined as the ratio of speedup to the number of processors. The efficiency indicates how well the processors are being used, that is, the higher the parallel efficiency, the better the use of the processors. Unfortunately, for a fixed size of problem, an increase in the number of processors generally causes the parallel efficiency to deteriorate. Therefore, it tends to be larger problems which benefit most from the use of parallel computers. The scalability of an algorithm is a measure that tries to express how much it is able to benefit from increased amounts of parallelism (which is what large machines are designed to provide [160]).

There are several issues which affect the performance of a parallel program and need to be taken into account. We address here those issues which we believe have a

substantial, direct or indirect, effect on the performance of our weakly overlapping preconditioning algorithm.

5.1.1 Decomposition

It has been observed in both Chapters 3 and 4 that the domain partitioning strategy that is used affects the number of iterations required to converge. In this context recall that for the isotropic problems, the number of iterations required for the RCB partition are generally less than those required for the anisotropic partition (see Tables 4.7 and 4.8 for example). Conversely, the anisotropic partition (appropriately aligned) tends to lead to fewer iterations than the RCB partition for the anisotropic problems (see Tables 4.11–4.16 and 4.17–4.22 for example). This is an important indication that the subdomain shape plays an important role in the convergence behaviour of our additive Schwarz preconditioning algorithm. Similar observations have been made for other DD–based solution algorithms (e.g. [23, 74, 76, 168, 169]) and has led to a number of lines of research into partitioning strategies that can take this into account [51, 155, 171].

Most automatic mesh partitioning algorithms attempt to minimize the cut–edge weight, a cost which approximates the communication volume required in the parallel application of Krylov subspace iterative methods such as CG or GMRES. Whilst minimization of the communication volume is an important achievement in any parallel application, when the convergence of the solver is influenced by the shape of the subdomains the overall computational cost may be more dependent on the number of iterations than on the communication overhead. Just as with the FEM, where the condition number of the discrete matrix system is determined by the aspect ratio of elements, the condition number of the preconditioned system can be dependent on the aspect ratio of subdomains [171]. From the point of view of this work, we observe from the results presented in Chapter 4, that for isotropic problems a small aspect ratio is better (e.g. the RCB partition), whereas for the anisotropic problems a large aspect ratio can be best (e.g. the anisotropic partition). This observation is taken into account in the presentation of the results quoted in §5.2 for a typical set

of test problems.

5.1.2 Communication

In this section we discuss some of the main implementation issues pertaining to the inter–processor communication required for the parallel implementation of our DD preconditioner. We also comment on the effects of the synchronization points which are a necessary part of the distributed inner products that are required for the parallel application of each Krylov subspace iteration.

We begin by discussing the inter–processor communication required for the restriction procedure (see §3.5 for details) which forms part of both the symmetric and the nonsymmetric preconditioning algorithms. The communication for the full interpolation procedure (see §3.7 for details) is required only by the symmetric preconditioning algorithm and has a similar pattern to that required by the restriction procedure. This is replaced in the generalized nonsymmetric preconditioning algorithm by the scaling of interface boundary terms (see §4.2 for details) which is discussed below. The restriction procedure consists of two phases; the *setup phase* and the *iteration phase*. In the setup phase on processor i, a data set corresponding to each subdomain Ω_j, for $j \neq i$, is selected and then communicated to processor j. Similarly, processor i receives a data set from every other processor $j \neq i$. In the iteration phase, a coarsening step is carried out on processor i for each processor $j \neq i$, and the resulting values are than communicated to processor j. Processor i then receives its own set of coarsened values from every other processor $j \neq i$. Thus every processor communicates (send and receive) with every other processor once in the setup phase and once in the iteration phase (which is repeated at each GMRES iteration). This quantity of global communication is due to our implementation of the additive Schwarz approach: where each processor undertake its own coarse grid solve in addition to its own weakly overlapping subdomain solve. A more conventional two level additive Schwarz implementation might perform the coarse grid solve on a single processor only, which would require less global communication. The remaining inter–processor communication would then be restricted to

immediate neighbours only, as illustrated in Figure 4.2. In retrospect this is a clear

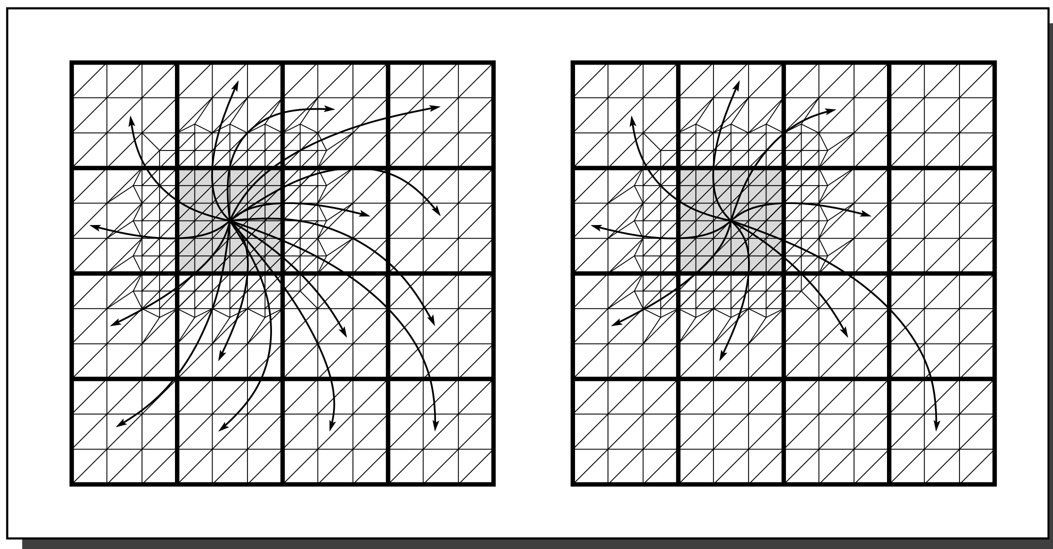

Figure 5.1: Communication required by a typical processor for our implementation of the two level additive Schwarz preconditioner (left), and the communication required by the same processor when the coarse grid solve is undertaken on a single processor (right).

disadvantage of our decision to incorporate the coarse grid solve into the local solve undertaken on each processor and almost affects the solution times which are shown in §5.2.

It should also be noted that although the anisotropic partition is better in terms of iteration counts for the anisotropic problems, it causes a higher volume of communication amongst the processors due to the larger subdomain interface boundary compared to the RCB partition. Also the size of subdomain problems becomes greater due to a larger overlap region, hence the computational cost per iteration increases in addition to the higher volume of inter–processor communication. This also proves to be significant in the results presented below.

Synchronization is defined as a point in a parallel algorithm that all processors are required to reach simultaneously. Having many synchronization points tends to increase the total time spent by processors waiting (rather than computing) [160], therefore a good implementation is usually one with as few synchronization points as

possible. For the GMRES algorithm at least one synchronization point is required at each iteration, when the inner products are carried out as part of the Gram–Schmidt orthogonalization step.

5.1.3 Load Balancing

For any parallel algorithm to perform well it is necessary to achieve a good load balance across the processors. Often allocation of one subdomain to each processor can result in an uneven work distribution since, in practice, there is often some variation in the size of individual subdomains. The coarse grid problem adds another source of load imbalance if solved on a single processor [160]. Today a number of heuristics and software tools (see §2.6 for details) are available which can partition almost any mesh into an arbitrary number of well–balanced submeshes. They can also maintain the balance dynamically in the situation of adaptive mesh refinement during the solution process. Dynamic load balancing is in itself a topic of much current research in parallel computing but is beyond the scope of this thesis. Here we restrict our calculations to the two representative partitioning strategies shown in Figure 3.9 and consider the consequences of global uniform and local non–uniform refinement of the subdomains in turn.

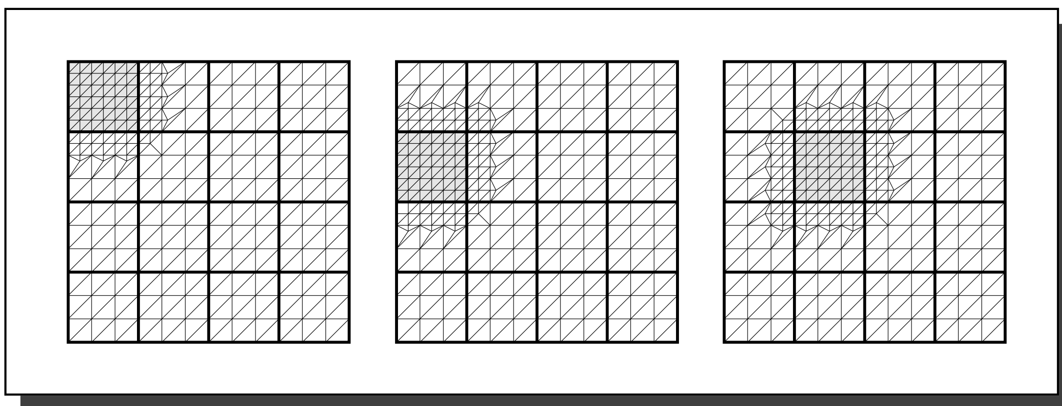

Figure 5.2: Global uniform refinement of three representative subdomains in an isotropic partition.

Consider the global uniform refinement of the three different subdomains shown

in the two dimensional example of Figure 5.2. In the situation where a subdomain at the corner is refined, a single layer of elements from the three neighbouring subdomains is also refined at each level. For a subdomain which shares one of its sides with the boundary of the domain Ω, there are five neighbouring subdomains which each require a layer of refinement at each level. The number of neighbouring subdomains increases to eight when a subdomain is completely inside the domain Ω. A similar sketch can be drawn for the three–dimensional RCB partition (also into sixteen subdomains) shown in Figure 5.3, where there are only two types of subdomain: those which are at a corner and those which are not at a corner of Ω. The subdomains at a corner each have seven neighbours and the others each have eleven neighbours.

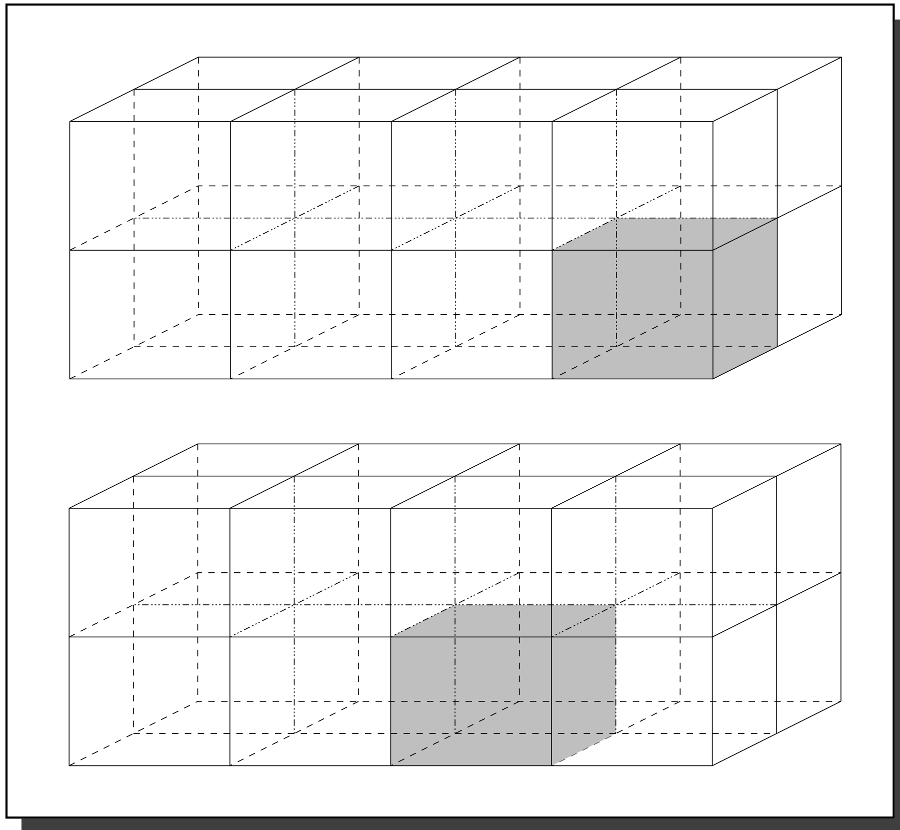

Figure 5.3: Two representative subdomains and their neighbouring subdomains for the RCB partition of $\Omega = (0,2) \times (0,1) \times (0,1)$ into sixteen subdomains.

We recall that in our weakly overlapping approach, each subproblem is solved

on the entire domain but only the subdomain Ω_i owned by the processor i is refined, along with a single layer of refinement at each mesh level of the neighbouring subdomains $\Omega_j \in N_i$. Due to this refinement of elements from the neighbouring subdomains the size of each subproblem is affected by the number of neighbouring subdomains and the overall length of the interface. Thus, even when each subdomain is an identical size, the higher the number of neighbouring subdomains, the larger the size of the subproblem and vice versa. For the RCB partition of the domain $\Omega = (0,2) \times (0,1) \times (0,1)$ shown in Figure 5.3, each subdomain has the same number of neighbouring subdomains when Ω is partitioned into 2, 4 or 8 subdomains (see Figure 3.9). However, the partition into 16 subdomains creates two types of subdomain: with either seven or eleven neighbouring subdomains. This is a source of slight load imbalance.

For the anisotropic partition the situation is slightly different. Each subdomain has the same number of neighbours only when the domain Ω is partitioned into 2 or 4 subdomains. The partition into 8 subdomains generates two types of subdomain: with either three or five neighbours. Finally, there are three types of subdomain generated when the Ω is partitioned into 16 subdomains: with either three, five or eight neighbours (again, see Figure 3.9). Refinement in this latter case leads to a significant load imbalance. For example, four levels of refinement of an initial grid with 768 elements leads to subproblems with between 300400 and 429816 elements in this case. (In contrast, when the RCB partition into 16 subdomains is used, the same refinement yields subproblems with between 276216 and 307406 elements. This difference in size is due to the subdomain boundary being longer for the anisotropic partition than the RCB partition.) Perhaps the best way to overcome this difficulty in practice would be to employ less processors than subdomains and solve more than one subproblem on each processor.

The situation of local non–uniform refinement is somewhat more complex in the sense that it is almost totally problem dependent. Recall that for Test Problem 3, the refinement of a thin layer (of width ≈ 0.02) near the boundary of the domain Ω is sufficient to resolve the solution to a reasonable accuracy. Hence, an optimal

partition for this problem will assign to each subdomain an approximately equal share of the domain boundary. However, in the case of Test Problem 7, mesh refinement near the boundary $x = 2.0$ only is sufficient to resolve the boundary layer. Therefore, the RCB partition of the coarse mesh would distribute the workload highly unevenly since only the subdomains containing the part of domain boundary at $x = 2.0$ would be refined. This is not the case for the anisotropic partition however which leads to a reasonably good load–balance in this particular situation.

When there is an imbalance in the workload of the processors some subproblem solves will be completed earlier than the others. When this occurs prior to a synchronization point these processors will have to wait for subproblem solves to be completed on all of the other processors. The issue of waiting, often referred to as idle time, is addressed briefly in the following section.

5.1.4 Parallel Overhead

Parallel overhead may be defined as the total time taken by all activities which are not required when the same problem is solved sequentially. This includes extra computation, inter–processor communication and waiting time. There are numerous opportunities for a parallel program to do extra computation. Some of the most obvious are performing calculations not required by the best sequential program and replicated calculations on more than one processor. Another example of parallel overhead comes in the call to reduction operations (e.g. `MPI_Reduce` or `MPI_Allreduce`), where the bulk of the time for such functions is taken up with communication. The need for inter–processor communication and its effects on parallel performance have been discussed in §5.1.2. Here we describe the time that some processors may spend waiting or idle. It should be noted that our implementation is such that waiting or idle time is minimized by overlapping the communication with useful computations whenever possible, as will be described shortly.

Recall from §5.1.3 that some processors are typically allocated a greater workload than others. Hence, in the preconditioning step, some processors complete the solution of their subdomain problem earlier than others. In the preconditioned

GMRES algorithm this preconditioning step is soon followed by the Gram–Schmidt orthogonalization step, which requires parallel inner products to be computed. At this point those processors completing their subdomain solve first must wait for the rest before they can continue.

A second source of idle time may be found in the coarsening procedure. Let us consider the coarsening that is being performed on processor i. Note that the number of coordinate triples from subdomain Ω_i on processor i that match coordinate triples from Ω_i on processor j ($j \neq i$) is greatest when j is neighbour of i due to the single layer of refinement around subdomain Ω_j on processor j. Therefore, the computational cost of the coarsening procedure on processor i depends upon the number of neighbouring subdomains that subdomain Ω_i has. When Ω is partitioned into 16 subdomains (as well as into 8 subdomains for the anisotropic partition), the number of neighbours per subdomain varies (see above) and therefore so does the total cost of the coarsening phase.

We note that processor i performs $p - 1$ coarsening operations, one for each other processor $j \neq i$ and then communicates the restricted residual to the same processor $j \neq i$. Similarly, each processor receives $p-1$ restricted residuals, one from every other processor. This communication (of restricted residuals) and computation (coarsening) are overlapped to minimize the parallel overhead. In the setup phase, communication (of selected data) and computation (data collection and matching of fine and coarse meshes) are also overlapped.

5.2 Sample Execution

In this section we assess the parallel performance of our generalized nonsymmetric preconditioning algorithm described in Chapter 4 by considering a specific implementation of the preconditioned GMRES algorithm using ANSI C and the MPI communication library [125, 126]. We consider a selection of representative test problems (previously considered in Chapters 3 and 4). Global uniform refinement is used for the first three of these test problems and local non–uniform refinement

for the last one. The calculations presented in Tables 5.1 to 5.7 are performed on a SG Origin 2000 computer which has a non–uniform memory access (NUMA) architecture. The non–uniform nature of the memory access means that timings of a given calculation may vary significantly between runs depending upon how memory has been allocated, hence, even for dedicated runs, timings of the same run may vary by a few percent according to allocation of the memory. For this reason, all timings quoted in the following tables represent the best time that is achieved over numerous repetitions of the same computation.

In this section we have chosen four representative test problems amongst those previously considered in Chapters 3 and 4. These consist of an isotropic problem (Test Problem 1), a nonsymmetric problem (Test Problem 4) and a convection–dominated problem (Test Problem 6). These test problems are solved using a fine conforming mesh of 3145728 tetrahedral elements obtained by global uniform refinement of the subdomains from a base level coarse mesh consisting of just 768 tetrahedral elements. The last problem considered is again convection–dominated (Test Problem 7) but is solved using a fine mesh of 153088 tetrahedral elements obtained through local non–uniform refinement. Again a base level coarse mesh of just 768 tetrahedral elements is used. For each of these test problems either the RCB partition or the anisotropic is used, with the exception of Test Problem 6 which is solved for both partitions. The choice of partitioning strategy is somewhat problem dependent and is briefly described when each problem is discussed in turn.

For each of the test problems considered here the figures quoted are referred to as Parallel Time and Sequential Time, and corresponding Speedups are provided. The Parallel Time consists of the time taken by the parallel preconditioned implementation of the GMRES solver: this includes the time required to refine the base level coarse mesh to the desired level of refinement (level four for the problems solved using global uniform or local non–uniform refinement) and to assemble the sparse matrices for the local and global systems of linear equations plus the time taken by both the setup phase (one time tasks only) and all operations in the iteration phase of the preconditioned GMRES algorithm. The Speedup is defined as the time

required to solve the problem using the best available (to us) sequential algorithm (and implementation) on a single processor divided by the time required to solve the problem using the parallel algorithm on p processors. The Sequential Time consists of the same ingredients which make up the Parallel Time but with the difference that this time is for the sequential implementation of the same preconditioner for p subdomains. The ratio of this Sequential Time to the Parallel Time is used to demonstrate the level of parallelism for the p subdomain implementation. This is referred as the Parallel Speedup in our tables. Note that one can assess the quality of the preconditioner for p subdomains by comparing the sequential time with that obtained for other choices of p and with the best sequential time.

Before presenting any of these results, we highlight a number of important issues relating to the solution of all of the problems considered in this section. These include the accuracy to which the subdomain problems at the preconditioning step are solved on each processor at each iteration and the drop tolerance and the level of fill–in that should be used in the sequential ILU preconditioner [151] that is used for these subproblems. The first of these issues that we address is the accuracy to which it is necessary to solve subproblems at the preconditioning step. If these subproblems are solved very accurately then unnecessary time is wasted since they are only an intermediate step in the overall solution process. On the other hand, highly inaccurate solutions lead to a reduction in the quality of the preconditioner and an increased number of GMRES iterations. As mentioned in the previous chapters, a reduction in the two norm of the residual by a factor of 10^2 (and some times even 10^1) appears to give near–optimal solution times. In the timings presented in this section, the quoted figures are always obtained for the best of these two parameter choices. This brings us on to the second of the issues noted above. It is well understood that the best sequential solution algorithm may vary from one problem to another within the wide class of elliptic PDEs considered in this thesis. The best general sequential linear system solver available to us is GMRES with an ILU preconditioner (as implemented in [151]), however this contains a number of adjustable parameters such as the amount of fill–in and the drop tolerance. Our choice of these

Processors	p=1	p=2	p=4	p=8	p=16
Parallel Time	751.80	567.79	489.87	331.99	131.70
Speedup	–	1.3	1.5	2.3	5.7
Sequential Time	–	1123.06	1908.46	2506.54	1814.79
Parallel Speedup	–	2.0	3.9	7.6	13.8

Table 5.1: The performance of the parallel solver for the Galerkin FE discretization when solving Test Problem 1 using the RCB partition. The times are quoted in seconds and the speedups are relative to the best sequential solution time.

parameters, determined empirically, may well also contribute to some variation in timings.

We conclude this preamble by noting that there are a number of additional factors that limit the efficiency of the current parallel implementation. In all cases the sequential execution time of the preconditioned solver (which is different for each choice of p) is greater than that of the best available sequential solver. Hence even a perfect parallel implementation would not deliver 100% efficiency. Furthermore, the deficiencies in our simple mesh partitioning and coarse grid solution strategies, outlined in §5.1, certainly contribute to an increased parallel overhead.

We now discuss in turn each of the test problems considered and the corresponding computational results. The figures in Table 5.1 correspond to Test Problem 1 which is solved using the RCB partition of the domain $\Omega = (0,2) \times (0,1) \times (0,1)$. The solution of this problem is smooth and isotropic therefore the RCB partition is preferred over the anisotropic partition. This choice is based not only on the number of iterations required to converge when using the two partitions (see Tables 4.1 and 4.2) but also because both the surface–area to volume ratio and total surface area of the subdomains is smaller for this partition. Hence the communication volume is smaller (see §5.1.2) and the overlapping subdomains have a better load balance (see §5.1.3). It is interesting to note from Table 5.1 that there is a large positive jump in performance between the 8 and 16 processors cases. This is because, as shown by the Sequential Time, 16 subdomains provide a more efficient algorithm

Processors	p=1	p=2	p=4	p=8	p=16
Parallel Time	804.16	443.39	321.44	218.07	105.90
Speedup	–	1.8	2.5	3.7	7.6
Sequential Time	–	882.39	1231.92	1628.51	1459.41
Parallel Speedup	–	2.0	3.8	7.5	13.8

Table 5.2: The performance of the parallel solver for the Galerkin FE discretization when solving Test Problem 4 using the RCB partition. The times are quoted in seconds and the speedups are relative to the best sequential solution time.

than 8 subdomains and so could lead to faster solutions on 8 processors if we were to allow more than one subdomain per processor. This is not surprising for the RCB partition given the subdomain shapes (see Figure 3.9).

A similar behaviour pattern can be observed for the results in Table 5.2 for Test Problem 4: a nonsymmetric problem where odd order derivatives are present. Recall that, for this problem, the number of iterations required to converge for the RCB partition (see Table 4.7) is always less than the number of iterations required for the anisotropic partition (see Table 4.8). Hence the isotropic nature of the solution of this problem appears to imply that the parallel performance using the RCB partition will always be the better of the two strategies considered here. The RCB partitions are therefore used for the results given in Table 5.2. We also observe that the performance for this particular problem is significantly better than that given in Table 5.1. Since the parallel speedups are about the same in this example however, we conclude that the reason for the improved performance in this case is solely due to the improvement in the Sequential Time of the p subdomain solver. As before, a large jump in performance between 8 and 16 processors is also observed for this problem, providing further evidence to suggest that the use of more subdomains than processors could lead to a better parallel performance.

Now we consider the more anisotropic situation given by Test Problem 6, where convection dominates along the x–axis. We consider this problem for two values of the diffusion coefficient ε, 1.0×10^{-2} and 1.0×10^{-3} for both the RCB and the

Processors	p=1	p=2	p=4	p=8	p=16
Parallel Time	770.65	497.39	341.78	227.98	113.81
Speedup	–	1.5	2.3	3.4	6.8
Sequential Time	–	984.72	1326.91	1725.23	1597.15
Parallel Speedup	–	2.0	3.9	7.6	14.0

Table 5.3: The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the RCB partition and $\varepsilon = 1.0 \times 10^{-2}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time.

Processors	p=1	p=2	p=4	p=8	p=16
Parallel Time	770.65	595.91	412.71	316.99	195.35
Speedup	–	1.3	1.9	2.4	3.9
Sequential Time	–	1178.91	1615.54	2231.65	2210.97
Parallel Speedup	–	2.0	3.9	7.0	11.3

Table 5.4: The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the anisotropic partition and $\varepsilon = 1.0 \times 10^{-2}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time.

anisotropic partitioning strategies. We recall from Chapter 4 that the number of iterations required to converge is always greater for the RCB partition than for the anisotropic partition (for $\varepsilon = 1.0 \times 10^{-2}$ see Tables 4.18 and 4.21 whereas for $\varepsilon = 1.0 \times 10^{-3}$ see Tables 4.19 and 4.22). This suggests that the anisotropic partition may be better for convection–dominated problems provided the subdomains are aligned appropriately. That is, if the dominating convection term is aligned with the x–axis, as considered here, the subdomains should also be aligned along the x–axis as shown in Figure 3.9 (RHS). However, if we look at the results in Tables 5.3 and 5.4 corresponding to Test Problem 6 when using $\varepsilon = 1.0 \times 10^{-2}$ for the RCB and the anisotropic partitions respectively, the actual timings contradict the above

Processors	p=1	p=2	p=4	p=8	p=16
Parallel Time	668.12	431.68	271.66	175.48	98.93
Speedup	–	1.5	2.5	3.8	6.8
Sequential Time	–	854.68	1062.95	1342.11	1398.79
Parallel Speedup	–	2.0	3.9	7.6	14.1

Table 5.5: The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the RCB partition and $\varepsilon = 1.0 \times 10^{-3}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time.

Processors	p=1	p=2	p=4	p=8	p=16
Parallel Time	668.12	481.61	297.77	191.86	134.22
Speedup	–	1.4	2.2	3.5	5.0
Sequential Time	–	952.01	1163.88	1377.96	1600.40
Parallel Speedup	–	2.0	3.9	7.2	11.9

Table 5.6: The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 6 using the anisotropic partition and $\varepsilon = 1.0 \times 10^{-3}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time.

observation in support of the anisotropic partition.

Thus our preconditioning algorithm for the convection–dominated problems may be evaluated in two different ways: if we are concerned about the number of iterations required by the preconditioning algorithm then the anisotropic partition is better whereas, in terms of the practical performance of the algorithm, the RCB partition is superior. This leads us to the important observation that for the RCB partition the number of iterations is larger but the cost per iteration is smaller than for the anisotropic partition. In this case the trade–off between these two factors favours the RCB partition. There are at least two reasons for this higher cost per iteration for the anisotropic partition: the surface–area of the subdomains is larger

(see Figure 3.9) and the imbalance in the workload distribution across the processors is greater (since the size of overlap regions varies more, see §5.1.3). The first of these factors causes both a higher volume of communication amongst the subdomains and larger individual subproblems to be solved on each processor since the overlap region is larger. The second factor causes a significant time to be spent waiting at synchronization points by some of the processors: those with the least computational load. For these reasons we see a drop in both the speedup and the parallel speedup when the anisotropic partition is used in these examples.

Although the specific solution times shown in Tables 5.5 and 5.6 for Test Problem 6 when $\varepsilon = 1.0 \times 10^{-3}$ are lower than the corresponding times when $\varepsilon = 1.0 \times 10^{-2}$, all of the above observations are valid in this case too. The improvement in the individual timings may be because the ILU preconditioner used for the problems on each processor performs better for more convection–dominated problems (as illustrated by the times for $p = 1$). Recall that in Chapter 4 such an improvement, when convection is more dominant, has already been observed.

Next we consider Test Problem 7, the last in this series of performance evaluations of our generalized preconditioning algorithm. This is again a convection–dominated problem but this time, instead of global uniform refinement, local non–uniform refinement is used in one particular part of the domain only. Consequently, both the total communication volume and the computational cost per iteration are decreased. This is evident from the timings presented in Table 5.7.

The solution of this problem is smooth everywhere except in a thin layer of size $O(\varepsilon)$ near the boundary $x = 2.0$, where it varies very rapidly. Thus refinement of the region near $x = 2.0$ is sufficient to accurately resolve the solution everywhere, including the boundary layer. This situation is totally unsuitable for the RCB partition of the coarse grid because of the highly uneven workload distribution that results from local non–uniform refinement. Hence the anisotropic partition is used for these calculations since it provides a relatively well–balanced workload distribution. From these results we see that the figures quoted for the the Parallel Time and the Sequential Time are very small. This is because the use of local mesh refinement

Processors	p=1	p=2	p=4	p=8	p=16
Parallel Time	29.19	14.61	8.73	6.30	4.80
Speedup	–	2.0	3.3	4.6	6.1
Sequential Time	–	28.90	33.63	43.72	59.84
Parallel Speedup	–	2.0	3.9	6.9	12.5

Table 5.7: The performance of the parallel solver for the stabilized FE discretization when solving Test Problem 7 using the anisotropic partition and $\varepsilon = 1.0 \times 10^{-2}$. The times are quoted in seconds and the speedups are relative to the best sequential solution time.

causes the size of the problem being solved to become relatively small.

We conclude this section with a few remarks concerning the issues which affect the performance of our weakly overlapping two level additive Schwarz DD preconditioning algorithm. Amongst these, the most important is the Sequential Time required to solve the problems using the p subdomain algorithm. We observe from the Tables 5.1–5.7 that these can vary significantly but that the Parallel Speedup is generally very good. This indicates that the parallel implementation of our weakly overlapping two level additive Schwarz DD preconditioning algorithm is good. It appears that the reasons for the undesirable variation in the Sequential Time as p varies are complex however. We make the following observations in this regard: the number of subproblems increase as the value of p increases but the size of each subproblem decreases. This is further complicated by the fact that the overall iteration count increases very slowly (at least for the values of p considered here). Hence the cost per iteration may decrease a little as p increases but more iterations may be needed to converge.

After the variation in the Sequential Time the next most important factor to influence the parallel efficiency is the occurrence of an uneven work load across the processors in certain examples. Although there is always some variation in the size of subdomain problems, we look at this issue here in the context of two partitioning strategies, the RCB partition and the anisotropic partition (see §3.8 for details).

In the first situation the resulting subproblems are reasonably well balanced for $p \in \{2, 4, 8\}$ whereas this is true only for $p \in \{2, 4\}$ for the anisotropic partition. Consider a base level coarse mesh of just 768 tetrahedral elements partitioned for 16 subdomains and refining each subdomain up to level four, the resulting meshes for the individual subdomains varies between 276216 and 307406 elements in case of the RCB partition whereas this variation is between 300400 and 429816 elements for the anisotropic partition. If the base level coarse mesh is partitioned for 8 subdomains than the final meshes consist of between 522530 and 526432 elements for the RCB partition and between 544560 and 653712 elements for the anisotropic partition. These imbalances are clearly reflected in the Parallel Speedup rows of all of the Tables in this section. When the load balance is good, the Parallel Speedups are quite close to being optimal, thus illustrating that the other parallel overheads and communication costs are quite small.

5.3 Discussion

Parallel performance of the generalized nonsymmetric preconditioner is evaluated for a variety of test problems using the RCB and the anisotropic partitioning strategies as appropriate. Experiments are performed for global uniform refinement and local non–uniform refinement. Some of the issues which have an impact on the performance are discussed in §5.1. These include decomposition of the domain into subdomains, where aspect ratios of the resulting subdomains influence the performance, discussed in §5.1.1, and inter–processor communication, discussed in §5.1.2. Here the size of interface boundary of the subdomains and the cost of all–to–all communications are amongst the primary concerns. In §5.1.3, the issue of load balance is also considered (this is a source of idleness for some processors when others are still doing useful computations) and other causes of parallel overhead in the context of our implementation are discussed in §5.1.4.

Sample performance results are presented in §5.2 for a selection of the test problems considered in this thesis. These results are encouraging but also highlight

a number of weaknesses in our parallel implementation that should be addressed. These include the need for a more robust partitioning strategy that is able to provide a good load balance when taking into account the elements in the overlap region. This strategy should also be able to balance the, possibly conflicting, requirements of yielding subdomains with low surface–area to volume ratios but that are also of an appropriate shape and alignment for anisotropic problems. Further issues raised by the results in §5.2 include the desirability of allowing more than one subdomain per processor and also the possible drawback of including the global coarse grid solve on each processor. Whilst this simplified the coding somewhat, it introduces the need for an all–to–all communication at each iteration rather than an all–to–one communication followed by a short broadcast.

The parallel preconditioning algorithms proposed in this thesis are designed to make maximum use of standard sequential algorithms: mesh adaption algorithms based upon local h–refinement [164] and iterative solution algorithms [151] for the solution of local subproblems. The use of these algorithms is intended with the requirement for little or no modifications. The behaviour of these sequential component algorithms clearly affects the overall parallel performance. Recall from §1.3.3 that green refinement of an element by introducing a new node at the centre of the element (when all six edges of the element are not marked for refinement) has a serious knock–on effect on the performance of the refinement algorithm: we also observe in this chapter its negative effect on the preconditioning algorithm. This is in addition to the extra coding effort that is needed to treat these green nodes (see §3.5.2.2). This adverse effect of green refinement could have been avoided if it were not for the introduction of a new node at the centre of the tetrahedral elements. Any continuation of this work should therefore address this issue or just abandon green refinement altogether (and treat hanging nodes as having their values prescribed by the values at the nodes at the ends of the edge). Similarly, the best sequential solver available to us for this work, used for the solution of subdomain problems on each processor, is an ILU preconditioned GMRES implementation [151]. It is certainly possible that the parallel performance could be significantly improved if

the sequential algorithm for solving the subdomain problems when $p \geq 1$ were to be improved (e.g. using multigrid [34]).

Chapter 6

Conclusion and Future Directions

This chapter concludes the research presented in this thesis. A brief summary of the work undertaken is given in §6.1 whereas in §6.2 suggestions to improve the parallel efficiency and a number of possible extensions of the work are outlined.

6.1 Summary

The basic components involved in this research work are introduced in Chapter 1 and then in Chapter 2 some of the well known domain decomposition solution techniques and related issues are discussed. The contribution of this thesis starts with a generalization, from two dimensions to three dimensions, of the symmetric weakly overlapping two level additive Schwarz preconditioner [11], undertaken in Chapter 3. Despite certain inconsistencies in the available tools for this generalized implementation in three dimensions with those used for two dimensions in [11], we have successfully demonstrated that the optimal convergence theory works in practice. The next step undertaken was the extension of this symmetric weakly overlapping preconditioner to a more general nonsymmetric case, undertaken in Chapter 4. This is achieved by replacing the full interpolation operation after each solve of the subdomain problems by a simple scaling of the subdomain interface boundary terms. At this time there is no theoretical evidence available to us for this generalization to the nonsymmetric preconditioning algorithm, which is simply based on empirical

observations. From the results so obtained it is observed that this generalized algorithm outperforms the original symmetric algorithm in every aspect. Hence, when applied to an SPD problem both the cost per iteration and the number of iterations are reduced.

In addition to the above comparison, the generalized algorithm is also applied to nonsymmetric and convection–dominated problems, with excellent results. Due to non–physical oscillations in the solution of convection–dominated problems when discretized by the Galerkin method, a more stable discretization technique, the streamline–diffusion method, is also implemented. Again, numerical evidence confirms the success of our generalized nonsymmetric weakly overlapping algorithm for this discretization technique and this class of problem. Furthermore, both of these three–dimensional algorithms perform well for local non–uniform as well as global uniform refinement of the subdomain meshes. All numerical results presented in this thesis, for a wide variety of test problems, are obtained by the parallel implementation of both algorithms. Parallel performance is discussed in Chapter 5 in the case of the generalized nonsymmetric algorithm only due to the demonstrated superiority of this algorithm over the original symmetric version. Some issues concerning the parallel performance are described and results are presented to show the speedups for a selection of typical test problems. An analysis of possible bottlenecks and parallel overheads is also provided.

6.2 Future Directions

There are numerous possible ways in which the research described in this thesis could be improved and extended. A few of these, those of most interest to the author, are suggested here.

One important possibility for improving the parallel efficiency of the described two level algorithms could be some modification of the coarse grid solve. In our existing implementation, each processor i maintains a coarse mesh which covers whole of the domain Ω but refines this only in the subdomain Ω_i and its neighbourhood.

In effect this means that each processor undertakes its own coarse grid solve at each iteration. A more conventional approach would be to only store on processor i the mesh on Ω_i and its neighbourhood, whilst maintaining a single copy of the coarse grid on just one processor. Instead of requiring an all–to–all communication at each iteration in order to restrict the residual to the coarse grid on each processor, this would require only an all–to–one communication at each iteration to restrict the residual to the single coarse grid. Neighbour–to–neighbour communications would still be required to deal with the overlap regions but the total amount of communication required at each iteration would decrease, together with a corresponding decrease in the computations and communication required in the setup phase. A similar improvement in the interpolation procedure would also be obtained for the symmetric preconditioning algorithm. The only disadvantage of this approach would be that a separate coarse grid solve will be required on just one processor and that it will be much harder to use this solver within the framework of [8], which was one of the original motivations for this work.

Since load imbalance is one of the major factors which effects the performance of a parallel program, we suggest the following two areas for further investigation.

1. Using fewer processors than subdomains, so that some or all of the processors solve more than one subproblem. It is possible that one processor may solve the largest subproblem and the smallest subproblem and other processors solve two subproblems of medium size for example.

2. Obtaining a more general partition of the mesh, so that each subdomain is of almost the same size and shape, with close to the minimum edge–cut. This might be achieved by using one of the software packages described in §2.6.5.

In other possible future work, our weakly overlapping approach could also be extended for the solution of nonlinear elliptic partial differential equations. When such an equation is discretized by the finite element method it results in a nonlinear algebraic system of the form

$$F(x) = 0, \qquad \text{where} \qquad F : \mathcal{R}^n \to \mathcal{R}^n, \tag{6.1}$$

and F is assumed to be continuously differentiable. Newton's method for solving (6.1) requires, at the kth step, the solution of the linear Newton equation

$$\frac{dF}{dx}(x_k)s_k = -F(x_k), \tag{6.2}$$

where x_k is the current approximate solution and $\frac{dF}{dx}$ is the Jacobian of F. A Newton iterative method, or quasi–Newton method, requires an iterative solver to determine an approximate solution of (6.2) at each step. This approximate solution, at each Newton iteration, may be determined by using a DD preconditioner based upon our weakly overlapping solution algorithm. In this regard we would propose making use of a nonlinear Krylov solver such as NITSOL [138], which is sufficiently flexible in its data structure to allow users to implement their preferred preconditioner for the solution of the linear Newton equations without any serious difficulty. The user is also allowed to specify an inner product and associated norm which allows easy adaptation to a parallel environment. Our preconditioned solution method should fit precisely within this framework.

A further extension could also be the generalization of the presented algorithms to the solution of systems of partial differential equations. In principal, we do not see any fundamental difficulty in undertaking this generalization: it should be just a matter of time and coding effort!

Bibliography

[1] Ainsworth, M. and Oden, J. T. A unified approach to a posteriori error estimation using element residual methods. *Numerische Mathematik*, 65:23–50, 1993.

[2] Alaghband, G. Parallel sparse matrix solution and performance. *Parallel Computing*, 21(9):1407–1430, 1990.

[3] Ashby, S. F. Minimax polynomial preconditioning for hermitian linear systems. *SIAM J. Matrix Analysis and Applications*, 12:766–789, 1991.

[4] Ashby, S. F., Manteuffel, T. A., and Taylor, P. E. A taxonomy for conjugate gradient methods. *SIAM J. Numer. Anal.*, 27:1542–1568, 1989.

[5] Axelsson, O. *Iterative Solution Methods.* Cambridge University Press, Cambridge, 1994.

[6] Babuška, I., Craig, A., Mandel, J., and Pitkäranta, J. Efficient preconditioning for the p-version finite element method in two dimensions. *SIAM J. Numer. Anal.*, 28(3):624–661, 1991.

[7] Babuška, I. and Suri, M. The p– and h–p versions of finite element methods, basic principles and properties. *SIAM Review*, 36(4):578–632, 1994.

[8] Bank, R. E. and Holst, M. J. A new paradigm for parallel adaptive meshing algorithms. *SIAM J. on Scientific Computing*, 22(4):1411–1443, 2000.

[9] Bank, R. E. and Jimack, P. K. A new parallel domain decomposition method for the adaptive finite element solution of elliptic partial differential equations. *Concurrency and Computation: Practice and Experience*, 13:327–350, 2001.

[10] Bank, R. E., Jimack, P. K., Nadeem, S. A., and Nepomnyaschikh, S. V. A weakly overlapping domain decomposition preconditioner for the finite element solution of elliptic partial differential equations. *SIAM Journal of Scientific Computing*, 2001. To appear.

[11] Bank, R. E., Jimack, P. K., and Nepomnyaschikh, S. V. A weakly overlapping domain decomposition for the adaptive finite element solution of elliptic partial differential equations. Research Report 1999.17, School of Computer Studies, The University of Leeds, Leeds, LS2 9JT, September 1999.

[12] Barnard, S. T. and Clay, R. L. A portable MPI implementation of the SPAI preconditioner in ISIS++. In Heath, M. *et al.*, editor, *Preceedings of the Eight SIAM Conference on Parallel Processing for Scientific Computing*. SIAM Press, 1997.

[13] Barret, R., Berry, M., Chan, T. F., 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, Philadelphia, PA, 1994.

[14] Ben-israel, A. and Greville, T. *Generalized Inverses: Theory and Applications*. John Wiley, 1973.

[15] Benzi, M. and Tuma, M. Numerical experiments with two sparse approximate inverse preconditioners. *BIT*, 38:234–241, 1998.

[16] Benzi, M. and Tuma, M. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Scientific Computing*, 19(3):968–994, 1998.

[17] Berzins, M. A solution–based triangular and tetrahedral mesh quality indicator. *SIAM. J. Sci. Comput.*, 19:2051–2060, 1998.

[18] Bhardwaj, M., Day, D., Farhat, C., Lesoinne, M., Pierson, K., and Rixen, D. Application of the FETI method to ASCI problems – scalability results on 1000 processors and discussion of highly heterogeneous problems. *Int. J. Numer. Meth. Engrg.*, 47:513–535, 2000.

[19] Biswas, R. and Strawn, R. C. A new procedure for dynamic adaption of three dimensional unstructured grids. *Appl. Numer. Math.*, 13:437–452, 1994.

[20] Bjorstad, P. E. A large scale, sparse, secondary storage, direct linear equation solver for structural analysis and its implementation on vector and parallel architectures. *J. Parallel Comput.*, 5, 1987.

[21] Bjorstad, P. E. and Widlund, O. B. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM J. Numer. Anal.*, 26(6):1093–1120, 1986.

[22] Bjorstad, P. E. and Widlund, O. B. To overlap or not to overlap: A note on a domain decomposition method for elliptic problems. *SIAM J. Sci. Stat. Comput.*, 10(5), September 1989.

[23] Blazy, S., Brochers, W., and Dralle, U. Parallelization methods for a charecteristic's pressure correction scheme. In Hirschel, E. H., editor, *Flow Simulation With High Performance Computers II*, Notes on Numerical Fluid Mechanics, 1995.

[24] Bornemann, F. A., Erdmann, B., and Kornhuber, R. A posteriori error estimates for elliptic problems in two and three space dimensions. *SIAM J. Numer. Anal.*, 33:1188–1204, 1996.

[25] Bornemann, F. A. and Yserentant, H. A basic norm equivalence for the theory of multilevel methods. *Numerische Mathematik*, 64:455–476, 1993.

[26] Bramble, J. H., Pasciak, J. E., and Schatz, A. H. The construction of preconditioners for elliptic problems by substructuring, I. *Mathematics of Computation*, 47:103–134, 1986.

[27] Bramble, J. H., Pasciak, J. E., and Schatz, A. H. An iterative method for elliptic problems on regions partitioned into substructures. *Math. Comp.*, 46:361–369, 1986.

[28] Bramble, J. H., Pasciak, J. E., and Schatz, A. H. The construction of preconditioners for elliptic problems by substructuring, II. *Mathematics of Computation*, 49:1–16, 1987.

[29] Bramble, J. H., Pasciak, J. E., and Schatz, A. H. The construction of preconditioners for elliptic problems by substructuring, III. *Mathematics of Computation*, 51:415–430, 1988.

[30] Bramble, J. H., Pasciak, J. E., and Schatz, A. H. The construction of preconditioners for elliptic problems by substructuring, IV. *Mathematics of Computation*, 53:1–24, 1989.

[31] Bramble, J. H., Pasciak, J. E., Wang, J., and Xu, J. Convergence estimates for product iterative methods with applications to domain decomposition. *Mathematics of Computation*, 57(195):1–21, 1991.

[32] Bramble, J. H., Pasciak, J. E., and Xu, J. Parallel multilevel preconditioners. *Mathematics of Computation*, 55:1–21, 1990.

[33] Brezzi, F., Franca, L., Hughes, T. J. R., and Russo, A. Stabilisation techniques and subgrid scales capturing. In Duff, I. S. and Watson, G. A., editors, *State of the Art in Numerical Analysis*, pages 391–406. Oxford University Press, 1997.

[34] Briggs, W. L., Henson, V, E., and McCormick, S. F. *A Multigrid Tutorial.* SIAM Books, Philadelphia, 2000. Second Edition.

[35] Cai, X.–C. An optimal two–level overlapping domain decomposition method for elliptic problems in two and three dimensions. *SIAM J. Sci. Comp.*, 14:239–247, 1993.

[36] Cai, X.–C. A family of overlapping Schwarz algorithms for nonsymmetric and indefinite elliptic problems. In Keyes, D., Saad, Y., and Truhlar, D., editors, *Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*. SIAM, 1994.

[37] Cai, X.–C., Dryja, M., and Sarkis, M. RASHO: A restricted additive schwarz preconditioner with harmonic overlap. In *Proceedings of the 13th International Conference on Domain Decomposition Methods*, France, 9–12 October 2000.

[38] Cai, X.–C. and Sarkis, M. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. on Sci. Comp.*, 21:792–797, 1999.

[39] Cai, X.–C. and Widlund, O. B. Multiplicative Schwarz algorithms for some nonsymmetric and indifinite problems. *SIAM J. Numer. Anal.*, 30(4):936–952, 1993.

[40] Cao, W., Huang, W., and Russell, R. D. An r–adaptive finite element method based upon moving mesh PDEs. *J. Comp. Phys.*, 149:221–244, 1999.

[41] Chan, T. F. and Goovaerts, D. On the relationship between overlapping and nonoverlapping domain decomposition methods. *SIAM Journal on Matrix Analysis and Applications*, 13:663–670, 1992.

[42] Chan, T. F. and Mathew, T. Domain decomposition algorithms. *Acta Numerica*, pages 61–143, 1994.

[43] Chan, T. F. and Resasco, D. C. Analysis of domain decomposition preconditioners on irregular regions. In Vichnevetsky, R. and Stepleman, R., editors, *Advances in Computer Methods for Partial differentials Equations*, IMACS, pages 317–322, 1997.

[44] Chan, T. F. and Zou, J. Additive Schwarz domain decomposition methods for elliptic problems on unstructured meshes. *Numerical Algorithms*, 8:329–346, 1994.

[45] Chow, E. and Saad, Y. Approximate inverse preconditioners via sparse–sparse iterations. *SIAM J. Scientific Computing*, 19:995–1023, 1998.

[46] Ciarlet, P. G. *The Finite Element Method for Elliptic Problems.* North-Holland Publishing Company, 1978.

[47] Cosgrove, J. D. F., Diaz, J. C., and Griewank, A. Approximate inverse preconditionings for sparse linear systems. *Int. J. Computer Math.*, 44:91–110, 1992.

[48] Cottle, R. W. Manifestations of the schur complement. *Linear Algebra and Its Applications*, 8:189–211, 1974.

[49] Davis, T. A. and Yew, P. C. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Analysis and Applications*, 11:383–402, 1990.

[50] Demmel, J. Applications of parallel computers. Available over the World Wide Web at http://www.cs.bekeley.edu/~demmel/cs267_Spr99, Spring 1999.

[51] Diekmann, R., Schlimbach, F., and Walshaw, C. Quality balancing for paralle adaptive FEM. In Ferreira, A. *et al.*, editor, *Irregular'98: Solving Irregularly Structured Problems in Parallel*, volume 1457 of LNCS, pages 170–181. Springer, 1998.

[52] Diniz, P., Plimpton, S., Hendrickson, B., and Leland, R. Parallel algorithms for dynamically partitioning unstructured grids. In *Seventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM Philadelphia, 1995.

[53] Dongarra, J. J., Duff, I. S., Sorensen, D. C., and Van der Vorst, H. A. *Solving Linear Systems on Vector and Distributed Memory Computers.* SIAM publications, Philadelphia, PA, 1991.

[54] Dryja, M. An additive Schwarz algorithm for two and three dimensional finite element elliptic problems. In Chan, T. F. *et al.*, editor, *2nd International Symposium on Domain Decomposition Methods*, SIAM, Philadelphia, 1989.

[55] Dryja, M., Smith, B. F., and Widlund, O. B. Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions. *SIAM J. Numer. Anal.*, 31(6):1662–1694, 1994.

[56] Dryja, M. and Widlund, O. B. An additive variant of the Schwarz alternating method for the case of many subregions. technical report 339, also ultracomputer note 131, Department of Computer Science, Courant Institute, 1987.

[57] Dryja, M. and Widlund, O. B. Some domain decomposition algorithms for elliptic problems. In Hayes, H. and Kincaid, D., editors, *Iterative Methods for Large Linear Systems*, pages 273–291, San Diego, Calefornia, 1989. Academic Press.

[58] Dryja, M. and Widlund, O. B. Multilevel additive methods for elliptic finite element problems. In Hackbusch, W., editor, *Parallel Algorithms for Partial Differential Equations, Proceedings of the Sixth GAMM–Seminar*, Vieweg, Braunschweig, Germany, 1990.

[59] Dryja, M. and Widlund, O. B. Towards a unified theory of domain decomposition algorithms for elliptic problems. In Chan, T. F., Glowinski, R., Périaux, J., and Widlund, O. B., editors, *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*, SIAM Philadelphia, PA, 1990.

[60] Duff, I. S. Direct methods for solving sparse systems of linear equations. *SIAM J. Sci. Statist. Comput.*, 5:605–619, 1984.

[61] Duff, I. S. MA32 – A package for solving sparse unsymmetric systems using frontal methods. Report 10079, HMSO, AERE, Harwell, 1989.

[62] Duff, I. S., Erisman, A. M., and Reid, J. K. *Direct Methods for Sparse Matrices.* Oxford University Press, London, 1986.

[63] Duff, I. S. and Reid, J. K. The multifrontal solution of unsymmetric sets of linear systems. *SIAM J. Sci, Statist. Comput.*, 1984.

[64] Eijkhout, V. and Chan, T. F. Parpre a parallel preconditioners package, reference manual for version 2.0.17. Tech. rep, CAM report 97–24, UCLA, 1997.

[65] Elman, H. C. and Shih, Y. T. Modified streamline diffusion schemes for convection–diffusion problems. *Comput. Methods Appl. Mech. Engrg.*, 174:137–151, 1999.

[66] Elman, H. C. and Streit, R. L. Polynomial iteration for nonsymmetric indefinite linear systems. In Hennart, J. P., editor, *Lecture Notes in Mathematics*, volume 1230. Springer–Verlag, Berlin, 1984.

[67] Faber, V. and Manteuffel, T. A. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal.*, 21:315–339, 1984.

[68] Farhat, C. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[69] Farhat, C. Lagrange multiplier based divide and conquer finite element algorithm. *J. Comput. Sys. Engrg.*, 2:149–156, 1991.

[70] Farhat, C., Chen, P. S., Risler, F., and Roux, F–X. A unified framework for accelerating the convergence of iterative substructuring methods with lagrange multipliers. *Int. J. Numer. Meth. Engrg.*, 42:257–288, 1998.

[71] Farhat, C. and Lesoinne, M. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Internet J. Numer. Meth. Engrg.*, 36(5):745–764, 1993.

[72] Farhat, C., Lesoinne, M., LeTallec, P., Pierson, K., and Rixen, D. FETI-DP: A dual-primal unified FETI method – part I: A faster alternative to the two–level FETI method. *International Journal for Numerical Methods in Engineering*, 50(7):1523–1544, 2000.

[73] Farhat, C., Lesoinne, M., and Pierson, K. A scalable dual–primal domain decomposition method. *Numerical Linear Algebra with Applications*, 7(7–8):687–714, 2000.

[74] Farhat, C., Maman, N., and Brown, G. Mesh partitioning for implicit computations via domain decomposition. *Int. J. Num. Meth. Engng.*, 38:989–1000, 1995.

[75] Farhat, C. and Mandel, J. Scalable substructuring by lagrange multipliers in theory and prectice. In Bjorstead, P., Espedal, M., and Keyes, D., editors, *DD9 Proceedings*. John Wiley & Sons Ltd., 1994.

[76] Farhat, C., Mandel, J., and Roux, F–X. Optimal convergence properties of the FETI domain decomposition method. *Comput. Methods Appl. Mech. Engrg.*, 115:367–388, 1994.

[77] Farhat, C. and Roux, F–X. A method of finite element tearing and interconnecting and its parallel solution algorithm. *Int. J. Numer. Meth. Engrg.*, 32:1205–1227, 1991.

[78] Farhat, C. and Roux, F–X. An unconventional domain decomposition method for an efficient parallel solution of large scale finite element system. *SIAM J. Sci. Stat. Comput.*, 13:379–396, 1992.

[79] Farhat, C. and Roux, F–X. Implicit parallel processing in structural mechanics. *Comput. Mech. Advances*, 2(1):1–124, 1994.

[80] Fiduccia, C. M. and Mattheyses, R. M. A linear time heuristic for improving network partitions. In *Proceedings of the Nineteenth IEEE Design Automation Conference*, pages 175–181. IEEE, 1982.

[81] Fischer, B., Ramage, A., Silvester, D. J., and Wathen, A. J. On parameter choice and iterative convergence for stabilized discretization of advection–diffusion problems. *Comput. Methods Appl. Mesh. Engrg.*, 179:179–195, 1999.

[82] Flynn, M. Very high–speed computing systems. In *Proceedings of the IEEE*, pages 1901–1909, December 1966.

[83] Franca, L. P., Frey, S. L., and Hughes, T. J. R. Stabilized finite element methods: I. Application to the advective–diffusive model. *Compt. Methods Appl. Mech. Engrg.*, 95:253–276, 1992.

[84] Freitag, L. A. and Ollivier–gooch, C. Tetrahedral mesh improvement using swapping and smoothing. *Int. J. Numer. Meth. Engrg.*, 40:3979–4002, 1997.

[85] George, A. and Liu, J. W. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, 1981.

[86] Golub, G. H. and Van Loan, C. F. *Matrix Computations.* John Hopkins University Press, Baltimore, 1989.

[87] Greibel, M. and Oswald, P. On the abstract theory of additive and multiplicative Schwarz algorithms. *Numer. Math.*, 70:163–180, 1995.

[88] Gresho, P. M. and Lee, R. L. Don't suppress the wiggles – they're telling you something. In Hughes, T. J. R., editor, *Finite Element methods for Convection Dominated Flows*, volume 34 of AMD, pages 37–61, ASME, New York, 1979.

[89] Gresho, P. M. and Sani, L. R. *Incompressible Flow and the Finite Element Method.* John Wiley and Sons, 1999.

[90] Grote, M. J. and Huckle, T. Parallel preconditionings with sparse approximate inverses. *SIAM J. Scientific Computing*, 18:838–853, 1997.

[91] Gustafsson, I. and Lindskog, G. A preconditioning technique based on element matrix factorization. *Comput. Methods Appl. Mech. Eng.*, 55:201–220, 1986.

[92] Hendrickson, B. and Leland, R. Multidimensional spectral load balancing. Technical Report SAND93–2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[93] Hendrickson, B. and Leland, R. The Chaco user's guide: Version 2.0. Technical report SAND95–2344, Sandia National Laboratories, Albuquerque, NM, July 1995.

[94] Hendrickson, B. and Leland, R. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16:452–469, 1995.

[95] Hodgson, D. C. *Efficient Mesh Partitioning and Domain Decomposition Methods on Parallel Distributed Memory Machines.* PhD thesis, School of Computer Studies, The University of Leeds, Leeds, 1995.

[96] Hodgson, D. C. and Jimack, P. K. Efficient mesh partitioning for parallel P.D.E. solvers on distributed memory machines. In *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, VA, 1993.

[97] Hodgson, D. C. and Jimack, P. K. A domain decomposition preconditioner for a parallel finite solver on distributed unstructured grids. *Parallel Computing*, 1997.

[98] Hood, P. Frontal solution program for unsymmetric matrices. *Int. J. Num. Meth. Eng.*, 10:379–400, 1976.

[99] Hooper, M. J. Harwell subroutine library: A catalogue of subroutines. Report AERE R–9185 9th ed., Harwell, 1989.

[100] Hughes, T. J. R. and Brooks, A. A multidimensional upwind scheme with no crosswind diffusion. In Hughes, T. J. R., editor, *Finite Element methods for Convection Dominated Flows*, volume 34 of AMD, pages 120–131, ASME, New York, 1979.

[101] Hughes, T. J. R., Franca, L. P., and Hulbert, G. M. A new finite element formulation for computational fluid dynamics: VIII. The Galerkin/Least squares methods for advective–diffusive equations. *Compt. Methods Appl. Mech. Engrg.*, 73:173–189, 1989.

[102] Hughes, T. J. R., Levit, I., and Winget, J. An element–by–element solution algorithm for problems of structural and solid mechanics. *J. Comput. Methods in Appl. Mech. Eng.*, 36:241–254, 1983.

[103] Irons, B. M. A frontal solution program for finite element analysis. *Int. J. Num. Meth. Eng.*, 2:5–32, 1970.

[104] Jimack, P. K. An optimal finite element mesh for elastostatic structural analysis problems. *Computers and Structures*, 64:192–208, 1997.

[105] Jimack, P. K. and Nadeem, S. A. A weakly overlapping parallel domain decomposition preconditioner for the finite element solution of elliptic problems in three dimensions. In Arabnia, H. R., editor, *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume III, pages 1517–1523. CSREA Press, USA, 2000.

[106] Jimack, P. K. and Nadeem, S. A. Parallel application of a novel domain decomposition preconditioner for the stable finite element solution of three–dimensional convection–dominated PDEs. In Sakellariou, R. *et al.*, editor, *Euro–Par 2001 Parallel Processing*, Lecture Notes in Computer Science 2150, pages 592–601. Springer, 2001.

[107] Jimack, P. K. and Nadeem, S. A. A weakly overlapping parallel domain decomposition preconditioner for the finite element solution of convection–dominated problems in three dimensions. In *Proceedings of International Parallel CFD 2001 Conference*, Egmond ann Zee, The Netherlands, 21–23 May 2001. To appear.

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

[109] Johnson, C. The streamline diffusion finite element method for compressible and incompressible fluid flow. In Griffiths, D. F. and Watson, G. A., editors, *Numerical Analysis 1989, Pitman Research Notes in Mathematics*, pages 155–181. Longman, London, 1989.

[110] Kahan,W. *Gauss–Seidel Methods for Solving Large Systems of Linear Equations.* PhD thesis, University of Toronto, 1958.

[111] Kallinderis, Y., Parthasarathy, V., and Wu, J. A new euler scheme and adaptive refinement/coarsening algorithm for tetrahedral grids. AIAA Paper 92–0446, 1992.

[112] Karypis, G. and Kumar, V. A software package for partitioning unstructured graphs, partitioning meshes and computing fill–reducing orderings of sparse matrices. Technical report, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN, September 1998.

[113] Kernighan, B. W. and Lin, S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[114] Klawonn, A. and Widlund, O. B. A domain decomposition method with lagrange multipliers and inexact solvers for linear elasticity. *SIAM J. Sci. Comput.*, 22(4):1199–1219, 2000.

[115] Klawonn, A. and Widlund, O. B. FETI and Neumann–Neumann iterative substructuring methods: Connections and new results. *Comm. Pure Appl. Math.*, 54:57–90, 2001.

[116] Kumar, V. and et al. *Introduction to Parallel Computing:Design and Analysis of Algorithms.* Benjamin/Cummings, Redwood City, CA, 1994.

[117] Kuznetsov, S., Lo, G. C., and Saad, Y. Parallel solution of large sparse linear systems. Tech. Rep. UMSI 97/98, Minnesota Supercomputer Institute, University of Minnesota,Minneapolis, MN, 1997.

[118] Larson, M. G. A posteriori and a priori error analysis for finite element approximations of self–adjoint elliptic eigenvalue problems. *SIAM J. Numer. Anal.*, 38(2):608–625, 2000.

[119] LeTallec, P. Domain decomposition methods in computational mechanics. *Compt. Mech. Adv.*, 2:121–220, 1994.

[120] Lohner, R. and Baum, J. D. Adaptive h–refinement on 3D unstructured grids for transient problems. *Int. J. Num. Methods Fluids*, 14:1407–1419, 1992.

[121] Mandel, J. Balancing domain decomposition. *Comm. Numer. Meth. Eng.*, 9:233–241, 1993.

[122] Mandel, J. and Tezaur, R. Convergence of a substructuring method with lagrange multipliers. *Numer. Math*, 73:473–487, 1996.

[123] Mandel, J. and Tezaur, R. On the convergence of a dual–primal substructuring method. *Numer. Math.*, 88(3):534–558, 2001.

[124] Matsokin, A. M. and Nepomnyaschikh, S. V. A Schwarz alternating method in a subspace. *Soviet Mathematics*, 29(10):78–84, 1985.

[125] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[126] Message Passing Interface Forum. MPI-2: Extensions to the message passing interface. *http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html*, 1997.

[127] Miller, K. Moving finite elements. II. *SIAM Journal of Numerical Analysis*, 18(6):1033–1057, 1981.

[128] Miller, K. and Miller, R. N. Moving finite elements. I. *SIAM Journal of Numerical Analysis*, 18(6):1019–1032, 1981.

[129] Morton, K. W. *Numerical Solution of Convection Diffusion Problems.* Chapman and Hall, London, 1996.

[130] Nadeem, S. A. and Jimack, P. K. Parallel implementation of an optimal two level additive Schwarz preconditioner for the 3-d finite element solution of elliptic partial differential equations. *Int. J. Num. Meth. Fluids*, 2001. Submitted.

[131] Ong, M. E. G. Uniform refinement of tetrahedron. *SIAM J. Sci. Comput.*, 15(5), 1994.

[132] Oswald, P. On discrete norm estimates related to multilevel preconditioners in the finite element method. In Ivanov, K. and Sendov, B., editors, *Proceedings of the International Conference on Constructive Theory of Functions*, pages 203–241, 1992.

[133] Pacheco, P. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[134] Paige, C., Parlett, B., and Van der Vorst, H. Approximate solutions and eigenvalue bounds from krylov subspaces. *Numer. Lin. Alg. Appls.*, 29:115–134, 1995.

[135] Paige, C. and Saunders, M. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[136] Pepper, D. W. and Heinrich, J. C. *The Finite Element Method: Basic Concepts and Applications.* Hemisphere Publishing Corporation, 1992.

[137] Perella, A. J. *A Class of Petrov–Galerkin Finite Element Methods for the Numerical Solution of the Stationary Convection–Diffusion Equation.* PhD thesis, Department od Mathematical Sciences, University of Durham, Durham, DH1 3LE, England, September 1996.

[138] Pernice, M. and Walker, H. F. NITSOL: A newton iterative solver for nonlinear systems. *SIAM J. Sci. Comp.*, 19(1):302–318, 1998.

[139] Preis, R. and Diekmann, R. The PARTY partitioning library: User guide version 1.1. Technical report, Heinz Nixdorf Institut, Univertität Paderborn, 1996.

[140] Przemieniecki, J. S. *Theory of Matrix Structural Analysis.* (Reprint of McGraw–Hill, 1968), 1985.

[141] Quarteoni, A. and Valli, A. *Domain Decomposition Methods for Partial Differential Equations.* Oxford University Press, UK, 1999.

[142] Ramage, A. A multigrid preconditioner for stabilized discretization of advection–diffusion problems. Mathematics research Report 33/98, University of Strathclyde, 1998.

[143] Ramage, A. A note on parameter choice and iterative convergence for stabilized discretization of advection–diffusion problems in three dimensions. Mathematics research Report 32/98, University of Strathclyde, 1998.

[144] Reddy, J. N. *An Introduction to the Finite Element Method.* McGraw–Hill, 1984.

[145] Ripa, S. and Schiff, B. Minimum energy triangulations for elliptic problems. *Computer Methods in Applied Mechanics and Engineering*, 84:257–274, 1990.

[146] Rivara, M.–C. A grid generator based on 4–triangles conforming mesh refinement algorithm. *Int. J. Numer. Meth. Eng.*, 24:1343–1354, 1987.

[147] Rixen, D. and Farhat, C. A simple and efficient extension of a class of substructure based preconditioners to heterogeneous structural mechanics problems. *Int. J. Numer. Meth. Engng.*, 44:489–516, 1999.

[148] Rixen, D., Farhat, C., Tezaur, R., and Mandel, J. Theoretical comparison of the FETI and algebraically partitioned FETI methods, and performance

comparisons with a dirichlet sparse solver. *Int. J. Numer. Meth. Engng.*, 46:501–534, 1999.

[149] Roos, H.–G., Stynes, M., and Tobiska, L. *Numerical Methods for Singularly Perturbed Differential Equations.* Springer–Verlag, Berlin, 1996.

[150] Saad, Y. A flexible inner–outer preconditioned GMRES algorithm. *SIAM Journal on Scientific and Statistical Computing*, 14:461–469, 1993.

[151] Saad, Y. SPARSKIT: A basic tool kit for sparse matrix computation, version 2. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign, Urbana, IL, 1994.

[152] Saad, Y. *Iterative Methods for Sparse Linear Systems.* PWS Publishing, New York, 1996.

[153] Saad, Y. and Malevsky, A. PSPARSLIB: A portable library of distributed memory sparse iterative solvers. In Malyshev, V. E. *et al.*, editor, *Proceedings of Parallel Computing Technologies (PaCT–95)*, St Petersburg, Russia, 1995. 3rd International Conference.

[154] Saad, Y. and Schultz, M. H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[155] Schlimbach, F. *Optimising Subdomain Aspect Ratios for Parallel Load Balancing.* PhD thesis, School of Computing and Mathematical Sciences, University of Greenwich, 2000.

[156] Schwarz, H. A. Gesammelte mathematische abhandlungen. *Vierteltahrsschrift der Naturgorschenden Gesellschaft in Zürich*, 15:272–286, 1870.

[157] Semper, B. Numerical crosswind smear in the streamline diffusion method. *Compt. Methods Appl. Mech. Engrg.*, 113:99–108, 1994.

[158] Simon, H. D. Partition of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[159] Smith, B. F. An optimal domain decomposition preconditioner for the finite element solution of linear elasticity problems. *SIAM J. Sci. Comput.*, 13(1):364–378, January 1992.

[160] Smith, B. F., Bjorstad, P. E., and Gropp, W. D. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1996.

[161] Smith, B. F., Gropp, W. D., and McInnes, L. C. PETSc 2.0 user's manual. Tech. rep., Argonne National Laboratory, 1995.

[162] Sonneveld, P. CGS, A fast lanczos–type solver for nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 10, 1989.

[163] Speares, W. E. and Berzins, M. A fast 3-D unstructured mesh adaptation algorithm with time-dependent upwind Euler shock diffraction calculations. In Hafez, M. and Oshima, K., editors, *Proc. of 6th Int. Symp. on Computational Fluid Dynamics*, volume III, pages 1181–1188, 1995.

[164] Speares, W. E. and Berzins, M. A 3-D unstructured mesh adaptation algorithm for time dependent shock dominated problems. *International Journal for Numerical Methods in Fluids*, 25:81–104, 1997.

[165] Strang, G. W. and Fix, G. J. *An Analysis of the Finite Element Method.* Prentice-Hall, 1973.

[166] Tong, C. H., Chan, T. F., and Jay Kuo, C. C. A domain decomposition preconditioner based on a change to a multilevel nodal basis. *SIAM J. Sci. Stat. Comput.*, 12(6):1486–1495, 1991.

[167] Van der Vorst, H. Bi–CGSTAB: A fast and smoothly converging variant of bi–cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 13(631–644), 1992.

[168] Vanderstraeten, D., Farhat, C., Chen, P. S., Keunings, R., and Zone, O. A retrofit based methodology for the fast generation and optimization of large–scale mesh partition: Beyond the minimum interface size criterion. *Comp. Meth. Appl. Mech. Engrg.*, 133:25–45, 1996.

[169] Vanderstraeten, D., Keunings, R., and Farhat, C. Beyond conventional mesh partitioning algorithms and the minimum edge cut criterion: Impact on realistic applications. In Bailey, D. *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 611–614. SIAM, 1995.

[170] Walshaw, C. and Berzins, M. Dynamic load balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995.

[171] Walshaw, C., Cross, M., Diekmann, R., and Schlimbach, F. Multilevel mesh partitioning for optimising subdomain aspect ratio. In *VecPar'98*, pages 285–300, 1998.

[172] Walshaw, C., Cross, M., and Everett, M. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Par. Dist. Comput.*, 47(2):102–108, 1997.

[173] Walshaw, C., Cross, M., and McManus, K. Multiphase mesh partitioning. *Appl. Math. Modelling*, 25(2):123–140, 2000.

[174] Widlund, O. B. Some Schwarz methods for symmetric and nonsymmetric elliptic problems. In Keyes, D. E. *et al.*, editor, *5th International Symposium on Domain Decomposition Methods*, SIAM, Philadelphia, 1992.

[175] Williams, R. D. Performance of dynamic load balancing for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3:457–481, 1991.

[176] Wohlmuth, B. I. *Discretization Methods and Iterative Solvers Based on Domain Decomposition.* Number 17 in Lecture Notes in Computational Science and Engineering. Springer, 2001.

[177] Xu, J. Iterative methods by space decomposition and subspace correction. *SIAM Review*, 34:581–613, 1992.

[178] Xu, J. and Zou, J. Some nonoverlapping domain decomposition methods. *SIAM Review*, 40(4):857–914, December 1998.

[179] Young, D. *Iterative Solution of Large Linear Systems.* Academic Press, New Yark, 1971.

[180] Zhang, X. *Studies in Domain Decomposition: Multilevel Methods and the Biharmonic Dirichlet Problem.* PhD thesis, Courant Institute, New York University, USA, September 1991.

[181] Zienkiewicz, O. C. *The Finite Element Methods in Engineering Science.* McGraw–Hill, London, 1971.

[182] Zienkiewicz, O. C. and Taylor, R. L. *The Finite Element Method*, volume I. McGraw–Hill Book Company, Europe, fourth edition, 1994.