Parallel Dynamic Load-Balancing for Adaptive Distributive Memory PDE Solvers

by

Nasir Touheed

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

The University of Leeds
School of Computer Studies
September 1998

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

Abstract

This thesis is concerned with the issue of dynamic load-balancing in connection with the parallel adaptive solution of partial differential equations (PDEs). We are interested in parallel solutions based upon either finite element or finite volume schemes on unstructured grids and we assume that geometric parallelism is used, whereby the finite element or finite volume grids are partitioned across the available parallel processors. For parallel efficiency it is necessary to maintain a well balanced partition and to attempt to keep communication overheads as low as possible. When adaptivity occurs however a given partition may deteriorate in quality and so it must be modified dynamically. This is the problem that we consider in this work.

Chapters one and two outline the problem in more detail and review existing work in this field. In Chapter one a brief history of parallel computers is presented and different kinds of parallel machines are mentioned. The finite element method is also introduced and its parallel implementation is discussed in some detail: leading to the derivation of a static load-balancing problem. A number of important static load balancing algorithms are then discussed. Chapter two commences with a brief description of some error indicators and common techniques for mesh adaptivity. It is shown how this adaptivity may lead to a load imbalance among the available processors of a parallel machine. We then discuss some ways in which the static load-balancing algorithms of Chapter one can be modified and used in the context of dynamic load-balancing. The pros and cons of these strategies are discussed and then finally some specific dynamic load-balancing algorithms are introduced and discussed.

In Chapter three a new dynamic load-balancing algorithm is proposed based upon a number of generalisations of existing algorithms. The details of the new algorithm are outlined and a number of preliminary numerical experiments are undertaken. In this preliminary (sequential) version the dual graph of an existing partitioned computational mesh is repartioned among the same number of processors so that after the repartitioning step each processor has an approximate equal load and the number of edges of this dual graph which cross from one processor to another are relatively small.

The remainder of the thesis is concerned with the practical parallel implementation of this new algorithm and making comparison with existing techniques. In Chapter four the algorithm is implemented for a 2-d adaptive finite element solver

for steady-state problems, and in Chapter five the generality of the implementation is enhanced and the algorithm is applied in conjunction with a 3-d adaptive finite volume solver for unsteady problems. In this situation frequent repartitioning of the mesh is required. In this chapter performance comparisons are made for the algorithm detailed here against new software that was developed simultaneously with the work of this thesis. These comparisons are very favourable for certain problems which involve very non-uniform refinement.

All software implementations described in this thesis have been coded in ANSI C using MPI version 1.1 (where applicable). The Portability of the load-balancing code has been tested by making use of a variety of platforms, including a Cray T3D, an SGI PowerChallenge, different workstation networks (SGI Indys and SGI O2s), and an SGI Origin 2000. For the purposes of numerical comparisons all timings quoted in this thesis are for the SGI Origin 2000 unless otherwise stated.

Acknowledgements

I would like to thank my supervisor Dr. Peter Jimack for his guidance and encouragement throughout the course of this research. Not only was he very helpful in guiding me throughout my stay at Leeds, but he was also very patient when it came to correcting my poorly drafted chapters as regards to the English language. Thanks also to Dr. Martin Berzins, Dr. David Hodgson and Dr. Paul Selwood for their helpful advice and discussions during this time.

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

Thanks are also due to my colleagues Idrees Ahmad, Syed Shafaat Ali, Muhammad Rafiq Asim, Fazilah Haron, Zahid Hussain, Jaw-Shyong Jan, Sharifullah Khan, Rashid Mahmood, Sarfraz Ahmad Nadeem, Allah Nawaz, Professor Muhammad Abdul-Rauf Quraishi, Shuja Muhammad Quraishi and Alex Tsai for matters not necessarily related to the research.

I would also like to acknowledge the Edinburgh Parallel Computing Centre at the University of Edinburgh for allowing me to use their parallel computing facilities, including the Cray T3D, and to thank Dr. Alan Wood of University of York, for his help and support in the early days of my stay in York.

Members of my extended family in Pakistan and of my immediate family here in the UK have been a great encouragement throughout the period of this research. I especially wish to thank my parents and sisters, brother-in-laws, mother-in-law and father-in-law for their encouragement. My two daughters Maryam and Sidrah (who was a much needed and welcome addition to our family in the middle of this project) have been most patient while I finished this task. This project would not have been completed without the constant love and support of my wife, Shagufta.

Finally, my thanks also go to the University of Karachi, the Government of Pakistan and the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom for supporting me financially throughout my research in the forms of Study-Leave, COTS and ORS Awards respectively.

At the very end I would like to thank The Almighty, for the much needed courage and strength which He granted me at this relatively old age of my life to finalise the project.

Contents

1 Introduction **1**

1.1 Introduction to Parallel Computers 3

1.1.1 SIMD Systems 4

1.1.2 General MIMD Systems 4

1.2 Comparison Between SIMD and MIMD Computers 7

1.3 Finite Element Methods for Elliptic PDEs 9

1.3.1 Piecewise Linear Finite Elements 11

1.3.2 Algorithmic Details 12

1.4 Time-Dependent Problems: The Linear Diffusion Equation 15

1.4.1 The Method of Lines 15

1.5 Parallel Finite Element and Load-Balancing 17

1.6 Recursive Graph Partitioning Heuristics 20

1.6.1 Recursive Coordinate Bisection (RCB) 20

1.6.2 Recursive Inertial Bisection (RIB) 21

1.6.3 Recursive Graph Bisection (RGB) 21

1.6.4 Modified Recursive Graph Bisection (MRGB) 21

1.6.5 Recursive Spectral Bisection (RSB) 22

1.6.6 Recursive Node Cluster Bisection (RNCB) 23

1.7 Multisectional Graph Partitioning Heuristics 24

1.7.1 Multidimensional Spectral Graph Partitioning 25

1.7.2 Stripwise Methods 25

1.8 Other Graph Partitioning Techniques 25

1.8.1 Greedy Algorithm (GR) 25

1.8.2 Kernighan and Lin Type Algorithms 26

1.8.3 State of the Art Software Tools for Graph Partitioning . . . 26

2 Adaptivity and Dynamic Load Balancing 29

2.1 Spatial Error Indicators 30

2.2 Different Types of Refinements 31

2.2.1 Regeneration Schemes 31

2.2.2 Local Mesh Adaptation Schemes : Hierarchical Refinement . 32

2.3 Relation Between Adaptivity and Dynamic Load Balancing 34

2.3.1 Generalisations of Static Algorithms 36

2.4 Diffusion Algorithms 39

2.4.1 Basic Diffusion Method 40

2.4.2 A Multi-Level Diffusion Method 40

2.4.3 Dimension Exchange Method 42

2.5 Minimising Data Migration 43

2.6 Two Parallel Multilevel Algorithms 45

2.6.1 ParMETIS 45

2.6.2 ParJOSTLE 46

2.7 Two Further Paradigms 47

2.7.1 Algorithm of Oliker & Biswas 47

2.7.2 Algorithm of Vidwans *et al.* 48

3 A New Dynamic Load Balancer 50

3.1 Motivation of the Algorithm 51

3.2 Description of the Algorithm 51

3.2.1 Group Balancing 52

3.2.2 Local Migration 53

3.3 Further Refinement of the Algorithm : Locally Improving the Partition Quality 56

3.4 Global Load-Balancing Strategy: Divide and Conquer Approach . . 58

3.5 Examples 60

3.6 Conclusions 73

4 Parallel Application of the Dynamic Load Balancer in 2-d 77

4.1 Introduction 79

4.2 A Parallel Dynamic Load-Balancing Algorithm 80

4.2.1 Group Balancing 81

4.2.2 Local Migration 82
4.2.3 Divide and Conquer and Parallel Implementation 84
4.3 Discussion of the Algorithm 87
4.3.1 Activity of Type 1 Processors : Packing the Load 88
4.3.2 Activity of Type 2 Processors : Unpacking the Load 88
4.3.3 Activity of Type 3 Processors : Third Party Adjustment . . 88
4.4 Description of Related Data Structures Associated With the Redistribution of the Mesh 89
4.5 Different Issues and Related Functions Used in the Main Algorithm By Processors of Type 1 93
4.5.1 Handling of Vertices 93
4.5.2 Handling of Edges 95
4.6 Different Issues Which are Related With Processors of Type 2 . . . 100
4.7 Different Issues Which are Related With Processors of Type 3 . . . 101
4.7.1 *insertion()* 101
4.7.2 *deletion()* 101
4.8 Use of Message Passing Interface (MPI) 101
4.9 Some Examples 103
4.9.1 Alternative Algorithms 103
4.9.2 Comparative Results 105
4.10 Discussion 121
4.10.1 Discussion I 121
4.10.2 Discussion II 122
4.11 Conclusions 123

5 Parallel Application of the Dynamic Load Balancer in 3-d 125
5.1 Introduction 126
5.2 A Parallel Adaptive Flow Solver 128
5.2.1 A Parallel Adaptive Algorithm 128
5.2.2 A Parallel Finite Volume Solver 131
5.3 Dynamic Load Balancing 132
5.4 Application of the Parallel Dynamic Load-Balancing Algorithm . . 133
5.4.1 Calculation of WPCG 133
5.4.2 Use of Tokens 134

5.4.3 No Colouring 135
5.4.4 Use of Global Communication 136
5.5 Computational Results 137
5.5.1 Examples 138
5.6 Discussion 150
5.6.1 Discussion I 150
5.6.2 Discussion II 152
5.7 Investigation into Scalability of the Algorithm 154
5.8 Conclusions 160

6 Conclusion and Future Areas of Research 161
6.1 Summary of Thesis 161
6.2 Possible Extensions to the Research 162

List of Figures

1.1 A linear array of processors. 6
1.2 A ring of processors. 6
1.3 Hypercubes of (a) dimension 1, (b) dimension 2 and (c) dimension 3. 7
1.4 (a) Two-dimensional mesh, (b) three-dimensional mesh. 8
1.5 Two-dimensional torus. 8
1.6 Entries of the Laplacian matrix. 23

2.1 Diffusion method. 41
2.2 Multi-level diffusion method. 42
2.3 Dimension exchange method. 43
2.4 The matrix A. 44

3.1 Calculation of Sender, Receiver and Mig_{tot}. 54
3.2 The calculation of gain. 55
3.3 Updation of gain densities and edges cut between the processors. . . 56
3.4 Initial version of load balancing of the two groups. 57
3.5 An algorithm for refining the partitions between a pair of processors. 58
3.6 Group-balancing algorithm: version two of load balancing of the two groups. 59
3.7 A divide & conquer type dynamic load-balancing algorithm. 61
3.8 The coarse mesh of Example 1. 63
3.9 The coarse mesh of Example 2. 65
3.10 The coarse "Texas" mesh of Example 4. 69
3.11 Coarse mesh of 5184 elements adapted to initial shock condition for Example 5. 72
3.12 Adapted mesh after 240 time-steps for Example 5. 72

4.1 Updating the gains. 84

4.2 Load balancing of the two groups. 85
4.3 Parallel dynamic load-balancing algorithm. 86
4.4 The array *nonodessuball* which can accommodate nine coarse elements. 92
4.5 The function *Shared*(). 94
4.6 The function *Shared2*(). 94
4.7 The function *Changenbhd*(). 96
4.8 The function *Changenbhd2*(). 97
4.9 The function *Changenbhd3*(). 98
4.10 The function *EdgeChange*(). 99
4.11 The function *DirichEdgeChange*(). 100
4.12 The coarse mesh of Example 3. 108
4.13 The partial view of the coarse mesh of Example 4. 109

5.1 Mesh data-structures in TETRAD 129
5.2 Regular refinement dissecting interior diagonal 130
5.3 Green refinement by the addition of an interior node 131
5.4 Calculation of weights of vertices and edges of the weighted dual graph.135
5.5 Calculation of a row of the weighted Laplacian matrix. 136
5.6 Scalability comparison using a re-balancing tolerance of 5% for Example 1 (where Time = RedTime + SolTime). 155
5.7 Scalability comparison using a re-balancing tolerance of 10% for Example 1 (where Time = RedTime + SolTime). 155
5.8 Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + SolTime). 155
5.9 Scalability comparison using a re-balancing tolerance of 5% for Example 2 (where Time = RedTime + SolTime). 156
5.10 Scalability comparison using a re-balancing tolerance of 10% for Example 2 (where Time = RedTime + SolTime). 156
5.11 Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + SolTime). 156
5.12 Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + 0.2 * SolTime). 157
5.13 Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + 5 * SolTime). 157

5.14 Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + 25 * SolTime). 157
5.15 Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + 0.2 * SolTime). 158
5.16 Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + 5 * SolTime). 158
5.17 Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + 25 * SolTime). 158

List of Tables

3.1 Partition generated in parallel on 8 processors along with our final partitions for Example 1. 64

3.2 Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.1) of Example 1. 64

3.3 Partition generated in parallel on 8 processors along with our final partitions for Example 2. 66

3.4 Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.3) of Example 2. 66

3.5 Partition generated in parallel on 8 processors along with our final partitions for Example 3. 68

3.6 Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.5) of Example 3. 68

3.7 Partition generated in parallel on 16 processors along with our final partitions for Example 4. 70

3.8 Summary of results when the New, Vidwans *et al.* and JOSTLE algorithms are applied to the initial partition (see Table 3.7) of Example 4. 70

3.9 Initial and final partitions (produced by the New algorithm) for Example 5. 73

3.10 Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.9) of Example 5. 73

3.11 Initial and final partitions (produced by the New algorithm) for Example 6. 74
3.12 Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.11) of Example 6. 74

4.1 Data for the partitions of Example 1 (involving parallel mesh generation and repartitioning on 2 processors). 107
4.2 Data for the partitions of Example 2 (involving parallel mesh generation and repartitioning on 4 processors). 107
4.3 Data for the partitions of Example 3 (involving parallel mesh generation and repartitioning on 4 processors). 109
4.4 Data for the partitions of Example 4 (involving parallel mesh generation and repartitioning on 4 processors). 110
4.5 Data for the partitions of Example 5 (involving parallel mesh generation and repartitioning on 2 processors). 110
4.6 Data for the partitions of Example 6 (involving parallel mesh generation and repartitioning on 4 processors). 111
4.7 Comparison of dynamic load-balancing results using four algorithms for Examples 1 to 6. 112
4.8 Data for the partitions of Example 7 (involving parallel mesh generation and repartitioning on 8 processors). 113
4.9 Data for the partitions of Example 8 (involving parallel mesh generation and repartitioning on 16 processors). 115
4.10 Data for the partitions of Example 9 (involving parallel mesh generation and repartitioning on 8 processors). 116
4.11 Data for the partitions of Example 10 (involving parallel mesh generation and repartitioning on 16 processors). 117
4.12 Data for the partitions of Example 11 (involving parallel mesh generation and repartitioning on 8 processors). 118
4.13 Data for the partitions of Example 12 (involving parallel mesh generation and repartitioning on 16 processors). 119
4.14 Comparison of dynamic load-balancing results using four algorithms for Examples 7 to 12. 120

5.1 Some partition-quality metrics immediately before and after a single re-balancing step for Example 1. 141
5.2 Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 5% for Example 1. 142
5.3 Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 10% for Example 1. 143
5.4 Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 15% for Example 1. 144
5.5 Some partition-quality metrics immediately before and after a single re-balancing step for Example 2. 146
5.6 Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 5% for Example 2. 147
5.7 Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 10% for Example 2. 148
5.8 Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 15% for Example 2. 149

Chapter 1

Introduction

Since the middle of the current century, breakthroughs in computer technology have made a tremendous impact on numerical methods in general and the numerical solutions of partial differential equations in particular. During the infancy period of computers they were serial in nature. This means that they were built using the von Neumann paradigm: with a single processor which runs as fast as possible and has as much memory as is possible (or affordable). The processor is commonly known as the central processing unit (CPU) and is further divided into a control unit and an arithmetic-logic unit (ALU). The memory stores both instructions and data. The control unit directs the execution of programs, and the ALU carries out the calculations called for in the program. When they are being used by the program, instructions and data are stored in very fast memory locations, called registers. As fast memory is quite expensive, there are relatively few registers.

The performance of such computers is clearly limited by physical laws. For example, the maximum speed at which the data can travel from memory to CPU is that of the speed of light, so in order to build a computer which is capable of carrying out three trillion copies of data between memory and registers per second say, one has to fit each 32-bit word into a square with side length of 10^{-10} meters (this is approximately equal to the size of a relatively small atom). This is simply not possible - see [79] for details.

In order to speed up the machine, one possibility is to reduce the transfer time taken by the data while travelling from memory to registers. This is achieved by the use of cache memory - which is implemented on the same chip as the CPU. The idea behind cache is the observation that programs tend to access both data and

instructions sequentially. Hence, if we store a small block of data and a small block of instructions in fast memory (cache), most of the program's memory accesses will use this cache memory rather than the slower main memory. This memory will be slower than registers but it will be faster than the main memory implemented outside the chip.

During the initial stages in the development of microchips, designers typically used an increased chip area to introduce new and sophisticated instructions, addressing modes and other mechanisms. These new features allowed the execution of high-level languages as well as the complex functions of operating systems, and this trend continued until the 1980s. At this time a new design philosophy called the reduced instruction set computer (RISC) emerged. The RISC supporters argue that all these new instructions complicate the design of the control unit, slowing down the execution of basic operations. A simple instruction set allows, in principle, a simple, fast implementation, so the larger number of instructions that is require can be more than compensated for by the increased speed. Another advantage claimed by RISC supporters is that the simplification of the control unit helps to save chip area for the control implementation. This can be used to implement special features in the operating unit, aimed at improving the execution speed. There are many variants of RISC processors, among them are Berkeley RISC, microprocessors without interlocked pipe stages (MIPS) and the Inmos Transputer (see Chapter 10 of [20]).

Even after all these advancements in the development of the computer industry there were and still are important classes of problem in science and engineering which practitioners have not been able to solve successfully. For example, to attack the "Grand Challenges" ([17]) months or even years are needed by the best of these computers. A grand challenge is a fundamental problem in science or engineering that has a broad economic and scientific impact, and whose solution could be advanced by applying high-performance computing techniques and resources ([67]). Many of these problems are basically large computational fluid dynamics problems which can be modelled by a set of partial differential equations (PDEs).

To have an idea of the computing requirements to solve such problems numerically we consider here two examples. The first one is studied by Case *et al.* ([16]) as mentioned in [26]. This is the simulation of a three-dimensional, fully resolved, turbulent flow as might occur in the design of a portion of a ship hull. The primary

parameter for characterising the turbulent fluid flow is the dimensionless quantity known as the Reynolds number (R). Such a simulation would have a Reynolds number of about 10^4 or greater. In order to fully resolve important disturbances of a small wave number in the flow one needs $R^{9/4}$ mesh points ([21]). That is N = 10^9 mesh points for each time step. Each mesh point has one pressure term and three velocity terms for both the current and the immediate past time step. This is a total of 8 $\times 10^9$ scalar variables. If temperature or other parameters must also be maintained for each point, then about 10^{10} words of data memory are required. The number of arithmetic operations varies widely, depending upon the solution method employed. One efficient approach that takes advantage of the problem geometry, has been estimated to require only about 500 additions and 300 multiplications per grid point. This leads to an operations count of $\approx 10^{12}$ operations per single time step (see [26] for details).

The second problem is that of modeling and forecasting of weather. Suppose we want to predict the weather over an area of 3000 $\times$ 3000 miles for two-day period and the parameters need to computed once every half hour. As mentioned in [67], if the area is being modeled up to a height of 11 miles and one wishes to partition this 3000 $\times$ 3000 $\times$ 11 cubic mile domain into segments of size 0.1 $\times$ 0.1 $\times$ 0.1 then there would be 10^{11} different segments. So we need at least 10^{11} words of data memory. It is also estimated in [67] that for this prediction the total number of operations is 10^{15}.

The world's most powerful computer of the mid 70's was the CRAY-1, which was not even close to having enough capability ([84]) to perform these calculations. It's primary memory was limited to 10^6 words, and the execution rate was about 10^8 operations/second.

Hence by this time it was clear that new, more powerful computer systems would be needed to solve this class of problems. Since the single processor machines had begun to start approaching its physical limits, the community had no choice but to consider alternative paradigms such as parallel machines.

1.1 Introduction to Parallel Computers

Parallel computers perform their calculations by executing different computational tasks on a number of processors concurrently. The processors within a parallel

computer generally exchange information during the execution of the parallel code. This exchange of information occurs either in the form of explicit messages sent by one processor to another or different parallel processors sharing a specified common memory resource within the parallel computer. The parallel load-balancing algorithms, proposed in this thesis, work very well on these paradigms.

In 1966 Michael Flynn ([33]) classified systems according to the number of instruction streams and the number of data streams. The two important systems are:

- SIMD - Single Instruction stream, Multiple Data stream,
- MIMD - Multiple Instruction stream, Multiple Data stream.

This section provides a brief introduction to these important classes of parallel computing architecture.

1.1.1 SIMD Systems

Such a system has a single CPU devoted to exclusively to control, and a large collection of subordinate processors, each having only ALUs, and their own (small amount of) memory. During each instruction cycle, the control processor broadcasts an instruction to all of the subordinate processors, and each of the subordinate processors either executes the instruction or is idle.

The most famous examples of SIMD machines are the CM-1 and CM-2 connection Machines that were produced by Thinking Machines. The CM-2 had up to 65,356 1-bit processors and up to 8 billion bytes of memory. Maspar also produced SIMD machines. The MP-2 has up to 16,384 32-bit ALUs and up to 4 billion bytes of memory.

1.1.2 General MIMD Systems

The key difference between MIMD and SIMD systems is that with MIMD systems, the processors are autonomous: each processor is a full-fledged CPU with both a control unit and an ALU. Thus each processors is capable of executing its own program at its own pace. The world of MIMD systems is divided into shared-memory and distributed-memory systems.

Shared-Memory MIMD

A generic shared-memory machine consists of a collection of processors and memory modules interconnected by a network. Each processor has access to the entire address space of the memory modules. So that any data stored in the shared memory is common to, and can be accessed by, any of the processors. This has the advantage of being very rapid (in principle) and is generally simpler to program. However, its main drawback is that there can be serious delays (contention time) if more than one processor wants to use the same location in memory at the same time. The simplest network connection is bus based. Due to the limited bandwidth of a bus, these architectures do not scale to large number of processors: the largest configuration of the currently popular SGI Challenge XL has only 36 processors. Recently Silicon Graphics, Inc. has designed and manufactured the Origin 2000 computer. The basic building block of the Origin is a node built upon two MIPS R10000 processors with a peak performance of 400 Mflop each. The computer utilises Scalable Shared-memory MultiProcessing (S2MP) architecture. Most other shared-memory architectures rely on some type of switch-based interconnection network. For example the basic unit of the Convex SPP1200 is a 5×5 crossbar switch.

Distributed-Memory MIMD

In distributed-memory systems, each processor has its own private memory. These processors are connected directly or indirectly by means of communication wires. From the performance and programming point of view the ideal interconnection network is a fully connected network, in which each processor is directly connected to every other processor. Unfortunately, the exponential growth in the size (and cost) of such a network makes it impractical to construct such a machine with more than a few processors. At the opposite extreme from a fully connected network is a **linear array**: a static network in which all but two of the processors have two immediately adjacent neighbouring processors (see Figure 1.1). A **ring** is a slightly more powerful network. This is just a linear array in which "terminal" processors have been joined (see Figure 1.2). These networks are relatively inexpensive; the only additional cost is the cost of p - 1 or p wires for a network of p processors. Moreover it is very cheap to upgrade the network - to add one processor we only

Figure 1.1: A linear array of processors.

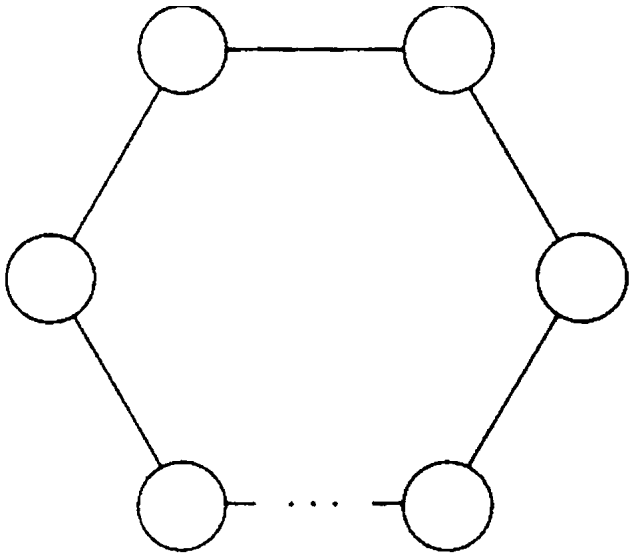

Figure 1.2: A ring of processors.

need one extra wire. There are two principal drawbacks:

- if two processors are communicating, it's very likely that this will prevent other processors which are also attempting to communicate from doing so,
- in a linear array two processors that are attempting to communicate may have to forward the message along as many as p - 1 wires, and in a ring it may be necessary to forward the message along as many as p/2 wires.

In between the two extremes a **hypercube** is a practical static interconnection network that gives a good balance between the high cost and high speed of the fully connected network and the low cost but poor performance of the linear array or ring. Hypercubes are defined inductively: a dimension 0 hypercube consists of a single processor. In order to construct a hypercube of dimension $d > 0$, we take two hypercubes of dimension $d-1$ and join the corresponding processors with communication wires (see Figure 1.3). It is clear that a hypercube of dimension

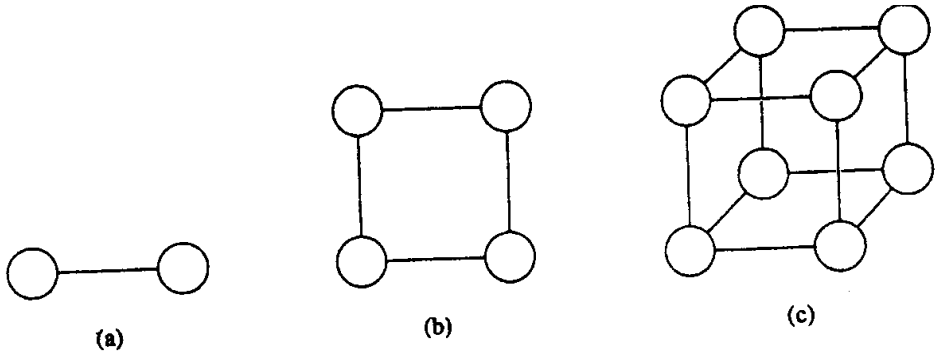

Figure 1.3: Hypercubes of (a) dimension 1, (b) dimension 2 and (c) dimension 3.

d will consist of 2^d processors. It is also clear that in a hypercube of dimension d each processor is directly connected to d other processors and that if we follow the shortest path then the maximum number of wires a message has to travel is d. This is much few than for the linear array of ring. The principal drawback to the hypercube is that it is not easy to upgrade the system : each time we wish to increase the machine size, we must double the number of processors and add a new wire to each processor. The first "massively parallel" MIMD system was a hypercube (an nCUBE 10 with 1024 processors).

Intermediate between hypercubes and linear arrays are the **meshes** and **tori** (see Figures 1.4 and 1.5), which are simply higher dimensional analogues of linear arrays and rings, respectively. Observe that an n-dimensional torus can be obtained from the n-dimensional mesh by adding "wrap-around" wires to the processors on the border. As far as upgrading is concerned meshes and tori are better than hypercubes (although not as good as linear arrays and rings). For example, if one wishes to increase the size of a q $\times$ q mesh, one simply adds a q $\times$ 1 mesh and q wires. Meshes and tori are currently quite popular. The Intel Paragon is a two-dimensional mesh, and the Cray T3D and T3E are both three-dimensional tori.

1.2 Comparison Between SIMD and MIMD Computers

In [67], Kumar *et al.* discuss the pros and cons of SIMD and MIMD computers. SIMD computers require less hardware and less memory than MIMD computers

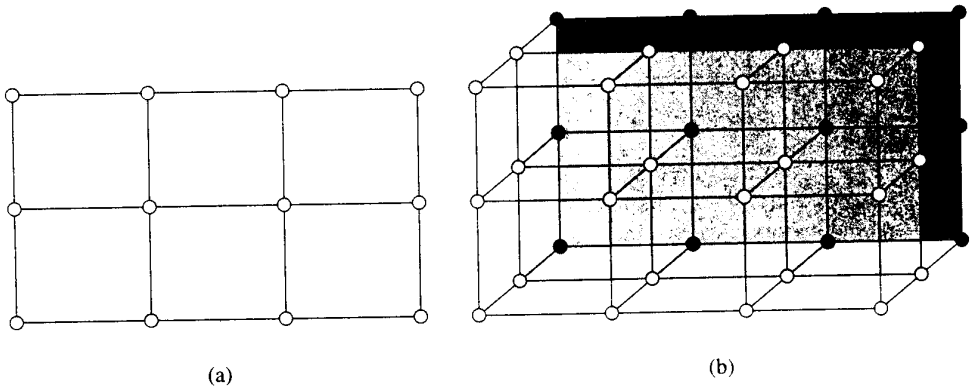

Figure 1.4: (a) Two-dimensional mesh, (b) three-dimensional mesh.

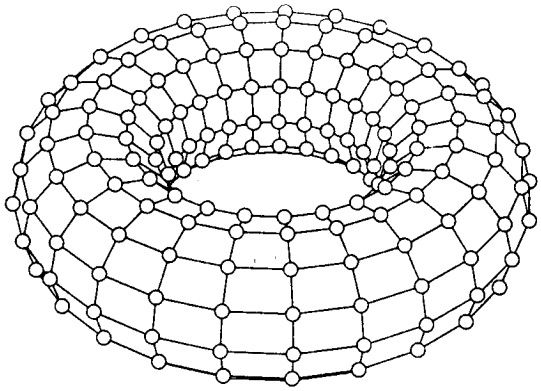

Figure 1.5: Two-dimensional torus.

because they have only one global control unit and only one copy of the program needs to be stored. On the other hand, MIMD computers store the program and operating system at each processor. SIMD computers are naturally suited for data-parallel programs; that is, programs in which the same set of instructions are executed on a large data set (which is the case in the field of image processing for example).

A clear disadvantage of SIMD computers is that different processors cannot execute different instructions in the same clock cycle, so if a program has many conditional branches or long segments of code whose execution depends on conditionals, it is entirely possible that many processors will remain idle for long periods of time. Data-parallel programs in which significant parts of the computation are contained in conditional statements are therefore better suited to MIMD computers than to SIMD computers.

Individual processors in an MIMD computer are more complex, because each processors has its own control unit. It may seem that the cost of each processor must be higher than the cost of a SIMD processor. However, it is possible to use general-purpose microprocessors as processing units in MIMD computers. In contrast, the CPU used in SIMD computers has to be specially designed. Hence, due to economies of scale, processors in MIMD computers may be both cheaper and more powerful than processors in SIMD computers.

1.3 Finite Element Methods for Elliptic PDEs

Probably the three most popular numerical techniques for solving partial differential equations are the finite difference, the finite element and the finite volume methods. In the finite difference approximation, the derivatives in a differential equation are replaced by difference quotients. The difference operators are usually derived from Taylor series and involve the values of the solution at neighbouring points in the domain. After taking the boundary conditions into account, a (sparse) system of algebraic simultaneous equations is obtained and can be solved for the nodal unknowns.

The finite differences method (FDM) is easy to understand and straightforward to implement on regular domains. Unfortunately this method is difficult to apply for systems with irregular geometries and/or unusual boundary conditions.

The finite element method (FEM) provides an alternative that is better suited for such systems. In contrast to finite difference techniques, the finite element method divides the solution domain into simply shaped regions or "elements". An approximate solution for the PDE can be developed for each of these elements. The total solution is then generated by linking together or "assembling" the individual solutions taking care to ensure continuity at the interelement boundaries. Thus the PDE is approximately satisfied in a piecewise fashion (see below).

The finite volume method (FVM) may also be applied on unstructured meshes. In this scheme the solution is represented as a series of piecewise constant elements. The discretised form of the PDE is found by integrating the equation over the elements (control volumes). For each control volume the area integral is converted into a line integral over its edges and the numerical flux at the boundaries also calculated.

A comprehensive description of the finite element method is beyond the scope of this thesis. The interested reader can consult the books of Johnson ([58]) and Strang & Fix ([95]). However we describe the method briefly in the case of a particular PDE; Poisson's equation in 2 dimensions :

$$-\nabla^2 u(\underline{x}) = f(\underline{x}), \qquad \text{for } \underline{x} \in \Omega \subset \Re^2. \tag{1.1}$$

For clarity we assume the following boundary conditions are imposed :

$$u = u_E \ \ on \ \ \Gamma_1 \ \ and \ \ \tfrac{\partial u}{\partial n} = g \ \ on \ \ \Gamma_2,$$
$$where \ \ \partial\Omega = \Gamma_1 \cup \Gamma_2 \ \ and \ \ \Gamma_1 \cap \Gamma_2 = \emptyset.$$

Note that this equation is a linear second order partial differential equation which arises in a large number of physical situations (e.g. flow of an ideal fluid).

The boundary condition $u = u_E \ on \ \Gamma_1$ is called a Dirichlet (or essential) boundary condition and $\frac{\partial u}{\partial n} = g \ on \ \Gamma_2$ is called a Neumann (or natural) boundary condition.

We first derive the weak form of the equation (1.1). To do this we multiply the equation (1.1) by a test function w and integrate over Ω to get,

$$-\int_\Omega w\nabla^2 u \ d\underline{x} = \int_\Omega wf \ d\underline{x}.$$

By using the divergence theorem we get,

$$\int_\Omega \underline{\nabla} u . \underline{\nabla} w \; d\underline{x} - \int_{\partial\Omega} \frac{\partial u}{\partial n} w \; ds = \int_\Omega w f \; d\underline{x}.$$

If we choose w $\in H_0^1(\Omega)$, (where $H_0^1(\Omega)$ stands for the space of all functions whose first derivatives are square integrable in Ω and which are zero everywhere on Γ_1) then above integral form reduces to,

$$\int_\Omega \underline{\nabla} u . \underline{\nabla} w \; d\underline{x} - \int_{\Gamma_2} g w \; ds = \int_\Omega w f \; d\underline{x}.$$

For simplicity we will assume g $\equiv$ 0 in which case the expression simplifies still further:

$$\int_\Omega \underline{\nabla} u . \underline{\nabla} w \; d\underline{x} = \int_\Omega w f \; d\underline{x}.$$

Now let $H_E^1(\Omega)$ be the space of all those functions whose first derivatives are square integrable in Ω and which satisfy the Dirichlet boundary condition everywhere on Γ_1. The above integral form then leads to the following weak form of the Poisson's PDE.

Find $u \in H_E^1(\Omega)$ such that

$$\int_\Omega \underline{\nabla} u . \underline{\nabla} w \; d\underline{x} = \int_\Omega w f \; d\underline{x}, \tag{1.2}$$

for all w $\in H_0^1(\Omega)$.
The rest of this section considers the finite element approximation to the solution of this weak form.

1.3.1 Piecewise Linear Finite Elements

The very first step in the approximation of u by the finite element method is to divide the domain Ω into a large number of small non-overlapping subdomains (in this section we will assume these are triangles). This is always possible provided that Ω is itself a polygon (i.e. there are no curved boundaries). There are methods to handle curved boundaries. One way is to approximate the curve boundary by means of a set of line segments in such a way that in the limit these line segments approach to the curve boundary (see [95] for details). Another way is to triangulate the domain Ω using isoparametric finite elements (see [19]).

Let us suppose that the vertices (nodes) of the triangles have been numbered from 1 to N = $n_B + n_E$ (where n_B is the number of vertices in the interior of the domain or on the Neumann boundary, Γ_2, and n_E is the number of vertices on

the Dirichlet boundary, Γ_1). On each triangle u is approximated by a low degree polynomial. Although any degree polynomials can be selected for simplicity we choose the first degree polynomials here (polynomials of degree zero can not be used since we require the derivatives to be square integrable).

We next define simple "basis" functions $P_i(\underline{x})$ for all nodes i from 1 to N. These functions are linear on each triangle and satisfy $P_j(\underline{x}) = 1$ if $\underline{x}$ is the position vector of the node j and $P_j(\underline{x}) = 0$ if $\underline{x}$ is the position vector of any of the other nodes. Now we can write $\bar{u}$ (an approximations to u) in terms of these basis functions as,

$$\bar{u} = \sum_{i=1}^{N} a_i P_i(\underline{x}), \tag{1.3}$$

where a_i are unknown (to be determined) for i = 1, ..., n_B, and are given by the Dirichlet boundary condition , u = u_E, for i = n_B+1, ...,$n_B + n_E$. (Note that, due to our choice of basis functions , a_i is the value of $\bar{u}$ when evaluated at the i^{th} node of the mesh). If we substitute the value of $\bar{u}$ from equation (1.3) for u and replace w by $P_j(\underline{x})$, for j = 1, ...,n_B, in equation (1.2) we then get a system of n_B equations for the unknowns $a_1, \ldots, a_{n_B}$. This system, known as the Galerkin finite element equations is given by:

$$\sum_{i=1}^{n_B} a_i \int_\Omega \underline{\nabla} P_i(\underline{x}).\underline{\nabla} P_j(\underline{x})\, d\underline{x} = \int_\Omega P_j(\underline{x}) f(\underline{x})\, d\underline{x} - \sum_{i=n_B+1}^{N} a_i \int_\Omega \underline{\nabla} P_i(\underline{x}).\underline{\nabla} P_j(\underline{x})\, d\underline{x}, \tag{1.4}$$

$$\text{for j} = 1, \ldots, n_B.$$

Typically this is written in matrix form as

$$K\underline{a} = \underline{f}, \tag{1.5}$$

where K is referred to as the "global stiffness matrix" (whose entries are given by $K_{ji} = \int_\Omega \underline{\nabla} P_j(\underline{x}).\underline{\nabla} P_i(\underline{x})\, d\underline{x}$) and $\underline{a}$ is a vector of the unknowns $a_1, \ldots, a_{n_B}$.

1.3.2 Algorithmic Details

Having derived the finite element equations (1.5) we now discuss how the matrix K and the vector $\underline{f}$ can be obtained systematically. The most important point to note is that the entry K_{ji} of the matrix K will always be zero if the vertices numbered j and i are not connected by an edge of the mesh. This is because the dot product of $\underline{\nabla} P_j(\underline{x})$ and $\underline{\nabla} P_i(\underline{x})$ will be zero on every triangular element in such a case. Since

this means that most of the entries of K will always be zero, we refer to this as a "sparse matrix".

Suppose that the finite element mesh consists of E triangular elements Ω_e (e = 1,...,E). Then each entry of K may be obtained from the following formula:

$K_{ji} = \int_\Omega \underline{\nabla} P_j(\underline{x}).\underline{\nabla} P_i(\underline{x})\, d\underline{x} == \sum_{e=1}^{E} \int_{\Omega_e} \underline{\nabla} P_j(\underline{x}).\underline{\nabla} P_i(\underline{x})\, d\underline{x}.$

Hence we may use the following pseudo-code to calculate K:

```
for(j = 1; j ≤ N; j++)
    for(i = 1; i ≤ N; i++){
        K(j,i) = 0
        for(e = 1; e ≤ E; e++)
            K(j,i) = K(j,i) + ∫_{Ω_e} ∇P_j(x).∇P_i(x) dx
    }.
```

The order of the loops can easily be re-arranged:

```
for(j = 1; j ≤ N; j++)
    for(i = 1; i ≤ N; i++)
        K(j,i) = 0
for(e = 1; e ≤ E; e++)
    for(j = 1; j ≤ N; j++)
        for(i = 1; i ≤ N; i++)
            K(j,i) = K(j,i) + ∫_{Ω_e} ∇P_j(x).∇P_i(x) dx.
```

Now we can make use of the sparsity caused by the local nature of $P_1, \ldots, P_N$:

```
for(j = 1; j ≤ N; j++)
    for(i = 1; i ≤ N; i++)
        K(j,i) = 0
for(e = 1; e ≤ E; e++)
    for(J = 1; J ≤ 3; J++){
        j = number of node which is J-th vertex of element e
        for(I = 1; I ≤ 3; I++){
            i = number of node which is I-th vertex of element e
            K(j,i) = K(j,i) + ∫_{Ω_e} ∇P_j(x).∇P_i(x) dx
        }
    }.
```

At this point we can make the following observations.

- It is necessary to number the vertices of each element, Ω_e, of the triangulation of Ω; 1,2,3. Also, an integer array, "icon" say, needs to be set up which stores the node number of each vertex of each element.
- It is also necessary to store an array of the position vectors, $\underline{s}_j$ say, of the vertices of the mesh.
- A similar arrangement can be made in order to calculate $\underline{f}$, where $f_j = \int_\Omega f(\underline{x})P_j(\underline{x})\, d\underline{x} = \sum_{e=1}^{E} \int_{\Omega_e} f(\underline{x})P_j(\underline{x})\, d\underline{x}$.

Now, if we assume that the n_E nodes on Γ_1 are numbered last, then the finite element pseudo-code should now look something like:

```
for(j = 1; j ≤ n_B; j++){
    f(j) = 0
    for(i = 1; i ≤ n_B ; i++)
        K(j,i) = 0
}
for(j = 1; j ≤ n_E ; j++)
     a(n_B + j) = u_E(s(n_B + j, 1), s(n_B + j, 2))
for(e = 1; e ≤ E; e++)
    for(J = 1; J ≤ 3; J++){
        j = icon(e,J)
        if(j ≤ n_B){
          f(j) = f(j) + ∫_Ωe f(x)P_j(x)dx
          for(I = 1; I ≤ 3; I++){
              i = icon(e,I)
              if(i ≤ n_B)
                K(j,i) = K(j,i) + ∫_Ωe ∇P_j(x).∇P_i(x) dx
              else
                f(j) = f(j) - a(i) ∫_Ωe ∇P_j(x).∇P_i(x) dx
          }
        }
    }
```

Solve the system: $K\underline{a} = \underline{f}$.

The parallel generation and solution of this system will be discussed in §1.5.

1.4 Time-Dependent Problems: The Linear Diffusion Equation

We now move on to consider how we may generalise the above theory to deal with a linear time-dependent differential equation. The simplest parabolic time-dependent differential equation is the linear diffusion equation:

$$\frac{\partial}{\partial t}u(\underline{x},t) = \nabla^2 u(\underline{x},t) + f(\underline{x},t) \quad for (\underline{x},t) \in \Omega \times (0,T], \tag{1.6}$$

subject to some initial condition, such as

$$u(\underline{x},0) = u^0(\underline{x}) \; for \; all \; \underline{x} \; \in \Omega,$$

and some boundary conditions, such as

$$u = u_E \; on \; \Gamma_1 \; and \; \frac{\partial u}{\partial n} = g \; on \; \Gamma_2 \; for \; all \; t \in (0,T),$$

where $\partial\Omega = \Gamma_1 \cup \Gamma_2$ and $\Gamma_1 \cap \Gamma_2 = \emptyset$, as before.

Note that in above all the spatial variables have been grouped together as $\underline{x}$ and the Laplacian operator, ∇^2, is assumed to apply only to these spatial variables and the time variable t is treated separately. This distinction is necessary as the variable t should not be thought of as being "just another independent variable", like x and y say, because the boundary conditions associated with this variable are not the same.

As far as 't' is concerned we only know the solution at the boundary t = 0 and would like to compute the solution for arbitrary values of t which are less than T (we have no idea about the behaviour of the solution at time T). This differs from the other variables where we generally know about the behaviour of the solution throughout the boundary of the spatial domain.

Keeping in mind the special nature of the variable 't' we need a practical method which treats the spatial variables and time variable independently. Fortunately the method of lines exactly does the same.

1.4.1 The Method of Lines

This is a general method which reduces a system of PDEs to a system of ordinary differential equations (ODEs), by only discretising in space in the first instance.

The spatial and temporal discretisation are thus independent, allowing a variety of spatial discretisations (e.g. finite element or finite volume) to be used with any standard ODE solver (see [9], for example). We attempt to follow this approach to obtain transient solutions using the finite element method presented in previous section.

This means we only triangulate the spatial part of the domain Ω, and then we multiply equation (1.6) by a test function, $P_j(\underline{x})$, which has no time dependence. This yields the following system of equations,

$$\int_\Omega \frac{\partial}{\partial t} u(\underline{x},t) P_j(\underline{x})\, d\underline{x} = \int_\Omega \nabla^2 u(\underline{x},t) P_j(\underline{x})\, d\underline{x} + \int_\Omega f(\underline{x},t) P_j(\underline{x})\, d\underline{x},$$

and making use of the divergence theorem as before this becomes,

$$\int_\Omega \frac{\partial}{\partial t} u(\underline{x},t) P_j(\underline{x})\, d\underline{x} = -\int_\Omega \nabla u(\underline{x},t).\nabla P_j(\underline{x})\, d\underline{x} + \int_{\partial\Omega} \frac{\partial}{\partial n} u(\underline{x},t) P_j(\underline{x}) ds + \int_\Omega f(\underline{x},t) P_j(\underline{x})\, d\underline{x}. \tag{1.7}$$

In two dimensions we may again divide the domain, Ω, into triangles and number the vertices of these triangles from 1 to N $= n_B + n_E$ where n_B *and* n_E are as defined in §1.3.1. Also let P_j be the usual basis functions centred on the j^{th} node of the mesh. Since we are interested in a time-dependent finite element solution we seek an approximation, $\bar{u}(\underline{x},t)$, to the true solution, u($\underline{x}$,t), of the form

$$\bar{u} = \sum_{i=1}^{N} a_i(t) P_i(\underline{x}),$$

where $a_i(t)$ are unknown (to be determined) for i = 1,...,n_B, and are given by the Dirichlet boundary condition, u = u_E, for i = n_B + 1,..., $n_B + n_E$.

Now, replacing u by $\bar{u}$ in equations (1.7) for j = 1,...,n_B we again obtain a system of n_B equations for n_B unknowns (in this case $a_1(t),\ldots,a_{n_B}(t)$). This system is given by

$$\sum_{i=1}^{n_B} \frac{da_i}{dt} \int_\Omega P_i P_j d\underline{x} = -\sum_{i=1}^{n_B} a_i \int_\Omega \nabla P_i.\nabla P_j d\underline{x} + \int_{\Gamma_2} g P_j ds + \int_\Omega f P_j d\underline{x} -$$

$$\sum_{i=n_B+1}^{n_B+n_E} a_i \int_\Omega \nabla P_i.\nabla P_j d\underline{x} - \sum_{i=n_B+1}^{n_B+n_E} \frac{da_i}{dt} \int_\Omega P_i P_j d\underline{x}.$$

As before we may express this in matrix notation, in which case it becomes,

$$M\frac{d\underline{a}}{dt} = -K\underline{a} + \underline{f}(t). \tag{1.8}$$

Again, K is the "global stiffness matrix" (whose entries are given by $K_{ji} = \int_\Omega \nabla P_i . \nabla P_j d\underline{x}$) and the matrix M is known as the"Galerkin mass matrix" (with entries given by $M_{ji} = \int_\Omega P_i P_j d\underline{x}$). In this case the vector $\underline{f}$ depends upon t through the possible dependence of the function f in equation (1.6) upon t, or the possible dependence of the Dirichlet boundary condition upon t (through the function u_E).

It should be noticed that the system of equations, (1.8), is not an algebraic system, it is a system of n_B ordinary differential equations for which we can easily obtain initial values for the unknowns $a_i(t)$ (from the function $u^0(\underline{x})$). There are many standard techniques (e.g. the software package SPRINT which is described in [9]) for dealing with equations such as these in an efficient manner (i.e. using local error through adaptive time-stepping). Nevertheless, at each time step a finite element calculations similar to that described in §1.3 must be undertaken.

1.5 Parallel Finite Element and Load-Balancing

For a small problem where the number of degrees of freedom is just a few thousand the system of equations (1.5) can be easily and quickly solved on a serial machine. But when the number of degrees of freedom is in excess of a million or so then the memory and speed of a serial machine start to become a serial bottleneck. Also for some applications where the size of the problem is not so big the time taken by a serial machine may still be very large (for non-linear problems for example, where the iterative methods for solving the corresponding system (1.5) are quite expensive). In these cases a promising way forward is to use a parallel architecture. By using such a machine not only can we hope to solve larger problems (e.g. in structural mechanics) but we can also hope to solve them more quickly.

In the rest of this section we discuss a method for assembling and solving the sparse system of equations (1.5) in parallel. Let us suppose the domain Ω has been divided into n subdomains Ω_1, Ω_2, ... ,Ω_n and the i^{th} subdomain Ω_i has been assigned to the i^{th} processor of a parallel machine. Let us assume that the unknowns on the interface between the subdomains are labelled $\underline{a}^*$ and the unknowns inside each subdomains are labelled $\underline{a}_1$, $\underline{a}_2$,....,$\underline{a}_n$. If we first number the unknowns in $\underline{a}_1$ then in $\underline{a}_2$,$\underline{a}_3$,...,$\underline{a}_n$ and lastly in $\underline{a}^*$ then the system of equations (1.5) can be written in the form,

$$\begin{bmatrix} A_1 & & & & & C_1 \\ & A_2 & & & & C_2 \\ & & . & & & . \\ & & & . & & . \\ & & & & A_n & C_n \\ B_1 & B_2 & . & . & B_n & A^* \end{bmatrix} \begin{bmatrix} \underline{a}_1 \\ \underline{a}_2 \\ . \\ . \\ \underline{a}_n \\ \underline{a}^* \end{bmatrix} = \begin{bmatrix} \underline{f}_1 \\ \underline{f}_2 \\ . \\ . \\ \underline{f}_n \\ \underline{f}^* \end{bmatrix}, \tag{1.9}$$

where A_i, B_i, C_i and A^* are themselves usually sparse. It is clear from the definition of the basis functions P_j that $\underline{f}_i$, A_i, B_i and C_i are totally housed by the i^{th} process and hence can be assembled independent of each other in parallel. But $\underline{f}^*$ and A^* are distributed across different processors. Each processor can compute and assemble its own contribution to them, independently, storing them in the blocks $\underline{f}_i^*$ and A_i^* say (so that $\underline{f}^* = \underline{f}_1^* + \underline{f}_2^* + ... + \underline{f}_n^*$ and $A^* = A_1^* + A_2^* + ... + A_n^*$).

In order to solve the system of equations (1.9) we first write it in component form:

$$A_i\underline{a}_i + C_i\underline{a}^* = \underline{f}_i,\ i = 1, 2, \ldots, n, \tag{1.10}$$

$$\sum_i B_i\underline{a}_i + A^*\underline{a}^* = \underline{f}^*. \tag{1.11}$$

If we substitute the value of $\underline{a}_i$ from equation (1.10) in (1.11) we get the following equation:

$$\sum_i B_i A_i^{-1}(\underline{f}_i - C_i\underline{a}^*) + A^*\underline{a}^* = \underline{f}^*, \tag{1.12}$$

On simplification this reduces to,

$$(A^* - \sum_i B_i A_i^{-1} C_i)\underline{a}^* = \underline{f}^* - \sum_i B_i A_i^{-1}\underline{f}_i. \tag{1.13}$$

If we define A_s by the equation,

$$A_s = A^* - \sum_i B_i A_i^{-1} C_i, \tag{1.14}$$

then the equation (1.13) can simply be written as,

$$A_s\underline{a}^* = \underline{f}^* - \sum_i B_i A_i^{-1}\underline{f}_i. \tag{1.15}$$

If equation (1.15) is then solved for $\underline{a}^*$ then this can be substituted into equation (1.10) and solved for $\underline{a}_i$ for all i. This approach is ideal for distributed memory parallel machines because each system in equation (1.10) is entirely independent

and may therefore be solved in parallel with the others when required. Moreover, if an iterative method, such as the conjugate gradient (CG) algorithm ([40]), is used to solve equation (1.15) then it is not necessary to explicitly form the matrix A_s of (1.14). The main step involved is the matrix vector multiplication of $\underline{w} = A_s\underline{p}$ where $\underline{p}$ is the direction vector obtained from the residual of the k^{th} iterates of $\underline{a}^*$, so we have

$$\underline{w} = A^*\underline{p} - \sum_i B_i(A_i^{-1}(\underline{C}_i\underline{p})). \tag{1.16}$$

From equation (1.16) it is clear that $\underline{w}$ can be obtained using only matrix-vector multiplication and subdomain solves (some local communication is also required between processors sharing interpartition boundary vertices).

From above discussion it is clear that the communication overhead is proportional to the number of vertices on the interpartition boundary, hence one should try to keep this boundary as small as possible. Also once the vector $\underline{a}^*$ is known each subdomain will try to solve the equation (1.10) in parallel, hence it is desirable that the number of unknowns in each of $\underline{a}_i$ is approximately same (otherwise some processors will be idle while others are still busy solving their systems).

Hence the decomposition of the elements of the mesh into subdomains should have two main features,

- each processor should store approximately the same number of vertices or elements (to ensure equal load),
- number of vertices which lie on the boundary between the processors should be kept low.

In order to achieve the above we first define the dual graph of a given mesh. The dual graph of a given mesh is obtained by replacing each element by a node, and that a pair of nodes is connected by an edge only if the corresponding elements are neighbours of each other, then above problem becomes a special case of a more general problem, namely the graph partitioning problem.

The n-way graph partitioning problem is defined as follows: Let G = G(N,E) be an undirected graph where N is the set of nodes with $\|N\|$ nodes and E is the set of edges with $\|E\|$ edges, partition N into n subsets, N_1, N_2, ...,N_n such that $N_i \cap N_j = \emptyset$ for i $\neq$ j, $\|N_i\| = \|N\|$ / n and $\bigcup_i N_i$ =N, and the number of edges

of E whose incident vertices belong to different subsets is minimised. The n-way partition problem is most frequently solved by recursive bisection. That is, we first obtain a 2-way partition of N, and then we further subdivide each part using 2-way partitions. After log n phases, graph G is partitioned into n parts. Thus, the problem of performing a n-way partition is reduced to that of performing a sequence of 2-way partitions or bisections.

Unfortunately this problem, which is well-known in the graph theory literature , is not solvable in polynomial time. It is in fact an NP-hard problem ([22, 36, 68]). Nevertheless there are heuristic approaches which perform well in most cases. In the next few sections we review some of the more important of these heuristics.

1.6 Recursive Graph Partitioning Heuristics

For the sake of simplicity (as mentioned above), many graph partitioning heuristics concentrate on bisecting the graph subject to the load balancing and cut-weight (the number of edges on the inter-partition boundary is called the cut-weight) minimisation constraints. When more than two subdomains are required, the procedure can be applied recursively on the recent subdomains. The main advantage of this approach is that it is easy to implement in parallel because of the divide and conquer nature, but the corresponding disadvantage is that the total number of subdomains thus produced must be a power of 2.

1.6.1 Recursive Coordinate Bisection (RCB)

Let G = G(N,E) be a given undirected graph. We must also assume that there are two or three-dimensional coordinates available for the nodes. A simple bisection strategy, due to Simon ([90]), which is a slight generalisation of an earlier method used by Williams in [114], for the graph G is to determine the coordinate direction of the longest expansion of the domain. Without any loss of generality, assume that this is the x-direction. Then all nodes are sorted with respect to their x-coordinate. Half of the nodes with small x-coordinate are assigned to one subdomain, the remaining half are assigned to the other subdomain.

Although easy to program, the principal drawback of RCB is that the method does not take advantage of the connectivity information given by the graph. It is

therefore unlikely that the resulting partition will have a low cut-weight and so this method is not generally suitable for our purpose.

1.6.2 Recursive Inertial Bisection (RIB)

This method is a generalisation of RCB techniques which is described in [28, 75] for example. Here, the vertices of the dual graph are considered as point masses located at the centroid of their corresponding initial element. The principal axis of inertia for these point masses is then calculated and the domain is bisected by making a cut which is orthogonal to this axis (with approximately equal weights on either side of it). This procedure is then repeated recursively for each subdomain.

This method is extremely fast, but like the RCB it also produces partitions with a relatively high cut-weight ([28]).

1.6.3 Recursive Graph Bisection (RGB)

Here the idea is to use the graph distance as opposed to Euclidean distance used in §1.6.1. Recall that the graph distance between the two nodes n_i and n_j is given by d(n_i, n_j) = number of edges in the shortest path connecting n_i and n_j.

Here the starting point is to find the diameter (or, since this is expensive to find, the pseudo-diameter) of the graph (see George and Liu ([37])) and then sort the nodes according to their distance from one of the extreme nodes. Half the vertices which are close to this extreme node are placed in one subdomain and the remaining half are placed in the other subdomain.

If we start out with a connected graph then by construction it is guaranteed that at least one of the two subdomains is connected. But it is still possible that the other subdomain may not be connected. Hence we may end up with a situation in which not all of the subdomains are connected.

1.6.4 Modified Recursive Graph Bisection (MRGB)

In [50] Hodgson and Jimack present their own graph bisection method MRGB. This method is a modification of the RGB method, which tries to improve on the original by attempting to produce subdomains which are all simply connected.

In MRGB each bisection begins by finding two approximately extremal nodes

of the graph and then builds a partition up around them by forming two sets. Each set first consists of those nodes which are at most one edge from one extremal node, then at most two edges, etc., until one partition contains half of all the nodes. The formation of the smaller partition is then continued until no more nodes can be claimed by it. If this partition also contains half of the nodes then the bisection is complete. Otherwise, there will remain unassigned nodes which are disconnected from the smaller partition. These nodes will be assigned to the larger partition thereby producing two connected subdomains which are not equal in their share of the nodes. If one insists on having balanced subdomains then some extra steps can be executed to transfer nodes from the larger subdomain to the smaller subdomain in such a way that the new improved subdomains are well balanced and that the increase in the cut-weight is not that large. Unfortunately due to the transfer step this method is still not guaranteed to produce simply connected subdomains but in practice (see [50] for details) the MRGB algorithm produces disconnected subdomains far less often than the RGB method and also produces subdomains which look more compact. Moreover the MRGB algorithm is computationally as cheap as the RGB method and nearly always produces partitioned subgraphs which have a smaller cost in terms of the cut-weight.

1.6.5 Recursive Spectral Bisection (RSB)

This method which was popularised by Pothen, Simon and Liou ([82]) is of a quite different nature to those above and is considerably less intuitive. It is in fact a continuous version of the following discrete optimal bisection problem.

With each node $n_i \in$ N we assign a weight x_i where x_i is +1 or -1 (where all nodes with $x_i = 1$ are in one subdomain and those with $x_i =$ -1 are in the other). Then the requirement of equal load among the two subdomains means $\sum_{i=1}^{N} x_i = 0$ and the requirement of minimal cut-weight demands that we should minimise the quadratic $\sum_{(v,w)\in E}(x_v - x_w)^2/\ 4$ (as the quadratic $\sum_{(v,w)\in E}(x_v - x_w)^2/\ 4$ is in fact the cut-weight).

Ignoring the factor of one quarter we note that $\sum_{(v,w)\in E}(x_v - x_w)^2 = \underline{x}^T L \underline{x}$, where L is the Laplacian matrix of the graph whose j^{th} entry of row i is given in Figure 1.6.

$$l_{ij} = \begin{cases} -1 & \text{if nodes i \& j are connected,} \\ \textit{degree of node } i & \text{if i = j,} \\ 0 & \text{otherwise.} \end{cases}$$

Figure 1.6: Entries of the Laplacian matrix.

Hence the discrete problem is:

minimise $\underline{x}^T L \underline{x}$ such that $\sum_i x_i = 0$ and $x_i = 1$ or -1.

But this is an NP-hard ([36]) problem, so the heuristic approach is to solve the following continuous version of above discrete problem:

minimise $\underline{x}^T L \underline{x}$ such that $\sum_i x_i = 0$ and $\| \underline{x} \|_2 = 1$.

Once we have the solution $\underline{x}^*$ of this continuous problem, the subgroups are made by sorting the |N| entries of $\underline{x}^*$ and placing nodes represented by x_i': $i = 1, \ldots, |\mathrm{N}|/2$ in one subgraph (with $\underline{x}'$ being the sorted vector) and those by x_i': $i = |\mathrm{N}|/2+1, \ldots, |\mathrm{N}|$ in the other (assuming, here that N is even).

It can be shown that ([50]) the vector $\underline{x}^*$ is in fact the second eigenvector of L, provided the graph is connected (i.e. it is that eigenvector of L which corresponds to the smallest positive eigenvalue). This is known as the Fiedler vector.

Experiments ([82]) has shown that RSB is an extremely good algorithm in terms of producing a small cut-weight. Unfortunately it is computationally expensive as it requires an eigenvector of a square matrix of size |N| which is often very large. Typically ([57]) a Lanczos algorithm is used to find the Fiedler vector but care is needed to ensure that the algorithm has genuinely converged before accepting the vector produced ([80]).

1.6.6 Recursive Node Cluster Bisection (RNCB)

In [50] Hodgson and Jimack present their own hybrid algorithm, which they call recursive node cluster bisection (RNCB), which attempts to combine features of the modified recursive graph bisection (MRGB) and recursive spectral bisection (RSB) algorithms. It relies on the concept of node clusters introduced by Walshaw and Berzins ([105]), who suggest that some connected elements of the mesh can be grouped together to form clusters (this idea will appear again in the multilevel algorithms outlined in §1.7.1). Such a cluster will have one corresponding node in

the dual graph but will have as many edges incident to it as there are elements adjacent to those elements forming the node cluster. The weight of this cluster's entry in the Laplacian matrix will be greater than for a single element. A partitioning algorithm which places the node cluster in a particular partition, places all of its corresponding elements in that partition.

The introduction of node clusters is an attempt to make the RSB method less expensive. The effect of creating node clusters is to lower the number of nodes within the dual graph and hence to decrease the size of the Laplacian matrix, so making the Lanczos method converge much faster. They report that for some cases RNCB with 33% clustering often produces better partitions than the spectral algorithm and at less cost. But for other meshes node clustering with 67% is better.

Using this approach, there is a major problem ensuring that the final decomposition is properly load balanced. See [50] which discusses a few recovery schemes which produces properly load balanced partitions.

The idea of clustering (also known as graph coarsening) introduced here has become a very popular method for reducing the computational cost of a partitioner. There are many methods based on this idea, some of them will be discussed shortly.

1.7 Multisectional Graph Partitioning Heuristics

There are two major drawbacks in bisection based methods.

- Lack of ability to decompose a given graph into an arbitrary number of subgraphs (as by construction the number of subgraphs produced is of the form 2^n).
- They do not attempt to produce a minimum cut-weight in the true global sense (since they only try to produce a small number of common edges on bisections of subgraphs at each recursive level, without paying any attention to the global scene).

To overcome these drawbacks many researchers have considered non-bisection, or multisection, techniques such as those we are now about to present. Here the basic idea is very simple, determine a "sort vector" to order all of the nodes in the graph and then split the graph into the desired number of subgraphs (rather than just two).

1.7.1 Multidimensional Spectral Graph Partitioning

In [45, 47] Hendrickson and Leland describe a multidimensional Spectral Load Balancing algorithm. Through a novel use of multiple eigenvectors, their algorithm can divide a computation into 4 (spectral quadrisection) or 8 (spectral octasection) pieces at once. These spectral partitions are further improved by a multidimensional generalisation of the Kernighan and Lin algorithm (see §1.8.2). They have shown that for some problems their multidimensional approach significantly outperforms spectral bisection.

1.7.2 Stripwise Methods

In this method one would sort the nodes in exactly the same manner as in RCB but then make the desired number of orthogonal cuts along the chosen axis to produce a specified number of equally sized subgraphs. As shown in [49] generating a strip-like decomposition of meshes is not generally advisable as this typically has an adverse affect on the scalability of the parallel solver.

1.8 Other Graph Partitioning Techniques

There are many other heuristics which are used by researchers in the field. We now describe some of the more popular.

1.8.1 Greedy Algorithm (GR)

Greedy algorithms have been around for decades. In [29] Farhat popularised their uses in the application area of finite element method (FEM). It is a greedy algorithm because it finds the first subdomain as well as it can without looking ahead. Once this is obtained it finds the next subdomain as best as it can. Hence the quality of the early subdomains is generally very good but if they are chosen too selfishly the quality of the later ones might be quite poor. Basically it is a graph based algorithm which uses the level-set principle of MRGB method to claim nodes in a walking tree fashion.

1.8.2 Kernighan and Lin Type Algorithms

In [65] Kernighan and Lin present a graph partitioning algorithm which is iterative in nature. It starts with an arbitrary partitioning of the graph. In each iteration tries to find a subset of vertices, from each part of the graph such that interchanging them leads to a partition with smaller edge-cut. If such subsets exist, then the interchange is performed and this becomes the partition for the next iteration. The algorithm continues by repeating the entire process. If it cannot find two such subsets, then the algorithm terminates, since the partition is at a local minima and no further improvement can be made by the algorithm. Unfortunately the complexity of the algorithm is nonlinear, as each iteration of the algorithm takes $O(\|E\|^2 \log \|E\|)$ time ([65]).

Several improvements to the original algorithm have been developed. One such algorithm is by Fiduccia and Mattheyses ([32]). Their algorithm fulfills the same purpose but its complexity is $O(\|E\|)$ which is linear. Nowadays, majority of the partitioning algorithms appear to include Kernighan and Lin type ideas as a post-processing step. The algorithm introduced in Chapter 3 also uses the philosophy of Fiduccia and Mattheyses in order to decrease the cut-weight.

1.8.3 State of the Art Software Tools for Graph Partitioning

During the last few years many public domain software tools have appeared for graph partitioning. We discuss a few of them. Others, not discussed here, include PARTY ([83]) and SCOTCH ([81]).

TOP/DOMDEC

In [30] Farhat *et al.* describe the basic features of TOP/DOMDEC (a Software Tool for Mesh Partitioning and Parallel Processing) and highlight their application of this tool in the parallel solution of computational fluid and solid mechanics problems. Basically in this software they have implemented the following algorithms.

- The Greedy algorithm (GR).
- The Reverse Cuthill-McKee algorithm (RCM) (see [18, 71]).

- The Recursive RCM algorithm (RRCM).
- The Principal Inertia algorithm (PI) (see [31]).
- The Recursive Principal Inertia algorithm (RPI).
- The Recursive Graph Bisection algorithm (RGB).
- The 1D Topology Frontal algorithm (1DTF) (see [103]).
- The Recursive Spectral Bisection algorithm (RSB).

This allow the users to select whichever algorithm they feel is the most appropriate for their applications.

Chaco

In [44] Hendrickson and Leland describe the capabilities and operation of Chaco. Chaco is a software package designed to partition graphs. Currently the following five classes of partitioning algorithms have been implemented in Chaco.

- Simple algorithms e.g. linear, random and scattered schemes.
- Spectral algorithms.
- Inertial algorithms (descriptions of these methods can be found in [75, 91]).
- Kernighan-Lin algorithms.
- Multilevel algorithms.

Their method of choice for large problems in which high quality partitions are sought is the multilevel algorithm. This algorithm is fully described in [46]. In this algorithm the original graph is approximated by a sequence of increasingly smaller weighted graphs. The coarsest graph is then partitioned using a spectral method, and this partition is propagated back through the hierarchy of graphs with load improvement at each level using, a variant of the Kernighan-Lin algorithm.

JOSTLE

In [106, 110, 111] Walshaw *et al.* outline the philosophy behind another method for solving this graph-partitioning problem. Their software tool, called JOSTLE, employs a combination of techniques including the Greedy algorithm to give an initial partitioning together with some powerful optimisation heuristics. The graph coarsening technique is additionally employed to speed up the whole processes. For time-dependent problems, unstructured mesh may be modified every few time-steps and so the load-balancing must have low cost relative to that of the solution algorithm in between the remeshing. Their algorithm tries to accomplish this task. Experiments on graphs with up to a million nodes indicate that the resulting code is up to an order of magnitude faster than existing state-of-the-art technique such as Multilevel Recursive Spectral Bisection, whilst providing partitions of equivalent quality. But JOSTLE is still much slower than METIS (another Software Package which is described below).

METIS

Recently Karypis and Kumar have released version 3 of their Software Package called METIS. This Package is designed for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. The algorithms implemented by METIS are also based on multilevel graph partitioning schemes, described in [60, 61, 62].

The underlying algorithm consists of three phases: coarsening, partition of the coarsest graph, and refinement. They claim that the partitions produced by METIS are consistently 10% to 50% better than those produced by spectral partitioning algorithms and 5% to 15% better than those produced by Chaco multilevel.

They also claim that it is extremely fast. Their experiments on a wide range of graphs have shown that METIS is one to two orders of magnitude faster than other widely used partitioning algorithms. They have found that graphs with over a million vertices can be partitioned into 256 parts in under 20 seconds on a Pentium Pro personal computer.

Chapter 2

Adaptivity and Dynamic Load Balancing

If one wishes to solve large computational fluid dynamics (CFD) or computational mechanics (CM) problems numerically using the FEM or FVM on a fixed unstructured mesh then the mesh should be dense enough to accurately reproduce the correct solution throughout the domain. Since, in many cases, the exact location of small-scale flow features such as shocks, vortices and wakes is not known in advance (and may vary with time), the mesh has to be fine everywhere (or at least in very large regions). This is not only very expensive computationally but also inefficient since it involves solving the problem for a great many unknowns which are not required in reality.

A remedy is to adapt the mesh in some way in order to maintain the quality of the solution (i.e. the solution error) whilst optimising the number of unknowns. Typically, for steady problems, this involves starting with an initial mesh (which we will call a coarse mesh) and solving the given problem on it. Based on some error criterion the mesh can now be adapted; i.e. it can be refined in that part of the region where the error indicator is high and possibly coarsened in that part of the region where the error indicator is very low. The process of solution followed by adaptivity can be repeated until a solution of the desired accuracy has been reached.

The above procedure is clearly only really valid for steady-state problems. For transient problem (where the small regions of the grid where there is a rapid spatial change in the solution (e.g. due to a shock etc.) may themselves change with time)

additional constraints have to be placed on the algorithms employed:

- the adaptive algorithm must be very fast as the adaptation is done very often (every 5-20 time steps for typical shock-interaction problems). For steady-state problems the speed is not a relevant issue as adaptation is performed only a few times (typically 3-5 times during the whole process),
- in the case where transient calculations are performed with an explicit time-marching scheme, the allowable time step will be governed by the smallest element in the mesh. Therefore, the minimum element size achieved by the adaptive refinement algorithm should not be too much smaller than the desired minimum element size. There is no such requirement for the steady-state problems as local time stepping may be employed.

In the next two sections we discuss some techniques for estimating the error locally and then introduce a few common methods for adapting a finite element mesh based upon this information. This will then be used to motivate the need for dynamic load balancing when parallel finite element solvers are being used (see §2.3). The rest of the chapter reviews some dynamic load balancing algorithms and software.

2.1 Spatial Error Indicators

As discussed above the use of adaptivity assumes that we can measure or estimate the error of a given numerical solution. Clearly this error is not known exactly. However we can attempt to approximate the error.

For the sake of simplicity we consider here the following simplified version of equation (1.1):

$$-\nabla^2 u(\underline{x}) = f(\underline{x}), \qquad \text{for } \underline{x} \in \Omega \subset \Re^2, \tag{2.1}$$

with u = 0 on $\partial\ \Omega$.
Then the corresponding weak form (equation (1.2)) simplifies to:
Find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \underline{\nabla} u . \underline{\nabla} w \ d\underline{x} = \int_\Omega w f \ d\underline{x}, \tag{2.2}$$

for all w $\in H_0^1(\Omega)$.

Let $u_h(\underline{x}) = \sum_{i=1}^{N} a_i P_i(\underline{x}), \in H_0^1(\Omega)$ be a piecewise polynomial (Lagrangian) approximation of the true solution u obtained by using the FEM on a given mesh. Also suppose that $\tilde{u}_h$ is the interpolant of u on this mesh, i.e. $\tilde{u}_h(\underline{x}) = \sum_{i=1}^{N} u(\underline{x}_i) P_i(\underline{x})$, where $\underline{x}_i$ is the position of the i^{th} node of the mesh. For the sake of simplicity we consider here the piecewise linear approximation only.

It is shown in [58] that the following inequality holds:

$$| u - u_h |_{H^1(\Omega)} \leq | u - \tilde{u}_h |_{H^1(\Omega)} \leq C [\sum_{e=1}^{E} (h_{\Omega_e} | u |_{H^2(\Omega_e)})^2]^{1/2}, \tag{2.3}$$

where $| v |_{H^r(\Omega)} = (\sum_{|\alpha|=r} \int_\Omega | D^\alpha v |^2 \ d\underline{x})^{1/2}$ (i.e. $| v |_{H^r(\Omega)}$ measures the $L_2(\Omega)$-norm of the partial derivatives of v of order exactly equal to r (hence it is a semi norm), and h_{Ω_e} is the longest side of the element e (which is called the diameter of the element e)).

In equation (2.3) $h_{\Omega_e} | u |_{H^2(\Omega_e)}$ is the contribution of the element Ω_e to the total error. So to keep the error small we must choose h_{Ω_e} small where $| u |_{H^2(\Omega_e)}$ is large. It is clear however that since u is unknown we can only try to approximate where $| u |_{H^2(\Omega_e)}$ is large by considering u_h instead of u. When u_h is piecewise linear the second derivatives of u are zero on each element hence we can estimate $| u |_{H^2(\Omega_e)}$ by considering the jumps in $\frac{\partial u_h}{\partial n}$ across each edge of Ω_e. This is an example of one of the simplest a posteriori estimates for the size of the error $| u - u_h |_{H^1(\Omega)}$. Many other, more complex and more general, algorithms have been proposed but these are outside the scope of this work (see [1, 7, 24, 25, 76, 96] for a number of examples however).

2.2 Different Types of Refinements

There are two popular classes of refinement algorithm that may be associated with unstructured meshes, namely regeneration schemes and mesh adaptation schemes. Below we describe these schemes briefly.

2.2.1 Regeneration Schemes

In this approach one starts with a uniform mesh over the entire domain. The problem is then solved on this mesh. If the error indicator is too large in any region then the mesh is discarded and a new mesh is generated over the domain which has

a non-uniform density (with most elements in the region where the error indicator was largest). The problem is then solved again and a new error estimate is again obtained. If the new indicator is satisfactory the problem is successfully solved. Otherwise we again discard the current mesh altogether and generate another new non-uniform mesh based on the latest error estimate and repeat the procedure. This method is recommended only if the procedure needs to be repeated just a few times (e.g. for steady state problems (see for example [49] and [99, Chapter 5])). It has the advantage of being straightforward to implement in parallel, provided one has a reliable parallel mesh generator. In some situations it is not necessary to discard the entire mesh, but only a portion of the mesh with a higher concentration of points in the high error regions and a lower concentration of points in the lower error regions may be regenerated. However in this case the newly generated portion of the mesh should smoothly paste together with the intact portion of the mesh.

In case it has to be repeated a large number of times (e.g. for time dependent problems), then due to the high computational cost associated with the generation of relatively refined meshes too many times, it is not beneficial and one may consider the second strategy described below.

2.2.2 Local Mesh Adaptation Schemes : Hierarchical Refinement

Let $\delta > 0$ be a given tolerance and suppose we want to obtain a FE approximation u_h such that,

$$|u - u_h|_{H^1(\Omega)} \leq \delta. \tag{2.4}$$

Relying on the error estimate (2.3) we see that (2.4) will be satisfied if the corresponding FE mesh Ω_e (e = 1, ..., E) is chosen so that

$$(h_{\Omega_e} |u|_{H^2(\Omega_e)})^2 \approx \frac{\delta^2}{EC^2}, \quad \textit{for all triangles } e. \tag{2.5}$$

There are several ways to determine a trial space satisfying (2.5), the three most common ones are briefly described here.

- Addition and deletion of points (h-refinement). In this case the mesh is refined by adding more points to the mesh in regions where the error indicator is large,

and removing points from regions where the indicator is low (for details see [8, 54, 69] for 2-d cases and [8, 11, 59, 93, 115] for 3-d problems).

- Movement of points (r-refinement). In this case no new points are added, only adjustment of the locations of existing points are made so as to ensure that the mesh density becomes higher where the error is large (e.g. [72, 74]).

- Order enrichment (p-refinement). The mesh remains the same, but we use changing degrees of interpolation polynomials, i.e. in elements where the error is large, higher order polynomial basis functions are used (e.g. [3, 119]).

It is also possible to combine these methods in a hybrid fashion to produce mixed schemes. For instance in [24, 25] Demkowicz *et al.* use hp-refinement in compressible flow problems and in [14] Capon uses hr-refinement for the compressible Navier-Stokes equations (see also [6, 15] for more hybrid applications). We now discuss the first method in detail as this method of refinement will be used in the examples in Chapter 5.

h-refinement

In this discussion, for simplicity, all the meshes are two dimensional and consist of triangles. We also continue to use the trivial problem given by the equation (2.1) by way of an example although the ideas generalise to more complex problems and corresponding generalised error estimates in a natural manner. Hence we proceed as follows: Choose a first mesh $\overline{\Omega}_e$ (e $= 1, \ldots, \overline{E}$) and compute a corresponding FE solution $\overline{u}_h$. Using $\overline{u}_h$ we can compute approximations to $(h_{\Omega_e} \mid u \mid_{H^2(\Omega_e)})^2$ denoted by $(h_{\overline{\Omega}_e} \mid \overline{u}_h \mid_{H^2(\overline{\Omega}_e)})^2$ for e $\in \{1, \ldots, \overline{E}\}$.

The quantity $(h_{\overline{\Omega}_e} \mid \overline{u}_h \mid_{H^2(\overline{\Omega}_e)})^2$ for e $\in \{1, \ldots, \overline{E}\}$ may be obtained using jumps in the normal derivatives across the edges as described in §2.1 (or more complex schemes such as in [7] for example).

We next construct a new mesh Ω_e (e $= 1, \ldots,$ E) by subdividing into four equal triangles each e $\in 1, \ldots, \overline{E}$ for which $(h_{\overline{\Omega}_e} \mid \overline{u}_h \mid_{H^2(\overline{\Omega}_e)})^2 > \frac{\delta^2}{\overline{E}C^2}$, where $\overline{E}$ is the number of triangles in $\overline{\Omega}_e$. Next compute the FE solution u_h on the new mesh Ω_e (e $= 1, \ldots,$ E) and repeat the process until,

$$(h_{\Omega_e} \mid u \mid_{H^2(\Omega_e)})^2 \approx \frac{\delta^2}{EC^2}, \quad \textit{for all triangles } e. \tag{2.6}$$

It is also possible to control the error in other norms than the one used in (2.4), for example we may want to use the maximum norm (for details see [58]). It may be pointed out here that the advantage of h-refinement is that relatively few mesh points need to be added or deleted at each refinement/coarsening step for time-dependent problems, but the disadvantages are the complicated logic and data structure that are required to keep track of the points that are added and removed. For details see [14, 112] (and references therein) for 2-d cases and [88, 93] for 3-d cases. In Chapter 4 we consider an example of mesh regeneration scheme, and h-refinement is considered in Chapter 5.

2.3 Relation Between Adaptivity and Dynamic Load Balancing

In the case of a uniprocessor machine there are no side affects to be dealt with when we make use of adaptive FEM or FVM to solve a PDE or a system of PDEs. On the other hand when we implement these adaptive methods on *parallel distributed-memory machines* (or, at least, when programming under this paradigm), then we immediately have the problem of load imbalance among the processors which are available on the parallel machine. This is due to the fact that the computational intensity is now both space and time dependent. Unless some corrective measures are taken, the current state of the load imbalance will significantly reduce the efficiency of the solver (as some processors will be idle whilst others are still doing some work). It is interesting to observe that this is a dynamic version of the static load balancing problem first encountered in §1.5 while we were discussing the parallel finite element method.

These corrective measures are known as *dynamic load-balancing algorithms*. A serial version of the algorithm is undesirable as it would carry a large communications overhead, become a serial bottleneck and would be constrained by the amount of memory available to a single processor. Hence a parallel load-balancing algorithm is required which is capable of modifying an existing partition in a distributed manner so as to improve the quality of the partition whilst keeping the amount of data relocation as small as possible (since there is a significant communication overhead associated with moving data between processors).

As described in §1.5 many heuristics have been devised to partition an initial unstructured mesh and hence minimise the load imbalance and interprocessor communication among processors. The redistribution of the refined mesh can also be achieved using some of these partitioning heuristics (with some modification) which we describe in the following section. We then describe and discuss in §§2.4–2.7 some dynamic load balancing algorithms specifically designed to regain the balanced load among the processors in parallel.

Before that we must discuss the ways in which we regain the balance. There are two possible ways of doing this. One way is to repartion the leaf mesh (i.e. the actual computational grid). This way it is possible to get a perfect load balance with a relatively low number of elements residing on the interpartition boundary. However the problem with this method is that we have to have complicated data structures to support this possibility (due to the hierarchical nature of the mesh). Also too much communication among the processors has to be performed (as we could end up with the parent and children elements being on different processors).

The other possibility is to repartition the initial coarse mesh (with any child elements being located accordingly (we are focusing here on h-refinement)). In this case one may not be able to get a perfect load balance among the processors if some elements are heavily refined and also the number of elements residing on the interpartition boundary may be little larger. However the relative advantage of this method over the previous one is that the parent and child elements are on the same processor and hence further adaptivity is easy to achieve (without too much communication). In this thesis (see Chapters 4 and 5) we always consider the second possibility, i.e. repartition the initial coarse mesh.

The task of repartitioning an adapted mesh can be converted into a graph partitioning problem by the introduction of a weighted dual graph. For each element, i, of the coarse mesh define a vertex of a dual graph and let this vertex have weight w_i, where w_i is the number of elements of the leaf mesh currently contained inside i. For each pair of face adjacent elements in the initial mesh define an edge, j, of the dual graph and let this edge have weight e_j, where e_j is the number of pairs of elements in the actual mesh which currently meet along edge j. At the end of each adaptive step the weights of all the vertices and edges of the dual graph must be updated. The amount of imbalance among the processors must be calculated. If this amount is more than a preset tolerance then a dynamic load balancing tech-

nique must be applied to regain the balance. It may be helpful to keep in mind that such a technique ideally should fulfill the following 4 objectives:

1. production of a load-balanced partition,
2. a minimal number of elements ending up on the partition boundary,
3. a minimal amount of data migration,
4. the possibility of an efficient parallel implementation.

It is clear that these four requirements are not always self-consistent. For example, an existing partition could be so far from optimal that there is no way that a near-optimal partition (in terms of 1 and 2) can be reached by only migrating a small proportion of the vertex weights. It is perhaps for this reason that quite a large number of dynamic load-balancing heuristics have been suggested in recent years ([23, 53, 104, 107] for example), each of which appear to put a slightly different emphasis on the relative importance of the four properties. The algorithm described in next chapter explicitly attempts to respect *all* of these requirements; however when conflicts do arise it is items 2 and 4, which relate more to the parallel overhead than the partition quality, which are the first to be relaxed. The motivation behind this is our decision that, when one is forced to choose between the two, robustness is more important than parallel efficiency in a parallel dynamic load-balancing algorithm. In the rest of this chapter we first discuss the possible generalisations of some of the static partitioning algorithms of §1.6 and then some dynamic load balancing algorithms specifically proposed for regaining the balanced load in parallel.

2.3.1 Generalisations of Static Algorithms

In this subsection we discuss the possibility of using the static partitioning algorithms referenced in §1.6 for the purpose of dynamically balancing the unbalanced load among the available processors which may have materialised as a consequence of the adaptivity. Our discussion is based on the presentation of Hendrickson ([43]) and the review paper of Jimack ([55]).

Recursive Coordinate Bisection (RCB)

Let us recall from §1.6.1 that in this method we first determine the coordinate direction of the longest expansion of the domain. Suppose that this is the x-direction. Then all nodes are sorted with respect to their x-coordinate. Half of the nodes with small x-coordinate are assigned to one subdomain, the remaining half are assigned to the other subdomain. The method is then repeated in a recursive manner until the desired number of subdomains are obtained.

The recursive nature of the algorithm makes it possible to run in parallel: at each level of the recursion, more and more subdomains may be bisected simultaneously. So the method is a potential candidate from a dynamic load balancing point of view. Also in case of local refinement there is a possibility of low migration as well. However the method is not used in principal due to the apparent inability to use the connectivity information given by the graph (hence the resulting partition may have a very high cut-weight (as defined in §1.6 the cut-weight means the number of fine edges on the inter-partition boundary)).

Recursive Inertial Bisection (RIB)

This method is first described in §1.6.2 and can also be used as a dynamic load balancer due to the following facts:

- the recursive nature of the algorithm leads to a straight forward parallel implementation,
- calculation of the principal axis of inertia for a given subdomain in parallel.

Also the data locality is generally preserved in this method as the principal axes will only change gradually, provided the mesh steadily refines and derefines.

Recursive Spectral Bisection (RSB)

The simplest static load balancing form of this algorithm was first discussed in §1.6.5. The weighted version of RSB is given in [47]. We now review this weighted version of RSB. In order to partition a weighted dual graph into two subdomains of equal size we start with a set of discreate variables x_i where each x_i corresponds to the i^{th} vertex of the dual graph whose weight is w_i. The only permissible values

of x_i are $\pm$ 1. The actual values of x_i are determined by solving the following optimisation problem:

$$\text{minimize } \frac{1}{4}\sum_j e_j(x_{j(1)} - x_{j(2)})^2 \ , \tag{2.7}$$

subject to,

$$\sum_i x_i w_i = 0 \ , \tag{2.8}$$

where $j(1)$ and $j(2)$ are the numbers of the two vertices at the ends of edge j. Once the optimisation vector of this problem is determined we form the two subdomains by placing all those vertices for which $x_i = +1$ in one subdomain whilst the remainder, for which $x_i = -1$, in the other.

It must be observed that the equation (2.8) guarantees that the two subdomains have equal weights while the minimisation criteria itself ensures that the total weights of all the edges on the boundary of the two subdomains is minimum. Just like the original simple version of the RSB, this weighted version is also NP-hard ([22, 36, 68]). So once again we are forced to look for heuristic to solve the above optimisation problem. In this heuristic version we allow x_i to be any real number.

As explained in [51], the partitioning vector $\underline{x} = (x_1, x_2, \ldots, x_m)^T$ is given by $x_i = u_{2_i}/\sqrt{w_i}$, where $\underline{u}_2$ is the Fiedler vector of the matrix $S = D^T L D$, L is the weighted Laplacian matrix of the dual graph and $D = \text{diag}\left(\frac{1}{\sqrt{w_i}}\right)$.

The subdomains are defined by sorting the m vertices of the dual graph according to the size of their entry in $\underline{x}$ and placing elements represented by $x_i^{'}$: $i = 1 \ldots n$ in one group (with $\underline{x}^{'}$ being the sorted vector) and those by $x_i^{'}$: $i = n+1, \ldots, m$ in the other, with n chosen so that

$$\left| \sum_{i=1}^{n} w_i^{'} - \sum_{i=n+1}^{m} w_i^{'} \right|$$

is as small as possible (where $w_i^{'}$ is the weight of the vertex represented by $x_i^{'}$).

Unfortunately the current form is not suitable for the dynamic load balancing point of view, since the new partition produced by the method has no correspondence with the existing partition. So a large amount of data may have to be migrated whenever this algorithm is used to repartition the current mesh.

In [101] Van Driessche and Roose present the modified version of spectral algorithm which is better suited for dynamic application by introducing some additional vertices and edges to the weighted dual graph. For each existing subgraphs they create a new vertex and join it with all the existing vertices in the subgraphs. It may be observed that none of these new edges are cut by the current partition of this extended graph. If we can repartition this graph in a way which dissallow the migration of newly generated vertices then it follows that any new edges which are now cut by the partition must correspond to the migration of a vertex of the dual graph from one subdomain to another. As the spectral algorithm performs well in terms of keeping the number of cut edges quite low it is to be expected that, when applied to this extended graph, it will lead to a small amount of data migration. However the success of this approach depends quite heavily on the choice of weight that are assigned to the new edges. The initial results reported in [101] are quite encouraging however.

The use of RCB and RIB as a dynamic load balancer is not so popular due to their apparent inability to use the connectivity information of the graph, which yields the relative high cut-weight. Also the use of RSB is quite sensitive to the weights of the new edges. Due to these reasons what is really needed is a different kind of heuristic that operates locally by migrating elements between the mapped neighbouring partitions. During the past few years there have been many such heuristics which are devised and implemented by the researchers working in this field. Some of the more important will be discussed in the remainder of this chapter.

2.4 Diffusion Algorithms

Here we assume that h-refinement approach has been used and a hierarchy is maintained. In order to illustrate the simple idea behind diffusion algorithms it is convenient to introduce a weighted graph which, following Vidwans *et al.* ([104]), we call a Weighted Partition Communication Graph (WPCG). This represents the face adjacency of the $\|P\|$ processors being used (processors that share at least one edge of a root element with a given processor are said to be face adjacent to that processor). A WPCG is obtained by having one vertex for every processor and an edge between two vertices if and only if they are face adjacent to each other. The weight w_{N_i} of the i^{th} vertex is equal to the number of leaf-level elements of the

mesh which reside on the i^{th} processor and the weight $w_{E_{ij}}$ of the edge connecting the i^{th} and j^{th} processors is equal to the number of leaf-level edges which lie on the interpartition boundary between the two processors. Diffusion methods correspond closely to simple iterative methods for the solution of diffusion problems; indeed, the surplus load can be interpreted as diffusing through WPCG towards a steady balanced state.

2.4.1 Basic Diffusion Method

This iterative approach, which is described in [12] for example, is a very simple and intuitive parallel method for dynamic load balancing. Here for each vertex in the WPCG we transfer an amount of work to each of its neighbours which is proportional to the load difference between them. In general this approach will not provide a balanced solution immediately, so the process has to be iterated a number of times until the load difference between any two processors is smaller than a specified value. In effect this method diffuses the load gradually amongst neighbours. If we denote by l_i the load of the processor p_i then the above basic diffusion method can be described algorithmically by the procedure given in Figure 2.1.

The main advantage of this method is that it only needs communications among neighbours (which may also be asynchronous). The main disadvantage is that the convergence can be slow (in the worst case the number of iterations needed to reach a given tolerance is $O(\|P\|^2)$ where $\|P\|$ is the total number of processors ([52])) and the method is neither able to detect a global imbalance nor able to remedy it (see [52] for an example). It may also be noted that a processor p_i essentially acts simultaneously on all its interprocessor communications channels. Even though a machine may have parallel hardware for communication, the communication will often have to be serialised with respect to an individual processor.

In order to avoid these shortcomings we consider another diffusion method, to be called the multi-level diffusion method ([52]).

2.4.2 A Multi-Level Diffusion Method

This is basically a divide-and-conquer type of approach. Let P be the WPCG (see §2.4 for the definition of WPCG) at a given stage and $\|P\|$ be number of processors

```
begin
    while (not converged) do
        for all processors p_i do
            for all N_i neighbours p_j of p_i do
                if l_i > l_j
                    transfer ⌊(l_i − l_j)/2⌋ load from p_i to p_j
            end for
        end for
    end while
end.
```

Figure 2.1: Diffusion method.

in the set P at that stage. The change in computational load on processor p_i is denoted by l_i. The sum of the load increments l_i of all subproblems p_i in the subset P_j of P is denoted by L_j. The procedure balance shown in Figure 2.2 achieves the desired load balance. It is important to note that the bisection step in Figure 2.2 means the following:

- $P_1 \cap P_2 = \emptyset$,
- $P_1 \cup P_2 = \mathrm{P}$,
- $\mid \|P_1\| - \|P_2\| \mid \leq 1$.

It is also important to note that no assumptions on the processor topology are made by the algorithm. Hence the user has the freedom to orient the bisection of the processor sets towards his/her processor topology if this is appropriate. It can easily be seen that the average case time complexity of this algorithm is $\mathrm{O}(\log \|P\|)$. The principle drawback of this algorithm is that it is not always possible to bisect a connected graph into two connected subgraphs. Also the condition $\mid \|P_1\| - \|P_2\| \mid \leq 1$ is too restrictive in the sense that relaxing this condition may improve the quality of the load balancer.

As a matter of fact the dynamic load balancing algorithm presented in forthcoming chapters relaxes this condition in addition to choosing the sorted version of

```
begin balance(P)
    if ‖P‖ = 1 then return
    bisect P into P1 and P2
    calculate L1 and L2
    transfer ⌊(L2‖P1‖ − L1‖P2‖)/(‖P1‖ + ‖P2‖)⌋ load from P2 to P1
    balance (P1)
    balance (P2)
end balance.
```

Figure 2.2: Multi-level diffusion method.

the Fiedler vector for the purpose of bisections.

2.4.3 Dimension Exchange Method

In [23] Cybenko shows that the basic diffusion algorithm is very slow to converge and therefore proposes an alternative version of the algorithm known as the dimension exchange method. This method is designed specifically with a hypercube architecture in mind.

Let us first define the edge-colouring of a graph G = (V,E). By this we mean that the edges of G are coloured with some minimum number of colours (say k) such that no two adjoining edges are of the same colour. A *dimension* is then defined to be the collection of all edges of the same colour. Let us assume that we have an edge-colouring of the WPCG. Then the dimension exchange method can be described in terms of the procedure shown in Figure 2.3.

Xu and Lau (see [117, 118]) have generalised the dimension exchange method by introducing an exchange parameter and called the new method the generalised dimension exchange method. In their paper they have also analyzed its properties and potential efficiency.

Unfortunately all of the above mentioned algorithms do not take into account one important factor, namely that the data movement resulting from the load balancing schedule should be kept to a minimum. Also no information is given about *which* elements should be transferred from one processor to another: one only cal-

```
Procedure for processor i ( 0 ≤ i < ||P||)

begin
    while (not Terminate)
        for( c = 1; c ≤ k; c++)
            if there is an incident edge coloured c
                load balance the two connected processors
            end if
        end for
    end while
end procedure.
```

Figure 2.3: Dimension exchange method.

culates the total weight to be transferred.

2.5 Minimising Data Migration

The basis of the algorithm due to Hu and Blake ([53]) is the minimisation of data migration. Let G = (V,E) be the WPCG of the problem. Also let ℓ_i be the total load on each processor (i.e. the weight of vertex i for $i = 1, \ldots, P$) and $\bar{\ell}$ the average load per processor. Suppose that δ_{ij} is the amount of the load that might be shifted from processor i to processor j (where the corresponding vertices of the WPCG are connected by an edge, (i, j)) in order to regain the load balance across the processors. The following equations must clearly be satisfied:

$$\sum_{(i,j)\in E} \delta_{ij} = \ell_i - \bar{\ell} \quad \text{for } i = 1, \ldots, \|P\| - 1, \tag{2.9}$$

where E is the set of all edges in the WPCG. (Note that if equations (2.9) are satisfied then it must also follow that the same equation also holds for $i = \|P\|$.) The variables δ_{ij} are directional, that is,

$$\delta_{ij} = -\delta_{ji}, \tag{2.10}$$

representing the fact that if processor i is to send the amount δ_{ij} to processor j, then processor j is to receive the same amount (to send -δ_{ij}). Because of (2.10) we

$$A_{ij} = \begin{cases} 1 & \text{if vertex i is the head of edge j,} \\ -1 & \text{if vertex i is the tail of edge j,} \\ 0 & \text{otherwise.} \end{cases}$$

Figure 2.4: The matrix A.

treat δ_{ij} as a variable only if $i < j$, for $i > j$ we merely replace δ_{ij} by -δ_{ji}.

Also note that in (2.9) we only have $\|P\|$ - 1 independent equations for a total of $\mid E \mid$ variables. In general $\mid E \mid$ is much larger than $\|P\|$ - 1, so the system (2.9) has infinitely many solutions. Hu and Blake decided to choose among these solutions one that minimises the data movement (it is important to note that there is always at least one solution, even if WPCG is a linear array : $\bullet---\bullet---\bullet \ldots \bullet---\bullet$, in which case $\mid E \mid = \|P\|$ - 1 so the minimisation problem does have a solution). If one writes the system (2.9) in matrix form:

$$\text{A } \underline{x} = \underline{b},$$

where A is the $\|P\|-1 \times \mid E \mid$ matrix associated with (2.9) and is given in Figure 2.4, and $\underline{x}$ and $\underline{b}$ are the vector of unknowns and the right-hand side of (2.9) respectively. Then the aforesaid minimisation problem can therefore be written mathematically as:

$$\begin{gathered} \text{minimise } 1/2\ \underline{x}^T \underline{x} \\ \text{subject to A } \underline{x} = \underline{b}. \end{gathered}$$

It is important to observe that in the above the minimisation is taken over the L_2 norm rather than L_1 or L_∞, as the computation over L_2 is easier as compared to the computations over other two norms.

As shown in [53] the above is equivalent to solving

$$\text{L } \underline{\lambda} = \underline{b},$$

where L is the Laplacian matrix of the WPCG and $\underline{\lambda}$ is the vector of Lagrange multipliers. Once we know the solution vector $\underline{\lambda}$ then $\delta_{ij} = \lambda_i - \lambda_j$.

The idea of minimising data movement is a promising one. And the above algorithm does accomplish this task. But the method still does not pinpoint which

elements to migrate. Also there is no control over the size of the interpartition boundary (as no provision is made in the algorithm for keeping the boundary to a minimum). A number of researchers have incorporated this idea into their own work. In [107] Walshaw *et al.* use the above method in their own quite sophisticated algorithm (see §2.6.2). Preliminary results reported in [107] are encouraging in the sense that the quality of the new partition is good and there is a relatively small amount of data migration.

2.6 Two Parallel Multilevel Algorithms

In recent years there have been a number of new multilevel algorithms which are designed for dynamic load balancing purposes. Of these, two such techniques called ParMETIS ([63]) and ParJOSTLE ([109]) have been implemented in parallel and released into the public domain (it may be noted that this work was all published after the start of the research undertaken in this thesis). The general idea behind these two algorithms (and also other multilevel algorithms) is to produce a hierarchy of coarsenings of the original weighted graph (where each level in the hierarchy is produced by merging together groups of neighbouring vertices of the graph at the previous level), and then to perform a *global* partition only for the coarsest graph. This partition is then projected onto the graph at the previous level and then modified using a *local* algorithm (such as [32, 65, 73]) in order to improve the partition quality. This step of projection onto the previous level followed by local improvement is repeated until the original graph has been recovered, when the algorithm terminates. It is possible to make quite an expensive choice for the global partitioner (a spectral algorithm for example [47]) since it is only applied once and to the coarsest graph. Moreover, the technique may either be applied to find a bisection of the original graph and then be repeated recursively on each subgraph, or it may be applied to find a k-way partition directly.

2.6.1 ParMETIS

In a series of papers ([61, 62, 63, 64, 85, 86]) Karypis *et al.* describe and justify the underlying theory of their software called METIS and ParMETIS. The algorithms in ParMETIS ([63]) are based on the multilevel partitioning and fill-reducing or-

dering algorithms which are implemented in the serial package called METIS (see §1.8.3). However, ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel computations and large scale numerical simulations.

In this parallel version the starting point is that an arbitrary weighted graph is already partitioned into k subgraphs (with an unacceptable load imbalance). The coarsening algorithm only permits vertices in the same subgraph to be merged together at each level. The coarsest graph in the hierarchy can then be repartitioned quite cheaply (since it is small) before the refinement stages begin. At each of these refinement stages the local transfer of some vertices is allowed to take place between subgroups so as to permit local improvements to the partition based upon the use of a greedy heuristic.

They claim that it can quickly compute high quality repartitions of adaptively refined meshes which optimise both the number of vertices that are moved as well as the edge-cut of the resulting partition. It may not always be possible however to produce a partition which satisfies a given tolerance as regard to load balance when working with non-uniformly weighted graphs (see Chapter 5).

Also, as discussed in [55], there are numerous difficulties associated with keeping the amount of interprocessor communication under control at the local improvement stages. In addition extensive experimentation is needed to establish the optimal degree of coarsening and the choice of partitioning algorithm for the coarsest graph.

2.6.2 ParJOSTLE

In [109] Walshaw *et al.* describe their own parallel multilevel method for the dynamic partitioning of graphs. Their method introduces a new iterative optimisation technique, called relative gain optimisation which both balances the workload and attempts to minimise the interprocessor communication overhead. They report in [109] that the application of the algorithm to graphs corresponding to a number of adaptively refined meshes leads to partitions of an equivalent, or higher quality, to those produced by static partitioners (which of course do not start from the existing partition) and much more quickly. They also point out that the algorithm results in only a small fraction of the amount of data migration compared to the static partitioners.

2.7 Two Further Paradigms

We complete this chapter with a discussion of two more algorithms concerning the application of dynamic load-balancing to adaptive solvers. These have been selected because they both relate to the use of mesh adaptivity for the parallel solution of time-dependent problems in three space dimensions and are hence of great practical interest.

2.7.1 Algorithm of Oliker & Biswas

In [77] Oliker and Biswas present an important method which dynamically minimises the amount of load imbalance which arises due to the adaptive nature of a particular solver used to solve a given class of CFD problems on parallel machines. This appears to be the very first complete algorithm which can accomplish all of the typical phases associated with such an adaptive approach; mesh adaptation, repartitioning, processor assignments, and remapping are all done rapidly and efficiently so as not to cause a significant overhead to the numerical simulation.

For the repartitioning stage they use ParMETIS ([77]). As far as the processor assignment stage is concerned they make use of two cost functions: TotalV and MaxV. TotalV minimises the total volume of data moved along all processors, while MaxV minimises the maximum flow of data to or from any single processor.

They finally execute a remapping phase which is responsible for physically moving data when it is reassigned to a different processor. This remapping phase is further divided into two sub-phases: marking and subdivision. In the marking stage the edges are simply marked for bisection (based on an error indicator). Once the marking stage is complete, the weight of the dual graph can be adjusted and based on the new weights the load balancer may proceed in generating a new partitioning. The newly redistributed mesh is then subdivided (and subsequently refined) based on the marking patterns. Since the actual refinment is performed only after the subdivision stage it is believed that a relatively small amount of data must be moved.

2.7.2 Algorithm of Vidwans *et al.*

In [104] Vidwans *et al.* present their own divide-and-conquer based algorithm designed to solve three-dimensional Navier-Stokes problems on unstructured meshes in parallel using adaptive techniques.

The initial computational domain is partitioned among the available processors (which they assume is a power of two) by a partitioning algorithm based on the orthogonal recursive bisection method. At some later stage of the solution process when there is a load imbalance due to adaptivity they split the processors into two equal groups based upon their IDs. The group with the higher load is termed as the *sender group* whilst the other is known as the *receiver group*. After that half of the difference of loads is transferred from the *sender group* to that of the *receiver group*. The recursion is continue in the sense that each of the two groups at a given stage are further divided into two subgroups. The recursion is terminated when the size of a particular group becomes 2, in which case the processors simply balance the sum of their individual loads by exchanging the loads across their common boundary. The algorithm is deterministic in the sense that recursion will terminate after log $\|P\|$ steps, with $\|P\|$ being the number of processors, irrespective of the amount of imbalance and its distribution across the processors.

As far as the migration of nodes (also known as cells) from the *sender group* to the *receiver group* are concerned, they employed two different approaches.

- The grid-connectivity-based approach. In this approach they start with those cells which have at least one face on the initial interprocessors boundary. After that they select those cells which are neighbours of the cells selected in the previous stage. The process is repeated until they have the desired number of cells to be migrated. This is obviously done in "layers" starting from the outermost layers of cells. In adapted regions a large number of layers relatively occupy small physical space. On the other hand in the dense region of the mesh small number of layers occupy the large physical space. So this method may leads to jagged and long boundaries.

- The coordinate-based approach. In this approach all those cells which have their centroids within a particular region in 3-d space are marked for migration. This region is defined to be adjoining the interprocessors boundary. As

the number of cells in the region is not known a priori, the width of the region has to be determined by trial-and-error approach so that it contains the required number of cells. They say that this method is better than the grid-connectivity-based approach. This trial and error approach requires manual intervention not only for different meshes, but also for a given mesh the manual intervention is required for ever changing boundaries of the partitions. So a dynamic load balancing algorithm based on coordinate-based approach can never be a robust one and also the trial and error nature of it will make the algorithm less efficient.

Apart from improving the selection criteria for migrating cells, there are two other steps in the algorithm which can be improved.

- The restriction that at each stage the two subdomains should have same number of processors may reduce the quality of the mesh-partition especially if the mesh is non-uniform in nature. If this restriction can be relaxed in a meaningful way than this may improve the quality of the mesh-partition.

- The division of a group into two groups which is based simply on the IDs of the processors in the original group may also reduce the quality of the load-balancer especially if this produces groups in which some of the processors are physically unrelated from the other processors in the group.

The new dynamic load balancing algorithm presented in the next chapter uses this philosophy of Vidwans *et al.* together with a significant number of improvements. There, the two subgroups are not required to have the same number of processors and the division of a given group into two groups is not simply based on the IDs of the processors present in the group. Instead a sorted version of the Fiedler vector is used for the purpose of bisection. Also a gain-based approach (see Chapter 3) is used as the basis for the purpose of migrating cells.

Experimentation shows that these three steps improve the quality of the new partitioner, especially in the case where the underlying mesh is of a non-uniform nature (i.e. in some region of the mesh the elements are much finer and in other parts of the region they are relatively coarse).

Chapter 3

A New Dynamic Load Balancer

As discussed in §1.4 an efficient and effective way to solve a large transient problem numerically is to use the FEM (or finite volume method) on an adapting unstructured mesh. Frequently the size of the problem is so large that it can also be advantageous to solve such a problem on a parallel machine. Quite often, in the case where a shock, or similar solution feature, is moving from one part of the domain to another as time goes by (see §2.3), effective adaptivity will be achieved by refining the mesh ahead of the shock and coarsening the mesh behind it. This basically involves adding points to the existing grid in regions where some error indicator is high, and removing points from regions where it is low. On a parallel architecture, where the mesh has been partitioned in some way, this in turn leads to a problem of load-imbalance.

As mentioned in Chapter 2 there are various general purpose load balancing algorithms for minimising load imbalance of a partition. However, they do not always produce satisfactory results for the above adaptive situation. The diffusion algorithm is very good as far as local improvement is concerned but is extremely slow in terms of global improvement. The multilevel algorithms are designed to improve the global rate of convergence but, due to the coarsening step built into the algorithm, are not always able to produce a repartitioning of the mesh which is below a given tolerance for the load-imbalance. For this reason we claim that there is a need for some sort of hybrid algorithm which combines the best features of existing techniques so that the load-imbalance may be reduced to an acceptable level without increasing the cut-weight (the cut-weight is being defined as the sum of the weights of all those edges in the weighted dual graph (see §2.3 for the definition of the

weighted dual graph) which cross between two different subdomains) substantially. In what follows we present a serial version of such an algorithm (improved parallel versions are described in the next two chapters).

3.1 Motivation of the Algorithm

As mentioned in §2.7.2, the new dynamic load balancing algorithm whose serial version is being presented below, and whose parallel versions will be presented in the next two chapters, uses the philosophy of Vidwans *et al.* together with a significant number of modifications. Here, the two subgroups are not required to have the same number of processors and the division of a given group into two groups is not simply based on the IDs of the processors present in the group. Instead we use a sorted version of the Fiedler vector for the purpose of bisection. We also use the concept of gain density (see §3.2.2) as the basis for the purpose of migrating cells. In order to demonstrate the fact that these modifications do improve the algorithm we also implemented a serial version of the original parallel algorithm of Vidwans *et al.* ([104]) which uses the grid-connectivity-based approach.

Experimentation (see §3.5) shows that these modifications do improve the quality of the new partition, in the sense that the New algorithm consistently produces better results than those produced by the serial version of the Vidwans *et al.* algorithm as far as the parameters MaxImb and CutWt (see §3.5) are concerned.

3.2 Description of the Algorithm

Let us assume for the sake of simplicity that we are required to solve a given partial differential equation on a 2-d domain subject to given boundary and/or initial conditions. Also suppose that there is a 2-d mesh of triangles which covers this domain. As the problem is computationally extensive we would like to use a parallel solver. So the first step in this direction is to partition the mesh into a number of subdomains.

Let us suppose the parallel solver assumes that each subdomain is assigned to a single processor of a parallel machine and that all the processors are identical (i.e. we do not consider here, the possibility of a parallel machine consisting of heterogeneous components). We assume that this partition is slightly unbalanced

(but has a reasonably low cut weight) and our job is to balance it dynamically. We wish to do this without re-partitioning the entire mesh from scratch and so we look for a more local approach. This method is described in what follows.

3.2.1 Group Balancing

In this chapter we are interested in those cases where there is a root (also called coarse) and a computational (also called fine) mesh where the latter is a refinement of the former. Suppose there are P processors involved and we have a coarse mesh which is already distributed across these processors (or the mesh may itself be generated on these processors in parallel). We define the weight of a coarse element to be the number of fine elements inside the coarse element and the weight of a coarse edge to be the number of fine mesh edges along the coarse element edge. From now onwards, by a node we shall mean a node of the weighted dual graph (see §2.3) of the coarse mesh (i.e. a coarse element).

Let us recall from §2.4 the definition of the Weighted Partition Communication Graph (WPCG) which represents the face adjacency of the partitions in the system (processors that share at least one edge of a coarse element with a given processor are said to be face adjacent to that processor): A WPCG is obtained by having one vertex for every subdomain in the partition and an edge between two vertices if and only if they are face adjacent to each other. The weight w_{N_i} of the i^{th} vertex is equal to the sum of weights of all coarse elements on the i^{th} processor (which is the same as the total number of "fine" elements on the i^{th} processor) and the weight $w_{E_{ij}}$ of the edge connecting the i^{th} and j^{th} processors is equal to the sum of weights of all coarse element edges on the interpartition boundary between the two processors.

We first divide the WPCG into two groups either by using processor IDs as in [104] or by using some other bisection method. In fact, we choose to use a weighted version of the method of spectral bisection (see [47]). This method is often considered to be computationally expensive, however it does perform better than most algorithms in minimising the cut-weight. Also the computational expense is not problematic for the WPCG as the number of processors in the system is always assumed to be small compared to the number of coarse elements in the mesh.

As explained in [51] and also in Chapter 2, the partitioning vector $\underline{x}$ is given by

$x_i = u_{2_i}/\sqrt{w_{N_i}}$, where $\underline{u_2}$ is the Fiedler vector of the matrix $S = D^T L D$, L is the weighted Laplacian matrix of the WPCG and $D = \text{diag}\left(\frac{1}{\sqrt{w_{N_i}}}\right)$.

The groups are defined by sorting the P vertices of the WPCG according to the size of their entry in $\underline{x}$ and placing elements represented by x_i': $i = 1 \ldots n$ in one group (with $\underline{x}'$ being the sorted vector) and those by x_i': $i = n+1, \ldots, P$ in the other, with n chosen so that

$$\left| \sum_{i=1}^{n} w'_{N_i} - \sum_{i=n+1}^{P} w'_{N_i} \right|$$

is as small as possible (where w'_{N_i} is the weight of the vertex represented by x_i'). These groups are called Group1 and Group2 respectively. Ideally we would like each group to contain the same number of processors and an equal total weight, but this may not be possible due to large variations in the weights when a mesh is locally refined and the fact that P need not be even. However the cut weight resulting from this bisection is generally relatively small. The group with the higher *average* load per processor is termed as the *larger* group and the other one is called the *smaller* group. In the second stage of the algorithm we try to use the idea of local migration from the "larger" to the "smaller" group so that after migration each group contains approximately the same *average* weight per processor without there being a significant increase in the cut-weight.

3.2.2 Local Migration

As mentioned above the groups formed in the last section may not be ideally balanced. To balance them we now migrate nodes from the "larger" to the "smaller" group. There are many ways to do this. Due to the non-linear complexity of the Kernighan and Lin algorithm ([65]) we decided to apply the ideas of Fiduccia and Mattheyses ([32]) who have suggested an algorithm for the same purpose but whose complexity is linear.

For obvious reason the "larger" group is called the Sender and the "smaller" group is called Receiver group respectively. The quantity Mig_{tot} stands for total weights of all the nodes which are to be migrated from the Sender to the Receiver group in order to leave them perfectly balanced.

Let N_1 and N_2 be the number of processors in Group1 and Group2 respectively. Also let Ave be the average weight per processor in the WPCG and Ave_1 & Ave_2

```
if(Ave1 ≤ Ave2){
        Sender = Group2;
        Receiver = Group1;
        Mig_tot = N2 * (Ave2 − Ave);
}
else{
        Sender = Group1;
        Receiver = Group2;
        Mig_tot = N1 * (Ave1 − Ave);
}
```

Figure 3.1: Calculation of Sender, Receiver and Mig_{tot}.

are the average weights per processor in Group1 and Group2 respectively. The calculation of Sender, Receiver and Mig_{tot} is shown in Figure 3.1 below.

Note that if we transfer a set of nodes from Sender to Receiver whose combined weight is nearly or exactly equal to Mig_{tot} then after the transferring process the average weights of both the groups will be equal to that of global average Ave, i.e. the two groups will be load balanced. The next issue which must be addressed is that of how much load from which processors in the Sender group should be transferred to which processors in the Receiver group. There are many possible ways of tackling this. Our choice is closely related to that of Vidwans *et al.* ([104]). Following [104] we define the concept of candidate processors. Processors in each group that are face-adjacent to at least one processor in the other group are called candidate processors. We only allow the candidate processors to be involved in the actual migration of nodes from Sender to Receiver. Let N_{tot} be the total weights on all candidate processors of the Sender group. Then if the i^{th} candidate processor in the Sender group is face adjacent to more than one candidate processor in the Receiver group we migrate nodes to that candidate processor in the Receiver group which has the least weight. The amount of load shifted from i^{th} candidate processor in Sender group is denoted by Mig_i and is given by,

$$\text{Mig}_i = \left(\frac{N_i}{N_{tot}}\right) * \text{Mig}_{tot}, \tag{3.1}$$

where N_i is the total weight of the i^{th} candidate processor in the Sender group. Now that we know how much to transfer and where to transfer, all that remains to

$$\text{gain}(k) \;=\; \sum_{(k,l)} \begin{cases} w_{E_{kl}} & \text{if } l \in j^{th} \text{ processor,} \\ -w_{E_{kl}} & \text{if } l \in i^{th} \text{ processor,} \\ 0 & \text{otherwise.} \end{cases}$$

Figure 3.2: The calculation of gain.

be decided is which nodes of the weighted dual graph (i.e. coarse elements) should actually be transferred. This is naturally accomplished by aiming to transfer those nodes which result in a new cut weight which is as low as possible.

The fundamental idea behind the algorithm for transferring these nodes which minimise the cut-weight is the concept of the *gain* and *gain density* associated with moving a node onto a different processor. Define the *gain(k)* of node k to be the net reduction in the cost of cut edges that would result if node k were to migrate from i^{th} candidate processor in the Sender group to the j^{th} candidate processor in the Receiver group. The calculation of gain(k) is shown in Figure 3.2. It is important to observe that in calculating the gain of a node k the sum is taken over all edges which have node k at one end. The *gain density* of a node is defined as the gain of the node divided by the weight of the node. It may also be pointed out here that we also calculate the gain densities of all the nodes of the j^{th} processor in the Receiver group (by definition the gain of a node in j^{th} processor is the net reduction in the cost of cut edges that would result if the node were to migrate from j^{th} processor to the i^{th} processor) as these densities are required in the §3.3 below where we move nodes around between the i, j processor pair to further minimise the number of cut edges whilst retaining the load balance.

The bulk of the work needed to make a move consists of selecting the base node (a node which is about to be shifted from one processor to another processor is called a base node), moving it, and then updating the gain densities of its neighbouring nodes. We solve the first problem, that of selecting a base node, by choosing the node with the largest gain density on the i^{th} processor and whose weight is less than or equal to Mig_i. We shift the node to the j^{th} processor and update the gain densities of its neighbouring nodes (observe that in general the node k will have three neighbours when we have two-dimensional domains and four neighbours when we have three-dimensional domains) by the logic explained in Figure 3.3 below,

```
For each n_k which is a neighbour of the node k {
    Let p_k be the processor to which n_k belongs;
    if (p_k == j) then
        decrement gain(n_k) by 2*w_{E_{n_k k}};
        find the new gain density of the node n_k;
    else if(p_k == i) then
        increment gain(n_k) by 2*w_{E_{n_k k}};
        find the new gain density of the node n_k;
    else
        increment the edges cut between p_k and j by w_{E_{n_k k}};
        decrement the edges cut between p_k and i by w_{E_{n_k k}};
}
change the sign of gain(k);
```

Figure 3.3: Updation of gain densities and edges cut between the processors.

which also update the cut weights between the processors which are affected by the move. Observe that, if the gain density associated with the base node is positive, then making that move will not only make the groups closer to load-balance but it will also reduce the total cost of the edges cut in between the two processors involved, and hence between the processors groups. The above logic is repeated for all possible candidate processors in the Sender group.

Now we are in a position to present version one of our group balancing algorithm. The main three steps of this version of the algorithm are summarised in Figure 3.4.

3.3 Further Refinement of the Algorithm : Locally Improving the Partition Quality

Throughout this discussion our purpose is to load-balance the two groups, minimise the edge cut-weight between the two groups, and keep migration local and as small as possible. In order to further minimise the edge cut-weight between the two groups we apply a local refinement strategy to each processor pair, after the pair contains the desired weights. For this purpose we use ideas similar to those of Hendrickson

```
Until(there are no unprocessed candidate processors in Sender group){
   Until(required load is transferred from processor i to processor j){
      Calculate gain densities;
      Move a node of highest possible gain density from i^th processor
      to j^th processor;
      Update gain densities and other variables of interests;
   }
}
```

Figure 3.4: Initial version of load balancing of the two groups.

and Leland([46]). This section describes the approach in detail.

Once the desired load Mig_i has been moved from the i^{th} processor in the Sender group to the j^{th} processor in the Receiver group it may still be possible to move nodes around between the i, j processor pair to further minimise the number of cut edges whilst retaining the load balance. We start with the current state of the two processors and save this state as a best-state so far achieved. We now move nodes between processors i and j in such a way that the absolute value of the difference in the weights between the two remains below a certain tolerance (e.g. below some fraction of the maximum weight of a node on either processor or below a fixed percentage of the combined weight of the two processors involved). Each time we pick a node of highest gain density, but we also allow the gain densities to be negative for a while in an attempt to avoid local minima traps. The algorithm in Figure 3.5 fully describes this strategy. The algorithm consists of two nested loops. The inner loop presides over a sequence of moves of nodes from one processor to another. The outer loop continues allowing attempted sequences until no further improvement is detected. To avoid thrashing we insist that a node may be moved at most once within the inner loop.

When a node is moved, it is locked and the gain densities of all its neighbours are modified. Then another move is selected and the process is repeated. The best partition that is encountered in this sequence is recorded and the data is moved into that configuration prior to the start of the next inner loop. In practice, the number of times the outer loop is executed tends to be quite small. The termination criteria used in the inner loop is simply to run the loop for a specified number of times (e.g.

```
Improvement := True;
Calculate all initial gain densities;
While(Improvement){
        Improvement := False;
        Unlock all the nodes;
        Best Partition := Current Partition;
        While(Termination criteria is not reached){
                Select node to move;
                Perform the move and lock the node;
                Update the gain densities of the moved node and all its neighbours;
                If(Current Partition is better than Best Partition){
                        Best Partition := Current Partition;
                        Improvement := True;
                }
        }
        Current Partition := Best Partition;
}
```

Figure 3.5: An algorithm for refining the partitions between a pair of processors.

25 percent of the total number of nodes involved).

We end this section with Figure 3.6 which describes version two of the algorithm to balanced the two groups of processors in such a way that the cut weight between them is as small as possible.

3.4 Global Load-Balancing Strategy: Divide and Conquer Approach

Once the algorithm of §3.3 has been applied to the two processors groups they will have approximately equal average weights. Hence, it is now possible to recursively apply the above technique to each of these two groups of processors in turn to bisect and then load balance them. This recursion will terminate when a group has less than three processors. In the case of a singleton group no action is required. For

```
Until(there are no unprocessed candidate processors in Sender group){
   Until(required load is transferred from processor i to processor j){
      Calculate gain densities;
      Move a node of highest possible gain density from i^th processor to
      j^th processor;
      Update gain densities and other variables of interest;
   }
   Apply the local refinement technique of the algorithm of Figure 3.5;
}
```

Figure 3.6: Group-balancing algorithm: version two of load balancing of the two groups.

a group consisting of two processors we simply divide the group into two singleton groups (the processor containing the larger weight is the Sender and the other processor forms the Receiver) and try to load balance them by using the algorithm of Figure 3.6. This divide and conquer algorithm is fully described in Figure 3.7 and discussed below.

To implement the above divide and conquer philosophy we make use of two stack data structures called Lower and Upper, controlled by a variable called Top. We also use an array called Index. At each stage of the recursion the entries of the Index between two given positions represent the identities of processors involved at that stage of the recursion in the order in which they appear in the sorted version of the last Fiedler vector. In fact these sets of processors will be divided into two groups for further load balancing. The two given positions are in fact the top entries of the two stacks Lower and Upper respectively and are called Left and Right respectively. We initially push 0 to Lower and P-1 to Upper (since we follow the convention of numbering the P processors from 0,1,2, ...,P-1). Also, initially the i^{th} entry of Index is equal to i, so at any stage of the algorithm the entries of Index are simply the re-arrangement of the integers 0,1, ...,P-1.

At each stage of the algorithm we pop one value from Lower and one value from Upper. Based upon these values we pick that group of processors from the array Index which lie in between the two given positions. We then form the weighted Laplacian matrix and calculate the corresponding Fiedler vector. Based upon the

sorted version of this Fiedler vector we divide the group into two as explained in §3.2.1. After that we load balance them as described in the algorithm of Figure 3.6. We also shuffle the corresponding entries of the Index array based on the sorted Fiedler vector. After this we push the starting positions of the two groups in the array Index on to the stack Lower and ending positions on to the stack Upper and update the variable Top. This logic is repeated until the two stacks become empty (i.e. we stop when Top becomes null).

3.5 Examples

In this section we present some numerical examples in which we compare new results with the results produced by the serial version of the Vidwans *et al.* (as mentioned in §3.1), Chaco ([48]) and JOSTLE Version-2 ([108]) algorithms (we are not able to use the METIS algorithm in this chapter as it does not provide the facility of improving a given partition serially (however, the parallel version of the METIS algorithm is tested against our parallel implementation in Chapter 5)). In the case of the Chaco algorithm we apply the Kernighan-Lin option after reading in the existing partition (and graph). The REFINE_PARTITION parameter in the Chaco algorithm controls number of sweeps which are made through the pairs of subgraphs with a non-zero common boundary in order to improve the partition once the Kernighan-Lin option is invoked. For our application we found that a value of 20 is reasonable.

As mentioned in §3.3, in our algorithm we move nodes between a pair of processors as long as the absolute value of the difference in the weights between them remains below a certain tolerance known as Tol. We take Tol as a percentage of the combined weight of the two processors involved. For the sake of uniformity we try to present the results for two different values of Tol. These values are respectively 1% and 2% of the combined weight of the two processors involved. In some particular cases we have also experimented with more than 2 values of the parameter Tol. The maximum imbalance in a group is denoted by MaxImb and is defined by,

$$\text{MaxImb} = ((Max - Ave) / Ave) * 100,$$

where Max represents the weight of the heaviest processor in the group and Ave is the average weight of a processor in the group (so MaxImb is the largest percentage by which the total weight on any single processor exceeds the average weight

```
Top = 0;
Lower[Top] = 0;
Upper[Top] = P - 1;
Increment Top;
Index[i] = i; for i = 0, 1, ..., P-1;
while(Top)
    Decrement Top;
    Left = Lower[Top];
    Right = Upper[Top];
    if(Left == Right) continue;
    if(Right == 1 + Left)
        weight1 = weight of processor Index[Right];
        weight2 = weight of processor Index[Left];
        if(weight1 ≤ weight2)
            Sender = Index[Left]; Receiver = Index[Right];
        else
            Sender = Index[Right]; Receiver = Index[Left];
        Apply the Group_Balancing algorithm of Figure 3.6;
        continue;
    Find the Weighted Laplacian L and Diagonal Matrix D of the group
        {Index[Left], ...,Index[Right]};
    Based on Sorted form of the scaled version of the Fiedler vector divide
        the group into two subgroups and update the array Index;
    Determine the Sender & Receiver group;
    Apply the Group_Balancing algorithm of Figure 3.6 to this pair of
        groups;
    Push the left side of Sender on Lower stack;
    Push the right side of Sender on Upper stack;
    Increment the variable Top;
    Push the left side of Receiver on Lower stack;
    Push the right side of Receiver on Upper stack;
    Increment the variable Top;
    continue.
```

Figure 3.7: A divide & conquer type dynamic load-balancing algorithm.

per processor). Also, CutWt stands for the cut-weight (let us recall from the beginning of the chapter that the cut-weight is defined as the sum of the weights of all those edges in the weighted dual graph which are cross between two different subdomains).

A group is only divided into two subgroups if the maximum imbalance MaxImb in the group is larger than a fixed percentage, say Tol2 (again unless stated otherwise Tol2 is taken to be fixed; at 3%). Finally, in the process of finding an optimal partitioning we allowed the swapping of at most 25% of the combined total number of nodes of the current i, j processor pair (as defined in §3.3) of the group involved.

Example 1. Here we consider a mesh of 161 coarse elements, for which the dual graph has 161 nodes and 217 edges (see Figure 3.8 for the corresponding coarse mesh). The element weights are assigned randomly in the range of 100 to 200 and the edge weights are also assigned randomly in the range of 10 to 150. Initially the dual graph is arbitrarily partitioned into eight subgraphs. Initial and final partitions produced by our algorithm and some associated features of these partitions are shown in Table 3.1. The total weights (i.e. the number of fine elements) of all the coarse elements is 23950, so the average weight per processor comes out to be 2994. Observe there is a difference in weights of 2070 between the processors with the maximum and minimum loads.

We also apply the Vidwans *et al.*, Chaco and JOSTLE algorithms to the initial partition and the results are summarised in Table 3.2. The initial value of MaxImb is 39.6% and of CutWt is 1730. It is important to observe that in all cases (excluding the Vidwans *et al.* algorithm) not only that we have a reduction in the MaxImb but we also have a reduction in the CutWt. The reduction in both of these parameters is not surprising as the initial partitioning was totally arbitrary. With Tol = 1% the New algorithm is able to reduce the MaxImb from 39.6% to 2.9% while with Tol = 2% MaxImb has been reduced only to 4.6%. The final values of Cut-weighs are 1380 for Tol = 1% and 1270 for Tol = 2%. When we use the Chaco algorithm the CutWt decreases to 1030 while the MaxImb reduces to 4.6%. When we apply the JOSTLE algorithm to the above initial partitioning the MaxImb is reduced to 8.2% only and the CutWt is reduced to only 1560. In the case of the Vidwans *et al.* algorithm the CutWt has increased by an amount of 400 (which is not surprising) while the MaxImb is reduced to 2.2%. In this case the Vidwans *et al.* algorithm produces a

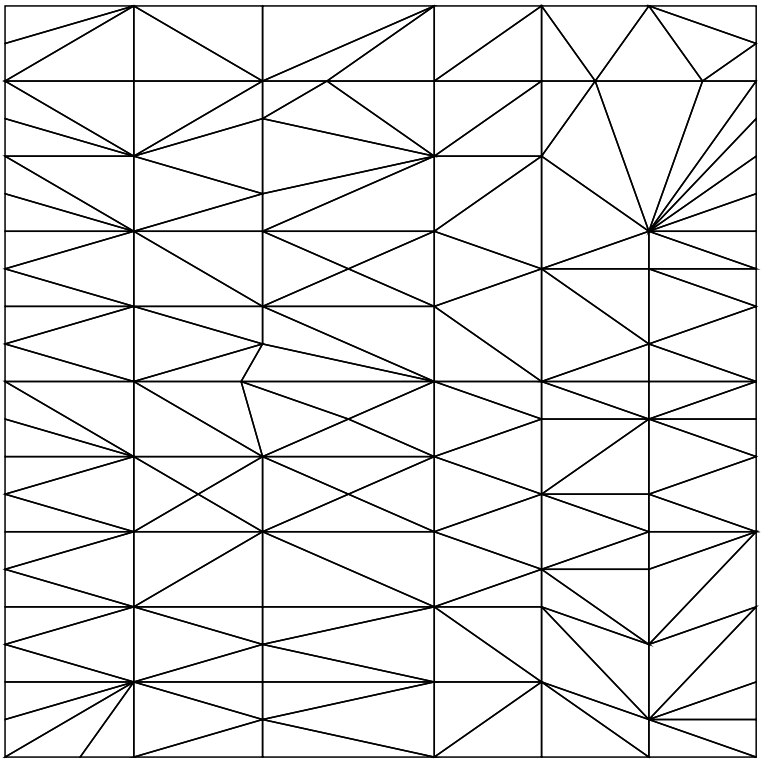

Figure 3.8: The coarse mesh of Example 1.

final partition with a least value of MaxImb whereas the Chaco algorithm produces a final partition with least amount of CutWt.

Example 2. Here we consider a mesh (see Figure 3.9 for the corresponding coarse mesh) which is generated by the method of Hodgson and Jimack ([51]) in parallel using 8 processors (Domain 1, Test 4 ("L-Shaped" geometry)). The initial and final situations are shown in Table 3.3. The total vertex weight is 836183, so that the average load of a processor is 104523. The initial value of MaxImb is 7.8% and that of CutWt is 3854. We apply our New algorithm to this moderately unbalanced partition for three different values of tolerance (Tol) namely 1% , 2%, and 4% of the combined weight of the two processors involved at each stage. Now the outcome was different for 1% and 2%, but the results for a tolerance of 4% were identical to those of 2%. We also apply the Vidwans *et al.*, Chaco and JOSTLE algorithms to the above initial partitioning and the results are summarised in Table 3.4.

It is interesting to observe that the final partition produced by the JOSTLE algorithm has smallest value of MaxImb (which is 2.4%) but the corresponding value of CutWt (which is 4087) is the second highest (the highest CutWt is produced by

Coarse mesh : 161 elements
Final mesh : 23950 elements
Average load : 2994 fine elements

Proc. Id.	Processor Load (initial)			Processor Load (final Tol = 1%)			Processor Load (final Tol = 2%)		
	fine	% imb.	coarse	fine	% imb.	coarse	fine	% imb.	coarse
0	3060	2.2	21	2970	-0.8	20	3060	2.2	21
1	2800	-6.5	19	2950	-1.5	20	3120	4.2	21
2	2220	-25.8	15	2970	-0.8	20	2990	-0.1	20
3	4180	39.6	29	3040	1.5	21	2910	-2.8	20
4	2660	-11.1	18	2950	-1.5	20	2800	-6.5	19
5	3040	1.5	20	3080	2.9	21	3070	2.5	21
6	2110	-29.5	14	2970	-0.8	19	3130	4.6	20
7	3880	29.6	25	3020	0.9	20	2870	-4.1	19

Table 3.1: Partition generated in parallel on 8 processors along with our final partitions for Example 1.

	MaxImb	Largest partition	Smallest partition	CutWt
Initial result	39.6%	3 (4180)	6 (2110)	1730
New result (Tol = 1%)	2.9%	5 (3080)	1 (2950)	1380
New result (Tol = 2%)	4.6%	6 (3130)	4 (2800)	1270
Vidwans *et al.* result	2.2%	0 (3060)	2 (2950)	2130
Chaco result	4.6%	6 (3130)	5 (2940)	1030
JOSTLE result	8.2%	1 (3240)	6 (2680)	1560

Table 3.2: Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.1) of Example 1.

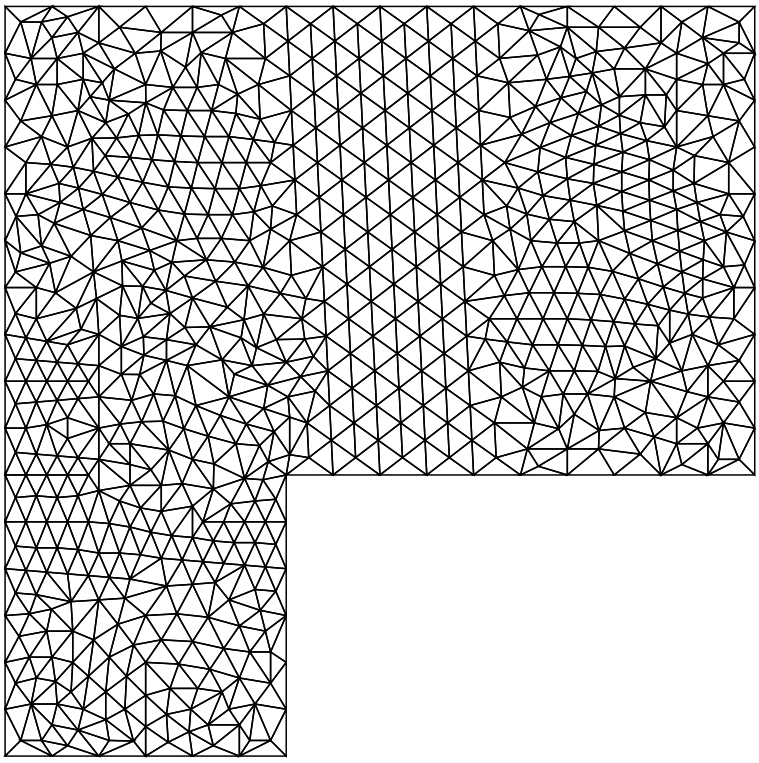

Figure 3.9: The coarse mesh of Example 2.

the Vidwans *et al.* algorithm which is 4999 where the value of MaxImb is 3.7%) as compared to other final partitions. On the other hand in the case of the Chaco algorithm, the value of MaxImb (which is 4.5%) is the highest as compared to other final partitions but the corresponding value of CutWt (which is 3517) is the smallest as compared to other final partitions. The result produced by the New algorithm are in between these two extremes. Our algorithm produces a final partition in which the value of MaxImb is 3.2% with the corresponding value of CutWt (with Tol as 2%) being 3857.

Example 3. Here we consider the same mesh as in Example 2, which is generated by the method of Hodgson and Jimack ([51]) in parallel using 8 processors. Unlike in the previous example however the weights correspond to the number of fine vertices generated inside each coarse element and subdomain rather than the number of fine elements. Characteristics of both the initial as well as the final partitions (produced by the New algorithm) are shown in Table 3.5. The combined total vertex weight is 468495, so that the average load per processor is 58562. The difference in weight between the processors with the maximum and minimum loads is 12637. In order

Coarse mesh : 1354 elements
Final mesh : 836183 elements
Average load : 104523 fine elements

Proc. Id.	Processor Load (initial)			Processor Load (final Tol = 1%)			Processor Load (final Tol = 2%)		
	fine	% imb.	coarse	fine	% imb.	coarse	fine	% imb.	coarse
0	103430	-1.0	52	103430	-1.0	52	103430	-1.0	52
1	98476	-5.8	607	106240	1.6	621	106240	1.6	621
2	105313	0.8	274	107850	3.2	325	107850	3.2	325
3	98017	-6.2	20	103854	-0.6	17	103854	-0.6	17
4	112686	7.8	20	106394	1.8	18	106394	1.8	18
5	104230	-0.3	305	102846	-1.6	246	102846	-1.6	246
6	105677	1.1	18	104821	0.3	19	100306	-4.0	18
7	108354	3.7	58	100748	-3.6	56	105263	0.7	57

Table 3.3: Partition generated in parallel on 8 processors along with our final partitions for Example 2.

	MaxImb	Largest partition	Smallest partition	CutWt
Initial result	7.8%	4 (112686)	3 (98017)	3854
New result (Tol = 1%)	3.2%	2 (107850)	7 (100748)	3900
New result (Tol = 2%)	3.2%	2 (107850)	6 (100306)	3857
Vidwans *et al.* result	3.7%	1 (108432)	6 (97659)	4999
Chaco result	4.5%	2 (109204)	3 (99651)	3517
JOSTLE result	2.4%	5 (107093)	3 (101665)	4087

Table 3.4: Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.3) of Example 2.

to further judge the New algorithm we also use the Vidwans *et al.*, Chaco and JOSTLE algorithms and the results are summarised in Table 3.6.

As is clear from Table 3.6, the initial value of MaxImb is 9.7% and that of CutWt is 3854. When the New algorithm is used with Tol = 1% the value of MaxImb reduced from 9.7% to 2.6% with a negligible increase in the value of CutWt. When the value of Tol is increased to 2% we see a deterioration in both the values of MaxImb as well as CutWt. When the value of Tol is further increased to 4% then the new partition is better in terms of CutWt but the value of MaxImb has gone up. The final partition produced by the Chaco algorithm has a value of 3.0% for the parameter MaxImb and a value of 3635 for the parameter CutWt. The corresponding values in the case of the JOSTLE algorithm are 2.7% and 3577 respectively. Overall the partition produced by the JOSTLE algorithm for this example is superior to the partitions produced by other tools. The values of the parameters MaxImb and CutWt are 5.5% and 4842 respectively in the final partition produced by the Vidwans *et al.* algorithm which are the highest as compared to the corresponding values produced by the other algorithms.

Example 4. Here we consider another mesh which is also generated by the method of Hodgson and Jimack ([51]) in parallel; this time using 16 processors. The geometry used is the "Texas" domain taken from PLTMG [5] (see Figure 3.10 for the corresponding coarse mesh). The initial and final situations are shown in Table 3.7. The combined load is 446151, so that the average load of a processor is 27884. We applied the New algorithm to this unbalanced partition for different values of tolerance (Tol) but the results were identical in each case. We also apply the Vidwans *et al.* and JOSTLE algorithms to the above initial partitioning and the results are summarised in Table 3.8. We are unable to compare our result with the result produced by the Chaco algorithm, as the option we choose to use in the Chaco algorithm, only supports 8 or less subdomains ([48, Subsection 4.4]).

As is clear from the Table 3.8, the initial value of MaxImb is 5.0% and that of CutWt is 4869. It is interesting to observe that the final partition produced by the Vidwans *et al.* algorithm has smallest value of MaxImb (which is 1.5%) but the corresponding value of CutWt (which is 5435) is the highest as compared to other final partitions. The common features of other tools are that we have reduction not only in the value of MaxImb but also in the value of CutWt. In these cases the new

Coarse mesh : 1354 elements Final mesh : 468495 vertices Average load : 58562 fine vertices

Proc. Id.	Processor Load (initial)			Processor Load (final Tol = 1%)			Processor Load (final Tol = 2%)		
	fine	% imb.	coarse	fine	% imb.	coarse	fine	% imb.	coarse
0	56570	-3.4	52	58264	-0.5	54	58264	-0.5	54
1	64255	9.7	607	58915	0.6	593	58915	0.6	593
2	61427	4.9	274	59512	1.6	276	56634	-3.3	276
3	51618	-11.9	20	57292	-2.2	33	60170	2.7	33
4	59344	1.3	20	58524	-0.1	20	58524	-0.1	20
5	61320	4.7	305	60057	2.6	297	60057	2.6	297
6	55598	-5.1	18	56418	-3.7	18	56418	-3.7	18
7	58363	-0.3	58	59513	1.6	63	59513	1.6	63

Table 3.5: Partition generated in parallel on 8 processors along with our final partitions for Example 3.

	MaxImb	Largest partition	Smallest partition	CutWt
Initial result	9.7%	1 (64255)	3 (51618)	3854
New result (Tol = 1%)	2.6%	5 (60057)	6 (56418)	3855
New result (Tol = 2%)	2.8%	3 (60170)	6 (56418)	3928
New result (Tol = 4%)	5.1%	2 (61540)	3 (55264)	3782
Vidwans *et al.* result	5.5%	2 (61795)	3 (55150)	4842
Chaco result	3.0%	2 (60332)	6 (55472)	3635
JOSTLE result	2.7%	6 (60138)	4 (56289)	3577

Table 3.6: Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.5) of Example 3.

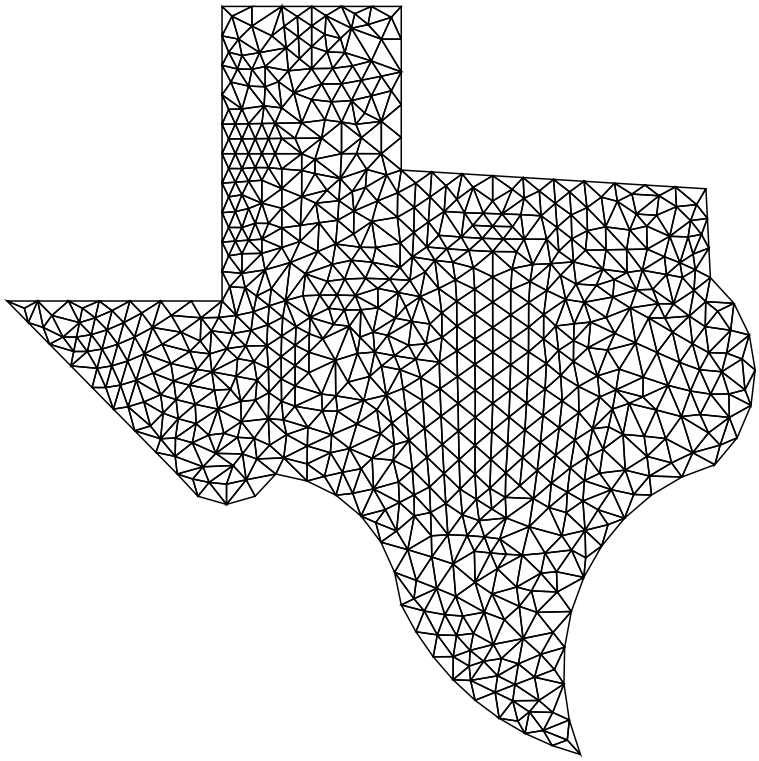

Figure 3.10: The coarse "Texas" mesh of Example 4.

values of MaxImb are similar but the CutWt produced by the JOSTLE algorithm is better than the corresponding value of CutWt produced by the other algorithms.

Example 5. In this example the coarse mesh consists of 5184 tetrahedral elements and corresponds to a discretisation of a 3-d space. This space is the union of two touching cuboids as used in [88, 93]. To simulate the shock wave diffraction around the 3-d right-angled corner formed between the cuboids, Selwood and Berzins employ a parallel adaptive Euler solver ([87]). We will return to this problem again in Chapter 5 where it will be discussed in much more detail.

Figures 3.11 and 3.12 illustrate how the mesh, which is distributed among 8 processors, adapts to the solution as the shock progresses through the domain. It is clear that although a partition of the mesh for the initial condition may be good, it is unlikely to remain so as the solution develops (as a matter of fact after 240 time steps an imbalance of 27.8% emerged) and thus dynamic load-balancing of the distributed data will be required. After 240 time steps there were 23274 fine elements in the mesh and the cut weight was 1332. We apply the New algorithm and partitions before and after the load balancing is shown in Table 3.9. We also use

Coarse mesh : 1331 elements Final mesh : 446151 elements Average load : 27884 fine elements

Proc. Id.	Processor Load (initial)			Processor Load (final Tol = 1%)			Processor Load (final Tol = 2%)		
	fine	% imb.	coarse	fine	% imb.	coarse	fine	% imb.	coarse
0	27985	0.4	94	27985	0.4	94	27985	0.4	94
1	28572	2.5	95	28572	2.5	95	28572	2.5	95
2	27536	-1.2	78	26765	-4.0	76	26765	-4.0	76
3	28238	1.3	89	27877	0.0	89	27877	0.0	89
4	29284	5.0	115	28287	1.4	112	28287	1.4	112
5	28310	1.5	86	27708	-0.6	85	27708	-0.6	85
6	27903	0.1	91	28494	2.2	91	28494	2.2	91
7	27886	0.0	88	27892	0.0	88	27892	0.0	88
8	28282	1.4	76	28282	1.4	76	28282	1.4	76
9	28036	0.5	69	28036	0.5	69	28036	0.5	69
10	26831	-3.8	71	26831	-3.8	71	26831	-3.8	71
11	26548	-4.8	72	28064	0.6	76	28064	0.6	76
12	28310	1.5	76	28310	1.5	76	28310	1.5	76
13	27737	-0.5	64	27737	-0.5	64	27737	-0.5	64
14	27625	-0.9	85	27625	-0.9	85	27625	-0.9	85
15	27068	-2.9	82	27686	-0.7	84	27686	-0.7	84

Table 3.7: Partition generated in parallel on 16 processors along with our final partitions for Example 4.

	MaxImb	Largest partition	Smallest partition	CutWt
Initial result	5.0%	4 (29284)	11 (26548)	4869
New result (Tol = 1%)	2.5%	1 (28572)	2 (26765)	4666
New result (Tol = 2%)	2.5%	1 (28572)	2 (26765)	4666
Vidwans *et al.* result	1.5%	12 (28310)	5 (27120)	5435
JOSTLE result	2.4%	7 (28578)	1 (26906)	4064

Table 3.8: Summary of results when the New, Vidwans *et al.* and JOSTLE algorithms are applied to the initial partition (see Table 3.7) of Example 4.

the Vidwans *et al.*, Chaco and JOSTLE algorithms and the results are summarised in Table 3.10.

As is clear from Table 3.10 the Chaco algorithm produces a final partition in which the value of MaxImb is smallest and the Vidwans *et al.* algorithm produces a final partition in which the value of of CutWt is highest. The least value of CutWt is enjoyed by the JOSTLE algorithm. The New algorithm produces the final partitions in which the values of MaxImb and CutWt are 2.5% and 1344 respectively when Tol is 1% and the values of these parameters are 4.7% and 1336 in the case of Tol being 2%. Again observe the affect of Tol on these parameters. As Tol increases so does MaxImb. But on the other hand the value of CutWt decreases as Tol increases.

Example 6. The underlying geometry of this problem is the same as that of previous example, but the initial coarse mesh is bigger. Here the coarse mesh consists of 34560 elements. At a certain time after applying the adaptive step of the solver, the maximum imbalance grew to 30.5%. To reduce this maximum imbalance among the processors the dynamic load balance algorithm is applied. Table 3.11 shows both the partition before, as well as after the application of the algorithm. We also use the Vidwans *et al.*, Chaco and JOSTLE algorithms and the results are summarised in Table 3.12.

It is clear from Table 3.12 that the initial values of the parameters MaxImb and CutWt are 30.5% and 3710 respectively. Just like Example 4, we apply the New algorithm to this unbalanced partition for different values of tolerance (Tol) but here the results were identical in each case. The partition produced by the Chaco algorithm has the best final value of MaxImb and the partition produced by the Vidwans *et al.* algorithm has the worst final value of CutWt. In the case of the JOSTLE algorithm, the final value of MaxImb is relatively high but the final value of the CutWt is fairly small. The value of MaxImb in the case of the New algorithm is in between the corresponding values produced by the other tools. And the CutWt produced by the New algorithm is better than that produced by the Chaco and Vidwans *et al.* algorithms but worse than that of produced by the JOSTLE algorithm.

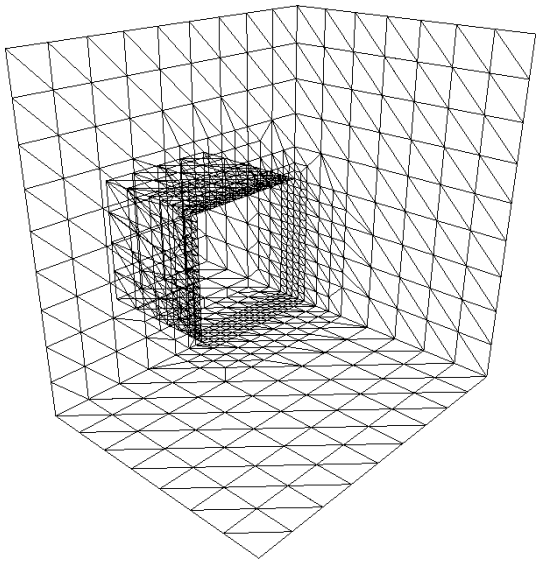

Figure 3.11: Coarse mesh of 5184 elements adapted to initial shock condition for Example 5.

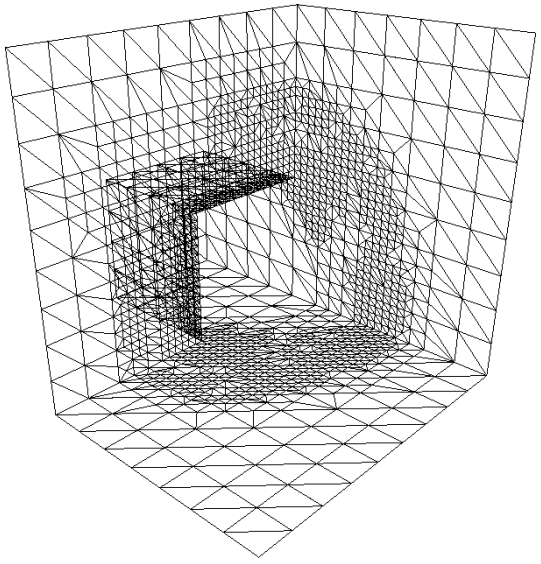

Figure 3.12: Adapted mesh after 240 time-steps for Example 5.

Coarse mesh : 5184 elements
Final mesh : 23274 elements
Average load : 2909 fine elements

Proc. Id.	Processor Load (initial)			Processor Load (final Tol = 1%)			Processor Load (final Tol = 2%)		
	fine	% imb.	coarse	fine	% imb.	coarse	fine	% imb.	coarse
0	2198	-24.4	247	2921	0.4	263	2921	0.4	263
1	2737	-5.9	104	2929	0.7	117	2929	0.7	117
2	2822	-3.0	2205	2890	-0.7	2131	2862	-1.6	2137
3	3719	27.8	84	2926	0.6	66	2926	0.6	66
4	3134	7.7	62	2897	-0.4	65	2897	-0.4	65
5	3201	10.0	137	2839	-2.4	131	2839	-2.4	131
6	2978	2.4	71	2983	2.5	50	3046	4.7	51
7	2485	-14.6	2274	2889	-0.7	2361	2854	-1.9	2354

Table 3.9: Initial and final partitions (produced by the New algorithm) for Example 5.

	MaxImb	Largest partition	Smallest partition	CutWt
Initial result	27.8%	3 (3719)	0 (2198)	1332
New result (Tol = 1%)	2.5%	6 (2983)	5 (2839)	1344
New result (Tol = 2%)	4.7%	6 (3046)	5 (2839)	1336
Vidwans *et al.* result	3.3%	2 (3005)	1 (2824)	1859
Chaco result	2.0%	3 (2966)	0 (2859)	1578
JOSTLE result	2.1%	3 (2973)	2 (2746)	1304

Table 3.10: Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.9) of Example 5.

3.6 Conclusions

In this chapter a serial version of a new dynamic load balancing algorithm has been presented. The algorithm is applied to six different test problems which are

Coarse mesh : 34560 elements
Final mesh : 83486 elements
Average load : 10436 fine elements

Proc. Id.	Processor Load (initial)			Processor Load (final Tol = 1%)			Processor Load (final Tol = 2%)		
	fine	% imb.	coarse	fine	% imb.	coarse	fine	% imb.	coarse
0	9871	-5.4	1378	10482	0.4	1522	10482	0.4	1522
1	8883	-14.9	288	10406	-0.3	910	10406	-0.3	910
2	10678	2.3	977	10516	0.8	1088	10516	0.8	1088
3	8880	-14.9	253	10280	-1.5	390	10280	-1.5	390
4	8519	-18.4	746	10498	0.6	2263	10498	0.6	2263
5	10677	2.3	6035	10403	-0.3	7486	10403	-0.3	7486
6	13620	30.5	13218	10451	0.1	10451	10451	0.1	10451
7	12358	18.4	11665	10450	0.1	10450	10450	0.1	10450

Table 3.11: Initial and final partitions (produced by the New algorithm) for Example 6.

	MaxImb	Largest partition	Smallest partition	CutWt
Initial result	30.5%	6 (13620)	4 (8519)	3710
New result (Tol = 1%)	0.8%	2 (10516)	3 (10280)	4629
New result (Tol = 2%)	0.8%	2 (10516)	3 (10280)	4629
Vidwans *et al.* result	1.0%	0 (10540)	3 (10328)	11463
Chaco result	0.5%	6 (10484)	3 (10403)	5915
JOSTLE result	1.4%	2 (10585)	7 (10148)	2718

Table 3.12: Summary of results when the New, Vidwans *et al.*, Chaco and JOSTLE algorithms are applied to the initial partition (see Table 3.11) of Example 6.

representative of typical practical situations in both 2-d and 3-d. In the majority of the cases our results are comparable with the results produced by other state of the art algorithms. In all cases the values of the parameter CutWt in the final partition produced by the New algorithm are much less than the corresponding values in the final partition produced by the Vidwans *et al.* algorithm and in the majority of cases the values of the parameter MaxImb produced by the New algorithm are also much less than the corresponding values produced by the Vidwans *et al.* algorithm. This shows that the modifications made in the Vidwans *et al.* algorithm do improve the quality of the new partitions.

In the last problem the cost of our algorithm is a little high. It should be noted that in this last example the coarse mesh is of relatively large size (34560 coarse elements) and initially 72% of the coarse mesh resides on just two processors (whose ID's are 6 & 7) as compared to only 31% of the fine mesh which reside on these processors. Unlike the JOSTLE algorithm, our algorithm has no graph coarsening feature built into it and so is likely to be most efficient for those problems where the size of coarse mesh is not too great. This is not a major restriction however, since starting with a smaller size mesh one can always get a fine mesh of much larger size by the repeated application of the adaptive refinement algorithm.

The main practical restriction on this algorithm as described in this chapter is that one should not rely upon a *serial* dynamic load-balancing algorithm to regain the balanced load of the processors while using a *parallel* adaptive solver. Such an approach presents a significant serial bottleneck; and in some cases it may not be possible at all if the size of the mesh is too huge to fit into the memory of a single processor. To avoid this serial bottleneck we present, in the next two chapters, practical parallel versions of our load-balancing algorithm which are designed for rebalancing hierarchical unstructured meshes in 2-d and 3-d space respectively.

In Chapter 4 the parallel algorithm is implemented for a specific class of 2-d meshes which are generated by the method of Hodgson and Jimack ([51]). The algorithm discussed there consists of two phases. In phase one we decide the new owners of the coarse elements and in second phase the actual migration of these elements is undertaken. Phase one remains the same irrespective of the generation methods of the partitioned meshes, however phase two does depend upon how the mesh is generated and maintained on a given processor, and the way in which connectivity information is stored across the inter-partition boundaries. So for

those types of unstructured mesh which are generated by other methods one has to modify phase two only. The performance of our algorithm is compared with the performance of the algorithms of Vidwans *et al.* ([104]) & Hu and Blake ([53]).

In Chapter 5 we present that version of the algorithm which is designed to rebalance the 3-d meshes which consists of tetrahedral elements. There are some further improvements and modifications to the algorithm for this work. The performance of this new modified algorithm is compared with the performance of the Vidwans *et al.* ([104]), ParJOSTLE ([109]) and ParMETIS ([63]) algorithms.

Chapter 4

Parallel Application of the Dynamic Load Balancer in 2-d

In this chapter we consider the parallel application of the algorithm presented in the previous chapter. This particular parallel application is basically a post-processing step which redistributes a 2-d mesh which has been created in parallel but is not perfectly load balanced among the number of available processors (due to reasons discussed below in §4.1). In the next chapter we discuss the parallel application of the algorithm presented in Chapter 3 for a more general 3-d problem which is designed to migrate some tetrahedral elements form one processor to another processor after an adaptivity step (see §2.3 for details) in order to regain a balanced load.

In order to solve large Computational Fluid Dynamics (CFD) and Computational Mechanics (CM) problems numerically on a parallel or sequential machine, the first step of most numerical methods is to generate a mesh of the underlying geometry (if one has not been generated already). For finite element solvers the popular choice is to generate an unstructured mesh. There are many advantages of unstructured meshes over structured ones. One advantage is that they are ideally suited for the discretisation of geometrically complicated domains, a second is that their use allows for the easy addition and removal of vertices and elements which are often required in an adaptive setup. There are many ways to generate unstructured meshes and many review papers about the development of mesh generation techniques (see [94, 97, 113] for example). Since we wish to solve problems in parallel the mesh should also be distributed among the number of available processors in

an efficient manner.

One way to obtain a distributed mesh is to generate the entire mesh on a single processor and then distribute it among the desired number of processors using any of the static graph partitioning algorithms discussed in Chapter 1. This approach results in a serial bottleneck at the generation stage however, and in some cases it may not even be possible due to the limited amount of memory that may be available on a single processor. Another possibility is to actually generate the mesh in a parallel and distributed manner.

Nowadays parallel mesh generation is becoming an important feature of any large distributed memory parallel CFD and CM codes as it ensures that:

- there is no sequential bottleneck at this point in the code,
- there is no parallel overhead incurred in partitioning an existing mesh,
- that no single processor is required to have enough local memory to be able to store the entire mesh.

In recent years numerous algorithms have been proposed for the generation of unstructured finite element and finite volume meshes in parallel (see [2, 38, 51, 66, 70, 116] for example). One of the main problems with many of these approaches however is that the final mesh, once generated, cannot generally be guaranteed to be perfectly load-balanced. Since an unbalanced load may adversely affect the performance of the solver there is a clear need for executing a post-processing step in order to get a well-balanced mesh at the end.

In this chapter we propose a post-processing step for the parallel mesh generator, based upon the cheap and efficient dynamic load-balancing technique of previous chapter. This technique is described and a number of numerical examples are presented in order to demonstrate that the quality of the partition of the mesh can be improved significantly at only a small additional computational cost. It should be mentioned here that this post-processing step is coupled with an existing parallel mesh-generator which is due to Hodgson and Jimack ([51]). However, the parallel application in the next chapter is quite general in the sense that it can be coupled with any parallel adaptive mesh-generator.

4.1 Introduction

This chapter is concerned with dynamic load-balancing algorithms in connection with the parallel generation of unstructured meshes of triangles for complex geometries in two dimensions (the parallel generation and adaptation of unstructured meshes of tetrahedra for complex geometries in three dimensions will be dealt with in the next chapter). A large number of algorithms and codes have been developed for parallel mesh generation in recent years and these may be divided into two broad categories: those based upon refinement of an initial coarse background mesh (e.g. [51, 70, 88, 98]), and those which mesh the domain in an alternative manner (e.g. [2, 66]). In this chapter we are concerned only with the first of these two categories and, for simplicity of exposition, we concentrate on the 2-d case. Extension to 3-d is possible and is the topic of next chapter.

The common feature of all of the parallel mesh generators based upon refinement of a background grid is that this grid must first be partitioned across the available processors. The techniques by which this is done vary significantly but they each have the same goal: to ensure that the total number of generated elements or points on each processor is about the same upon completion of the parallel mesh generation. Hence, if a mesh of uniform density is being generated and the background grid is also of uniform density then we would expect each processor to be assigned about the same number of coarse elements. If, on the other hand, a mesh of non-uniform density is being generated from a uniform background grid then we would expect a potentially different number of coarse elements to be assigned to each processor. A secondary objective when partitioning the background grid is to ensure that the number of generated elements which have an edge on the boundary between two processors is as small as possible. This will ensure that the amount of communication required by the finite element or finite volume solver will be minimised (see §2.3 for details).

In order to attempt to achieve these objectives, *a priori* estimates need to be made about how many elements, edges and nodes will be generated within each coarse element. Inevitably the actual values of these three numbers after generation will not precisely match these estimates. In order to keep the differences as small as possible some authors have developed quite elaborate schemes for improving the quality of their estimates; including the use of *neural networks* [98] or *virtual*

refinement [51] for example. Even with these schemes however final load imbalances of up to 10% are frequently observed in practice.

In this chapter we suggest that, so long as a reasonable partition is produced *a priori*, a more profitable use of resources is to improve the quality of the partition after the mesh has been generated in parallel through the use of a parallel post-processing step. This step simply involves making local modifications to the load-balance before the solution phase commences. As will be demonstrated these local modifications are made in a manner designed to strike a balance between the potentially conflicting requirements of

1. improving the load-balance,
2. maintaining data locality,
3. minimising the number of fine edges shared by two processors,
4. avoiding sequential bottlenecks,

which were first discussed in §2.3. In the following sections we outline a parallel dynamic load-balancing algorithm which is designed to meet these objectives. This algorithm is a modification of the algorithm presented in the previous chapter. It is based upon earlier work of Vidwans *et al.* [104]. It should be noted that the algorithm introduced may be directly applied to a more general class of dynamic load-balancing problem than that introduced above. In particular, we may have an adaptive hierarchical mesh, (see [92] for example) and/or an adaptive p-version of the finite element algorithm (as in [3] for example). The next chapter describes in details the version of the algorithm designed for dynamically rebalancing the 3-d adaptive hierarchical meshes.

4.2 A Parallel Dynamic Load-Balancing Algorithm

Suppose we have a hierarchically refined root mesh which is distributed across p processors. As in §3.2.1 we define the weight of a root element as the number of leaf elements inside it and the weight of an edge of the root mesh as the number of leaf element edges along it. We may also recall from §2.3 the definition of the weighted dual graph of the root mesh: each node of this graph corresponds to an

element of the root mesh (whose weight is same as the weight of the corresponding root element), $\mathcal{T}_0$, and two nodes are connected if the corresponding root elements are neighbours (the weight of this edge is being the weight of the common edge of the two corresponding root elements). The task of repartioning the triangulation $\mathcal{T}$ may therefore be represented in terms of this weighted graph. In particular, we require an algorithm for repartitioning such a graph which satisfies the four criteria enumerated in §4.1.

4.2.1 Group Balancing

Let us recall from §2.4 the definition of the Weighted Partition Communication Graph (WPCG): A WPCG is obtained by having one vertex for every processor and an edge between two vertices if and only if they are face adjacent to each other. The weight w_{N_i} of the i^{th} vertex is equal to the sum of weights of all root elements on the i^{th} processor and the weight $w_{E_{ij}}$ of the edge connecting the i^{th} and j^{th} processors is equal to the sum of weights of all root element edges on the interpartition boundary between the two processors.

We next divide the WPCG into two subgroups denoted by Group1 and Group2 by using the same procedure of §3.2.1. It may be pointed out here that this division is performed on a single processor. Let us recall from §3.2.1 that during the division process of WPCG, one requires the weighted Laplacian matrix of the WPCG. In the present context the assembly of the i^{th} row of this Laplacian is performed locally by the i^{th} processor. Each processor after assembling its own row sends it to one processor (which we call a master processor). The master processor after receiving all the contributions from all other processors forms the Laplacian and then divides the WPCG in to 2 subgroups. Finally the master processor broadcasts this decision to all other processors.

If the leaf mesh is quite uniformly distributed across the processors then we would expect each group to contain about the same number of processors and an almost identical total weight. If the existing partition is not well load balanced however then the number of processors in each group may be very different. In either case the cut-weight resulting from this bisection will generally be very small. In the next stage of the algorithm we use the idea of local migration from the "larger" to the "smaller" group so that after migration each group contains approximately the

same average weight per processor without there being a significant increase in this cut-weight.

4.2.2 Local Migration

As mentioned above the subgroups formed in the last subsection may not be ideally balanced. To balance them we now migrate nodes of the weighted dual graph from the "larger" to the "smaller" group by using the logic of §3.2.2. We call "larger" the Sender and "smaller" the Receiver group respectively. The actual determination of Sender and Receiver groups are performed in Figure 3.1 which also calculates the quantity Mig_{tot} which stands for the total weight of all the nodes which are to be migrated from the Sender to the Receiver. Note that if the combined weight of the nodes transferred from the Sender to the Receiver is nearly or exactly equal to Mig_{tot} then the two groups will be load-balanced upon completion.

Having established the required load to be transferred, the next issue to address is that of how many nodes each processor in the Sender group should actually send and which processors in the receiver group they should be sent to. We again use the same idea of candidate processors as defined in §3.2.2. Recall that processors in each group that are face-adjacent to at least one processor in the other group are called candidate processors. We only allow the candidate processors to be involved in the actual migration of nodes from Sender to Receiver. Let N_{tot} be the total weight on all candidate processors of the Sender group. Then if the i^{th} candidate processor in the Sender group is face adjacent to more than one candidate processor in the Receiver group we migrate nodes to that candidate processor which has the "longest" boundary (by this we mean that the cut-weight between the two processors involved is maximum as compared to other possible pairs). The amount of load shifted from the i^{th} candidate processor in Sender group is denoted by Mig_i and is calculated in precisely the same way as in §3.2.2 (equation 3.1).

As far as the actual migration of the nodes in the weighted dual graph is concerned, we follow precisely the same logic as of §3.2.2. Recall from §3.2.2 that in the process of migration of nodes we use the concepts of gain and gain density associated with the moving nodes. As in §3.2.2 by *gain(k)* of node k we mean the net reduction in the cost of cut edges that would result if node k were to migrate from i^{th} candidate processor in the Sender group to the j^{th} candidate processor in

the Receiver group (see Figure 3.2 for the calculation of gain(k)). The gain density of a node is defined as the gain of the node divided by the weight of the node. The bulk of the work needed to make a move consists of selecting the base node (a node which is about to be shifted from one processor to another processor is called a base node), moving it, and then updating the gains of its neighbouring nodes.

Just like §3.2.2 the selection of a base node is made by choosing the node with the largest gain density on the i^{th} processor whose weight is less than or equal to Mig_i. We shift the node to the receiving processor and update the gains of its neighbouring nodes using the algorithm outlined in Figure 4.1. Observe that, if the gain associated with the base node is positive, then transferring it will not only improve the load-balance but will also reduce the total cut-weight between the two groups. The above logic of balancing the two groups is presented in Figure 4.2.

At this point we should discuss some major differences between the current method of calculating and updating the gains and the previous method of calculating and updating the gains used in the previous chapter. We first wish to emphasise the use of the word processor. In the last chapter the word processor was used in a symbolic sense. This was nothing to do with the physical processor of the machine. On the other hand in the present context the word processor is both symbolic as well as physical. At present each subdomain of the original domain is assigned to a unique physical processor of the parallel machine. In the previous chapter in principle only one physical processor was responsible to achieve the entire task of load-balancing the dual graph. In doing so the concept of recursion was used extensively. Starting with one group the problem was divided into two subgroups. To balance the original group some load was shifted from the Sender to the Receiver subgroups. This was achieved by considering a pair of processors called i, j processor pair at a time. The actual migration of the load depends upon the gain densities of the nodes which can be calculated by using the gains shown in Figure 3.2 (which only calculates the gains of those nodes which belong to the i^{th} processor). As pointed out in §3.2.2 there was also calculations of gains of all the nodes belonging to the j^{th} processor (recall from §3.2.2 that the gain of a node belonging to j^{th} processor is define to be the net reduction in the cost of cut edges that would result if the node were to migrate from j^{th} processor to the i^{th} processor). The calculation of gains for all the nodes of the i, j pair of processors facilitated the reshuffling process of §3.3 (where we performed a sequence of moves of nodes from

For each $n_k \in i^{th}$ processor which is a neighbour of the node k {
increment gain(n_k) by $2^* w_{E_{n_k k}}$;
find the new gain density of the node n_k;
}

Figure 4.1: Updating the gains.

one processor to another).

In the current situation each processor only calculates and updates the gains of those nodes which it owns. A processor does not attempt to calculate the gains of nodes owned by other processors. This is due to the fact that in the parallel implementation we decided not to execute the re-shuffling step of the serial version. The rational behind this decision is the fact that such a step is not possible without a substantial amount of communication among the processors. We believe that the cost of such communication involved in the re-shuffling steps will slow down the performance of the parallel load-balancer substantially, whilst experience suggest it only provides a small improvement in the partition quality.

4.2.3 Divide and Conquer and Parallel Implementation

Once we have obtained Sender and Receiver groups with the same average weights, it is possible to recursively apply the above splitting algorithm to each of these two processor groups in parallel: bisecting them and load-balancing them. The recursion terminates when every group consists of a single processor: each with approximately the same load.

This divide and conquer approach naturally permits a certain degree of parallelism in its implementation. Further parallelism is also facilitated by the fact that it is possible for more than one sending processor in a Sender group to migrate data onto its corresponding receiving processor at any given time.

This divide and conquer algorithm is described fully in Figure 4.3 and discussed below.

```
While(there are unprocessed candidate procs. in Sender group){
    Let Mig_i (as explained above) be the amount of load to be
    shifted from the i^th candidate proc. in Sender group to the
    j^th candidate proc. in Receiver group,
    Calculate gain densities.
    Until(required load is transferred from the i^th proc. to the j^th proc. ){
        Move a node of highest possible gain density from the i^th proc.
        to the j^th proc.,
        Update gain densities of neighbours of the moved node.
    }
}
```

Figure 4.2: Load balancing of the two groups.

```
While (Any Groups contain two or more processors){
   Find the maximum load Max and the average load Ave of the Group,
   Find the percentage of maximum imbalance MaxImb in the Group
   by using the formula:
                    MaxImb = ( (Max - Ave) / Ave) * 100.
   If (The Group has more than one processors){
      Send the contribution of the Laplacian to the processor 0.
      If (Rank of the processor is 0){
         Form the Laplacian matrix after receiving the contribution
         from other processors,
         Find the Fiedler vector and by using it decide the Receiver
         and Sender groups.
      }
      If (MaxImb is more than a given tolerance){
         Move some load from the processors in the Sender Group to the processors
         in the Receiver Group in such as way that after the migration two
         Groups have the same average load and the increase in the cut weight
         is as small as possible.
      }
   }
   If (The migration affects the current processor){
      Modify the necessary data structures to reflect the migration.
   }
   Divide the Group into two Groups (i.e. from now onwards both Sender
   and Receiver will be called Group).
}
```

Figure 4.3: Parallel dynamic load-balancing algorithm.

4.3 Discussion of the Algorithm

After deciding how much to shift and where to shift we face practical implementation difficulties. For example, if two neighbouring coarse elements (note that two elements are neighbours of each other if they have a common vertex or a common edge) migrate then it is very hard to modify the data structures (this is due to the fact that when two neighbouring elements migrate we have to use forwarded messages to other subdomains to tell them the rapidly changing situation, sending this type of message is a complicated business). To avoid this difficulty we decided to not move any neighbouring elements simultaneously. For this reason we colour the coarse mesh and allow the simultaneous migration of elements of the same colour only. In case of triangular meshes only 6 to 10 colours are required to colour the mesh (in three dimensions the number of colours may change dramatically from one mesh to another mesh!). This means the migration step will take in general between 6 to 10 phases. In our implementation all of the processors must be synchronised after the migration of elements of the same colour.

Another point to be discussed here is the movement of data objects associated with the migrating nodes. As is clear from the description of the load-balancer the nodes of the dual graph will be migrating from one processor to another processor before they necessarily reach their final destination. Ideally we should pass some sort of token (as we do in the next chapter) to the transitory processor of the migrating node rather than passing the entire data objects associated with the migrating node to the transitory processor. Only at the very end should we pass entire data objects from the original processor of the migrating node to the final destination of the node. However in the current implementation we decided to pass the entire data objects associated with the migrating node to the transitory processor of the node. In 2-d, this strategy does not increase the cost of the migration process substantially (as is clear from Tables 4.7 and 4.14 where the maximum rebalancing time is just 1.3 seconds involving a mesh of 0.7 million fine elements). However in the next chapter such a strategy would be very costly (partly because of the use of halo elements) so there we only pass tokens when a node is migrated from one processor to another one and the entire data objects are passed only at the end of the process from the very original processor of the migrating node to the final destination of the node.

In the above algorithm there are basically three types of processor. The first

type of processor are those processors from which there is actual migration of coarse elements. The second type of processors are those which actually receives some coarse elements. The third type of processors are those processors which are affected by the above migration of elements (these are those processors which have at least one coarse element which is a neighbour of the migrated coarse elements). Observe that during the migration of coarse elements the intersection of type 1 and type 2 processors is empty but it is possible for a type 3 processors to be a type 1 or type 2 processor as well. We briefly discuss the activities of these three types of processor here; more details are given in the next section when we also discuss the accompanying data structures.

4.3.1 Activity of Type 1 Processors : Packing the Load

The type 1 (sending) processors will pack the following which will be sent to the type 2 (receiver) processors.

- Connectivity information of the fine mesh inside the coarse element.
- Coordinates of the interior points of the coarse element.
- Coordinates of the unshared corner points of the coarse element.
- Sharing information (by this we mean the IDs of those processors with which the vertices and edges are shared) of all the vertices and edges of the coarse element.

4.3.2 Activity of Type 2 Processors : Unpacking the Load

The type 2 (Receiver) processors will receive and unpack the above message to establish the new coarse element and associated fine mesh.

4.3.3 Activity of Type 3 Processors : Third Party Adjustment

When a coarse element migrates from one processor to another processor, not only these two processors are affected but also other processors may be affected in the sense that they may also have to modify their neighbourhood relations.

4.4 Description of Related Data Structures Associated With the Redistribution of the Mesh

As stated earlier the dynamic load-balancing algorithm presented in this chapter is coupled with a particular mesh generator which is due to Hodgson and Jimack ([51]). We describe here existing data structures used by this mesh generator and by the dynamic load-balancing algorithm which is associated with a typical subdomain.

- *nocrseelems* By *nocrseelems* we mean the total number of coarse elements in the current subdomain.

- *nonodessub* By *nonodessub*[i] we mean the number of vertices inside the i^{th} coarse element. These interior vertices are called vertices of "type 1".

- *noelemssub* By *noelemssub*[i] we mean the number of fine elements inside the i^{th} coarse element.

- *finevts* By *finevts*[i] we mean a pointer leading to the information about the co-ordinates of all the vertices of "type 1" in the i^{th} coarse element (so *finevts*[i][2j-1] and *finevts*[i][2j] represent respectively the x and y co-ordinates of the j^{th} vertex of "type 1" in the i^{th} coarse element).

- *crsetri* By *crsetri*[i] we mean a pointer leading to the information about the i^{th} coarse element (this information is basically about the connectivity and the colour of the element).

- *crsedges* By *crsedges*[i] we mean a pointer leading to the information about three edges of the i^{th} coarse element (this information is basically the IDs of these edges).

- *finetri* By *finetri*[i] we mean a pointer leading to the information about the fine elements of the i^{th} coarse element (this information is basically about the connectivity of these fine elements).

- *noedges* By *noedges* we mean total numbers of non-Dirichlet coarse edges in the subdomain.

- *noedgevts* By *noedgevts* we mean the total number of edge vertices (an internal vertex of an edge of a coarse element, which does not lie on the boundary of the domain is called an edge vertex). All the edge vertices are stored in a two dimensional array called *edgevts*. For example *edgevts*[i][2j-1] and *edgevts*[i][2j] represent the x and y coordinates of the j^{th} internal vertex of the i^{th} coarse edge of the subdomain respectively; an edge vertex is also referred to as being a vertex of "type 2".

- *nocrsevts* By *nocrsevts* we mean total number of coarse vertices (a vertex of a coarse element which does not lie on the boundary of the domain is called a coarse vertex); a coarse vertex is also referred as being a vertex of "type 3". All coordinates of the coarse vertices are stored in an array called *crsevts*. For example *crsevts*[2i-1] and *crsevts*[2i] represent the x and y coordinates of the i^{th} coarse vertex of the subdomain.

- *nobndvts* By *nobndvts* we mean total number of boundary vertices in the fine mesh (a vertex which lies on the external boundary is called a boundary vertex); a boundary vertex is also referred as being a vertex of "type 4". All coordinates of the boundary vertices are stored in an array called *bndvts*. For example *bndvts*[2i-1] and *bndvts*[2i] represent the x and y coordinates of the i^{th} boundary vertex of the subdomain.

- *transfer* By *transfer*[i][0] we mean the total number of coarse vertices common with i^{th} subdomain and *transfer*[i][j] gives the number of the j^{th} such coarse vertex. If nid represents the ID of the current subdomain then by *transfer*[nid][0] we mean the total number of coarse vertices in the subdomain and *transfer*[nid][j] gives the multiplicity of the j^{th} coarse vertex (by multiplicity we mean the number of subdomains with which the vertex is shared). For the corresponding information about the boundary vertices we use *transfer2*.

- *nomyintf* By *nomyintf* we mean number of vertices of "type 3" which are common with other subdomains.

- *nomyintf2* By *nomyintf2* we mean number of vertices of "type 4" which are common with other subdomains.

- *numngbrs* By *numngbrs* we mean number of neighbouring subdomains and the IDs of these subdomains are stored in the array *myngbrs*.

Since during the course of the algorithm the coarse elements from one subdomain will migrate to other subdomains, we face a twofold problem:

- each subdomain of type 2 must be ready to accommodate an unspecified number of coarse elements coming from subdomains of type 1,
- these coarse elements must be linked together with the existing elements already contained in the subdomain in an efficient manner.

To overcome this we decided to extend certain arrays to accommodate new items if the size of an element of the array is not too big - if the size is too large then we create a companion pointer array which is large enough to contain the addresses of the current elements as well as that of forthcoming ones (e.g. the size of an element of the array *nonodessub* is just four bytes (the size of an integer) so we simply extend this array, on the other hand the size of an element of the array *crsetri* is twenty four so we do not extend this array, instead we create a companion pointer array which is large enough to contain the addresses of the current elements as well as that of forthcoming ones). The rational behind this decision is to have a trade off between the available memory and the execution speed of the algorithm - creating large arrays to accommodate the worst possible scenario would reduce the amount of the memory for other tasks (in some cases it may not even be possible to do so), on the other hand creating the arrays which are capable of handling current elements only and extending them in future, should the need arise, will slow down the execution of the algorithm as the dynamically creating / extending arrays is a painfully slow process. Let *nocrseelemsall* be the variable which represents the maximum number of coarse elements which are possible in a certain subdomain. We extend the array *nonodessub* from *nocrseelems* to *nocrseelemsall*+1, with the new index set from 0 to *nocrseelemsall*. The elements from 1 to *nocrseelems* contain the old values and the remaining elements form a linked list ready for new arrivals according to the following recipe:

- *nonodessub*[0] = *nocrseelems* + 1,
- *nonodessub*[i] = -(i+1) for i from *nocrseelems* + 1 to *nocrseelemsall*.

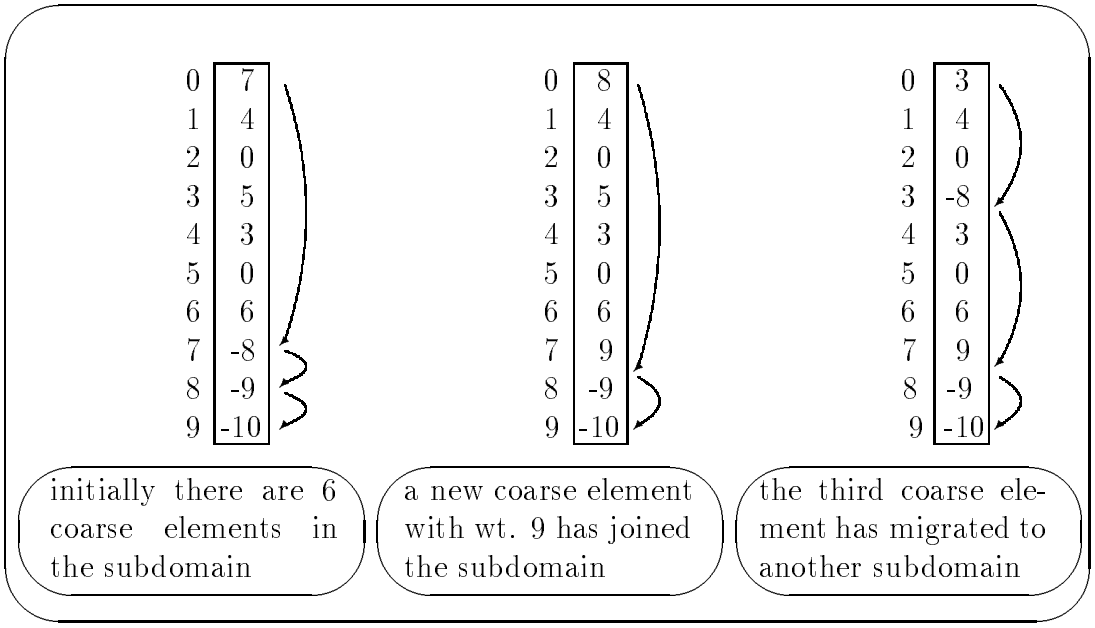

Figure 4.4: The array *nonodessuball* which can accommodate nine coarse elements.

The new coarse element will always be given the number stored in *nonodessub*[0] and the linked list will be updated accordingly (i.e. we update the value of *nonodessub*[0] by using the formula *nonodessub*[0] = -*nonodessub*[*nonodessub*[0]]). If a coarse element (say i) migrates from the current subdomain then its position will always be inserted at the beginning of the linked list (i.e. we execute the statements : *nonodessub*[i] = -*nonodessub*[0] and *nonodessub*[0] = i) (Figure 4.4 describes the situation for a small mesh in a subdomain). This way we can minimise the gap in numbering the coarse elements in a subdomain. We also extend the array *noelemssub* from *nocrseelems* to *nocrseelemsall*, with the new index set from 1 to *nocrseelemsall*. The elements from 1 to *nocrseelems* contain the old values and the remaining elements are reserved for future arrivals (note that there is no need to form a linked list here as this array (like the following four pointer arrays) is controlled by means of the pointer entries of the array *nonodessub*!). The extended version of these two arrays are called *nonodessuball* and *noelemssuball* respectively. We next define the following four pointer arrays each consisting of *nocrseelemsall* elements.

- *finevtsall* The first *nocrseelems* elements contain the addresses of the corresponding elements of the array *finevts*.

- *crsetriall* The first *nocrseelems* elements contain the addresses of the corresponding elements of the array *crsetri*.
- *crsedgesall* The first *nocrseelems* elements contain the addresses of the corresponding elements of the array *crsedges*.
- *finetrigroup* The first *nocrseelems* elements contain the starting addresses of the corresponding group of fine elements belonging to the corresponding coarse elements.

Note the remaining elements in the above four arrays are reserved for future entries. We also extend the arrays *nonodesdirichedge* and *nonodesedge* just like the array *nonodessub*, again with the understanding that the negative entries form the linked list for future arrivals.

4.5 Different Issues and Related Functions Used in the Main Algorithm By Processors of Type 1

There are various issues related with the transfer of data items. Suppose, as a result of applying the algorithm, we discover that coarse element i from a subdomain *nid* (say) of type 1 is to be migrated to another subdomain j (say) of type 2. Then the following issues are to be considered.

4.5.1 Handling of Vertices

When a coarse element migrates from one subdomain to another subdomain we have to examine its three vertices carefully (recall from §4.4 that these are the vertices of type greater than 2: type 3 means coarse vertex and type 4 means boundary vertex). This is the job of the present section. Let v be a vertex of type 3 or 4 of the element i in the subdomain *nid*. It is important to know if the vertex v is shared between *nid* and j before and/or after the migration.

```
if(transfer[nid][v] == 1 or !transfer[j][0])      /* either v is not shared with */
   return(0);                  /* anyone, or shared with others but not with j */
else                        /* would like to see if v is shared with the neighbour j */
   for(m = 1; m <= transfer[j][0]; m++)
      if(v == transfer[j][m]) /* yes, v is shared with the neighbour j and */
         return(m);                                  /* the relative position is m */
return(0);                              /* no, v is not shared with the neighbour j */
```

Figure 4.5: The function *Shared*().

```
for(ce = 1; ce ≤ nocrseelems; ce++) /* go through all the coarse elements */
   if(nonodessub[ce] ≥ 0)      /* ignore the element which is already gone */
      for(k = 0; k < 3; k++)            /* check if the element has vertex v
                                                              of desired type */
         if(v == the k^th vertex of the coarse element ce )
            return(1);                                  /* search is successful */
return(0);                                          /* search is not successful */
```

Figure 4.6: The function *Shared2*().

Shared() and *Shared2*() Functions

To check if it is shared before the migration we have to use the array *transfer*. The function *Shared*() shown in Figure 4.5 does this job. It returns 0 if v is not shared before the migration and returns m otherwise, where m is the relative sharing position (i.e. v == *transfer*[*j*][m]).

To check if v is shared after the migration we have to examine if it also lies on another coarse element of the subdomain *nid*, this is the job of the function *Shared2*() which is shown in Figure 4.6. This function returns 1 as soon as it discovers that even after migration of coarse element i the vertex v is still shared between both the subdomains and it returns 0 otherwise, i.e. after the migration the two subdomains do not share vertex v.

Changenbhd(), *Changenbhd2*() and *Changenbhd3*() Functions

When the element i goes from *nid* to j there are 4 possibilities concerning each vertex v of type 3 (a coarse one) or type 4 (a boundary one). These possibilities are described below.

- v is shared among the subdomains *nid* and j both before as well as after the migration. In this case we simply replace v by its counterpart in the subdomain j.
- v is shared among the subdomains *nid* and j before the migration (at the relative position m) but not after it. Here we remove it from *nid* and adjust the neighbouring relation. This is the job of the function *Changenbhd*() which is shown in Figure 4.7.
- v is not shared among *nid* and j before the migration but shared after it. Here we create a new vertex v_j in j and make necessary changes in the subdomain j. This is the job of the function *Changenbhd2*() which is shown in Figure 4.8.
- v is neither shared before nor shared after the migration. Here we create a new vertex v_j in j and make necessary changes in the subdomain j, we remove v from *nid* and make some changes in the subdomain *nid* as well. This is the job of the function *Changenbhd3*() which is shown in Figure 4.9.

4.5.2 Handling of Edges

When a coarse element goes from one subdomain to another subdomain we have to consider various issues related to the edges of the element. The following two subsections discuss this situation.

Non-Dirichlet Edge: EdgeChange() Function

Let e be a non-Dirichlet edge of the element i which is about to be migrate from subdomain *nid* (say) of type 1 to another subdomain j (say) of type 2. Then the function *EdgeChange*() shown in Figure 4.10 makes the necessary changes which are related with this migration.

1. {Check all the neighbours} Repeat the following steps for k = 0 to p - 1.
   (a) if(k == *nid*) continue,
   (b) if(k == *j*) mk = m else mk = *Shared*(),
   (c) if(mk == 0) continue,
   (d) Remove the mk^{th} vertex from its neighbouring row which corresponds to subdomain k,
   (e) Send a message to the subdomain k which will also remove the mk^{th} vertex from its neighbouring row which corresponds to subdomain nid,
   (f) transfer[*nid*][v] = transfer[*nid*][v] - 1,
   (g) if(transfer[*nid*][v] == 1) break.
2. {Remove v from *nid*}
   (a) *transfer*[*nid*][v] = - *transfer*[*nid*][0],
   (b) *transfer*[*nid*][0] = v.
3. {Update no. of boundary vertices} *nomyintf* = *nomyintf* - 1.

Figure 4.7: The function *Changenbhd*().

1. {to check if it is a newly shared vertex} if(*transfer*[*nid*][v] == 1) *nomyintf* = *nomyintf* + 1.

2. {*nid* is a new neighbour at v to *j*} *transfer*[*j*][0] = *transfer*[*j*][0] + 1.

3. {insert v in the neighbouring list which corresponds to the neighbour *j*} *transfer*[*j*][*transfer*[*j*][0]] = v.

4. {pack the ID (the integer *nid*) and the multiplicity of v (the integer *transfer*[*nid*][v]) for the neighbour *j*}

5. {initialise the variable k1} k1 = 2.

6. {Check all the neighbours of *nid* other than *j*} Repeat the following steps for k = 0 to *p* - 1.

   (a) if(k1 >*transfer*[*nid*][v]) break,

   (b) if(k == *j* or k == *nid*) continue,

   (c) mk = *Shared*(),

   (d) if(mk == 0) continue,

   (e) {pack the ID (the integer k) of the subdomain k for the subdomain *j*},

   (f) {store the information for the type 3 processor k so that it will be able to update his neighbouring list which corresponds to the subdomain *j*},

   (g) {increment the variable k1} k1 = k1 + 1.

7. {Increment the multiplicity of v} *transfer*[*nid*][v] = *transfer*[*nid*][v] + 1.

Figure 4.8: The function *Changenbhd2*().

1. {check if v is shared among *nid* and some other subdomain}

   if(*transfer*[*nid*][v] > 1) *nomyintf* = *nomyintf* - 1.

2. {pack the integer *transfer*[*nid*][v] - 1 for the subdomain *j* which will use this to make the insertions in his corresponding neighbouring lists}.

3. {check all the neighbours} Repeat the following steps for k = 0 to *p* - 1.

   (a) if(*transfer*[*nid*][v] == 1) break,

   (b) if(k == *nid* or k == *j*) continue,

   (c) mk = *Shared*(),

   (d) if(mk == 0) continue,

   (e) {store the information so that *nid* will delete latter on v from its neighbouring list which corresponds to the subdomain k},

   (f) {send the information to the subdomain k so that it will delete latter on the counter part of v from its neighbouring list which corresponds to the subdomain *nid*},

   (g) {pack the ID (the integer k) of the subdomain k for the subdomain *j*}.

4. {Remove v from *nid*}

   (a) *transfer*[*nid*][v] = - *transfer*[*nid*][0],

   (b) *transfer*[*nid*][0] = v.

Figure 4.9: The function *Changenbhd3*().

1. Pack the ID of the subdomain with which e is shared (in case e is an internal edge pack the integer -1 (which means e is an internal edge)) for subdomain j.

2. If e is not shared between any two subdomains then do the following:

   (a) Pack the necessary information for the subdomain j so that it will create and establish the necessary edge data which corresponds to the edge e,

   (b) Include e into the neighbouring list which corresponds to subdomain j and pack the necessary information for the subdomain j so that it will do the same for the the neighbouring list which corresponds to subdomain *nid*,

   (c) Pack the number of edges on the edge e and their coordinates for the subdomain j.

3. If e is shared between the subdomains *nid* and j then do the following:

   (a) Pack the relative sharing position of the edge e with the subdomain j for the subdomain j,

   (b) Store the informations for the subdomain *nid* so that it will remove e from the subdomain *nid* and also from the neighbouring list which corresponds to subdomain j,

   (c) Pack the informations for the subdomain j so that it will remove the counter part of e from the neighbouring list which corresponds to subdomain *nid*,

   (d) Update the counter for the number of vertices of type 2.

4. If e is shared between the subdomains *nid* and k ($\neq j$) then do the following:

   (a) Pack the necessary information for the subdomain j so that it will create and establish the necessary edge data which corresponds to the edge e,

   (b) Store the information so that e will be removed from the subdomain *nid* and also from the neighbouring list which corresponds to subdomain k and also send the necessary information to the subdomain k so that it will also remove the counter part of e from the neighbouring list which corresponds to subdomain *nid* and include it in the neighbouring list which corresponds to the subdomain j,

   (c) Pack the number of edges on the edge e and their coordinates for the subdomain j,

   (d) Store the information for the subdomain j so that it will insert the newly created edge into the neighbouring list which corresponds to the subdomain k,

   (e) Update the counter for the number of vertices of type 2.

Figure 4.10: The function *EdgeChange*().

1. Pack the integer representing the number of vertices and their coordinates for the subdomain *j*.
2. Update the counter for the number of vertices of type 4.
3. Remove the Dirichlet edge e from the subdomain *nid*.

Figure 4.11: The function *DirichEdgeChange*().

Dirichlet Edge: DirichEdgeChange() Function

Let e be a Dirichlet edge of the element *i* which is about to be migrate from subdomain *nid* to subdomain *j*. Since this type of edge is never shared between any pair of subdomains, it must be removed from *nid* and be inserted into *j*. The function *DirichEdgeChange*(), shown in Figure 4.11, makes the necessary changes which are related with this situation.

4.6 Different Issues Which are Related With Processors of Type 2

Recall that the type 2 processors are those processors which receive migrating elements from processors of type 1. The information about forthcoming coarse elements is received by means of packed messages. The job of this type of processor is to receive the packed message and unpack the data and modify or establish the necessary data structures. It starts creating a new coarse element by first modifying the arrays *nonodessub* and *noelemssub*. After that if there are incoming vertices of type 1 it will create the location for them and will save the corresponding coordinates. After that it will create the necessary space for the associated fine elements and move the received fine mesh into the space. It then starts observing the three coarse vertices so that the necessary neighbourhood relations with other subdomains may be established. In likewise manner it examines the three old edges one by one and establish the edge data and other neighbouring relations with other subdomains.

4.7 Different Issues Which are Related With Processors of Type 3

Recall that the type 3 processors are those processors whose data structures are affected by the migration of coarse elements from processors of type 1 to processors of type 2. The basic job of such processors are to modify the neighbourhood relations arising due to above migrations. This modification is accomplished by means of following two functions:

4.7.1 *insertion()*

The job of this function is to make required insertions in the neighbouring lists associated with other subdomains.

4.7.2 *deletion()*

The job of this function is to make required deletions in the neighbouring lists associated with other subdomains.

4.8 Use of Message Passing Interface (MPI)

The message passing paradigm which is well known and well understood has been widely used on parallel machines since their inception. However there have been many different versions of it over the years, each designed with a specific hardware in mind. This lack of standardisation used to be a major cause of not being able to produce portable software and libraries for message-passing machines. MPI (like PVM [41] which came before it) is a step forward in bringing a unified standard into the parallel community. Basically MPI is a set of routines which are useful to a wide range of users and implemented efficiently on a wide range of computers. MPI is intended to become the de facto standard, gradually replacing vendor-specific and other interfaces used by C/C++ and Fortran/Fortran90 programs today.

The MPI standardisation effort began in 1992 when an MPI Forum was established consisting about 60 people from 40 organisations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories,

and industry. In designing MPI they sought to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. By 1994 an initial MPI standard was published [34] and since that time many efficient implementations have been released for all types of parallel architecture.

Since the original publication of the MPI standard in 1994 ([34]) there have been a number of enhancements, and so this original version of the library is referred to as MPI-1.1 (which also includes a small number of clarifications and minor corrections to the original document [34]). There is now also an MPI-1.2 and an MPI-2. The former contains further clarifications and corrections to MPI-1.1 whilst the latter includes a number of significant extensions (see [35] for complete details of both MPI-1.2 and MPI-2). These extensions include: support for parallel i/o, dynamic processes, extended collective operations, and one-sided communication, as well as specific bindings for both Fortran90 and C++. (Although MPI-2 has yet to be accepted with the same enthusiasm as MPI-1 was when it was first announced.)

The implementation of the dynamic load-balancing algorithm that is used for the numerical experiments described in the next section was completed using only the MPI-1.1 version (as this was the only version available at the start of the project). Incidentally, this version is also ideally suited to the divide and conquer philosophy since it provides explicit mechanisms for the definition and splitting of processor groups.

To implement the above divide and conquer philosophy we make use of the function MPI_Comm_split() available in the MPI library. This function takes as input a communicator, a colour, and a key. All processors with the same colour are placed into the same new communicator, which is returned in the fourth argument. The processes are ranked in the new communicator in the order given by the key. In our application we assign the value 1 (value 0) if the processor is in the Sender group (Receiver group) to colour and the key is taken to be the ID (rank) of the processor.

When a coarse element migrates from the Sender group to the Receiver group we have to update numerous complicated data structures (which not only involve the processors on the Receiver and the Sender groups but also involve the processors outside these two groups), so we need to maintain the presence of the very initial group. This means each processor is a member of two groups, the initial

group (called the I_Group) which consists of all the p processors involved and which remains the same throughout the discussion and the current group (known simply as the Group) which is a variable group and changes with each application of the Divide and Conquer algorithm. This is because each application of the Divide and Conquer Approach means dividing the Group into two Groups (i.e. we essentially have the identical first and fourth parameter in the function MPI_Comm_split()).

Note that the above Divide and Conquer Approach is repeated until all the Groups have exactly one processor. If at some stage a few groups still have more than one processors in them then they will need to balance themselves; but after this rebalancing step even the groups consisting of singleton processor may have to update their data structures (if a migrated coarse element has something common with these processors), hence the singleton groups are not entirely idle at this stage.

4.9 Some Examples

In this section we describe some computations in which a parallel implementation of our dynamic load-balancing algorithm is tested and contrasted with parallel implementations of other alternative methods (which have been described and discussed in detail in §2.7.2 and §2.5 respectively): those of Vidwans *et al.* [104] and Hu and Blake [53].

Computational results which corresponds to some non-uniformly refined meshes are dynamically load-balanced on between 2 and 16 processors of the SGI Origin 2000 system. We believe that the use of more than 16 processors for solving a 2-d steady state problem is not justified on this particular machine. However in the next chapter we do use more than 16 processors to solve a time dependent problem in 3-d as the size of such problems are so huge that the use of more than 16 processors is totally justifiable.

4.9.1 Alternative Algorithms

Let us recall from §2.7.2, that our dynamic load-balancing algorithm which is described in §4.2 is an improved and modified version of the dynamic load-balancing algorithm of Vidwans *et al.* ([104]). These modifications are basically concerned in modifying and improving a number of steps proposed by Vidwans *et al.* The first

two modifications are concerned with bisecting a processor group into Sender and Receiver subgroups. In the case of the New algorithm we relax the condition that the size of these two new subgroups should be the same. Also instead of using the processors IDs we use sorted version of Fiedler Vector for the purpose of bisecting the original processor group. The final substantial modification is that the notion of gains and gain densities is used in the local migration phase of our algorithm. (Note Vidwans *et al.* propose two different methods for this purpose whose prons and cons are discussed in §2.7.2.)

In order to show the gradual improvement of these modifications we have implemented two versions of the algorithm of Vidwans *et al.* which we call VKV0 and VKV1. The VKV0 version is the original algorithm of Vidwans *et al.* ([104]). In the VKV1 version we use a slightly more sophisticated mechanism for dividing the processors into two equally-sized groups than simply using the processors IDs. This is achieved by considering the weighted partition communication graph (WPCG) (see §4.2.1) of the initial partition of the mesh $\mathcal{T}_0$. We then use a weighted version of the spectral bisection algorithm (see, for example, [46]) to order the processors (as opposed to just using their IDs) before dividing them into two groups of equal size. The algorithm then proceeds as in the VKV0 algorithm above (but with the modified scheme which uses the same concept of "gain" and "gain density" as used in the case of the New algorithm for selecting which cells to migrate at each level of the recursion). As clear from §4.10 the partitions produced by the VKV0 version is of poor quality while those produced by the VKV1 version are of comparable quality as compared to the partitions produced by the other algorithms.

The other algorithm that we use for the purposes of comparison is denoted by HB which is described by Hu and Blake in [53] (see §2.5 for details). This is not based upon recursive splitting but instead seeks to calculate the optimal route from an existing partition to one in which the weighted dual graph of the root mesh is perfectly load-balanced. For the purposes of this algorithm the term "optimal" is used to mean that the Euclidean norm of the migrated load is minimised. It should be noted that the algorithm doesn't take into account that the load on each processor comes in discrete units (i.e. the weights of the root elements) and so it may require that the weight to be transferred between two processors is a value that is not actually achievable in practice. Clearly transferring an approximation to this load is the best that can be practically achieved (and on occasions where

the root mesh is very heavily locally refined this may be far from ideal).

In the original description of the algorithm, Hu and Blake only determine the load which needs to be migrated from a set of processors to another set of processors. The question of which elements are to be migrated is not addressed there. In order to be as fair as possible in our parallel implementation of Hu and Blake's method we take great care to ensure that the particular root elements that are transferred between processors are chosen with the overall cut-weight in mind: again making use of the notion of the "gain" and "gain-density" of each node on a sending processor. Hence we are really comparing our algorithm with *improved version* of Hu and Blake's algorithm.

At this point we would like to recall from §2.6 another two important software tools which are *now* available in the public domain; namely ParJOSTLE ([109]) and ParMETIS ([63]). At the time that the work of this chapter was undertaken these had not yet been released and so we postpone comparison of the performance of our algorithm with the performance of these algorithms until the next chapter.

4.9.2 Comparative Results

We are now in a position to compare our New dynamic load balancing algorithm with the three algorithms described above. In this subsection we consider two different sets of examples, each consisting of six test problems, which are related to different geometries Ω and different root meshes. In first set the number of subdomains used are relatively small (2 and 4) whilst the second set uses relatively large number of subdomains (8 and 16). The common feature of Examples 1, 2, 3, 6, 7, 8, 9, and 10 is that these root meshes are all subject to extremely non-uniform local refinement which leads to a large proportion of the leaf elements being contained on a relatively small number of root elements (this is typical in an adaptive finite element or finite volume solver). In addition, the initial partition of the leaf mesh is not very well load-balanced but it does have a short interprocessor partition boundary (i.e. a small cut-weight). The other four problems (Examples 4, 5, 11 and 12) do not have such extreme non-uniform local refinement.

In order to demonstrate the practical utility of our algorithm we also programmed a simple parallel FE solver for the Laplace's equation subject to simple Dirichlet boundary conditions. For the purpose of comparing we attempt to find

the numerical solution to an accuracy of 4 decimal places throughout the chapter. Also all our timings are wall clock timings (as the time is calculated by using the MPI function, MPI_Wtime()).

The abbreviations used in Tables 4.1 to 4.6 and 4.8 to 4.13 have the following meanings:

- %imb - which stands for the percentage imbalance of fine elements and it is the percentage by which the total weight (number of fine elements) on the current processor exceeds the average weight of a processor (mathematically %imb = 100 * (weight - average weight) / average weight).
- crse - total number of coarse elements on the current processor.

Example 1. In this case the root mesh contains 792 elements and the leaf mesh consists of 317911 elements which are split across 2 processors. The geometry used is the "L-Shaped" domain taken from [51] and the initial partition has 198 root elements in one subdomain and 594 root elements in the other subdomain (see Figure 3.9 which shows a slightly bigger coarse mesh of same geometry). The initial maximum imbalance and cut-weight are 1.6% and 387 respectively. The initial partition is shown in Table 4.1, which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.7 and results are discussed in §4.10.

Example 2. The geometry of the mesh is the same as in the previous example, but we increase the size of the root mesh as well as the number of processors (to 4). The new mesh now has 1354 root elements and 340294 fine elements in it. The initial partition has between 76 and 661 root elements in each subdomain. The initial maximum imbalance and cut-weight are 3.2% and 1359 respectively. The initial partition is shown in Table 4.2 which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.7 and results are discussed in §4.10.

Example 3. Here the root mesh contains 1259 elements and the leaf mesh has 256440 elements which are split across 4 processors. The geometry is taken from [51]

Coarse mesh : 792 elements Final mesh : 317911 elements Average load : 158956 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	1.6	198	0.0	192	0.0	156	0.0	192	0.0	192
1	-1.6	594	0.0	600	0.0	636	0.0	600	0.0	600

Table 4.1: Data for the partitions of Example 1 (involving parallel mesh generation and repartitioning on 2 processors).

Coarse mesh : 1354 elements Final mesh : 340294 elements Average load : 85074 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	-4.1	661	0.0	670	0.0	788	0.0	670	0.0	675
1	0.6	292	0.0	330	0.0	194	0.0	291	0.0	334
2	0.2	325	0.0	286	0.2	325	0.2	325	0.0	280
3	3.2	76	0.1	68	-0.2	47	-0.2	68	0.0	65

Table 4.2: Data for the partitions of Example 2 (involving parallel mesh generation and repartitioning on 4 processors).

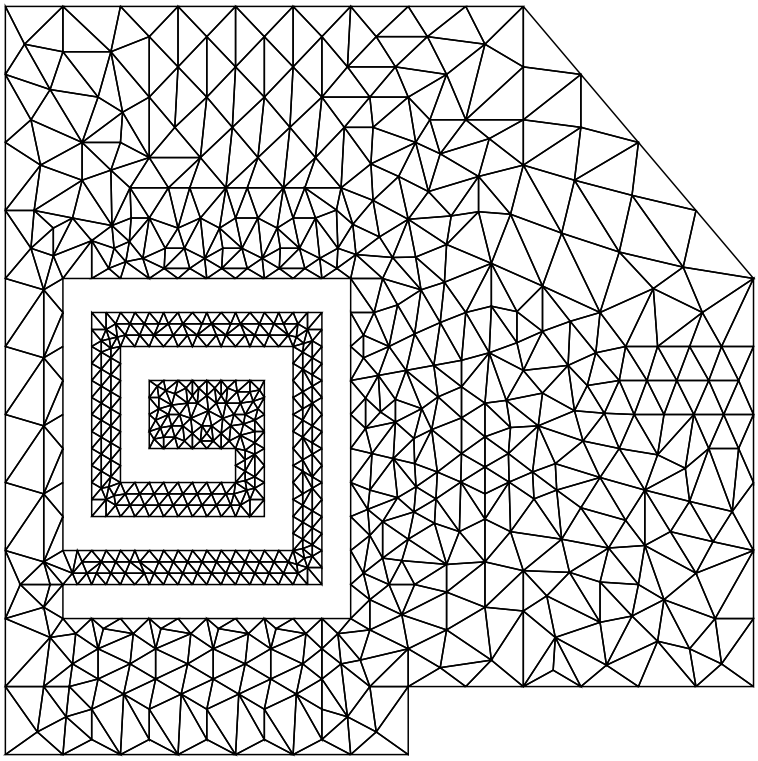

Figure 4.12: The coarse mesh of Example 3.

(geometry 2) and features a complex hole in the interior of the region. The initial partition has between 34 and 685 root elements in each subdomain (see Figure 4.12 for coarse mesh). The initial maximum imbalance and cut-weight are 3.4% and 1093 respectively. The initial partition is shown in Table 4.3 which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.7 and results are discussed in §4.10.

Example 4. Here the root mesh contains 3305 elements and the leaf mesh contains 275535 elements which are split across 4 processors. The geometry is the region around a NACA0012 aerofoil (see Figure 4.13 for a partial view of the coarse mesh around the cavity) and the initial partition has between 669 and 984 root elements in each subdomain. The initial maximum imbalance and cut-weight are 1.4% and 1104 respectively. The initial partition is shown in Table 4.4 which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.7 and results are discussed in §4.10.

Example 5. Here the root mesh contains 1210 elements and the leaf mesh contains 255093 elements which are distributed among 2 processors. The geometry used is

Coarse mesh : 1259 elements
Final mesh : 256440 elements
Average load : 64110 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	0.1	685	-0.1	698	-0.7	536	-0.7	681	-0.7	681
1	1.5	34	0.3	30	0.9	31	0.8	31	0.8	31
2	-5.0	449	-0.2	464	-0.2	650	0.0	464	0.0	464
3	3.4	91	0.0	67	0.0	42	0.0	83	0.0	83

Table 4.3: Data for the partitions of Example 3 (involving parallel mesh generation and repartitioning on 4 processors).

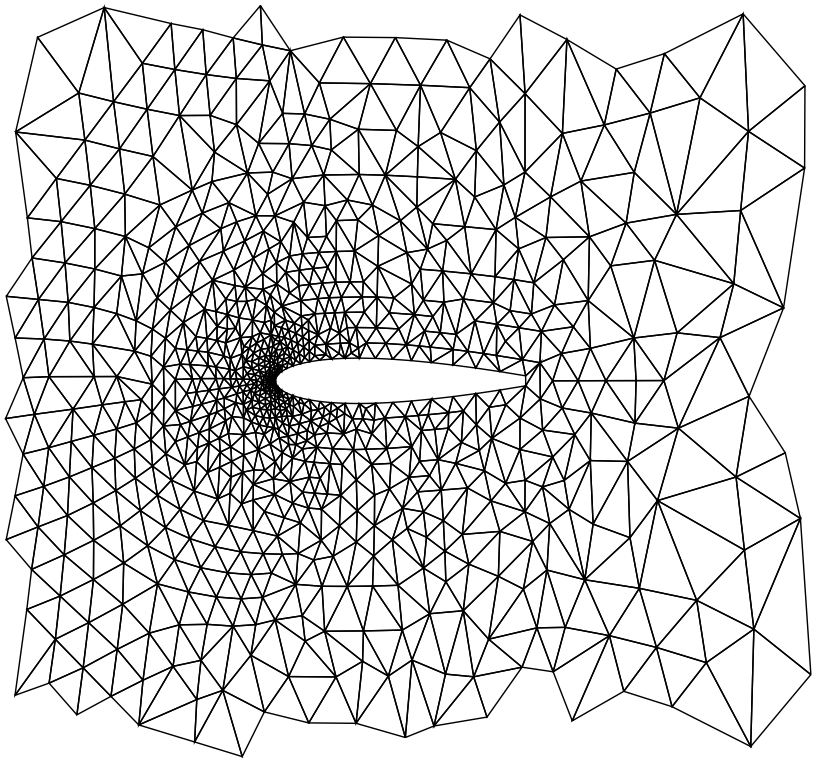

Figure 4.13: The partial view of the coarse mesh of Example 4.

Coarse mesh : 3305 elements Final mesh : 275535 elements Average load : 68884 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	-0.7	984	0.0	1068	0.0	825	0.0	982	0.0	982
1	-0.8	823	0.0	839	-0.4	1090	-0.4	832	-0.5	832
2	0.1	829	0.0	750	0.5	995	0.5	840	0.5	840
3	1.4	669	0.0	648	0.0	395	0.0	651	0.0	651

Table 4.4: Data for the partitions of Example 4 (involving parallel mesh generation and repartitioning on 4 processors).

Coarse mesh : 1210 elements Final mesh : 255093 elements Average load : 127546 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	1.3	494	0.0	488	0.0	476	0.0	488	0.0	488
1	-1.3	716	0.0	722	0.0	734	0.0	722	0.0	722

Table 4.5: Data for the partitions of Example 5 (involving parallel mesh generation and repartitioning on 2 processors).

the "Texas" domain taken from PLTMG [5] (see Figure 3.10 for the corresponding coarse mesh) and the initial partition has 494 root elements in one subdomain and 716 root elements in the other subdomain. The initial maximum imbalance and cut-weight are 1.3% and 355 respectively. The initial partition is shown in Table 4.5 which also contains the final partitions produced by all four algorithms. A summary of some of the salient features of these partitions are given in Table 4.7 and results are discussed in §4.10.

Coarse mesh : 1568 elements
Final mesh : 362329 elements
Average load : 90582 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	3.8	184	0.2	175	0.3	173	0.2	175	0.2	175
1	-2.8	504	-0.2	506	-0.2	504	-0.2	508	-0.2	508
2	3.4	258	0.1	249	0.0	241	0.0	249	0.0	249
3	-4.4	622	-0.1	638	0.0	650	0.0	636	0.0	636

Table 4.6: Data for the partitions of Example 6 (involving parallel mesh generation and repartitioning on 4 processors).

Example 6. Here the geometry of the mesh is the same as of previous example but the size of the mesh is bigger and we use more processors. Now the root mesh contains 1568 elements and the leaf mesh contains 362329 elements which are distributed among 4 processors. The initial partition has between 184 and 622 root elements in each subdomain. The initial maximum imbalance and cut-weight are 3.8% and 1141 respectively. The initial partition is shown in Table 4.6 which also contains the final partitions produced by all four algorithms. A summary of some of the salient features of these partitions are given in Table 4.7 and results are discussed in §4.10.

Example 7. Here the root mesh contains 1371 elements and the leaf mesh has 847659 elements which are split across 8 processors. The geometry is the same as of Example 3. The initial partition has between 13 and 408 root elements in each subdomain (see Figure 4.12 for coarse mesh). The initial maximum imbalance and cut-weight are 3.1% and 3329 respectively. The initial partition is shown in Table 4.8 which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.14 and results are discussed in §4.10.

Example	Feature	Initial	HB	VKV0	VKV1	New
I	MaxImb	1.6%	0.0%	0.0%	0.0%	0.0%
	CutWt	387	413	613	413	413
	G_R_Time	22.7	0.0	0.0	0.0	0.0
	SolTime	264.7	258.5	264.1	258.6	258.7
II	MaxImb	3.2%	0.1%	0.2%	0.2%	0.0%
	CutWt	1359	1523	1769	1487	1426
	G_R_Time	13.8	0.1	0.1	0.1	0.0
	SolTime	172.4	137.4	145.9	137.7	138.8
III	MaxImb	3.4%	0.3%	0.9%	0.8%	0.8%
	CutWt	1093	1107	1267	1157	1157
	G_R_Time	13.9	0.1	0.1	0.1	0.1
	SolTime	113.1	104.7	107.7	101.6	102.5
IV	MaxImb	1.4%	0.0%	0.5%	0.5%	0.5%
	CutWt	1104	1110	1265	1080	1079
	G_R_Time	19.7	0.2	0.1	0.1	0.1
	SolTime	83.4	82.8	83.1	82.9	82.8
V	MaxImb	1.3%	0.0%	0.0%	0.0%	0.0%
	CutWt	355	308	502	308	308
	G_R_Time	15.78	0.0	0.0	0.0	0.0
	SolTime	166.2	164.1	165.9	164.1	164.0
VI	MaxImb	3.8%	0.2%	0.3%	0.2%	0.2%
	CutWt	1141	1076	1537	1031	1032
	G_R_Time	13.3	0.1	0.1	0.1	0.1
	SolTime	146.5	144.3	145.9	142.2	142.8

Table 4.7: Comparison of dynamic load-balancing results using four algorithms for Examples 1 to 6.

Coarse mesh : 1371 elements
Final mesh : 847659 elements
Average load :105957 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	-3.8	408	0.0	421	-0.2	529	-0.6	416	-0.6	416
1	-2.1	385	-0.5	392	-0.2	343	-0.6	393	-0.6	393
2	-1.6	13	0.1	18	0.2	35	-0.3	13	-0.3	13
3	-0.3	19	0.4	20	0.2	37	1.4	19	1.4	19
4	3.0	368	0.0	360	-0.9	263	-0.9	364	-0.9	364
5	3.1	112	-0.2	100	1.0	83	1.0	109	1.0	109
6	1.8	28	0.3	27	0.1	26	0.2	25	0.2	25
7	-0.2	38	0.0	33	-0.1	55	-0.1	32	-0.1	32

Table 4.8: Data for the partitions of Example 7 (involving parallel mesh generation and repartitioning on 8 processors).

Example 8. The geometry of the mesh is the same as in the third and previous examples, but we increase the size of the mesh as well as the number of processors to achieve the load balance and getting the numerical solution of the above PDE. The new mesh now has 2071 root elements and 1776023 fine elements in it. The initial partition has between 4 and 643 root elements in each subdomain. The initial maximum imbalance and cut-weight are 6.9% and 8389 respectively. The initial partition is shown in Table 4.9 which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.14 and results are discussed in §4.10.

Example 9. Here the root mesh contains 4153 elements and the leaf mesh contains 566919 elements which are split across 8 processors. The geometry is the same as of Example 4. The initial partition has between 102 and 1113 root elements in each subdomain. The initial maximum imbalance and cut-weight are 2.1% and 3327 respectively. The initial partition is shown in Table 4.10 which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.14 and results are discussed in §4.10.

Example 10. We consider here the same mesh as of Examples four and nine, but with yet more elements and more processors. Now the root mesh contains 4701 elements and the leaf mesh contains 736255 elements which are distributed among 16 processors. The initial partition has between 54 and 1061 root elements in each subdomain. The initial maximum imbalance and cut-weight are 2.3% and 9340 respectively. The initial partition is shown in Table 4.11 which also contains the final partitions produced by four algorithms. A summary of some of the salient features of these partitions are given in Table 4.14 and results are discussed in §4.10.

Example 11. Here the root mesh contains 1784 elements and the leaf mesh contains 1116372 elements which are distributed among 8 processors. The geometry used is the same as of Examples five and six above. The initial partition has between 104 and 437 root elements in each subdomain. The initial maximum imbalance and cut-weight are 4.2% and 4097 respectively. The initial partition is shown in Table

Coarse mesh : 2071 elements
Final mesh : 1776023 elements
Average load : 111001 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	-0.6	643	0.0	646	-0.6	643	-0.6	643	-0.6	643
1	0.6	153	0.0	151	0.6	153	0.6	153	0.6	153
2	-0.8	140	0.0	169	-0.8	140	-0.8	140	-0.8	140
3	-1.1	483	-2.0	424	-0.3	382	0.0	483	0.0	488
4	6.9	14	2.1	9	2.3	9	2.1	10	0.1	9
5	-0.8	4	-0.8	4	1.2	6	1.4	5	0.0	5
6	-6.0	15	-1.6	26	-0.8	20	-0.9	20	-2.4	21
7	3.5	13	0.5	10	-0.5	8	-0.7	8	0.5	10
8	2.5	16	2.5	16	2.5	16	2.5	16	2.5	16
9	0.6	469	-1.3	456	-2.8	504	-2.8	462	0.0	454
10	-0.3	24	-0.1	25	-0.1	34	-0.1	25	0.0	28
11	-1.4	23	-0.1	29	-0.1	66	-0.1	26	0.2	25
12	-4.9	35	-2.0	63	-1.7	39	-1.7	39	-0.8	34
13	0.5	10	1.5	12	0.5	10	0.5	10	0.5	10
14	-3.7	13	0.6	20	0.2	23	0.3	15	0.1	17
15	5.0	16	0.6	11	0.4	18	0.3	16	0.1	18

Table 4.9: Data for the partitions of Example 8 (involving parallel mesh generation and repartitioning on 16 processors).

Coarse mesh : 4153 elements
Final mesh : 566919 elements
Average load : 70865 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	2.1	109	0.2	105	0.4	105	0.3	106	0.0	106
1	-1.5	707	-0.4	695	-0.1	571	0.0	691	0.1	698
2	0.6	102	0.6	102	0.6	102	0.6	102	0.0	126
3	-1.0	1082	-0.2	1056	-0.8	1169	-0.8	1086	-0.1	1054
4	2.0	103	0.5	100	0.1	99	0.1	100	0.1	100
5	-1.8	1113	-0.2	1147	-0.1	922	-0.2	1115	-0.1	1118
6	1.7	114	0.2	112	0.0	109	0.0	112	0.0	111
7	-2.1	823	-0.6	836	-0.1	1076	0.0	841	0.0	840

Table 4.10: Data for the partitions of Example 9 (involving parallel mesh generation and repartitioning on 8 processors).

Coarse mesh : 4701 elements
Final mesh : 736255 elements
Average load : 46016 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	0.5	55	-0.2	55	-0.4	54	-0.5	54	-0.4	54
1	1.1	73	0.2	72	1.1	73	1.1	73	0.1	72
2	2.3	103	0.2	99	0.8	98	0.9	101	0.9	101
3	-5.1	860	-0.5	892	-0.6	1040	-0.7	878	-0.3	890
4	0.4	55	0.4	55	-0.6	54	-0.6	54	0.9	55
5	1.8	71	0.9	70	0.6	69	0.5	69	0.3	70
6	2.0	102	0.2	98	0.3	95	0.5	99	0.1	100
7	-2.8	1061	-1.0	1042	0.6	919	0.6	1061	-0.8	1022
8	1.0	63	0.2	62	-0.4	63	-0.4	63	0.6	63
9	0.5	62	-0.1	61	-0.1	61	-0.3	61	-0.5	61
10	2.3	111	0.7	106	0.9	107	0.9	109	-0.1	118
11	-3.7	1056	-0.7	1072	-0.5	942	-0.6	1058	-0.3	1045
12	-0.7	54	0.1	56	-0.7	54	-0.7	54	-0.7	54
13	1.8	80	0.4	78	0.1	79	0.4	80	-0.1	78
14	1.6	101	0.0	96	-0.2	95	-0.3	97	-0.3	129
15	-2.8	794	-0.9	787	-0.7	898	-0.6	790	0.7	789

Table 4.11: Data for the partitions of Example 10 (involving parallel mesh generation and repartitioning on 16 processors).

Coarse mesh : 1784 elements
Final mesh : 1116372 elements
Average load : 139546 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	-3.6	111	0.1	117	-0.5	117	-0.4	116	-0.4	116
1	-5.8	104	-0.2	112	0.3	116	0.2	113	0.4	113
2	4.2	342	0.1	331	0.1	327	0.2	332	0.1	332
3	1.8	189	0.1	193	0.0	193	0.0	191	0.0	191
4	-2.3	144	0.0	149	0.0	151	0.0	149	0.0	149
5	-0.7	155	0.0	158	0.1	159	0.1	159	0.0	158
6	2.3	437	0.0	430	0.0	429	-0.1	431	0.0	432
7	4.1	302	-0.1	294	0.1	292	0.1	293	0.1	293

Table 4.12: Data for the partitions of Example 11 (involving parallel mesh generation and repartitioning on 8 processors).

4.12 which also contains the final partitions produced by all four algorithms. A summary of some of the salient features of these partitions are given in Table 4.14 and results are discussed in §4.10.

Example 12. Here the geometry of the mesh is the same as of Examples five, six and eleven. But the size of the mesh is bigger and we use more processors. Now the root mesh contains 3896 elements and the leaf mesh contains 2329856 elements which are distributed among 16 processors. The initial partition has between 93 and 539 root elements in each subdomain. The initial maximum imbalance and cut-weight are 3.5% and 9612 respectively. The initial partition is shown in Table 4.13 which also contains the final partitions produced by all four algorithms. A summary of some of the salient features of these partitions are given in Table 4.14 and results are discussed in §4.10.

Coarse mesh : 3896 elements
Final mesh : 2329856 elements
Average load : 145616 fine elements

Proc. Id.	Proc. Load (Initial)		Proc. Load (HB)		Proc. Load (VKV0)		Proc. Load (VKV1)		Proc. Load (New)	
	%imb	crse	%imb	crse	%imb	crse	%imb	crse	%imb	crse
0	-6.1	101	0.0	107	-0.3	112	-0.2	107	-0.2	107
1	1.8	164	-0.1	166	0.2	160	-0.1	164	0.0	164
2	-3.3	93	-0.1	99	0.0	99	0.1	99	0.0	97
3	-3.9	111	-0.2	116	-0.1	121	-0.1	115	-0.1	116
4	2.1	430	0.0	421	0.1	420	0.0	421	0.0	421
5	3.5	325	0.0	317	0.0	324	0.0	323	0.0	324
6	-0.9	173	-0.1	185	-0.1	182	-0.1	179	-0.1	179
7	1.1	241	0.0	244	0.0	250	0.0	243	0.1	244
8	-2.7	137	-0.3	142	-0.3	142	-0.1	142	-0.1	142
9	2.3	184	0.1	179	0.2	180	0.0	180	0.0	180
10	-1.7	154	0.1	158	0.0	160	0.1	160	0.0	158
11	1.4	200	0.4	199	0.0	199	0.1	197	0.1	199
12	0.3	539	0.0	537	0.3	539	0.3	539	0.3	539
13	1.9	368	0.1	361	0.6	362	0.7	364	0.7	364
14	1.4	433	0.1	425	-0.5	412	-0.4	425	-0.5	424
15	2.8	243	0.0	240	-0.3	234	-0.2	238	-0.2	238

Table 4.13: Data for the partitions of Example 12 (involving parallel mesh generation and repartitioning on 16 processors).

Example	Feature	Initial	HB	VKV0	VKV1	New
VII	MaxImb	3.1%	0.4%	1.0%	1.4%	1.4%
	CutWt	3329	3909	4215	3436	3436
	G_R_Time	30.9	0.2	0.3	0.3	0.3
	SolTime	479.2	455.8	484.6	442.3	437.5
VIII	MaxImb	6.9%	2.5%	2.5%	2.5%	2.5%
	CutWt	8389	9425	9837	9272	9049
	G_R_Time	44.9	0.9	0.9	0.8	0.8
	SolTime	592.3	583.5	588.7	532.4	558.2
IX	MaxImb	2.1%	0.6%	0.6%	0.6%	0.1%
	CutWt	3327	3235	3601	3193	3356
	G_R_Time	30.4	0.4	0.4	0.3	0.3
	SolTime	139.8	142.0	150.6	118.2	118.5
X	MaxImb	2.3%	0.9%	1.1%	1.1%	0.9%
	CutWt	9340	9379	9988	9370	9327
	G_R_Time	64.7	2.1	1.1	0.9	1.3
	SolTime	406.3	400.8	430.8	419.7	355.4
XI	MaxImb	4.2%	0.1%	0.3%	0.2%	0.4%
	CutWt	4097	3868	5199	3862	3815
	G_R_Time	19.4	0.2	0.4	0.3	0.3
	SolTime	404.3	390.9	401.7	392.6	392.4
XII	MaxImb	3.5%	0.4%	0.6%	0.7%	0.7%
	CutWt	9612	10056	12956	9677	9632
	G_R_Time	32.3	0.8	1.3	1.1	1.1
	SolTime	836.3	818.2	831.7	828.4	809.9

Table 4.14: Comparison of dynamic load-balancing results using four algorithms for Examples 7 to 12.

4.10 Discussion

For simplicity and clarity we split our discussion in two parts. Part one of the discussion corresponds to Examples 1 to 6 above, where we have only 2 or 4 processors involved. The second part of the discussion corresponds to Examples 7 to 12 above, where the number of processors involved are either 8 or 16.

4.10.1 Discussion I

The results of applying the four algorithms to each of the first six test problems are summarised in Table 4.7. In the table MaxImb stands for the maximum imbalance (see §3.5 for a quantitative definition of this). Qualitatively, this means the largest percentage by which the total weight on any single processor exceeds the average weight per processor. Also, CutWt stands for the cut-weight. As in Chapter 3 it is defined as the total weight of all of those edges of the weighted dual graph of the root mesh which are cut by the partition boundary. The entry G_R_Time means the generation or rebalance time of the corresponding mesh and SolTime means the time taken by the solver to numerically solve the simple PDE mentioned above.

We start our discussion with Examples 1 and 5. In these problems the number of subdomains are exactly 2. As expected all algorithms (except the VKV0 algorithm) produce identical results (as in the case of 2 processors all they have to do (except the VKV0 algorithm) is to shift some load from the heavily loaded processor to the lightly loaded processor using the concept of "gain density"). In the case of the VKV0 algorithm results are different as it does not use the concept of "gain density". Upon completion all four algorithms produce perfectly load balanced partitions.

There is a substantial increase in the cut weight for the VKV0 algorithm in both Examples 1 and 5 whereas for the other algorithms there is only a small increase in Example 1 and a significant decrease in the cut weight for Example 5. There is very little saving in the solution time. This is to be expected on the grounds that the initial imbalance was not that high (being less than 1.7% in both the cases). As a matter of fact the particular mesh-generator is very good in terms of producing well balanced meshes in case of two subdomains (this is why we present only two examples using 2 processors).

For other problems where we use four processors the VKV0 algorithm produces

final partitions with relatively high cut-weight. In the case of the VKV1 and New algorithms the cut-weight is relatively smaller and almost identical (except for Example 2 where cut-weight produced by the New algorithm is smallest compared to the other three). The cut-weights produced by the algorithm of Hu and Blake are in between these two extremes. Also the solutions times are all roughly same. Apart from Examples 2 and 3 the reduction in the solution time is not that significant. This can be explained by observing that although the initial imbalance in the current situation is higher than the corresponding imbalance in case of two subdomains (except Example 4 where the initial imbalance is slightly less than that of initial imbalance of Example 1) it is still not high enough to produce any significant increase in the solution time when solver is applied on the modified meshes. Also the time to rebalance the meshes is negligible, always less than or equal to 0.1 seconds (except Example 4 where the algorithm of Hu and Blake took 0.2 seconds).

4.10.2 Discussion II

The results of applying the four algorithms to each of the last six test problems are summarised in Table 4.14. All the headings in this table are exactly the same as given in Table 4.7 and also have same meanings as described in §4.10.1.

There are a number of comments which need to be made concerning these results. Firstly, the cut-weights produced by the VKV0 algorithm are higher than the corresponding cut-weights produced by the VKV1 algorithm (as a matter of fact cut-weights produced by the VKV0 algorithm are highest as compared to the cut-weights produced by all other algorithms). This clearly shows the effect of using the concept of gains in the migration phase of the VKV1 algorithm. Also, by looking at second problem (e.g. problem 8), it may appear on first inspection that all four of these techniques perform quite poorly in terms of the size of the maximum imbalance (the final maximum imbalance is well above the desired allowable target of 1% (which we maintained throughout the chapter)). This is not really the case however since the mesh refinement in this example is highly localised (as is typical in the adaptive solution of partial differential equations) and so some root elements have extremely large weights compared with others. This makes it impossible to achieve an exact load-balance in this case without increasing the cut-weight massively. However the situation in all other five problems is not so bad. In these

examples each algorithm consistently achieves a maximum imbalance of well under 1% (with two exceptions - in Example 7 a highest maximum imbalance of 1.4% is produced by both the VKV1 and New algorithms and in Example 10 both the VKV0 and VKV1 algorithms achieve a maximum imbalance of exactly 1.1%).

As mentioned above in all problems the VKV0 algorithm produces the highest cut-weight as compared to the other three algorithms. Apart from Examples 7 and 9 the New algorithm has the least amount of cut-weight. In Example 9 the VKV1 algorithm enjoys the least amount of cut-weight. In Example 7 both the VKV1 and New algorithms produce the least amount of cut-weight.

It is interesting to observe that the parallel execution times for all of these algorithms are generally quite similar with only one exception, the exception being Example 10 (which is rather surprising) in which case the algorithm of Hu and Blake is taking twice as much time as taken by other two algorithms. But in either case the cost of rebalancing the mesh is only a fraction of the cost of generating the mesh. Hence for this class of problem with these reasonably good initial partitions, it would appear that minimising data migration is not as important as obtaining a high quality partition.

A final note for this section is to analyse the parallel execution time taken by our simple solver. This is the most important parameter of any dynamic load balancing algorithm. As far as the New algorithm is concerned the net saving in solver time ranges from 3% to 15%. Except for Example 11 the SolTime of the solver using the partition of the New algorithm is less than the corresponding SolTime of the solver using the partition algorithm of Hu and Blake. The corresponding difference in time in case of Example 11 is negligible. In all cases the SolTime taken by the VKV0 algorithm is higher than the corresponding time of other algorithms.

4.11 Conclusions

In this chapter we have introduced a post-processing algorithm for the parallel generation of unstructured meshes for use in parallel finite element or finite volume analysis. The algorithm is based upon a parallel implementation of the dynamic load-balancing algorithm of Chapter 3 so as to perform a local modification of the partition of an underlying background grid from which the mesh was generated in parallel. This modification aims to improve the load-balance whilst respecting data

locality and ensuring that the length of the partition boundary is not increased unnecessarily.

We have successfully demonstrated an implementation of this algorithm in two dimensions. In addition it has been shown that the execution time of the code, implemented in C using MPI, is extremely competitive. It should be noted however that the post-processing step described here can only be as effective as the coarse mesh allows it to be. For example, if the background grid only has a small number of elements which are evenly spread across the domain and the fine mesh is very fine in some particularly local regions, then it is possible that even an optimal solution of the corresponding load-balancing problem may have a very large imbalance and/or cut-weight (e.g. Example 8 above).

As mentioned earlier, at the time of undertaking the work of this chapter no public domain dynamic load-balancing algorithm was available to compare with our algorithm. Recently, parallel versions of the publicly available software packages METIS [63] and JOSTLE [109] have been released and so it would now also be possible to make use of these within the post-processing step and compare the performance of these with the above algorithm. We have not made these comparisons however since extensive use of both of these packages is made in the next chapter in which the load-balancing algorithms are applied to a problem arising in the adaptive solution of 3-d time-dependent equations. Moreover, some further modifications to our new dynamic load-balancer have been made for this 3-d application and it is with this final version that we compare the parallel versions of METIS and JOSTLE.

Chapter 5

Parallel Application of the Dynamic Load Balancer in 3-d

As mentioned in previous chapters the objective here is to demonstrate the performance of our new dynamic load-balancing algorithm when used in conjunction with any parallel, adaptive, time-dependent, 3-d flow solver. In order to get some numerical results we decided to couple it with a particular parallel, adaptive and time-dependent solver that has recently been developed at Leeds by Selwood *et al.* ([87, 88]) which is a parallel version of a serial code also developed at Leeds by Speares and Berzins ([93]). An overview of this adaptive solver is given along with a detailed description of the application of the the new dynamic load-balancing algorithm. The effectiveness of this algorithm is then assessed when it is coupled with the solver to tackle a model 3-d flow problem in parallel. Three alternative parallel dynamic load balancing algorithms are also described and tested on the same flow problem. Perhaps we should mention here the two major differences of this solver as compared to the solver used in §4.9.2 (apart from the obvious fact that we are now solving a 3-d problem over a 3-d mesh). The first major difference is the use of *halo* objects. These halo objects in a particular subdomain are the copies of those objects which are owned by other subdomains but which share a common boundary with the current subdomain. The use of these halo objects simplifies the parallel solver, albeit at the expense of more communication and more reallocation of data objects during the execution of the dynamic load-balancing algorithm. The other major difference is that the solution technique used by this solver is that of finite volume method (see §1.3 and §5.2.2).

5.1 Introduction

As stated in previous chapter the use of multiprocessor computers for the solution of large, complex CFD and CM problems has great potential for both significant increases in mesh sizes and the significant reduction of solution times. For transient problems accuracy and efficiency constraints also require the use of mesh adaptation since solution features on different length scales are likely to evolve. Significantly, the meshes that are generally used for these problems on parallel machines are typically too large for serial adaptivity to be viable in conjunction with a parallel solver without causing a major serial bottleneck and a large communication overhead. In addition the size of the final mesh would be artificially constrained by the amount of memory available to a single processor. There is therefore a clear need for parallel adaptivity procedures to be supplied in addition to the parallel solver itself. This adaptivity should allow both the addition and deletion of degrees of freedom across the solution domain in a distributed manner, without ever requiring the entire mesh to be held on a single processor – see [56] for a discussion of some examples of such techniques.

In order for the parallel solver to perform efficiently however it is necessary that, at each stage of the solution process, the work load of each processor should be about equal (or proportional to its computational power in the case of an heterogeneous system). If this equality of load is initially achieved through appropriately partitioning the original finite element/volume mesh across the processors then it is clear that the use of parallel adaptivity will eventually cause the quality of the partition to deteriorate. For the same reasons that it is undesirable to perform mesh adaptivity on a single processor it is also undesirable to re-partition the mesh using just one processor: it would carry a large communication overhead, become a serial bottleneck and would be constrained by the amount of memory available to just one processor. Hence we again conclude that a parallel load balancing technique is required which is capable of modifying an existing partition in a distributed manner so as to improve the quality of the partition (see §5.3 below) whilst keeping the amount of data relocation as small as possible.

In this chapter we consider the dynamic load balancing problem which arises in the adaptive solution of time-dependent partial differential equations using a particular parallel adaptive algorithm based upon hierarchical mesh refinement.

This algorithm is applicable to problems in *three* space dimensions of the form

$$\frac{\partial \underline{u}}{\partial t}(\underline{x}, t) = \mathcal{L}(\underline{u}(\underline{x}, t)) \qquad \text{for } (\underline{x}, t) \in \Omega \times (0, T], \tag{5.1}$$

where $\Omega \subset \Re^3$ and $\mathcal{L}$ is some spatial operator. It is based upon the adaptive refinement of a coarse root mesh, $\mathcal{T}_0$ say, of tetrahedra which covers the spatial domain Ω. The flexibility of the data structures held within the adaptivity code (see §5.2.1 below) means that the exact nature of the parallel solver may vary (e.g. finite element or finite volume) provided it uses a tetrahedral mesh and is able to work with a partition of the elements of this mesh.

In the following section an overview of this parallel adaptive algorithm is given, along with a brief description of a particular parallel solver based upon a cell-centred finite volume scheme. In §5.3 we re-visit the dynamic load balancing problem. The final version of the parallel dynamic load-balancing algorithm is introduced in §5.4, where its implementation is also outlined. The chapter then concludes by reporting the results of a number of numerical tests which are used to contrast the four load balancing algorithms (our algorithm, the Vidwans *et al.* algorithm ([104]) and two very recent software tools for tackling this distributed problem in parallel) considered for this particular adaptive solver.

To conclude this introductory section we observe that the parallel dynamic load balancing problem addressed in this chapter can arise in numerous other contexts in parallel computational mechanics. As well as the use of local h-refinement, other algorithms which permit the distribution of the computational load across the domain Ω to vary as the simulation proceeds will require a dynamic load balancing strategy. This includes algorithms based upon p-refinement (see [24, 27] for example) or those for solving systems, such as those arising in phase-change problems for example (e.g. [4]), in which the computational nature of the solution can change with time at each point in Ω. Another important situation in which dynamic load balancing often arises is when a parallel code is executed on an heterogeneous network (such as a cluster of workstations for example), in which the performance of each processor may vary with time as its load increases or decreases. All of these situations may be treated by the algorithms discussed in §5.3 and §5.4 below however, for the sake of clarity, we now restrict all further discussion, examples and comparisons to the three-dimensional h-refinement code described in the following section.

5.2 A Parallel Adaptive Flow Solver

5.2.1 A Parallel Adaptive Algorithm

The software outlined in this subsection (called PTETRAD) was written by Selwood *et al.* [87, 88] and is based upon a parallel implementation of a general purpose serial code, TETRAD (TETRahedral ADaptivity), for the adaptation of unstructured tetrahedral meshes [93]. The technique used is that of local refinements/derefinements of the mesh to ensure sufficient density of the approximation space throughout the spatial domain, Ω, at all times. A more complete discussion of the parallel algorithms and data structures may be found in [87, 88, 89, 100].

Data structures

One of the major issues involved in parallelising an adaptive code such as TETRAD is how to treat the existing data-structures. TETRAD utilises a complex tree-based hierarchical mesh structure, with a rich interconnection between mesh objects. Figure 5.1 indicates the mesh object structures used in TETRAD. In particular, note that the main connectivity information used is 'node to element' and that a complete mesh hierarchy is maintained by both element and edge trees. Furthermore, as the meshes are unstructured, there is no way of knowing *a-priori* how many elements share any given edge or node.

For parallelisation of TETRAD, there are two main data-structure issues. The first is how to partition a hierarchical mesh, the second is that specific new data-structures are required to support parallel partitioning of the mesh.

1. As we first discussed in §2.3 there are two options for partitioning a hierarchical mesh. The first is to partition the grid at the root or coarsest level, $\mathcal{T}_0$. This has a number of advantages. The local hierarchy is maintained on a processor and thus all parent/child interactions (such as refinement/derefinement) are local to a processor. The partitioning cost will also be low, as the coarse mesh is generally quite small. The main disadvantage of this approach however is that, for comparatively small coarse meshes with large amounts of refinement, it may be difficult to get a good partitioning, both in terms of load balance and communication requirements.

   The other main approach is to partition the leaf-level mesh, i.e. the actual

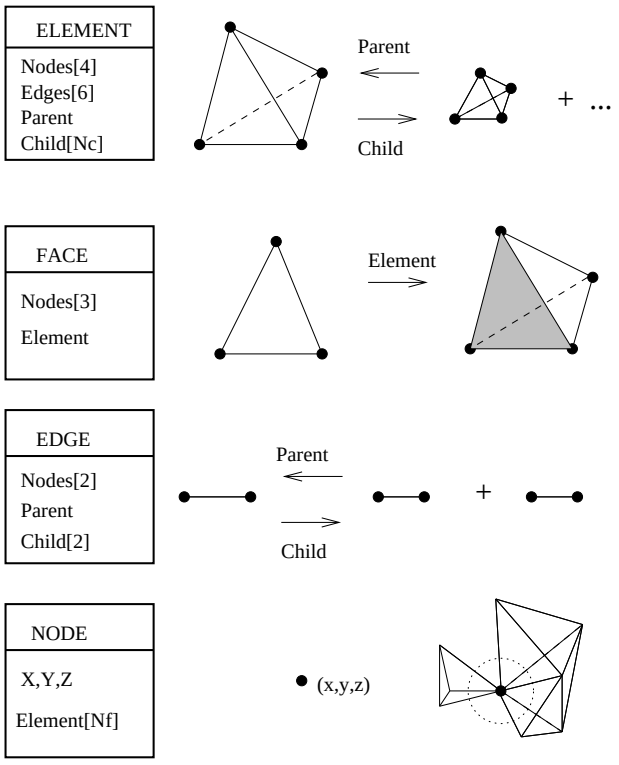

Figure 5.1: Mesh data-structures in TETRAD

computational grid. The pros and cons of this approach are the opposite of those with the coarse level partitioning. In particular, the quality in terms of load balance and cut-weight of the partition is likely to be better, albeit at the expense of a longer partitioning time. However, the data-structures have to be more complicated as hierarchical operations, such as multigrid V-cycles and derefinement for example, are no longer necessarily local to a processor (and are therefore likely to be slower).

The approach taken for parallelising TETRAD is that of partitioning the coarse mesh. The only disadvantage of this, that of possible suboptimal partition quality, can be avoided if the initial, coarse mesh is scaled as one adds more processors.

2. Given a partitioned mesh, data-structures are required in order to support inter-processor communication and to ensure data consistency. Data consistency is handled by assigning ownership of mesh objects (elements, faces, edges and nodes). As is common in many solvers such as those used by [13, 87] PTETRAD uses halo elements: these are copies of inter-processor boundary

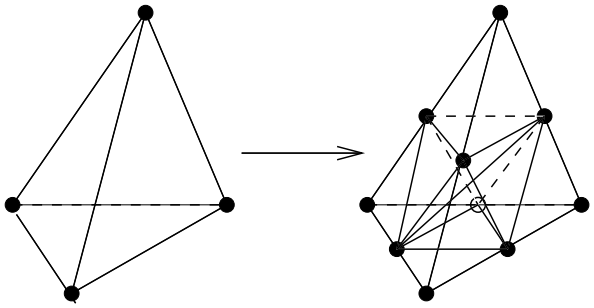

Figure 5.2: Regular refinement dissecting interior diagonal

elements (with their associated data) used to reduce communication overheads. In order to have complete data-structures (e.g. elements have locally held nodes) on each processor, halo copies of edges, nodes and face objects are also used. If a mesh object shares a boundary with many processors, it may have a halo copy on each of these. All halos have the same owner as the original mesh object. In situations where halos may have different data than the original, the original is used to overwrite the halo copies and thus is definitive. This is used to help prevent inconsistency between the various copies of data held.

Adaptivity Algorithms

Both TETRAD [93] and its parallel implementation, PTETRAD [87, 88], use a similar strategy to that outlined in [70] to perform adaptivity. Edges are first marked for refinement/derefinement (or neither) according to some estimate or indicator (provided as part of the parallel solver: see §5.2.2 below for example). Elements with all edges marked for refinement may then be refined regularly into eight children. To deal with the remaining elements which have one or more edge to be refined so-called "green" refinement is used. This places an extra node at the centroid of each element and provides a link between regular elements of differing levels of refinement. The types of refinement are illustrated in Figures 5.2 and 5.3. An important restriction that is made is that green elements may not be further refined as this may adversely affect mesh quality ([78]). Instead, they are first removed and then uniform refinement applied to the parent element.

Immediately before the refinement of a mesh, the derefinement stage occurs. This may only take place when all edges of all children of an element are marked

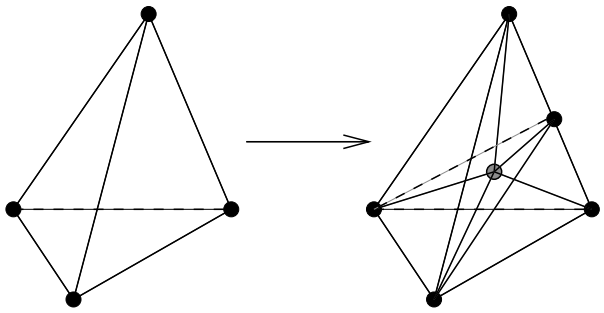

Figure 5.3: Green refinement by the addition of an interior node

for derefinement and when none of the neighbours of an element to be deleted are green elements or have edges which have been marked for refinement. This is to prevent the deleted elements immediately being generated again at the refinement stage which follows. A further necessary constraint is that no edges or elements at the coarsest level, $\mathcal{T}_0$, may be derefined.

For further details of the implementation of these adaptive algorithms using MPI ([34]) please refer to [87, 88]. These papers discuss important issues such as performing parallel searches in order to allow refinement of edges of green elements (which requires coarsening followed by regular refinement), maintaining mesh consistency and dealing with halo data in parallel.

5.2.2 A Parallel Finite Volume Solver

In order to apply the above adaptive algorithm to systems of PDEs of the form (5.1) a parallel solver is also required. The data structures supported by TETRAD have been used with both finite element and finite volume solvers (cell-centred and cell-vertex), however in the examples used in this chapter numerical experiments are based only around a cell-centred finite volume scheme.

The scheme used is applicable when (5.1) represents a system of hyperbolic conservation laws of the form

$$\frac{\partial \underline{u}}{\partial t} + \frac{\partial \underline{F}(\underline{u})}{\partial x} + \frac{\partial \underline{G}(\underline{u})}{\partial y} + \frac{\partial \underline{H}(\underline{u})}{\partial z} = 0 \ , \tag{5.2}$$

such as the three-dimensional Euler equations for example, and is a parallel version of the algorithm described in detail [93]. This is a conservative cell-centred scheme which is a second-order extension of Gudunov's Riemann problem-based scheme ([39]), using MUSCL-type piecewise linear reconstructions of the primitive variables

within each element ([102]). Although the time-stepping is explicit it is executed in two distinct phases: a non-conservative predictor-type update (referred to in [102] as the "Hancock step") followed by a second half-time-step based upon the application of the underlying conservation law. Implicit in this numerical method is the need to solve a Riemann problem at each element interface at each time-step – although this is only done approximately using a modified form of the approximate solver described in [42].

The parallel version of the solver was also implemented by Selwood and Berzins ([87]). This implementation was quite straightforward due to the face data structure that exists within the adaptivity software (see Figure 5.1 for example). To avoid any conflicts at the boundary between two subdomains a standard "owner computes" rule is used for each of the faces when solving the approximate Riemann problems to determine fluxes. The use of halo elements ensures that the owner of each face has a copy of all of the data required to complete these flux calculations provided the halo data is updated twice for each time-step (i.e. immediately before the Hancock step and then again before the second half-time-step).

5.3 Dynamic Load Balancing

As explained in §5.2.1 above parallel solvers such as PTETRAD require the computational domain to be partitioned into subdomains. In the case of PTETRAD this partition should be applied to the coarse root mesh $\mathcal{T}_0$ (as discussed above in §5.2.1). It is usual to express the requirements of such a partition in terms of the weighted dual graph of this mesh. Let us recall from §2.3 the definition of the weighted dual graph - for each element, i, of the root mesh define a corresponding vertex of the dual graph and let this vertex have weight v_i, where v_i is the number of leaf-level elements of the current mesh which lie within root element i. For each pair of face adjacent elements in the root mesh define an edge, j, of the dual graph and let this edge have weight e_j, where e_j is the number of pairs of leaf-level elements in the current mesh which meet along face j. We may also recall from §2.3 that, for a homogeneous network of processors, we would like to be able to partition this graph so that at all times the following four conditions are satisfied:

1. the total vertex weight in each subgraph is approximately equal,

2. the total cut-weight of the partition is kept to a minimum,
3. there is a minimal amount of migration of data between subgraphs,
4. the load balancing should be completed in parallel.

Note that the first two constraints on the partition of $\mathcal{T}_0$ (or its dual graph) should hold at each time-step. However, when parallel adaptivity occurs it is likely that the weights v_i and e_j will change. In particular, changes in the vertex weights v_i are liable to cause an existing well-balanced partition of $\mathcal{T}_0$ to become unbalanced. The objective of a dynamic load balancing algorithm is to modify an existing, inadequate, partition of the dual graph so as to meet objectives 1 and 2 above but in such a way that the last two constraints are also satisfied. The motivation behind third requirement is simply that there is a significant communication overhead associated with moving data between processors and this overhead should not be allowed to nullify the computational advantages of obtaining an improved partition. And without the last requirement there would be a sequential bottleneck in the whole solution procedure at the load balancing stage which could seriously reduce the overall efficiency and performance of the adaptive parallel solver.

5.4 Application of the Parallel Dynamic Load-Balancing Algorithm

In this section we are going to modify the parallel dynamic load-balancing algorithm already introduced in Chapters 3 and 4. These modifications are required to add generality to the code due to the different nature of solver as well as underlying meshes. This modified algorithm is similar in strategy to that of Chapter 4 with Group Balancing, Data Migration and a Divide and Conquer philosophy. However, as far as the implementation is concerned there are major differences between this and Chapter 4. We now discuss these differences in details.

5.4.1 Calculation of WPCG

The idea of the Weighted Partition Communication Graph (WPCG) was first introduced in §2.4 and since then has been used extensively in the previous two chapters. Let us recall from §2.4 that a WPCG is obtained by having one vertex for every

processor and an edge between two vertices if and only if they are face adjacent to each other. Although the definition of WPCG is quite standard the actual determination may vary from one context to another. In the present situation the concept of 'halo' elements together with the hierarchical nature of the mesh plays an important role in the determination of the WPCG. During the bisection process of the WPCG it is necessary to calculate the weighted Laplacian of WPCG as well as the weights of all the vertices of WPCG. This is done in two phases. In phase one, which is shown in Figure 5.4, we calculate the weights of all the vertices and edges of the dual graph with the help of the c-style function FindWeights(). In phase two, which is shown in Figure 5.5, we use these edge weights to find the weighted Laplacian of the WPCG. It is interesting to observe the recursive nature of the function FindWeights() shown in Figure 5.4 (the very first call to this function is of the form FindWeights($Element$,$Element$,$WeightOfElement$,$WeightOfElementEdge$)). It may also be observed that as the original mesh is already distributed across a number of processors of a parallel machine, the calculations in Figure 5.4 are performed by each processor on those roots elements which it currently owns. Also, the k^{th} row of the WPCG is assembled by the k^{th} processor with the help of Figure 5.5 (note that in this figure the weight of j^{th} element face is the same as the weight of the corresponding edge in the dual graph). Just like in §4.2.1 each processor, after assembling its own row, sends it to one processor (which we call a master processor). The master processor, after receiving all the contributions from all other processors, forms the Laplacian and then divides the WPCG into two subgroups denoted by Group1 and Group2 using the same procedure as in §3.2.1.

5.4.2 Use of Tokens

At each level of the recursive re-balancing algorithm, after deciding how much to shift between the different processors, we face the same practical difficulties as encountered in the previous chapter. However to overcome these difficulties we take a different approach to that taken in the previous chapter. The first difference, which is a major one, is concerned with passing the information associated with moving a coarse element. In the previous chapter, whenever a coarse element goes from one subdomain to another subdomain all of its associated data structures are sent, even if the new home is only a transitory one. In the current implementation

```
FindWeights(ElmC,Elm,WeightOfElm,WeightOfElmEdge[]){
    if (ElmC has no child){
      WeightOfElm++;
      if (ElmC has a face which is contained in j^th face of Elm)
        WeightOfElmEdge[j]++;
    }
    for(i = 0; i < Children of ElmC; i++) {
      ElmC2 = i^th child of ElmC;
      FindWeights(ElmC2,Elm,WeightOfElm,WeightOfElmEdge[]);
    }
}
```

Figure 5.4: Calculation of weights of vertices and edges of the weighted dual graph.

the communication of the full coarse element hierarchies is left until the very end of the load-balancing process, with much smaller tokens being passed instead during the transitory stages. There are many reasons for doing this. One reason is that in the previous chapter the load-balancing algorithm was used only once, usually at the end of the generation of the mesh and before the solution process commences, but in the current context it can be used possibly after each adaptation step (in case the resulting imbalance is greater than a predefined tolerance). The second reason is that in the 2-d application the size of the accompanying data structure of a moving element is much smaller than the corresponding size in this 3-d application. In fact the re-balancing times in all the problems of Chapter 4 were less than a second (except problems 10 and 12 where it was slightly higher than a second), showing that in 2-d steady state cases moving all the information associated with a migrating coarse element is indeed not that costly.

5.4.3 No Colouring

The other major difference between the current implementation and that of the previous chapter is to drop the colouring methodology. Note that the colouring scheme in the 2-d case was implemented in order to avoid the complication involved in the simultaneous migration of two neighbouring elements (without it, complicated

```
for(e = 1; e ≤ total no. of root elements; e++ ){
   Weight of the k^th vertex of WPCG = Weight of the root element e;
   if (element e is not halo)
     for(j = 1; j ≤ 4; j++)
        if (j^th face of element e touches the boundary of i^th processor) {
           Lapr[i] -= weight of j^th element face;
           Lapr[k] += weight of j^th element face;
        }
}
```

Figure 5.5: Calculation of a row of the weighted Laplacian matrix.

forwarding messages would have been required to accomplish the same task). The decision to drop the colouring approach in the current context is based upon the fact that in the 3-d setup it is computationally more expensive to implement. In 2-d the average number of colours used was never more than 10. But in 3-d a given vertex may have 100 (or even more) common tetrahedral elements. So the colouring scheme would almost certainly adversely affect the performance of the dynamic load balancer. An alternative to the colouring approach is discussed below.

5.4.4 Use of Global Communication

Rather than use the colouring scheme introduced in Chapter 4 in order to to avoid data conflicts (see §4.3) we decided to make use of global arrays, whereby each processor knows the owners of every coarse element in the entire coarse mesh. These arrays are updated after each level of the recursive step. Note that each level of the algorithm starts with a given group. Based upon the sorted version of the Fiedler vector it is divided into two subgroups which are called Sender and Receiver groups respectively. A certain number of coarse elements are marked for migration from the Sender to the Receiver groups in an attempt to balance the computational load evenly.

Initially each processor in a group is assigned as the owner of all non-halo coarse elements which reside within the processor: the unique owner for each coarse element being the ID of the processor itself. Soon after these assignments, which

are local to each processor, the list of elements owned by each processor is broadcasted globally within the I_Group (let us recall from §4.8 that each processor is a member of two groups, the initial group (called the I_Group) which consists of all the p processors involved and which remains the same throughout the discussion and the current group (known simply as the Group) which is a variable group and changes with each application of the Divide and Conquer algorithm). As a result of this broadcast every processor now knows the owner of every coarse element in the entire mesh. When a coarse element in the Sender group is marked for a migration, the owning processor of the coarse element keeps a note of this change (i.e. it records the new owner of the coarse element (which is the ID of some processor in the Receiver group)).

At the end of this symbolic migration step each candidate processor in the Sender group broadcasts globally within the I_Group the new owners of those coarse elements which are marked for migration from the processor. As a result of this broadcast every processor knows not only the IDs of those processors from which there would be migration of coarse elements but they also know the new owners of these coarse elements. Soon after the broadcast each processor updates its own version of the array which keeps track of the owner of all the coarse element in the entire coarse mesh.

It may be pointed out here the this broadcasting step consists of two separate steps. In the first step each candidate processor in the Sender group broadcasts only the number of coarse elements marked for the migration. In the second step it broadcasts the new owners of these coarse elements. This division of the broadcasting was necessary. The first step is necessary for the second step. After the first step each processor can create the necessary temporary arrays in order to accommodate the new owners of the marked coarse elements (which will be broadcast in the second step).

An overview of the whole algorithm is given in Figure 4.3.

5.5 Computational Results

We now present some computational results produced by the new dynamic load-balancing algorithm when used in conjunction with Selwood's parallel adaptive flow solver outlined in §5.3. We also compare our results with the results produced by the

original algorithm of Vidwans *et al.* (for details see [104] and §2.7.2) which uses the grid-connectivity-based approach for the purpose of the migration of nodes and two publically available software tools; namely the ParJOSTLE ([107]) and ParMETIS ([63]) algorithms. It should be noted that this flow solver requires a partition of the root mesh, $\mathcal{T}_0$, such that the total number of leaf-level elements on each processor is approximately equal. When there is heavy local refinement in some regions of the spatial domain Ω (as in the examples below) it follows that the dual graph of $\mathcal{T}_0$ will have highly disparate weights. Hence, in this chapter we are only testing the performance of the dynamic load balancing algorithms for one specific class of problem: the repartitioning of highly non-uniformly weighted graphs.

5.5.1 Examples

For these examples we apply the parallel adaptive Euler solver to model a shock wave diffraction around the 3-d right-angled corner formed between two cuboid mesh regions (taken from [88, 93]). The initial condition is of Rankine-Hugoniot shock data at the interface of the two cuboid regions with a shock speed of Mach 1.7. Figures 3.11 and 3.12 illustrate how the mesh adapts to the solution as the shock progresses through the domain. It is clear that although a partition of the mesh for the initial condition may be good, it is unlikely to remain so as the solution develops and thus dynamic load balancing of the distributed data will be required. It should be noted that for all the calculations described below, the ParJOSTLE algorithm was used with its graph reduction threshold parameter set to 300 (see [87] where Selwood and Berzins use the same value) which appeared to give consistently better results than either of the other values tried: 20 (the default) and 50. All other parameters in both the ParJOSTLE and ParMETIS algorithms were left at their default values. It should be observed that in the case of Example 1 the ParMETIS and Vidwans *et al.* algorithms were unable to produce any results when we used 32 processors (in such cases the ParMETIS algorithm was showing the message 'Too much suppression' and the results were absurd). Very recently (and after the completion of this chapter) two alpha versions of the ParMETIS 2.0 algorithm have been released which we hope would produce satisfactory results in this situation (at the time of writing we have not yet experiment with these however). The abbreviations used in Tables 5.1 to 5.8 have the following meanings:

- MaxImb - which stands for maximum imbalance and it is the largest percentage by which the total weight on any single processor exceeds the average weight per processor (see §3.5 for quantitative definition).

- CutWt - which stands for cut-weight and is defined as the sum of the weights of all those edges in the weighted dual graph which cross between two different subdomains (see Chapter 3).

- SolTime - which stands for the solution time and represents the wall-clock time (in seconds) taken by the parallel finite volume solver, either using the initial partition or using a new partition after application of one of the dynamic load-balancing algorithms.

- RedTime - which stands for the repartitioning and redistribution times and represents the wall-clock time (in seconds) that is required in calculating the new partitioning vector and redistribution (i.e. copying) of mesh objects across the machine as a result of applying one of the dynamic load-balancing algorithms.

- Migration - total number of fine elements which are migrated by one of the dynamic load-balancing algorithms.

- MigFreq - which stands for the migration frequency and represents the number of times the repartitioning needed to be undertaken throughout the 300 time-steps as a result of applying one of the dynamic load-balancing algorithms (note the maximum value of MigFreq is 10 here).

Example 1. In the computations whose results are tabulated in Tables 5.1 to 5.4 the root mesh, $\mathcal{T}_0$, contained 5148 elements (see Figures 3.11 and 3.12 which illustrate how the mesh adapts to the solution as the shock progresses through the domain). Up to three levels of refinement are allowed which leads to an initial fine mesh containing between 84446 to 91008 elements depending upon the number of processors used (it is interesting to observe that the initial size depends upon the number of processors used due to a variable number of green elements at the boundary which depends on the number of processors used!) with many more elements appearing

in this leaf-level mesh at later times (or there may be a drop in the size in case of heavy coarsening). Note that throughout these calculations, the adaptive mesh has resolution equivalent to a mesh of 2.6 million uniform, regular elements.

Table 5.1 presents a comparison of some partition-quality metrics when four different dynamic load-balancing algorithms are applied using 2, 4, 8, 16 and 32 processors of an SGI Origin 2000 computer. In each case the initial partition has a maximum imbalance of over 32%. The solution times quoted represent the wall-clock time (in seconds) taken by the parallel finite volume solver for the next 30 time-steps, either using the initial partition or using a new partition after application of one of the dynamic load-balancing algorithms. Finally, when load-balancing has been performed, the total weight of all of the root elements of $\mathcal{T}_0$ that have been migrated from one processor to another is quoted.

An alternative form of comparison between the four dynamic load-balancing algorithms is provided by Tables 5.2, 5.3 and 5.4. For these results sequences of 300 time-steps were taken with adaptivity taking place on up to ten occasions (after every 30 time-steps). Whenever the maximum imbalance exceeds a prescribed tolerance (which is 5% for Table 5.2, 10% for Table 5.3 and 15% for Table 5.4) after mesh adaptivity has taken place the dynamic load-balancing algorithm is called. The solution times quoted are the total times for the finite volume solver to complete the 300 time-steps *excluding* the repartitioning and redistribution times (which are also quoted separately). This gives an indication of the quality of the dynamic load-balancing algorithm. As additional, architecture independent, comparison of their overheads these tables also show the total weight of all of the root elements that were migrated throughout the 300 time steps (Migration) as well as the number of times that repartitioning needed to be undertaken (MigFreq). See §5.6 for a discussion of these results.

Example 2. In the computations whose results are tabulated in Tables 5.5 to 5.8 the root mesh, $\mathcal{T}_0$, contained 34560 elements (this is a little more than for the illustrative examples shown in Figures 3.11 and 3.12). Up to three levels of refinement are allowed which leads to an initial fine mesh containing between 291094 to 304186 elements depending upon the number of processors used (again the initial size depends upon the number of processors used due to a variable number of green elements at the boundary) with many more elements appearing in this leaf-level

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	2				
MaxImb	40%	5%	0%	0%	0%
CutWt	958	1709	959	2387	2127
SolTime	126.0	96.4	91.3	95.2	93.8
Migration	–	17909	44955	16780	16786
Processors	4				
MaxImb	34%	4%	1%	0%	0%
CutWt	1720	2423	2399	3958	2791
SolTime	62.0	48.3	46.4	47.4	46.6
Migration	–	18831	27122	23750	34445
Processors	8				
MaxImb	33%	5%	3%	2%	0%
CutWt	3064	4139	4129	5634	4905
SolTime	32.0	25.3	26.2	27.5	27.3
Migration	–	21435	44982	26569	25182
Processors	16				
MaxImb	55%	9%	9%	2%	4%
CutWt	4760	6439	5525	8010	7510
SolTime	17.0	12.1	12.0	12.1	11.9
Migration	–	35001	63580	31507	30424
Processors	32				
MaxImb	143%	–	42%	–	24%
CutWt	5616	–	7103	–	8959
SolTime	11.2	–	6.7	–	6.2
Migration	–	–	43884	–	36714

Table 5.1: Some partition-quality metrics immediately before and after a single re-balancing step for Example 1.

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	2				
SolTime	1054.0	848.6	811.1	808.3	818.9
RedTime	–	9.3	26.5	21.8	30.3
Migration	–	17909	57892	18891	21560
MigFreq	–	1	2	2	3
Processors	4				
SolTime	518.3	413.9	406.6	420.9	408.4
RedTime	–	32.8	18.5	14.1	11.8
Migration	–	30296	65719	25222	40382
MigFreq	–	7	3	2	2
Processors	8				
SolTime	276.4	205.3	214.5	239.2	234.8
RedTime	–	17.3	56.7	27.7	33.8
Migration	–	48636	309312	35063	36976
MigFreq	–	4	7	5	7
Processors	16				
SolTime	146.2	107.0	108.9	124.8	113.9
RedTime	–	34.8	51.4	43.1	28.1
Migration	–	133303	498071	57339	47317
MigFreq	–	10	10	10	7
Processors	32				
SolTime	104.3	–	66.5	–	71.3
RedTime	–	–	36.9	–	32.1
Migration	–	–	263986	–	69507
MigFreq	–	–	10	–	10

Table 5.2: Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 5% for Example 1.

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	2				
SolTime	1054.0	848.6	810.9	833.2	826.3
RedTime	–	9.3	12.3	12.9	22.7
Migration	–	17909	44955	16780	20949
MigFreq	–	1	1	1	2
Processors	4				
SolTime	518.3	416.6	414.8	429.3	408.4
RedTime	–	15.6	11.6	8.8	11.8
Migration	–	27616	48654	23750	40382
MigFreq	–	3	2	1	2
Processors	8				
SolTime	276.4	206.1	212.3	240.9	235.5
RedTime	–	11.5	14.7	16.6	16.4
Migration	–	26948	95260	32477	34547
MigFreq	–	3	2	3	3
Processors	16				
SolTime	146.2	109.1	110.2	125.8	115.0
RedTime	–	21.4	30.9	22.3	16.1
Migration	–	103898	294794	50006	46165
MigFreq	–	5	6	5	4
Processors	32				
SolTime	104.3	–	64.2	–	69.7
RedTime	–	–	32.3	–	33.0
Migration	–	–	251810	–	68871
MigFreq	–	–	10	–	10

Table 5.3: Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 10% for Example 1.

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	2				
SolTime	1054.0	848.6	810.9	833.2	831.7
RedTime	–	9.3	12.3	12.9	12.5
Migration	–	17909	44955	16780	16786
MigFreq	–	1	1	1	1
Processors	4				
SolTime	518.3	432.1	414.7	429.3	408.4
RedTime	–	17.4	11.7	8.8	11.8
Migration	–	29258	48654	23750	40382
MigFreq	–	3	2	1	2
Processors	8				
SolTime	276.4	224.7	219.6	222.6	215.1
RedTime	–	8.9	15.4	13.2	15.2
Migration	–	25407	102105	32268	35409
MigFreq	–	2	2	2	3
Processors	16				
SolTime	146.2	112.8	113.8	124.1	115.4
RedTime	–	14.9	14.7	17.8	11.9
Migration	–	91719	164763	46843	44615
MigFreq	–	3	3	4	3
Processors	32				
SolTime	104.3	–	66.1	–	71.2
RedTime	–	–	34.3	–	23.1
Migration	–	–	238785	–	63810
MigFreq	–	–	10	–	6

Table 5.4: Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 15% for Example 1.

mesh at later times (or there may be a drop in the size in case of heavy coarsening). Note that throughout these calculations, the adaptive mesh has resolution equivalent to a mesh of 17.7 million uniform, regular elements.

Table 5.5 presents a comparison of some partition-quality metrics when the four different load-balancing algorithms are applied using 2, 4, 8, 16 and 32 processors of an SGI Origin 2000 computer. In each case the initial partition has a maximum imbalance of over 28%. The solution times quoted represent the wall-clock time (in seconds) taken by the parallel finite volume solver for the next 30 time-steps, either using the initial partition or using a new partition after application of one of the dynamic load-balancing algorithms. Finally, when load-balancing has been performed, the total weight of all of the root elements of $\mathcal{T}_0$ that have been migrated from one processor to another is quoted. On this occasion the ParMETIS and Vidwans *et al.* algorithms did work satisfactorily (due to there being a larger coarse mesh).

An alternative form of comparison between the four load-balancing algorithms is provided by Tables 5.6, 5.7 and 5.8. For these results sequences of 300 time-steps were taken with adaptivity taking place on up to ten occasions (after every 30 time-steps). Whenever the maximum imbalance exceeds a prescribed tolerance (which is 5% for Table 5.6, 10% for Table 5.7 and 15% for Table 5.8) after mesh adaptivity has taken place the dynamic load-balancing algorithm is called. The solution times quoted are the total times for the finite volume solver to complete the 300 time-steps *excluding* the repartitioning and redistribution times (which are also quoted separately). This gives an indication of the quality of the dynamic load-balancing algorithms. As additional, architecture independent, comparison of their overheads these tables also show the total weight of all of the root elements that were migrated throughout the 300 time steps (Migration) as well as the number of times that repartitioning needed to be undertaken (MigFreq). Note that Table 5.8 has no entries which corresponds to two or four processors as these entries are identical to the corresponding entries of Table 5.7 (as a matter of fact if tolerance is ≥ 10 the load-balancing algorithm is called only at the beginning of the whole process, provided the number of processors are ≤ 4). See §5.6 for a discussion of these results.

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	2				
MaxImb	29%	4%	0%	0%	0%
CutWt	1660	1626	1793	5456	3358
SolTime	417.2	355.7	375.4	376.8	345.3
Migration	–	53844	73808	44539	44539
Processors	4				
MaxImb	43%	5%	1%	0%	0%
CutWt	3334	5155	3873	9637	7815
SolTime	251.7	181.4	185.7	191.4	192.2
Migration	–	79133	75248	70754	78642
Processors	8				
MaxImb	48%	5%	2%	0%	2%
CutWt	6776	8699	7156	15947	11395
SolTime	139.1	87.7	91.6	90.2	91.5
Migration	–	83249	220105	89811	77261
Processors	16				
MaxImb	91%	7%	2%	0%	1%
CutWt	9183	12056	10719	20490	16027
SolTime	97.3	48.1	42.4	44.6	42.3
Migration	–	131248	192875	115058	117723
Processors	32				
MaxImb	96%	9%	9%	2%	2%
CutWt	12875	16162	15009	23457	22950
SolTime	42.5	23.8	24.7	24.7	23.8
Migration	–	134191	176209	129931	128408

Table 5.5: Some partition-quality metrics immediately before and after a single re-balancing step for Example 2.

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	2				
SolTime	3013.4	2920.2	2873.7	2876.1	2567.6
RedTime	–	52.8	34.7	44.7	28.8
Migration	–	60242	73808	44539	44539
MigFreq	–	2	1	1	1
Processors	4				
SolTime	1856.7	1495.1	1260.1	1314.0	1323.9
RedTime	–	25.8	42.1	128.6	44.1
Migration	–	79133	157523	92664	83911
MigFreq	–	1	2	5	2
Processors	8				
SolTime	1264.6	776.2	809.2	822.7	804.5
RedTime	–	68.5	113.8	109.3	64.0
Migration	–	129401	682913	125472	110418
MigFreq	–	6	4	6	4
Processors	16				
SolTime	876.6	414.1	413.1	473.4	444.5
RedTime	–	84.8	100.1	104.1	115.3
Migration	–	326707	889233	162362	185772
MigFreq	–	9	5	6	8
Processors	32				
SolTime	422.4	214.9	225.3	263.1	239.9
RedTime	–	72.3	128.0	113.7	93.4
Migration	–	249751	1672496	186305	192690
MigFreq	–	10	10	9	9

Table 5.6: Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 5% for Example 2.

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	2				
SolTime	3013.4	2880.1	2873.7	2876.1	2567.6
RedTime	–	30.5	34.7	44.7	28.8
Migration	–	53844	73808	44539	44539
MigFreq	–	1	1	1	1
Processors	4				
SolTime	1856.7	1495.1	1475.1	1501.7	1493.8
RedTime	–	25.8	19.2	43.0	25.0
Migration	–	79133	75248	70754	78642
MigFreq	–	1	1	1	1
Processors	8				
SolTime	1264.6	805.8	833.2	880.1	814.1
RedTime	–	39.9	42.3	56.9	40.4
Migration	–	110435	362968	118299	99744
MigFreq	–	3	2	3	2
Processors	16				
SolTime	876.6	414.2	430.2	468.3	432.6
RedTime	–	48.8	63.2	70.2	75.1
Migration	–	217530	586957	158496	169039
MigFreq	–	4	3	4	4
Processors	32				
SolTime	422.4	217.6	211.5	265.3	232.3
RedTime	–	54.6	77.9	88.0	53.2
Migration	–	292716	1184563	180421	173236
MigFreq	–	6	7	6	5

Table 5.7: Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 10% for Example 2.

	Initial	ParMETIS	ParJOSTLE	Vidwans *et al.*	Recursive
Processors	8				
SolTime	1264.6	791.9	817.5	805.7	801.6
RedTime	–	27.1	40.2	48.3	16.2
Migration	–	95914	373527	114166	77261
MigFreq	–	2	2	2	1
Processors	16				
SolTime	876.6	426.1	401.6	451.1	420.8
RedTime	–	36.3	27.0	52.2	56.9
Migration	–	171606	390243	157631	170719
MigFreq	–	3	2	3	3
Processors	32				
SolTime	422.4	211.1	227.5	279.8	245.6
RedTime	–	46.0	56.6	69.1	48.8
Migration	–	209883	668654	161495	160964
MigFreq	–	6	4	3	3

Table 5.8: Solution times, redistribution times, total migration weights and migration frequencies for 300 time-steps using a re-balancing tolerance of 15% for Example 2.

5.6 Discussion

For simplicity and clarity we split our discussion in two parts. Part one of the discussion corresponds to Example 1 above, where we have a relatively smaller coarse mesh. The second part of the discussion corresponds to Example 2 above, where the coarse mesh is relatively larger. Also our dynamic load-balancing algorithm is referred as the Recursive algorithm. These discussions basically involve the efficiency and performance of the four dynamic load-balancing algorithms. The other important factor of the parallel adaptivity itself is not discussed here. The issues related to adaptivity are addressed by Selwood and Berzins and can be found in [87].

5.6.1 Discussion I

We first discuss the results shown in Table 5.1. It is clear that the cut-weights produced by the Vidwans *et al.* algorithm are highest as compared to the cut-weights produced by all other algorithms. The Recursive algorithm produces the best load-balance after re-partitioning (except the case of 16 processors where the Vidwans *et al.* algorithm produces the best load-balance after re-partitioning), however this is always achieved at the expense of a larger cut-weight as compared to the ParMETIS and ParJOSTLE algorithms. In general the ParMETIS algorithm tends to migrate the least amount of data albeit at the expense of a larger value of MaxImb. The ParJOSTLE algorithm produces a new partitioning in which the value of the parameter CutWt is the least. However it also tends to move the highest number of elements (with one exemption, the exemption being the case of 4 processors where the Recursive algorithm migrates the highest number of elements). The ParJOSTLE algorithm also produces a new partitioning in which the value of MaxImb is better then the corresponding value produced by the ParMETIS algorithm but worse than that produced by the Recursive and Vidwans *et al.* algorithms.

As far as the parameter SolTime is concerned it is roughly the same for all techniques. It also scales well (as it reduces approximately to half when the number of processors are doubled).

We next consider the results shown in Table 5.2 where we are using a re-balancing tolerance of 5%. From this table it is clear that the value of SolTime is roughly the same (except two cases: in the case of 2 processors it is slightly

higher for the ParMETIS algorithm and in the case of 8 processors it is slightly higher for the Vidwans *et al.* and Recursive algorithms) for all four techniques. It is also clear from this table that the the ParJOSTLE algorithm produces the most data migration. In the case of 2 processors it is the ParMETIS algorithm who migrates the least amount of data whilst in the cases of 4 and 8 processors this property is enjoyed by Vidwans *et al.* algorithm and in the last two cases it is the Recursive algorithm who migrates the least amount of data. As far as the value of the parameter RedTime is concerned, it is smallest in the cases of 2 and 8 processors for the ParMETIS algorithm (in other 3 cases this property is enjoyed by the Recursive algorithm). In the last 3 cases (where we use more than 4 processors) this value is highest in the case of the ParJOSTLE algorithm. Also this value is highest for the Recursive and ParMETIS algorithms in the cases of 2 and 4 processors respectively.

In the case of Table 5.3 where we are using a re-balancing tolerance of 10%, the value of the parameter SolTime is relatively high for the Vidwans *et al.* algorithm (except in the case of 2 processors where it is relatively high for the ParMETIS algorithm). For other techniques it is roughly the same (except two cases: in the case of 2 processors it is relatively small for the ParJOSTLE algorithm and in the case of 8 processors it is the second highest for the Recursive algorithm). The value of the parameter RedTime is smallest respectively for the ParMETIS algorithm in the cases of 2 and 8 processors, for the Vidwans *et al.* algorithm in the case of 4 processors, for the Recursive algorithm in the case of 16 processors and for the ParJOSTLE algorithm in the case of 32 processors. As far as the smallest value of the parameter Migration is concerned its behaviour is exactly the same as the behaviour of the RedTime above (except two cases: in the case of 2 processors the smallest value is produced by the Vidwans *et al.* algorithm whilst in the case of 32 processors the smallest value is produced by the Recursive algorithm). The value of the parameter Migration is still highest for the ParJOSTLE algorithm.

We next turn to the Table 5.4 where we are using a re-balancing tolerance of 15%. Here the values of the parameter SolTime are relatively high in the cases of the ParMETIS and Vidwans *et al.* algorithms and are relatively low for other techniques (except two cases: for 16 processors case this value is smallest for the ParMETIS algorithm and for 2 processors case this value is also high for the Recursive algorithm). The value of the parameter RedTime is smallest respectively

for the ParMETIS algorithm in the cases of 2 and 8 processors, for the Vidwans *et al.* algorithm in the case of 4 processors and for the Recursive algorithm in the last 2 cases. The behaviour of the parameter Migration is exactly the same as in Table 5.3.

It is clear from above analysis that every algorithm has some plus points and some minus points. In general the Recursive, Vidwans *et al.* and ParMETIS algorithms tend to migrate the least amount of data whilst the ParJOSTLE algorithm tends to migrate the huge amount of data. The Recursive and Vidwans *et al.* algorithms lead to the best load-balance whilst the ParMETIS algorithm leads to the worst load-balance after re-partitioning. The ParJOSTLE algorithm produces a partitioning in which this indicator is better than the ParMETIS algorithm and worse than other techniques. As far as the the value of the parameter CutWt is concerned the ParJOSTLE algorithm is a clear cut winner while the Vidwans *et al.* algorithm finishes at the bottom place – with the ParMETIS algorithm taking the middle ground. But they all have one common property, the sum of RedTime and SolTime is much less than the corresponding value of SolTime (except that in the case of 16 processors the ParJOSTLE (with a re-balancing tolerance of 5%) and Vidwans *et al.* (with a re-balancing tolerance of 5% and 10%) algorithms take a little higher time) where no dynamic load-balancing algorithm is called upon (see Figures 5.6 to 5.8 which clearly supports the conjecture).

Finally, it is interesting to observe the behaviour of the parameter MigFreq. The value of MigFreq increases as one increases the number of processors. This is due to the fact that the more processors are used, the more quickly imbalance is generated by the adaptation.

5.6.2 Discussion II

We next discuss the results shown in Table 5.5. Let us recall that these results corresponds to a larger base mesh which has 34560 coarse elements. Just like the smaller mesh of Example 1, here too, the Recursive and Vidwans *et al.* algorithms tend to produce the best load-balance after re-partitioning, however this is achieved at the expense of a larger cut-weight. In particular the cut-weights produced by the Vidwans *et al.* algorithm are highest as compared to other techniques. Except the cases of 4 and 16 processors, the Recursive algorithm migrates the least amount

of data (this property is enjoyed by the Vidwans *et al.* algorithm in the cases of 4 and 16 processors). Also the value of SolTime is roughly same for all algorithms (except that in the case of 2 processors it is relatively high for the ParJOSTLE algorithm and relatively small for the Recursive algorithm). The ParJOSTLE algorithm produces new partitioning in which the value of the parameter CutWt is the least one (except 2 processors case where this property is enjoyed by the ParMETIS algorithm) but this is always achieved at the expense of huge data migration. As in the case of smaller mesh, the ParMETIS algorithm tends to produces the worst load-balance after re-partitioning. The amount of imbalance produced by the ParJOSTLE algorithm is roughly the same as produced by the Recursive and Vidwans *et al.* algorithms (except 32 processors case in which it produces an imbalance of 9% as oppose to 2% produced by the Recursive and Vidwans *et al.* algorithms).

We next turn to the Table 5.6 which corresponds to a re-balancing tolerance of 5%. Here the value of the parameter SolTime is highest respectively for the ParMETIS algorithm in first two cases and for the Vidwans *et al.* algorithm for last three cases. There is no clear pattern as far as the parameters RedTime and Migration are concerned. The first parameter is small for the Recursive algorithm in the cases of 2 and 8 processors and this property is enjoyed by the ParMETIS algorithm for remaining 3 cases. The second parameter is small respectively for the ParMETIS algorithm in the case of 4 processors, for the Vidwans *et al.* algorithm in the cases of 16 and 32 processors and for the Recursive algorithm in the cases of 2 and 8 processors. As usual heavy migration results in the case of the ParJOSTLE algorithm.

We next analyse Table 5.7 where the results correspond to a re-balancing tolerance of 10%. Here the value of the parameter SolTime is relatively high for the Vidwans *et al.* algorithm (except in the case of 2 processors where it is relatively high for the ParMETIS algorithm). Also the value of the parameter RedTime is relatively high for the Vidwans *et al.* algorithm (except in the case of 16 processors where it is relatively high for the Recursive algorithm). Apart from 4 processors case the value of the parameter Migration is high for the ParJOSTLE algorithm. In the case of 4 processors this parameter is high for the ParMETIS algorithm. The value of the parameter Migration is smallest for the Recursive algorithm (except 4 and 16 processors cases where it is smallest for the Vidwans *et al.* algorithm).

As far as the Table 5.8 is concerned (where we use a re-balancing tolerance of

15%) the value of SolTime is smallest for the ParMETIS algorithm in the cases of 8 and 32 processors and for the ParJOSTLE algorithm in the case of 16 processors respectively. The value of the parameter RedTime is smallest for the Recursive algorithm in the case of 8 processors, for the ParJOSTLE algorithm in the case of 16 processors and for the ParMETIS algorithm in the case of 32 processors respectively. As usual heavy migration results in the case of the ParJOSTLE algorithm. Least amount of migration materialised by the Vidwans *et al.* algorithm in the case of 16 processors whilst for other than 16 processors case this property is enjoyed by the Recursive algorithm.

As in the case of the smaller mesh here too, no single algorithm emerges as a clear cut winner in all respects. The Recursive and Vidwans *et al.* algorithms produce the best load-balance after re-partitioning but suffers by the fact that it has a relatively high value of CutWt. In the case of the ParJOSTLE algorithm CutWt is relatively low but resulting migration is huge one. Resulting value of MaxImb in the case of the ParMETIS algorithm is relatively high as compared to the other three algorithms. Once again they all have same common property, the sum of RedTime and SolTime is much less than the corresponding value of SolTime where no dynamic load-balancing algorithm is called upon (see Figures 5.9 to 5.11 which clearly supports the conjecture).

In all cases the qualitative behaviour of the parameter MigFreq is exactly the the same as for the smaller mesh (i.e. the value of MigFreq increases as one increases the number of processors).

A final important question is what is the optimal value of the re-balancing tolerance parameterΓ From Tables 5.6 to 5.8 it is clear that in cases where large number of processors are used it does not pay to call the dynamic load-balancer at lower values of the parameter. Using the tolerance of 5% is more costly as compared to the other values tried (namely 10% and 15%). So in such case a smaller values of the parameter should be avoided.

5.7 Investigation into Scalability of the Algorithm

It is clear that the scalability of any dynamic load-balancing algorithm which is used in an adaptive mesh solver is a complex issue. Clearly the performance of the dynamic load-balancer depends upon how often the adaptivity has taken place. But

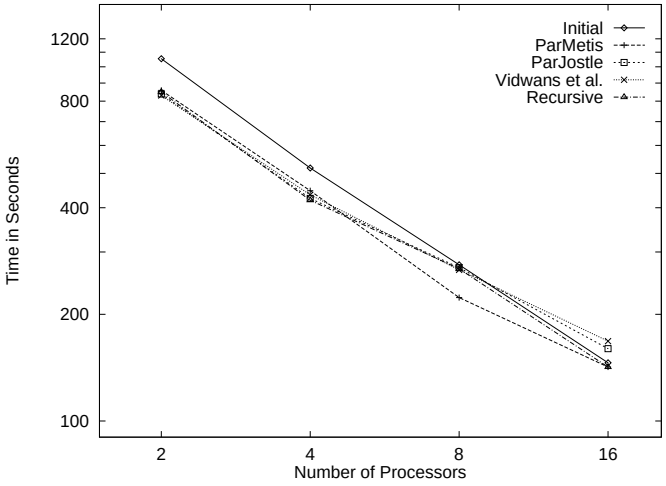

Figure 5.6: Scalability comparison using a re-balancing tolerance of 5% for Example 1 (where Time = RedTime + SolTime).

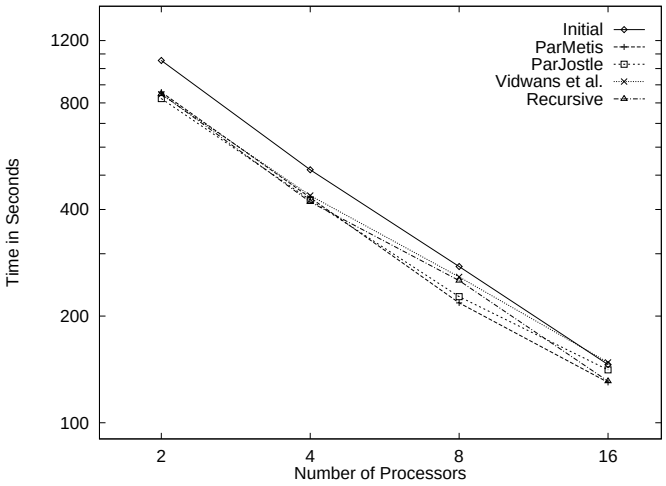

Figure 5.7: Scalability comparison using a re-balancing tolerance of 10% for Example 1 (where Time = RedTime + SolTime).

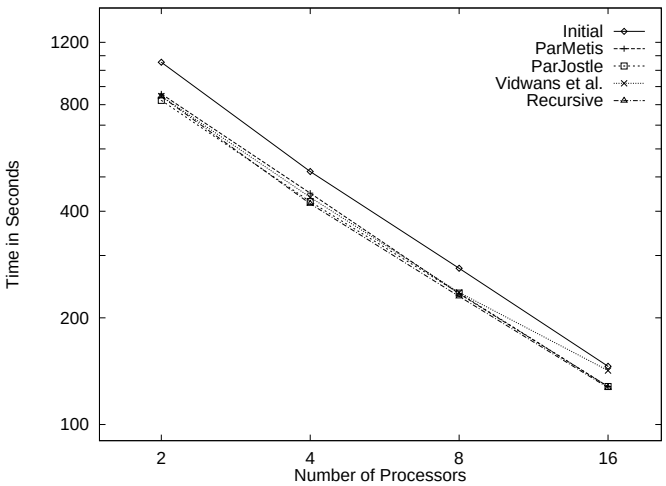

Figure 5.8: Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + SolTime).

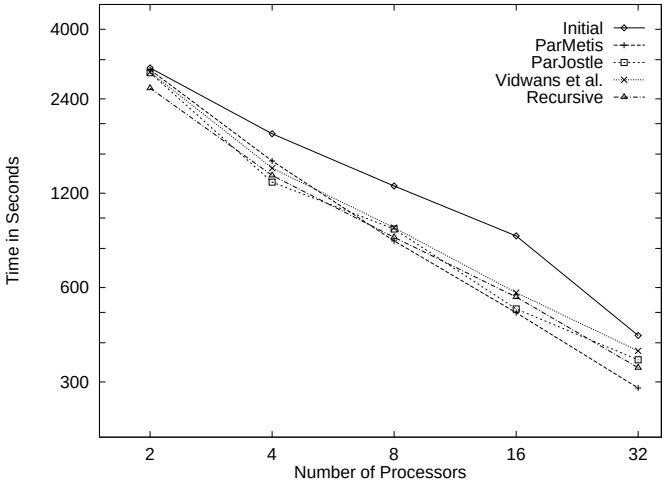

Figure 5.9: Scalability comparison using a re-balancing tolerance of 5% for Example 2 (where Time = RedTime + SolTime).

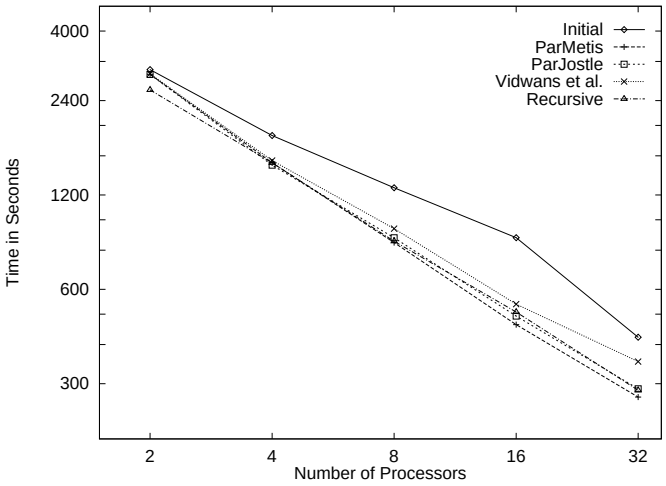

Figure 5.10: Scalability comparison using a re-balancing tolerance of 10% for Example 2 (where Time = RedTime + SolTime).

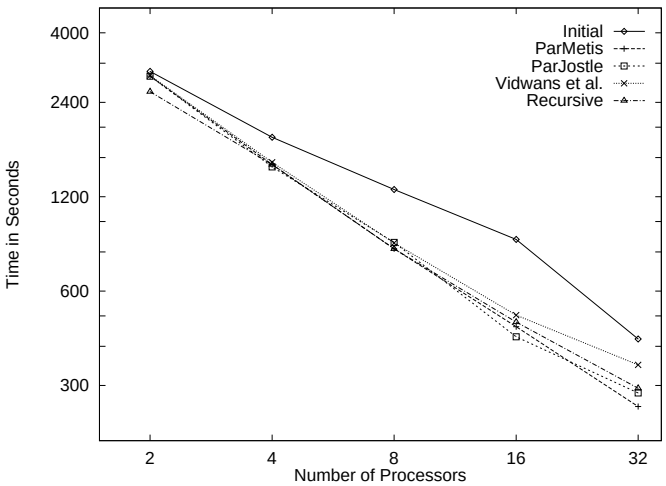

Figure 5.11: Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + SolTime).

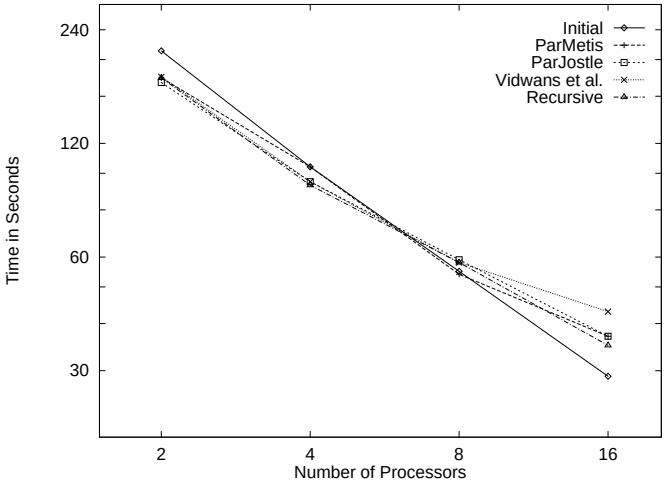

Figure 5.12: Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + 0.2 * SolTime).

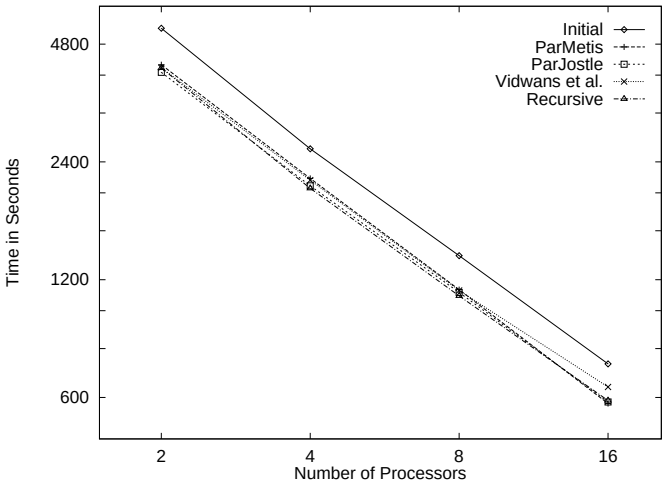

Figure 5.13: Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + 5 * SolTime).

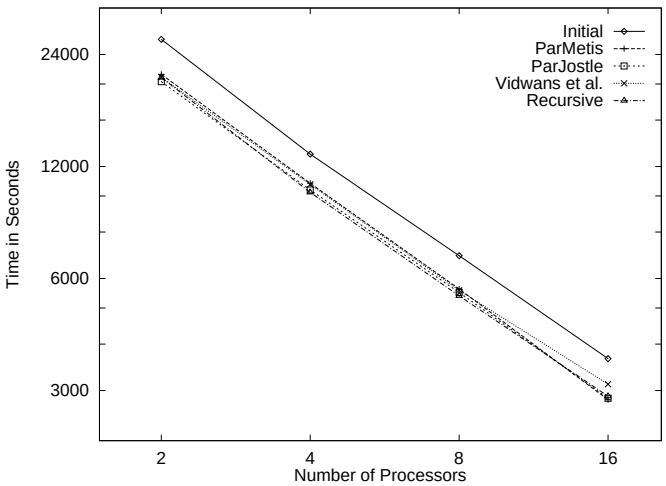

Figure 5.14: Scalability comparison using a re-balancing tolerance of 15% for Example 1 (where Time = RedTime + 25 * SolTime).

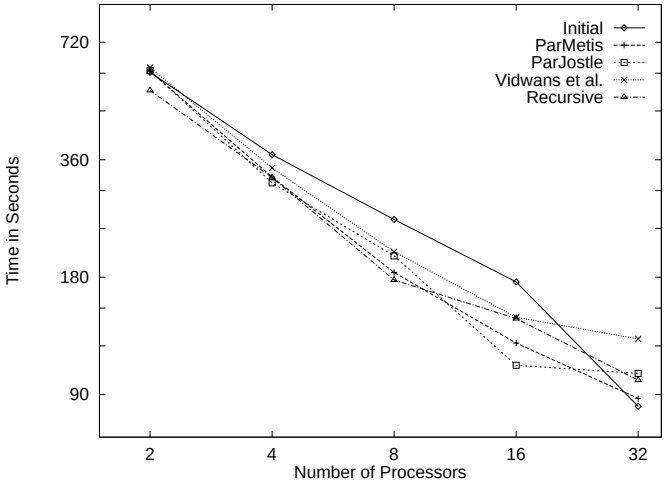

Figure 5.15: Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + 0.2 * SolTime).

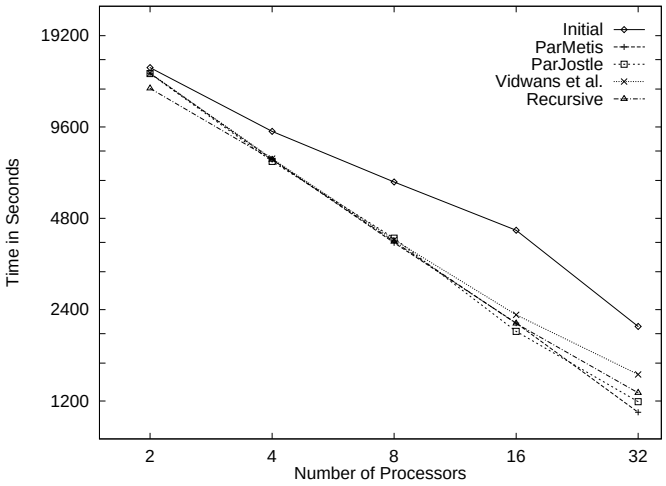

Figure 5.16: Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + 5 * SolTime).

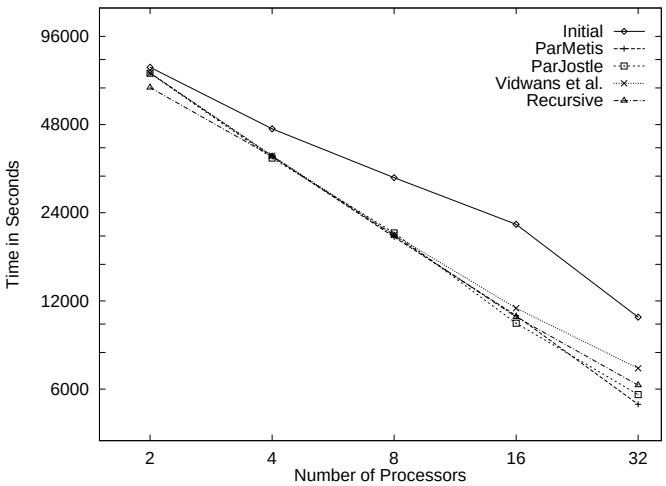

Figure 5.17: Scalability comparison using a re-balancing tolerance of 15% for Example 2 (where Time = RedTime + 25 * SolTime).

it also depends upon other parameters. For example if after the adaptation stage, the amount of imbalance remains below a specified tolerance no redistribution of data will occur. The other parameter upon which it depends is the number of processors used. The numerical results shown above suggest that the more processors are used, the more quickly imbalance is generated by the adaptation. Another parameter is the number of levels of refinements. Also, repartitioning consists almost entirely of communication and since the work involved is not evenly distributed, it tends not to scale well.

The more important quantity to consider is the combined performance of the solver and the dynamic load-balancer. In practice one wishes to see the net saving which results in using a solver combined with a dynamic load-balancer. In Figures 5.6 to 5.17 we plot the time (which is a function of RedTime and SolTime) against the number of processors used. In Figures 5.6 to 5.11 the time function is simply the sum of RedTime and SolTime. In Figures 5.12 to 5.17 the time function is equal to RedTime + α SolTime, where α is a positive constant other than unity. Note that by varying α we can analyse the equivalent effect of other simpler (for which $\alpha < 1$) or more expensive (for which $\alpha > 1$) solvers. (Alternatively, we can view the variation of α as representing slower or faster inter-processor communications respectively.)

It is clear from Figure 5.6 that in this case the use of the ParJOSTLE and Vidwans *et al.* have an adverse affect and there is a small saving in case one uses the ParMETIS or Recursive algorithm when 16 processors are used. In other cases (see Figures 5.7 to 5.11) it pays off to use the dynamic load-balancing algorithm. As pointed out above, for the larger base mesh it is better to use a relatively larger value of the re-balancing tolerance parameter in those cases where the number of processors are greater than 8 (see Figures 5.9 to 5.11).

We next discuss the relationship between the intensity of the solver and the use of the dynamic load-balancer. From Figures 5.12 and 5.15 it is clear that calling a dynamic load-balancer in the case of a very cheap solver does not pay anything. It in fact increases the cost. However, in the case of an expensive solver it pays off to use a dynamic load-balancer (especially when using a large number of processors). For example if we are having a solver which is 25 times more expensive than the current solver and we are using 16 processors to solve the problem then at least there is a 15% saving in the case of the smaller base mesh and a 48% saving in the

case of the larger base mesh as a result of using a dynamic-load balancing algorithm.

5.8 Conclusions

In this chapter we have demonstrated that good load-balancing results can be obtained provided one choses the re-balancing thresholds carefully. As one would expect, we have also shown that the more expensive is the solver the more would be the saving that can be achieved by using a dynamic load-balancer. Moreover, we have demonstrated that the Recursive algorithm is very competitive when contrasted with other parallel dynamic load-balancing algorithms developed simultaneously with this work. In some situations it gives the best performance of all four approaches and in all cases the performance is comparable. Clearly the choice of which of the four algorithms is the best to use will depend on the precise nature of the problem being solved and on the exact nature of the parallel architecture.

Chapter 6

Conclusion and Future Areas of Research

6.1 Summary of Thesis

The main contribution of this thesis has been to present the development of a new parallel dynamic load-balancing algorithm designed specifically for application in the conjunction with parallel adaptive finite element or finite volume codes. We first discussed a uniprocessor version in Chapter 3 which was tested on a number of different model problems. It was observed in this chapter that the algorithm produces satisfactory results in the sense that the final partitions are reasonably well balanced and the relative increase in the cut-weights are also small (in some cases the cut-weight is even smaller than the original value). In few cases, where the coarse mesh is large with the majority of coarse elements having unit weight, the cost of this algorithm is a little high (this issue is discussed in the next section below).

In Chapter 4 we presented an initial parallel dynamic load-balancing version of the the algorithm of Chapter 3. This new version of the algorithm was successfully applied to a variety of 2-d unstructured meshes which were generated in parallel but were not always perfectly balanced (for reasons mentioned in the chapter). This version is not only very fast (in the majority of the cases it took less than a second to get the balanced mesh on the test problems studied) but also produces a final partition which is reasonably load balanced. A net saving in solver time ranging from 3% to 15% was also achieved.

In Chapter 5 a further, slightly more general, version of the algorithm of Chapter 3 was introduced. This new version was developed to allow, amongst other applications, possible use after the adaptivity steps of a 3-d parallel adaptive solver (which inevitably has some sort of imbalance after the adaptivity step). It was applied on a number of 3-d problems on a differing number of processors. It is shown that results produced are compatible with those produced by other state-of-the-art techniques (which were developed simultaneously with this work). The resulting amount of imbalance was generally less than the corresponding value produced by other techniques and in many cases less data migration is required (certainly in comparison with the ParJOSTLE algorithm). In contrast with the other algorithms considered the resulting cut-weights after repartitioning are relatively high (but not in comparison with the other algorithms considered in Chapter 4). Overall, for the type of application of interest here there was little to choose between the performance of the new algorithm presented here and that of the other algorithms considered (with each method, including the new algorithm, performing best in some particular cases).

A final point to be mentioned concerns the availability of the new algorithm as public domain software. Both ParMETIS and ParJOSTLE are publicly available (although, as we have seen in Chapter 5, their current implementations are far from stable) whilst the new algorithm is not. Unfortunately such software development was beyond the scope of this project.

6.2 Possible Extensions to the Research

There are number of ways in which the research described in this thesis could be extended. One important possibility that I would like to explore is to build into the scheme a *partial coarsening*. Recall that this possibility is first discussed at end of Chapter 3 where it is suggested that the use of a partial coarsening may increase the efficiency of the algorithm. Observe that in Example 6 of this chapter 72% of the coarse mesh was residing on 2 (out of 8 possible) processors whereas 31% of the fine mesh was contained on these two processors. In a situation like this a large portion of the coarse mesh consists of coarse elements whose weights are very low (the majority of the elements have only unit weights). So if these nodes can be combined into some sort of coarse, higher-level, nodes then it is hoped that a

significant improvement in terms of speed may be obtained.

A major change from Chapter 3 to Chapters 4 and 5 was the absence of the swapping steps once the load balance among two subgroups is achieved. Recall that in Chapter 3 such a step was executed in hope of finding any "hidden" minima as regard to the cut-weight. The algorithm of Chapter 3 was a serial one so no communication cost was involved with this. The versions of Chapters 4 and 5 are parallel ones however. The major decision of dropping this step was taken due to the communication cost involved in keeping it. This decision was based upon personal judgement. In order to prove (or disprove) this conjecture, this extra step could be implemented and the cost of implementing this step verses the resulting net saving in the solver time (due to a reduced cut-weight) may be examined.

Apart from above there is a clear need for further research in other directions as well. In general the repartitioning thresholds have a very significant impact on the performance and scalability of the whole repartitioning process. More investigation could certainly be undertaken in this area. Smaller thresholds may result in too frequent re-mapping of the mesh (which results in a higher communication cost) and larger thresholds obviously lead to keeping an unbalanced mesh for a much larger time (which results in a relatively higher solver cost). Even a fixed choice of the threshold may be problematic. It my be worthwhile to consider other options such as to calculate the redistribution cost and compare it with the saving (or some estimate of the saving) which may result from running the adapted mesh on the new partition rather than on the old partition. Only if this is favourable would the repartitioning step be executed. Pioneering work in this field has already been started by Biswas and Oliker [10, 77]. More investigation is needed in this area (e.g. the impact on future costs of the current decision: a too greedy saving may result in paying more cost in future in some situations!). Also the relation between the size of the mesh, the number of processors used and the repartitioning thresholds could be examined in further detail. Finally, there is also a need to explore some other metrics for the cost of balancing distributed loads: the current trend is to minimise the total amount of migration but an arguably more important metric would be to minimise the maximum amount of migration for any single processor.

Another factor to be further explored is of the number of levels of refinement that are used, especially in three-dimensional problems. In Chapter 5 we considered up to 3 levels of refinement, which means the maximum possible weight of a coarse

element was 512. Considering a fourth level of refinment would mean up to 4096 fine elements inside each coarse element. Thus, in order to consider this level of refinement the underlying coarse mesh would have to be very large which could add significantly to computational costs (although this overhead could be almost totally eliminated if the partial refinement idea described above could be successfully implemented).

Other investigations that would be worthwhile to undertake include the application of the dynamic load-balancing algorithm to a wider variety of problems. In Chapter 5 we only considered the solution of one particular equation, it would be worthwhile to try a wider variety of equations and parallel solvers using the same parallel adaptivity code. Also different refinement algorithms (such p-refinement or hp-refinement) could be considered, as well as the use of heterogeneous networks. Note that in situations like these one has to generalise the definition of the weights of coarse elements. It is no longer equal to the number of fine elements present but has to be adjusted to incorporate the differing amounts of work required by different coarse elements and the varying speeds of the individual component of the heterogeneous network involved.

One approach that has not been considered at all in this thesis is to perform load-balancing based upon a partition of the fine mesh in a grid hierarchy rather than the coarse mesh. There are many implications to considering such a strategy, which appears to allow more flexibility in the partition than by distributing the coarse grid, and it is quite possible that a full study of this would require another thesis.

Bibliography

[1] M Ainsworth and J T Oden. A unified approach to a posteriori error estimation using element residual methods. *Numerische Mathematik*, 65(1):23–50, 1993.

[2] T Arthur and M J Bockelie. A comparison of using APPL and PVM for a parallel implementation of an unstructured grid generation program. Technical Report 191425, NASA Computer Sciences Corporation, Hampton, Virginia, 1993.

[3] I Babuska, B A Szabo, and I N Katz. The p-version of the finite element method. *SIAM Journal on Numerical Analysis*, 18(3):515–545, 1981.

[4] C Bailey, P Chow, M Cross, Y Fryer, and K Pericleous. Multiphysics modelling of the shape casting process. In *Proceedings of the Royal Society 0f London Series A-Mathematical Physical and Engineering Sciences*, volume 452, pages 459–486, 1996.

[5] R E Bank. *PLTMG Users' Guide 7.0.* SIAM, Philadelphia, 1994.

[6] R E Bank. *PLTMG Users' Guide 8.0.* SIAM, Philadelphia, 1998.

[7] R E Bank and A Weiser. Some a posteriori error estimates for partial differential equations. *Mathematics of Computation*, 44:283–301, 1985.

[8] P Bastian, K Birken, K Johannsen, S Lang, K Eckstein, N Neuss, H Rentz-Reichert, and C Wieners. UG - A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science (To appear)*, 1998.

[9] M Berzins and R M Furzeland. A user's manual for SPRINT - A versatile software package for solving systems of algebraic, ordinary and partial differential equations: Part 1 - algebraic and ordinary differential equations. Technical Report TNER.85.058, Thornton Research Centre, Shell Research Limited, 1985.

[10] R Biswas and L Oliker. Load balancing unstructured adaptive grids for CFD problems. In M Heath et al., editors, *Eighth SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia*, 1997.

[11] R Biswas and R C Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.

[12] J E Boillat. Load balancing and Poisson equation in a graph. *Concurrency: Practice and Experience*, 2:289–313, 1990.

[13] J Cabello. Parallel explicit unstructured grid solvers on distributed memory computers. *Advances in Engineering Software*, 23:189–200, 1996.

[14] P J Capon. *Adaptive Stable Finite Element Methods for the Compressible Navier-Stokes Equations*. PhD thesis, School of Computer Studies, University of Leeds, 1995.

[15] P J Capon and P K Jimack. An adaptive finite element method for the compressible Navier-Stokes equations. In M J Baines and K W Morton, editors, *Numerical Methods for Fluid Dynamics 5*, pages 327–334. OUP, 1995.

[16] K M Case et al. Problems in transition from laminar to turbulent Flow-II. Technical Report JASON JSR-77-18, SRI International, Arlington VA, 1978.

[17] Grand Challenges. High performance computing and communications. The fiscal year 1992 U.S. research and development program. Technical report, by the Committee on Physical, Mathematical and Engineering Sciences, NSF Washington, 1992.

[18] W Chan and A George. A linear time implementation of the Reverse Cuthill McKee algorithm. *BIT*, 20:8–14, 1980.

[19] P G Ciarlet. *The finite element method for elliptic problems.* North-Holland Publishing Company, 1978.

[20] L Ciminiera and A Valenzano. *Advanced microprocessor architectures.* Addison-Wesley Publishing Company, 1987.

[21] S Corrsin. Turbulent flow. *American Scientist*, 49:300–325, 1961.

[22] D M Cvetkovic, M Doob, I Gutman, and A Torgasev. Recent results in the theory of graph spectra. *Annal Discrete Mathematics*, 36, 1988.

[23] G Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[24] L Demkowicz, J T Oden, and W Rachowicz. A new finite element method for solving compressible Navier-Stokes equations based on an operator splitting method and h-p adaptivity. *Computer Methods in Applied Mechanics and Engineering*, 84:275–326, 1990.

[25] L Demkowicz, J T Oden, W Rachowicz, and O Hardy. An h-p Taylor-Galerkin finite element method for compressible Euler equations. *Computer Methods in Applied Mechanics and Engineering*, 88:363–396, 1991.

[26] A M Despain. A massive multiple microcomputer system. In James Clark Solinsky, editor, *Advanced Computer Concepts (LJI Conference Proceedings)*, pages 91–105, La Jolla Institute, 1981.

[27] K D Devine and J E Flaherty. Parallel adaptive hp-refinement techniques for conservation laws. *Applied Numerical Mathematics*, 20:367–386, 1996.

[28] P Diniz, S Plimpton, B Hendrickson, and R Leland. Parallel algorithms for dynamically partitioning unstructured grids. In D H Bailey et al., editors, *Seventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1995.

[29] C Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[30] C Farhat, S Lanteri, and H D Simon. TOP/DOMDEC - a software tool for mesh partitioning and parallel processing. *Computing Systems in Engineering*, 6(1):13–26, 1995.

[31] C Farhat and M Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36(5):745–764, 1993.

[32] C M Fiduccia and R M Mattheyses. A linear_time heuristic for improving network partitions. In *Proceedings of the Nineteenth IEEE Design Automation Conference*, pages 175–181. IEEE, 1982.

[33] M Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, 1966.

[34] Message Passing Interface Forum. MPI: A Message Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[35] 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.

[36] M Garey, D Johnson, and L Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[37] A George and J W Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981.

[38] G Globisch. PARMESH: A parallel mesh generator. *Parallel Computing*, 21(3):509–524, 1995.

[39] S K Godunov. A finite difference method for the numerical computation of discontinuous solutions of the equations of fluid dynamics. *Mat. Sb.*, 47:357–393, 1959.

[40] G H Golub and C F Van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, 1989.

[41] B K Grant and A Skjellum. The PVM system: An in-depth analysis and documenting study – concise edition. Technical Report UCRL-JC-112016, LLNL, Livermore, CA, 1992.

[42] A Harten, P D Lax, and B van Leer. On upstream differencing and Godunov type schemes for hyperbolic conservation laws. *SIAM Review*, 25(1):36–61, 1983.

[43] B Hendrickson. Can static load balancing algorithms be appropriate in a dynamic settingΓ Dynamic Load Balancing on MPP Systems: Progress, Challenges and Issues. A 1-day meeting at CCLRC Daresbury Laboratory, Daresbury, Warrington, UK, 1995.

[44] B Hendrickson and R Leland. The Chaco user's guide, version 1.o. Technical Report SAND93–2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[45] B Hendrickson and R Leland. Multidimensional spectral load balancing. Technical Report SAND93–0074, Sandia National Laboratories, Albuquerque, NM, 1993.

[46] B Hendrickson and R Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93–1301, Sandia National Laboratories, Albuquerque, NM, 1993.

[47] B Hendrickson and R Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16:452–469, 1995.

[48] B Hendrickson and R Leland. The Chaco user's guide, version 2.o. Technical Report SAND95–2344, Sandia National Laboratories, Albuquerque, NM, 1995.

[49] D C Hodgson. *Efficient Mesh Partitioning and Domain Decomposition Methods on Parallel Distributed Memory Machines.* PhD thesis, School of Computer Studies, University of Leeds, 1995.

[50] D C Hodgson and P K Jimack. Efficient mesh partitioning for parallel P.D.E. solvers on distributed memory machines. In *Sixth SIAM Conference Parallel Processing for Scientific Computing*, Norfolk, VA, 1993.

[51] D C Hodgson and P K Jimack. Efficient parallel generation of partitioned, unstructured meshes. *Advances in Engineering Software*, 27(1/2):59–70, 1996.

[52] G Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 19(2):209–218, 1993.

[53] Y F Hu and R J Blake. An optimal dynamic load balancing algorithm. Technical Report DL-P-95-011, The Central Laboratory for the Research Councils, Daresbury Laboratory, Daresbury, Warrington, Cheshire, UK (To be published in Concurrency: Practice and Experience), 1995.

[54] P K Jimack. A new approach to finite element error control for time-dependent problems. In Baines and K W Morton, editors, *Numerical Methods for Fluid Dynamics 4*, pages 567–573. OUP, 1993.

[55] P K Jimack. An overview of dynamic load-balancing for parallel adaptive computational mechanics codes. In B H V Topping, editor, *Parallel and Distributed Processing for Computational Mechanics I*. Saxe-Coburg Publications, 1997.

[56] P K Jimack. Techniques for parallel adaptivity. In B H V Topping, editor, *Parallel and Distributed Processing for Computational Mechanics II*. Saxe-Coburg Publications, 1998.

[57] Z Johan. *Data Parallel Finite Element Techniques for Large-scale Computational Fluid Dynamics*. PhD thesis, Stanford University, 1992.

[58] C Johnson. *Numerical solution of partial differential equations by the finite element method*. Cambridge University Press, 1987.

[59] Y Kallinderis and P Vijayan. Adaptive refinement-coarsening scheme for 3-D unstructured meshes. *AIAA Journal*, 31(8):1440–1447, 1993.

[60] G Karypis and V Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95–035, Department of Computer Science, University of Minnesota, Minneapolis, USA, 1995.

[61] G Karypis and V Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, USA, 1995.

[62] G Karypis and V Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, USA, 1996.

[63] G Karypis and V Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In M Heath et al., editors, *Eighth SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia*, 1997.

[64] G Karypis, K Schloegel, and V Kumar. ParMETIS, parallel graph partitioning and sparse matrix ordering library, version 1.0. Department of Computer Science, University of Minnesota, Minneapolis, USA.

[65] B W Kernighan and S Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[66] A I Khan and B H V Topping. Parallel adaptive mesh generation. *Computer Systems in Engineering*, 2(1):75–101, 1991.

[67] V Kumar, A Grama, A Gupta, and G Karypis. *Introduction to parallel computing: design and analysis of parallel algorithms.* The Benjamin/Cumming Publishing Company, Inc., 1994.

[68] T Lengauer. *Combinatorical algorithms for integrated circuit layout.* Teubner-Verlag, Stuttgart, 1990.

[69] R Löhner. An adaptive finite element scheme for transient problems in CFD. *Computer Methods in Applied Mechanics and Engineering*, 61:323–338, 1987.

[70] R Löhner, J Camberos, and M Merriam. Parallel unstructured grid generation. *Computer Methods in Applied Mechanics and Engineering*, 95:343–357, 1992.

[71] J G Malone. Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessors computers. *Computer Methods in Applied Mechanics and Engineering*, 70(1):27–58, 1988.

[72] K Miller and R N Miller. Moving finite elements, Part I. *SIAM Journal on Numerical Analysis*, 18(6):1019–1032, 1981.

[73] B Monien and R Diekmann. A local graph partitioning heuristic meeting bisection bounds. In M Heath et al., editors, *Eighth SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia*, 1997.

[74] M C Mosher. A variable node finite element method. *Journal of Computational Physics*, 57:157–187, 1985.

[75] B Nour-Omid, A Raefsky, and G Lyzenga. Solving finite element equations on concurrent computers. In A K Noor, editor, *Parallel Computations and Their Impact on Mechanics*, New York, 1986. American Soc. Mech. Eng. 209–227.

[76] J T Oden, T Strouboulis, and P H Devloo. Adaptive finite elements for high-speed compressible flow. *International Journal for Numerical Methods in Fluids*, 7:1211–1228, 1987.

[77] L Oliker and R Biswas. Efficient load balancing and data remapping for adaptive grid calculations. Technical report, NASA Ames Research Center, Moffett Field, CA, USA, 1997.

[78] M E G Ong. Uniform refinement of tetrahedron. *SIAM Journal on Scientific Computing*, 15(5):1134–1144, 1994.

[79] P Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[80] B N Parlett, H D Simon, and L Stringer. Estimating the largest eigenvalue with the Lanczos algorithm. *Mathematics of Computation*, 38:153–165, 1982.

[81] F Pellegrini. SCOTCH 3.2 user's guide. Université Bordeaux I, 1997.

[82] A Pothen, H D Simon, and K P Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11(3):430–452, 1990.

[83] R Preis and R Diekmann. PARTY - A software library for graph partitioning. In B H V Topping, editor, *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 63–71. Civil–Comp Press, Edinburgh, 1997.

[84] R M Russell. The CRAY-1 computer system. *Comm. ACM*, 21(1):63–72, 1978.

[85] K Schloegel, G Karypis, and V Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97–013, University of Minnesota, Department of Computer Science, Minneapolis, USA, 1997.

[86] K Schloegel, G Karypis, and V Kumar. Parallel multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97–014, University of Minnesota, Department of Computer Science and Army HPC centre, Minneapolis, USA, 1997.

[87] P M Selwood and M Berzins. Parallel unstructured tetrahedral mesh adaptation: Algorithms, implementation and scalability. *Submitted to Concurrency*, 1998.

[88] P M Selwood, M Berzins, and P M Dew. 3-D parallel mesh adaptivity: Data-structures and algorithms. In M Heath et al., editors, *Eighth SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia*, 1977.

[89] P M Selwood, N A Verhoeven, J M Nash, M Berzins, N P Weatherill, P M Dew, and K Morgan. Parallel mesh generation and adaptivity : Partitioning and analysis. In A Ecer, J Periaux, N Satufoka, and P Schiano, editors, *Parallel CFD – Proc. of Parallel CFD 96 Conference*. Elesvier Science BV, 1997.

[90] H D Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[91] H D Simon. Partitioning of unstructured problems for parallel processing. In *Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergammon Press, 1991.

[92] W E Speares and M Berzins. A fast 3-D unstructured mesh adaptation algorithm ith time-dependent upwind Euler shock diffraction calculations. In M Hafez and K Oshima, editors, *Proc. of 6th Int. Symp. on Computational Fluid Dynamics*, volume III, pages 1181–1188, 1995.

[93] W E Speares and M Berzins. A 3-D unstructured mesh adaptation algorithm for time dependent shock dominated problems. *International Journal for Numerical Methods in Fluids*, 25:81–104, 1997.

[94] J L Steger. Technical evaluation report: AGRD fluid dynamics panel specialist meeting on application of mesh generation to complex 3-D configurations. Technical Report AGARD-AR-268, AGARD, 1991.

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

[96] E Suli and P Houston. Finite element methods for hyperbolic problems: A posteriori error analysis and adaptivity. In I Duff and G A Watson, editors, *State of the Art in Numerical Analysis*, pages 441–471. OUP, 1997.

[97] J F Thompson and N P Weatherill. Aspects of numerical grid generation: Current science and art. In *Invited paper, 11th AIAA Applied Aerodynamics Conference, Monterey, Ca*, 1993.

[98] B H V Topping and A I Khan. Sub-domain generation method for non-convex domains. In B H V Topping and A I Khan, editors, *Information Technology for Civil and Structural Engineering*, pages 219–234. Civil–Comp Press, 1993.

[99] B H V Topping and A I Khan. *Parallel Finite Element Computations.* Saxe-Coburg Publications, Edinburgh,UK, 1996.

[100] N Touheed, P M Selwood, P K Jimack, M Berzins, and P M Dew. Parallel dynamic load-balancing for the solution of transient CFD problems using adaptive tetrahedral meshes. In D R Emerson et al., editors, *Parallel Computational Fluid Mechanics (Proc. of Parallel CFD 97 Conference, May, 1997, Manchester, UK)*, pages 81–88. Elsevier, Amsterdam, 1998.

[101] R Van Driessche and D Roose. An improved spectral bisection algorithm and its application to dynamic load balancing. *Parallel Computing*, 21:29–48, 1995.

[102] B van Leer. On the relation between upwind difference schemes. *SIAM Journal on Scientific and Statistical Computing*, 5:1–20, 1984.

[103] D Vanderstraeten, O Zone, R Keunings, and L Wolsey. Non-deterministic heuristics for automatic domain decomposition in direct parallel finite element calculations. In R F Sincovec et al., editors, *Parallel Processing for Scientific Computing*, pages 929–932. SIAM, 1993.

[104] A Vidwans, Y Kallinderis, and V Venkatakrishnan. Parallel dynamic load-balancing algorithm for 3-dimensional adaptive unstructured grids. *AIAA Journal*, 32(3):497–505, 1994.

[105] C Walshaw and M Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995.

[106] C Walshaw, M Cross, and M G Everett. A parallelisable algorithm for optimising unstructured mesh partitions. Mathematics Research Report, School of Mathematics, Statistics and Scientific Computing, University of Greenwich, London, UK, 1995.

[107] C Walshaw, M Cross, and M G Everett. Dynamic load-balancing for parallel adaptive unstructured meshes. In M Heath et al., editors, *Eighth SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia*, 1997.

[108] C Walshaw, M Cross, and M G Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In B H V Topping, editor, *Advances in Computational Mechanics with Parallel and Distributed Processing (Proc. Parallel & Distributed Computing for Computational Mechanics, Lochinver, Scotland)*, 97–103, Edinburgh, 1997. Civil–Comp Press.

[109] C Walshaw, M Cross, and M G Everett. Parallel dynamic graph-partitioning for unstructured meshes. *Journal of Parallel and Distributed Computing (in press)*, 1998.

[110] C Walshaw, M Cross, S Johnson, and M G Everett. A parallelisable algorithm for partitioning unstructured meshes. In *Irregular '94: Parallel Algorithms for Irregularly Structured Problems*, Geneva, 1994.

[111] C Walshaw, M Cross, S Johnson, and M G Everett. JOSTLE: Partitioning of unstructured meshes for massively parallel machines. In N Satofuka

et al., editors, *Parallel Computational Fluid Dynamics: New Algorithms and Applications (Proc. Parallel CFD'94, Kyoto, 1994)*, pages 273–280. Elsevier, Amsterdam, 1995.

[112] J M Ware. *The Adaptive Solution of Time-Dependent Partial Differential Equations in Two Space Dimensions*. PhD thesis, School of Computer Studies, University of Leeds, 1993.

[113] N P Weatherill. A review of mesh generation. In *Advances in Finite Element Technology*, pages 1–10. Civil–Comp Press, Budapest, 1996.

[114] R D Williams. Performance of dynamic load balancing for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3:457–481, 1991.

[115] J K Wilson and B H V Topping. A new element bisection algorithm for unstructured adaptive tetrahedral mesh generation. *Engineering Computations*, 15(5):588–615, 1998.

[116] P Wu and E N Houstis. Parallel dynamic mesh generation and domain decomposition. Technical Report, Computer Sciences Dept., Purdue University, 1993.

[117] C Z Xu and F C M Lau. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 16:385–393, 1992.

[118] C Z Xu and F C M Lau. The generalized dimension exchange method for load balancing in k-ary ncubes and variants. *Journal of Parallel and Distributed Computing*, 24:72–85, 1995.

[119] O C Zienkiewicz, D W Kelly, and J P Gago. The hierarchical concept in finite element analysis. *International Journal of Computers and Structures*, 16:53–65, 1983.