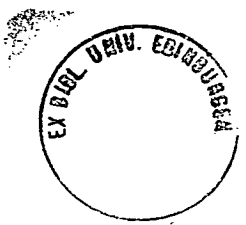
Decomposition of Unstructured Meshes for Efficient Parallel Computation

Robert A. Davey

Doctor of Philosophy

University of Edinburgh

1997



To Ros

Abstract

This thesis addresses issues relating to the use of parallel high performance computer architectures for unstructured mesh calculations. The finite element and finite volume methods are typical examples of such calculations which arise in a wide range of scientific and engineering applications.

The work in this thesis is focused on the development at Edinburgh Parallel Computing Centre of a software library to support static mesh decomposition, known as PUL-md. The library provides a variety of mesh decomposition and graph partitioning algorithms, including both global methods and local refinement techniques. The library implements simple random, cyclic and lexicographic partitioning, Farhat's greedy algorithm, recursive layered, coordinate, inertial and spectral bisections, together with subsequent refinement by either the Kernighan and Lin algorithm or by one of two variants of the Mob algorithm. The decomposition library is closely associated with another library, PUL-sm, which provides run-time support for unstructured mesh calculations.

The decomposition of unstructured meshes is related to the partitioning of undirected graphs. We present an exhaustive survey of algorithms for these related tasks. Implementation of the decomposition algorithms provided by PUL-md is discussed, and the tunable parameters that optimise the algorithm's behaviour are detailed. On the basis of various metrics of decomposition quality, we evaluate the relative merits of the algorithms and explore the tunable parameter space. To validate these metrics, and further demonstrate the utility of the library, we examine how the runtime of a demonstration application (a finite element code) depends on decomposition quality. Additional related work is presented, including research into the development of a novel 'seed-based' optimisation approach to graph partitioning. In this context gradient descent, simulated annealing and parallel genetic algorithms are explored.

Acknowledgements

This work would not have been possible without the generous assistance of many of my colleagues at EPCC. I would like to thank everyone who has contributed to the PUL-md and PUL-sm libraries; in particular Robert Baxter and Shari Trewin. I am also indebted to Arthur Trew and Mark Sawyer, both for ensuring I didn't get side-tracked, and for making time available for me to work on this thesis. I would further like to thank; Chris Wendl (Harvard) and David Henty, for collaboration on the GA's decomposition project; Mark Parsons and Julian Parker, for help and encouragement along the way; and also Harvey Richardson, for getting me going on the CM at the beginning.

My parents I must thank for their support, not only during the course writing this thesis, but also throughout my previous education.

Of my friends, I should mention; Claire Cummings and Glenn Marion, for the MSc days and beyond; and also Jo[h]ns Mather and Shapcott for Computer Science consultancy.

My supervisors, Asif Usmani and J. Michael Rotter, deserve my thanks for having more faith that I would finish this than I did.

I would also like to mention my MSc supervisor, Des Johnston, as I feel I should spell his name right in at least one acknowledgements section.

Finally, I would like to thank everyone who has put up with me talking about nothing else for the last few years...

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Robert A. Davey*)

Contents

Notation and Terminology

Terminology

- An unstructured mesh consists of *nodes* and *elements.*
- A dual graph consists of *vertices* and *edges.*
- We find a *decomposition* of a mesh.
- We find a *partition* of a graph.
- Where the distinction is irrelevant we use *decomposition* and *partition* interchangeably.
- k-way, a decomposition over k processors.
- A *sub-domain* is the part of a mesh (graph) assigned to an individual processor.
- EPCC, Edinburgh Parallel Computing Centre.
- PUL, the EPCC Parallel Utilities Libraries.
- PUL-md, the Mesh Decomposition library.
- PUL-sm, the Static Mesh runtime support library.

Notation

General notation:

x a scalar.

$|x|$ absolute value.

$\boldsymbol{x}$ a vector.

$\boldsymbol{X}$ a matrix.

$|\boldsymbol{x}|_2$ or $|\boldsymbol{X}|_2$ Euclidean norm.

Set notation:

$|S|$ the size of the set S.

$\{a, b, c, d\}$ list of members of a set.

$\{s \in S : conditions\}$ sub-set of S for which the conditions are true.

Parallel computing notation:

P set of processors.

k the number of processors (sub-domains).

d_{net} dimensionality of a hypercube network.

h_{ij} hypercube hops between processors i and j.

$t_{latency}$ network latency.

s_m size of a message.

β network bandwidth.

Partitioning notation:

M_{part} a partition (decomposition).

S_p the sub-domain on processor $p \in P$.

l recursive equivalent of k.

Mesh notation:

$\mathcal{M}$ an unstructured mesh.

$\mathcal{N}$ the set of mesh nodes.

η_i a mesh node.

n_η the number of nodes.

$\mathcal{E}$ the set of mesh elements.

$\varepsilon_i = \{\eta_a, \eta_b, \ldots\}$ a mesh element (a set of nodes).

n_ε the number of elements.

d dimensionality of the mesh.

Graph notation:

G a dual graph.

V the set of vertices.

v_i a graph vertex.

n_v the number of vertices.

E the set of edges.

$e_{ij} = \{v_i, v_j\}$ a graph edge between v_i and v_j.

n_e the number of edges.

$w_v(v_i)$ the weight of vertex v_i.

$w_e(e_{ij})$ the weight of edge e_{ij}.

E_{cut} the set of cut edges.

$|A|_v$ the total vertex weight of $A \subseteq V$.

$|B|_e$ the total edge weight of $B \subseteq E$.

Finite element notation:

$\boldsymbol{K}$ the global stiffness matrix.

$\boldsymbol{K}^e$ an elemental stiffness matrix.

Pseudocode:

$x \mathrel{+}= y$ is $x \rightarrow x + y$.

$x \mathrel{-}= y$ is $x \rightarrow x - y$.

$S \uplus A$, for the sets S and A, $S \rightarrow S \cup A$.

Miscellaneous notation:

H an objective function.

$\boldsymbol{K}_m$ the Krylov matrix.

Chapter 1

Introduction

The subject of this thesis is, as its title suggests, efficient parallel computation involving unstructured meshes. Meshes of this type are a common discretisation technique used by numerical methods such as the finite element and finite volume methods. As unstructured mesh calculations arise in a wide variety of scientific and engineering application areas, and parallel computers are now a commonplace high performance computing platform, this is of significant practical importance.

If an unstructured mesh calculation is to make use of a parallel computer, then parts of the mesh must be assigned to the individual processors of the machine; a process referred to as decomposition. For this to make efficient use of the machine the decomposition must be such that the work load on each processor is as even as possible, so that no processor is left idle while waiting for another to complete its work, and such that communication between processors is minimised. The problem of finding a decomposition of a mesh that satisfies these requirements is closely associated with that of partitioning a graph. The graph partitioning problem does not just arise in parallel unstructured mesh calculations, but is also of importance in the design of integrated circuits, task scheduling, sparse matrix factorisation and several other areas.

Given the number of important applications that result in problems of this type, it is unsurprising that considerable research has gone into the development of algorithms for mesh decomposition and graph partitioning. Unfortunately, it can be shown that the computational complexity of solving these problems for large problem sizes is such that no exact solution is ever likely to be found. We must therefore resort to heuristic approaches which give an acceptably good

approximate solution in a reasonable time.

In this thesis we survey existing algorithms that address these issues and examine their merits qualitatively. We then detail the algorithms we have chosen to implement and examine their merits quantitatively, based on various metrics of decomposition quality. To validate these metrics of quality we investigate how the performance of a typical application is related to them, and also examine certain features of the algorithms that are otherwise difficult to assess. We also introduce a novel partitioning algorithm based on optimisation techniques.

1.1 Research Approach and Background

The research approach taken in this thesis results from the background of library development undertaken over a number of years at Edinburgh Parallel Computing Centre (EPCC), hence we will say a little about this background so that the work presented here and the role of the author in this development may be put in context.

The Parallel Utilities Libraries (PUL) project encompasses the development of portable parallel libraries which support application development and free the application programmer from re-implementing basic parallel utilities. The PUL project was one of the original Key Technology Programmes at EPCC and has been ongoing since 1991. PUL draws upon the machine independence and library support offered by MPI, and offers facilities for a range of programming paradigms, including task-farming, regular domain decomposition, parallel I/O and unstructured meshes.

The libraries of interest to us here are PUL-md and PUL-sm; the libraries which EPCC has developed to support unstructured mesh applications. These two libraries respectively address mesh decomposition (hence 'md') and runtime support of static mesh applications (hence 'sm'), and are closely coupled, with the former acting as a serial preprocessor for the latter.

Two industrial collaborations have driven the development of these libraries; namely those between EPCC and British Aerospace, and also with Fujitsu Parallel Computing Research Centre, Kawasaki, Japan.

The collaboration with British Aerospace, which took place in 1995 and 1996, centred around the FLITE3D project [BMT96], which involved the parallelisation

of an existing computational fluid dynamics code used in the design of commercial aircraft. The porting of this code was greatly facilitated by use of the PUL libraries, and in turn furthered their development. The resulting parallel code is now run on by British Aerospace on the Farnborough Supercomputing Centre 256 processor T3D, and forms a valuable part of their aircraft production cycle.

The collaboration with Fujitsu, which ran from 1993 to 1996, focused on the porting, development and optimisation of PUL software on the Fujitsu AP1000 [DPPS95, BD96], a distributed memory MIMD parallel computer. In particular, the PUL-md and PUL-sm libraries were greatly improved and expanded in the the course of the Fujitsu project, and it is during this phase of development that the current author was most deeply involved. At the conclusion of this project a demonstration application was sought to illustrate the use of the libraries, which is the HEAT2D finite element code we shall encounter in chapter 9.

1.1.1 Library Development and Related Work

Work on the PUL-sm library began in late 1992, and predates PUL-md by two years. Much of the initial work for both libraries was carried out by Shari Trewin, with further contributions from Simon Chapple and Killian Murphy. More recent development of PUL-sm, in particular extensions for mesh halos, have been carried out by Shari Trewin and Robert Baxter.

Prior to development of PUL-md proper an EPCC Summer Scholarship[1] project undertaken by Malcolm Allen (Napier University, UK) investigated parallel methods for static mesh decomposition [All93]. In the course of this project parallel methods were investigated but not implemented. However, serial implementations of recursive coordinate bisection (see section 6.6.1) and the Kernighan and Lin local refinement algorithm (see section 6.8.1) were undertaken, as was a mesh registration interface to PUL-sm. While this project influenced subsequent development of PUL-md, neither its code nor implemented algorithms were included in the initial version of the library.

The initial version of PUL-md, as implemented by Shari Trewin, forms the background for the work in this thesis. It implemented dual graph extraction, recursive layered bisection (see section 6.7.1) without Cuthill-McKee (see section

[1]The Summer Scholarship program allows students to work with EPCC for ten weeks over the summer on a variety of topics relating to High Performance Computing.

6.3.4), the initial implementation of the Mob algorithm (see sections 6.8.2 and 7.5.1), and I/O functions designed to interface with PUL-sm. In particular, the partition data structure (see section 7.3.2) and closely related recursive software design (see section 7.3.3) of the decomposition functions date from this version.

The current author began work on PUL-md in 1995, and has focused on further development of the decomposition functions. This has entailed improvements to the initial algorithms (addition of Cuthill-McKee to recursive layered bisection and implementation of the improved 'Mob Complete' version of that algorithm), and the implementation of a substantial number of additional algorithms (all those except the two mentioned above). Concurrent with this Robert Baxter added additional I/O functions relating to mesh halos to the library. This latter stage of development was primarily driven by the Fujitsu project outlined above.

Other, related, work at EPCC includes the development of the parallel version of the HEAT2D demonstration code, which was undertaken by Robert Baxter, and the development of a parallel implementation of the Jostle sub-domain heuristic (see section 6.8.3), known as 'Refine,' undertaken by Mark Parsons. The latter parallel algorithm is not included in PUL-md proper.

The final piece of work we must mention is the 1996 Summer Scholarship project [Wen96] which forms the basis for chapter 10 of this thesis. The concept for this work was originated by the current author and David Henty, who jointly supervised the project. The actual implementation of the algorithms described are entirely due to the student, Chris Wendl (Harvard University, USA).

The version of PUL-md described in this thesis constitutes release version 'PUL-md-2-2.' Like all PUL software it is available free of charge to UK academics. Interested parties (including companies) may also obtain the software under evaluation license.

1.1.2 Publications

Four publications have resulted from the current author's involvement in the development of PUL-md:

[DPPS95] **Unstructured Mesh Partitioning and Improvement on the AP1000**

A short overview of PUL-md and PUL-sm development presented at PCW'95, the Fourth International Parallel Computing Workshop (1995) hosted by Imperial College in conjunction with the Fujitsu Parallel Computing Research Centre. This overview was published in the proceedings of that workshop.

[BD96] **Unstructured Mesh Libraries for the AP1000**

A paper detailing the full three years of collaboration between EPCC and the Fujitsu Parallel Computing Research. This again focused on the PUL-md and PUL-sm libraries and was presented at PCW'96, the Sixth Parallel Computing Workshop (1996) and appears in the proceedings of that workshop.

[BDH97] **Unstructured Mesh Applications at Edinburgh Parallel Computing Centre: Libraries, Applications and Interactive Learning**

A paper which surveys all work carried out at EPCC relating to unstructured mesh applications to date. This covers the PUL-md and PUL-sm libraries, the HEAT2D demonstration code, the FLITE3D project, the 1997 Summer Scholarship project and the EPIC [Wes96, MW97] interactive courseware which allows Web based learning, in this case pertaining to mesh decomposition. This paper was presented at the First Euro-Conference on Parallel and Distributed Computing for Computational Mechanics 1997, Pre-Processing and Solution Procedures, held at Lochinver, Scotland. It is published in 'Advances in Computational Mechanics with Parallel and Distributed Processing.'

[BDT96] **PUL-md Prototype User Guide**

The library's User Guide, which details the application program interface and decomposition algorithms.

In addition to these publications, an EPCC course currently entitled 'Unstructured Meshes: Generation and Decomposition' contains much work originated by the current author, as does the EPIC on-line course material referred to above.

1.2 Thesis Summary

The content of this thesis is arranged in the following chapters:

2 Unstructured Mesh Calculations

We introduce the numerical methods which give rise to unstructured mesh calculations, and contrast them with other approaches. These methods are typified by the finite element and finite volume methods, which we introduce and briefly discuss here. This provides motivation and perspective for subsequent discussions.

3 High Performance Computing

We review the current state of high performance computing, with particular reference to large scale parallelism. We discuss both hardware and software issues relevant to parallel unstructured mesh calculations. In terms of hardware, we contrast the SIMD and MIMD parallel architectures, and examine how network topologies have a considerable bearing on the cost of communication. In terms of software, we contrast the data parallel and message passing programming paradigms.

4 Decomposition

We introduce the factors that affect the performance of a parallel computation, and see that these are, to a first approximation, load balance and communication costs. We then examine, in the most general terms, how computation may be spread across the processors of a parallel platform in an efficient manner.

5 Parallel Unstructured Mesh Calculations

We look at the details of the implementation of parallel unstructured mesh calculations and introduce the halo and shadow node models of mesh distribution. We look in detail at the decomposition of unstructured meshes and see how it is related to the graph partitioning problem. This chapter both motivates the problem this thesis addresses, namely the decomposition of unstructured meshes for efficient parallel computation, and also phrases the problem in such a way as to abstract it from any one particular application or platform.

6 Decomposition Algorithms

We review the algorithms that have been developed for mesh decomposition and graph partitioning, classifying them broadly as global methods or local refinement techniques. This review aims to be as complete as possible, with particular attention placed on those algorithms which are implemented in PUL-md, as the subsequent chapter will make reference to the discussion here whenever some feature of an algorithm implemented is well known.

7 Development of a Mesh Decomposition Library

We examine the programming methodology used in the development of PUL-md, both in terms of its application program interface and the underlying data structures and software design. We discuss each of the algorithms implemented in the library in turn, and define the tunable parameters which control their behaviour.

8 Evaluation and Discussion of Decomposition Algorithms

The title of this chapter is largely self-explanatory; for three example datasets of varying sizes, we have employed PUL-md using a variety of algorithms and a range of settings of their associated tunable parameters and recorded various metrics of quality for the resulting decompositions. This raw data we consign to a later appendix, as it is quite voluminous, even though only a representative range of numerical experiments were performed. Where possible we extract the relevant data and present it graphically, but where this is not possible we refer the reader to the specific data in the appendix. We present our conclusions as to the relative merits of these algorithms and the most favourable setting of their tunable parameters in the summary at the end of this chapter.

9 A Demonstration Application

While the evaluation in the previous chapter is based on purely theoretical metrics of quality, here we study the effects of decomposition quality on the actual runtime of a real application. We use this to investigate the validity of the metrics of quality previously used and to investigate some features of our decomposition algorithms that are difficult to assess in the absence of a real application.

10 A Seed-Based Optimisation Approach to Partitioning

We introduce a novel partitioning algorithm which uses optimisation techniques, in particular genetic algorithms, to find favourable seed vertices in the graph whose positions then determine the full partition. A qualitative comparison of the various optimisation techniques and seed-based partitioning algorithms employed is presented.

11 Conclusions

We summarise our conclusions and review outstanding issues in this chapter.

Appendix A Decomposition Statistics

In this appendix we present the data that forms the basis of our evaluation of the decomposition algorithms implemented in PUL-md, as referred to above.

Chapter 2

Unstructured Mesh Calculations

In this chapter we introduce the numerical methods which give rise to unstructured mesh calculations, and contrast them with other approaches. The class of problems that these numerical methods address are primarily those that can be expressed as partial differential equations (PDEs). This constitutes a very large class of problems, as many of the general laws of nature are most naturally expressed in this form. Areas of application range from structural analysis and fluid mechanics to solid state physics and quantum mechanics.

In all of these cases, we are dealing with some region in which the problem is defined, namely the *simulation domain.* If, as is often the case, no analytical solution exists for the problem at hand, then numerical methods may be employed to give an approximate solution. While PDEs view the simulation domain as a continuum, the numerical methods we are interested in discretise the domain into smaller regions so that valid approximations to the PDEs may be made and the overall solution obtained.

2.1 Discretisation

There are two main types of discretisation that are employed in the solution of PDEs; *regular grids* and *unstructured meshes*, as illustrated in figure 2.1.

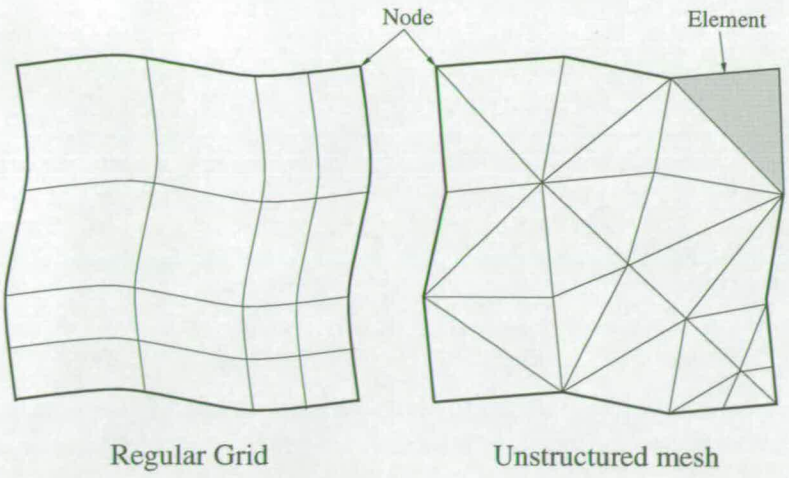


Figure 2.1: Structured and unstructured discretisations

2.1.1 Regular Grids

Regular grids arise from the *finite difference method* of solving PDEs numerically. The approach is to approximate the derivatives in the PDE in terms of the finite difference in the values of the problem variables at neighbouring grid points. The differential equations are thus transformed to a set of algebraic equations accurate only at the finite number of grid points. The algebraic equations may then be solved to give the overall solution.

We have illustrated a situation in figure 2.1 where the grid has been distorted by a simple mapping, but in most cases the grid is orthogonal and regularly spaced. Even with a distorted mesh there are clearly limits as to how complex a geometry can be modelled, which is one of the disadvantages of the method. A partial solution to this is to use many such grids, arranged in such a way that they meet in a congruent manner and fill the simulation domain. The finite difference method may then be employed within each regular grid, so long as the nodes that are shared between grids are treated in such a way as to take this into account. This approach is the multi-block method, which is often used for computational fluid dynamics (CFD) simulations. Although multi-block allows for more complex geometries, the definition of the block structure is a time consuming process, usually undertaken by hand.

2.1.2 Unstructured Meshes

Unstructured meshes arise from the *finite element* and *finite volume methods*, which we shall discuss shortly. The approach taken here is to discretise the simulation domain into a mesh of elements each with simple geometry. These elements are arranged in a completely arbitrary manner and may be of widely varying sizes. Examining the figure once more, we see that a node in the unstructured mesh is a member of a variable number of elements, indicating that, while the array is the natural data-structure for the regular grid, it can not be used to represent an unstructured mesh in a straightforward way. Typical element geometries are triangles and quadrilateral in two dimensions, and tetrahedra, hexahedra and triangular prisms in three dimensions. Generally speaking element types will be homogeneous for a given mesh, although meshes of mixed element type are not unknown. The element geometry may be linear, as shown in the figure, or given in terms of a simple function, such a low-degree polynomial.

Flexibility in fitting an unstructured mesh to a geometry is not the only advantage gained by this approach. We may also fit the mesh to the expected solution, so there is greater accuracy where the solution is changing most rapidly or is of greatest interest. Comparing the two diagrams in figure 2.1, we see that both have greatest density in the lower left hand corner. However, in order for the regular grid to accomplish this, other parts of the grid must also be made more dense unnecessarily, whereas the unstructured mesh can focus on the region of interest alone.

Before moving on we introduce the terminology and notation we shall employ in later discussion of unstructured meshes, as follows.

2.1.2.1 Mesh Notation

An unstructured mesh, $\mathcal{M}$, may be defined in terms of a set of nodes, $\mathcal{N}$, and a set of elements, $\mathcal{E}$, so that $\mathcal{M} = (\mathcal{N}, \mathcal{E})$. An element, $\varepsilon_i \in \mathcal{E}$, consists of a set of nodes, $\eta_a, \eta_b, \ldots \in \mathcal{N}$, so that $\varepsilon_i = \{\eta_a, \eta_b, \ldots\}$, where the ordering of the nodes in ε_i determines their connectivity.

The number of nodes is then $n_\eta = |\mathcal{N}|$ and the number of elements $n_\varepsilon = |\mathcal{E}|$.

2.1.2.2 Static and Dynamic Meshes

If the mesh does not change in the course of the simulation then we say that it is a *static mesh*, if the mesh does evolve then we say that it is a *dynamic mesh* (or sometimes *adaptive*). A static mesh is typically generated by some preprocessor to the main solution program, and may be used for many simulations where boundary conditions or material properties change, but where the geometry is fixed. As a dynamic mesh evolves through the course of the simulation, the mesh generation must become part of the simulation. The mesh and the simulation are coupled by some measure of error in the solution, and the mesh refined in regions where the error is high to give greater accuracy, or coarsened in regions where it is very low to speed execution. This may either be done by disregarding the existing mesh and generating another essentially unrelated to it, or by making local alterations to the existing mesh where necessary, which is usually more economic.

2.2 Finite Element Calculations

The finite element method is the archetypal unstructured mesh calculation. It was originated for structural stress analysis, and was quickly recognised as a very general technique for solving PDEs.

The method is based on extrapolating values at the nodes of an element to give values at any point within the element via some *shape function.* For example, if we consider a triangular element used in an elasticity problem, we may, for given displacements of the element's nodes, write down expressions for the strain at any point within the triangle if we make the assumption that the displacement varies, say, quadratically (i.e. a quadratic shape function). We may then write down an expression for the stress at any point within the element based on its material properties. If we integrate over the element (which may need to be done numerically) then we arrive at a simple matrix which completely defines the element's behaviour in response to nodal forces; this is the *element stiffness matrix*, often denoted $\boldsymbol{K}^e$.

Clearly, the behaviour of the element is related to that of any other element with which it shares a common node, as it is at the nodes that the problem variables are defined, so we must combine the element stiffness matrices into a larger

matrix which describes the whole simulation domain. This process is referred to as *assembly* into the *global stiffness matrix*, often denoted $\boldsymbol{K}$. A matrix equation results, which may be solved to give the overall deformation of the structure being simulated in response to applied loads.

While we have presented a specific example of the use of the finite element method for elasticity, the approach is similar in other application areas, although terminology may vary. For a complete study of the finite element method we refer the reader to [Zie89].

2.3 Finite Volume Calculations

While the finite element method originates in structural mechanics, and is most easily explained in that context, the finite volume method originates in CFD and is most easily explained in terms of fluid flow.

In the finite volume method we consider fluxes in and out of the element[1]. Where the PDE we are studying represents some set of conservation principles, this method is particularly suitable. For example, consider flow of a fluid through the faces of a three dimensional element. We may integrate the flow normal to an element face to give the total flow through that face. If mass, for instance, is conserved within the element, then the total of these flows over all of that element's faces must be zero. Moreover, the flow entering an element through one face must equal the flow leaving another element with which it shares that face. In this way an overall algebraic system of equations may be built up to give the flow in the simulation domain in terms of the flow defined by its boundary conditions.

For a good introduction to the finite volume method we refer the reader to [MV95].

2.4 Summary

We have seen that unstructured mesh calculations are employed to solve a variety of problems of great importance in science and engineering, and so are a topic worthy of study. While we have scarcely done justice to the numerical methods

[1]More correctly control volume, but we wish to use consistent terminology.

outlined in this chapter, and these are not the only applications of unstructured meshes[2], we hope that we have presented enough of their general features in order to put our subsequent discussions in perspective.

[2]Dynamically triangulated random surfaces [BJW90, DM91], with applications in fundamental physics and biology, are just one other example.

Chapter 3

High Performance Computing

In this chapter we introduce both hardware and software considerations which arise as a result of the present day nature of high performance computing (HPC). We review computer architectures with particular emphasis on large scale parallelism and network design, as these will provide the primary motivating factors for the discussion of decomposition that follows in subsequent chapters. We then look at the programming models and languages that have evolved to program such machines. We will see that the development of hardware and software have often gone hand-in-hand, but that emerging standards in programming now allow considerable portability of programs, demonstrating that parallel processing is now a mature discipline of wide applicability.

3.1 Architectures

A computer, in its most basic and general form, receives two types of input; its instruction stream and its data stream. The instruction stream carries the code the computer is to execute, while the data stream carries the input data from memory that the instructions are to be applied to. The computer, having transformed the input data according to the instructions, then outputs the results, typically back down the data stream to memory.

These notions of instruction and data streams allow us to classify computer architectures according to the multiplicity of these streams, as we shall see in the following section.

3.1.1 Flynn's Taxonomy

A classification scheme of the type just outlined may be formed by differentiating between single and multiple instruction streams and also between single and multiple data streams. The four combinations that result give rise to Flynn's taxonomy [Fly66], as illustrated in figure 3.1.

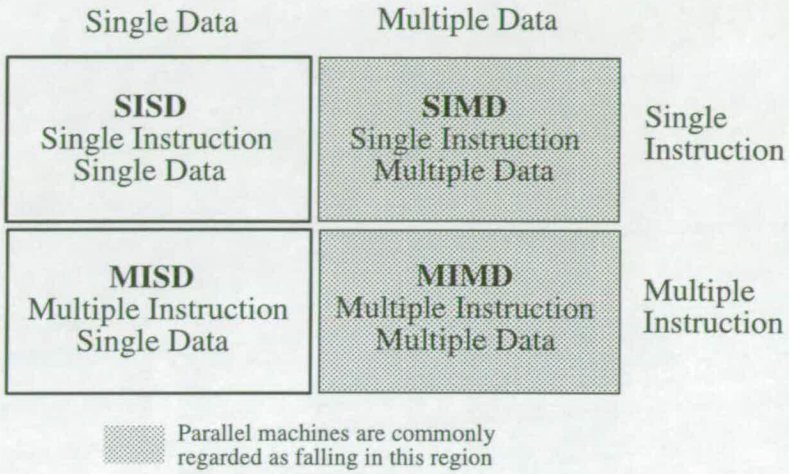


Figure 3.1: Flynn's taxonomy of computer architectures.

We will now examine each of the four architectures defined by figure 3.1 in turn.

3.1.1.1 SISD

The SISD architecture is the traditional computer architecture, otherwise referred to a *serial*, *sequential* or *Von Neumann machine*. Here there is a single instruction stream and a single data stream, both feeding a single processor. The majority of present day computers still fall into this category, as typified by desk-top workstations and personal computers.

3.1.1.2 MISD

The MISD architecture is the first parallel architecture we shall consider; parallel in that more than one processor is used. Here each processor has its own instruction stream, but all processors share a single data stream. Hence each processor is performing a potentially different operation on its own copy of the same data. While a fairly free interpretation of this category allows the inclusion of pipelined architectures (which are essentially a hardware implementation of

the sort of functional parallelism we shall later discuss in section 4.2.2), it is difficult to find good examples of architectures which have been constructed that are clearly in the MISD category.

While MISD is perhaps the least useful category in Flynn's taxonomy, the following two categories, SIMD and MIMD, describe faithfully the two main types of parallel architecture and it is largely for this reason that this taxonomy is useful.

3.1.1.3 SIMD

In a SIMD architecture there are multiple processors, each with its own data stream, but all sharing a single instruction stream. This is often referred to as *synchronous* parallelism, in that the operation of all processors is necessarily synchronised by the instruction stream they share. A typical SIMD architecture is illustrated in figure 3.2. In that figure each *processor element* (PE) consists of a processor and local memory[1]. The program which the PE's execute is held on some *host* or *front end* machine, and is broadcast through the controller over a dedicated control bus (the single instruction stream). The PE's then apply the broadcast instructions to the data in their local memory (the multiple data streams).

This type of architecture would be of limited usefulness if all PE's always performed the same operation, therefore the hardware permits a sub-set of PE's to be inactive and ignore (or not receive) the broadcast instructions. Thus, if we wish half the PE's to perform instruction A and the other half instruction B, then we would have to broadcast instruction A while one half of the PE's were inactive, then broadcast instruction B while the other half of the PE's were inactive. This is clearly only economic if we mostly wish to perform the same operation on a large number of different data items, but a surprisingly large class of problems satisfy this requirement.

As well as operating on the data resident in local memory, PE's may communicate with each other through the network which connects them. As the instructions to do so must also be issued in the manner just described, the architecture is best suited to applications where any communication that occurs is of a similarly uniform nature.

[1]It is therefore technically a *distributed memory* SIMD architecture. See section 3.1.2.

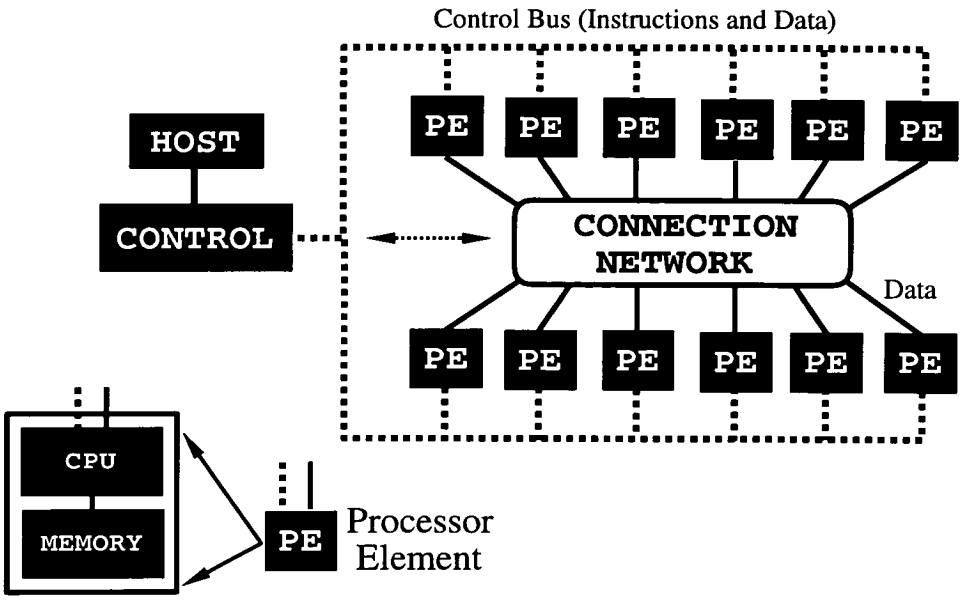


Figure 3.2: A distributed memory SIMD architecture. Each PE consists of processor and memory.

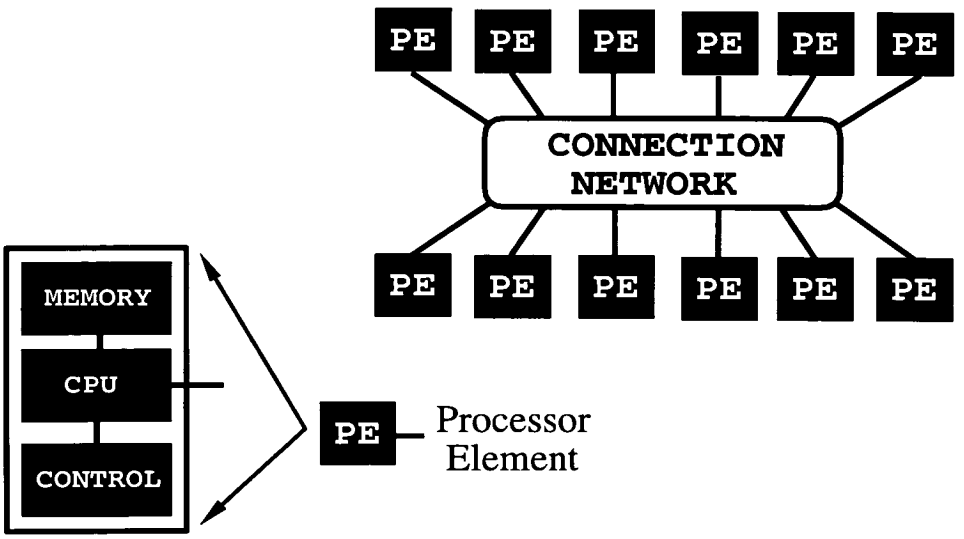


Figure 3.3: A distributed memory MIMD architecture. Each PE consists of processor, control and memory.

These considerations lead to SIMD architectures being particularly suited to operating on large arrays of data and indeed often have connection networks which reflect this (see section 3.1.3), hence they are sometimes referred to as *array processors.*

This architecture is typified by machines such as the Thinking Machines Corporation Connection Machine 2 series (CM-2 and CM-200), the ICL/AMT Distributed Array Processor (DAP), both of which are no longer in production. While the MIMD architecture which we shall discuss next is now more prevalent, SIMD machines are still produced, for example the Alenia Spazio/Quadrics APE100 and its proposed successor, the APEmille.

3.1.1.4 MIMD

In a MIMD architecture there are again multiple processors, but now each has its own data *and* instruction stream. There is no longer any synchronisation imposed by the instruction streams, as each is independent; this may therefore be termed *asynchronous* parallelism. A typical MIMD architecture is illustrated in figure 3.3, where we see that there is no longer any front end machine controlling the PE's. If, as in the figure, each PE has its own local memory, then each is a SISD device in itself, each obeying its own program and operating on its own data.

Returning to the example used in the previous section, where half the processor perform instruction A and half instruction B, we see that these instructions may now be carried out concurrently in the MIMD architecture by simply running two different programs (one containing instruction A, the other B) on the two halves of the machine.

If the PE's are to cooperate on some task then they must communicate but, unlike the SIMD architecture, the PE's must coordinate this communication amongst themselves without the aid of a front end machine to oversee the process; for example, if PE 1 wishes to pass some data it holds to PE 2, then not only does the program for PE 1 need to contain instructions to send the data, but the program for PE 2 will generally need to contain instructions to receive it. While this *message passing* is more complex than the uniform communication typical of SIMD architectures, it is also much more flexible, as essentially arbitrary communication patterns are possible.

These considerations make MIMD the most flexible and widely applicable parallel

architecture; certainly any task that can be accomplished on a SIMD machine can be accomplished just as easily on a MIMD machine, but the reverse is not true. While this partially explains the predominant use of the architecture, another consideration is the fact that each PE is a SISD device and, since such devices are commonplace, a MIMD machine may be built from proprietary components (off the shelf processors and even proprietary interconnect) making construction more economic than SIMD machines that may require bespoke components.

The most familiar example of a MIMD architecture is the simple network of workstations; with the appropriate software communication across standard Ethernet allows message passing and the use of the network for concurrent processing. For higher performance dedicated parallel machines with high speed networks are required, as typified by the Cray T3D and T3E, the Meiko Computing Surfaces such as the CS-2, the Intel Paragon, the IBM SP2, the nCUBE, the Thinking Machines Corporation Connection Machine 5 (CM-5) and many others.

3.1.2 Shared versus Distributed Memory

In the example architectures we used to illustrate SIMD and MIMD (figures 3.2 and 3.3) each PE has its own local memory and can only access data residing in another PE's memory by communicating with that PE. When each PE has its own local memory like this we say that memory is *distributed.* However, for many machines this is not the case and memory is *shared* by all PEs, each having access to all memory addresses via the connection network.

A potential disadvantage to this approach arises when all processors need to communicate with a single bank of physical memory and there is the danger of memory access bottlenecks. This makes it difficult to construct shared memory architectures of this type with very large numbers of processors. Nonetheless, this architecture, know as the *symmetric multi-processor* (SMP) model, has proved very successful for small scale parallelism. Many machines operating as servers can benefit from this architecture, with the operating system scheduling the many independent processes that are typically run on such machines to run on different processors concurrently, and in a way that is transparent to the user (or users). SMP machines of this type include the Cray CS-6400, the DEC AlphaServer, the Sequent Symmetry, the SGI Challenge/POWER Challenge and the Sun Enterprise Server, to name but a few.

To get around the difficulties encountered if all processors access a single bank of physical memory, it is possible to physically distribute the memory (in the manner of a distributed memory MIMD design), but still maintain a *shared* global address space. Machines of this type are known as non-uniform memory access (NUMA) machines, as the access time for a memory address physically local to a processor is faster than for a remote access. The distinction between distributed memory MIMD and NUMA is subtle, but the latter allows one PE to access memory addresses physically located on another without the second PE's cognisance, while the former does not. The Cray/SGI Origin2000, the HP-Convex Exemplar and the Sequent NUMA-Q Series are good examples of this architecture.

3.1.3 Network Topologies

So far we have treated the connection network for a parallel architecture as a black box, in that its structure has had no bearing on the taxonomy we have been discussing. However, the details of the network's construction will have particular bearing on subsequent discussions, especially when we turn our attention to the cost associated with communication. In any event, the efficiency of the connection network is fundamental to the efficiency of any parallel machine as a whole, and so we will now examine some typical network structures.

There are three main types of network:

- Bus networks
- Switching networks
- Point-to-point networks

In a bus network, all PEs share one communication channel which can only carry one message at a time. Thus the performance is limited by the *bus bandwidth* (i.e. the amount of data it can transmit in unit time), which will not scale as the number of PEs attached to it increases and is therefore of relatively little interest to us, although it is worth noting that Ethernet falls into this category.

Switching networks are dynamic connection structures with active elements, such that varying connection patterns can be engaged during execution. Examples of such networks include crossbar switches, delta networks, Clos networks and fat trees. Of these, we shall only detail the latter, as illustrated in the left-most

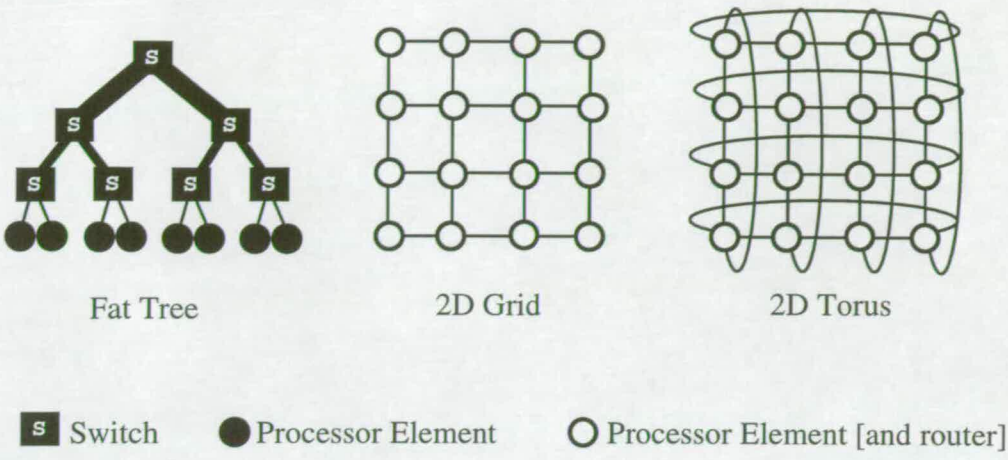


Figure 3.4: A fat tree switched network (where line thickness indicates bandwidth), and 2D point-to-point networks

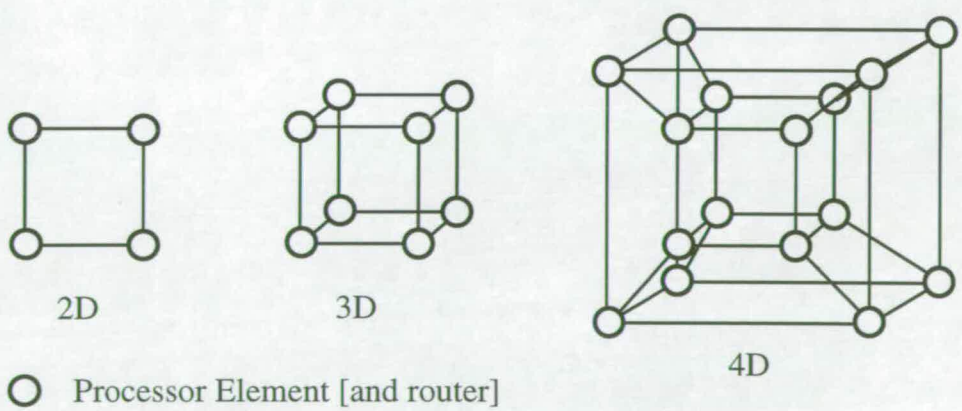


Figure 3.5: Hypercube point-to-point networks in 2, 3 and 4D.

diagram of figure 3.4. The fat tree illustrated is binary but this is by no means always the case; the important concept is that a message from one PE to another travels up the tree only so far as is needed in order for the switches to pass it on to its destination. The tree is termed 'fat' as a switch higher up the tree has more connections routed through it, although not always as many as the sum of the branches below, as most communication is likely to be local. This type of network is employed in the Thinking Machines CM-5.

Point-to-point networks are most common, where connections are directly from PE to PE, although special routing hardware often takes care of communication destined for other PE's (*cut through routing*), so that computation on the PE need not be halted for it to handle the message itself. The structure of the network can vary widely, from vectors and arrays (or grids), to rings, complete graphs or hypercubes. Often, if a grid is used, the connections at the edges of the array wrap round to form a torus, as shown in the right-most diagram of figure 3.4. This type of network is employed by the Cray T3D and T3E, where a torus in three dimensions is used.

The hypercube, in varying dimensions, is an attractive structure for network design, as no two PEs are ever further apart than a number of connections equal to the dimensionality of the network. This can be seen in figure 3.5; if we consider the square, which is the 2-dimensional hypercube, then we see that the greatest distance is from opposite corner to opposite corner, which involves traversing 2 network connections, or *hypercube hops* as they are known; similarly, opposite corner to opposite corner for the cube takes 3 hypercube hops; finally, the tesseract, or 4-dimensional hypercube is 4 hops from corner to corner.

We shall have more to say on the cost associated with network distance in section 5.3.1, but for now will simply comment that it is generally true that the time taken for a message to reach its destination increases with network distance for almost any type of network, particularly when there is contention for network resources.

We note that if a binary tree or hypercube network is used, then the number of PEs in the machine must be a power of two. This is often the case for machines that use other network structures as well, for this permits a fixed number of binary digits to specify each PE's identity. It is also often the case that when a large parallel machine is shared by many users the number of PEs assigned to each user is a power of two, for the same reason (this is true for the Cray

T3D). We emphasise this now, as we will later encounter methods for dividing a computational problem up over a number of processors which is assumed to be a power of two.

3.2 Programming Paradigms

We now turn our attention from the hardware of parallel platforms to the programming paradigms that have evolved to make use of them. Historically, although parallel programming languages and libraries fell clearly into one of two camps; the data parallel and the message passing, there was a lack of standards that hindered the portability of parallel codes between platforms manufactured by different vendors, as each had their own proprietary software. Today emerging standards are gaining wide popularity and allow a well written parallel program to be run on a variety of platforms with relative ease.

3.2.1 SPMD

Sometimes used as a classification of hardware, the single program multiple data (SPMD) programming paradigm is more correctly a style of MIMD programming, although one so widely accepted that many operating systems enforce its use. SPMD simply entails the execution of the same program by every PE of a MIMD machine. This does not reduce the flexibility of the resulting code, as can be seen if we return to the example we used in sections 3.1.1.3 and 3.1.1.4. So long as each PE is aware of its own identity and, if we may assume for the moment that this is given as an integer, then the following pseudocode fragment illustrates this:

```
if (PE identity even)
   execute instruction A
else
   execute instruction B
endif
```

Just as in the MIMD case the instructions A and B are executed concurrently, each on half the PEs. We see that any MIMD set of heterogeneous programs may be combined into one larger program and run in SPMD style in this way

without loss of generality. The only down-side to this is that the executable code occupies more memory, but it is rare that more than a handful of different operations need be executed concurrently, so this is not often a consideration.

3.2.2 Data Parallel

In the *data parallel* programming paradigm the programmer writes code in a language designed with parallel execution in mind, but leaves many of the details of how that execution is performed to the compiler. A single program results, where concurrent operations are carried out in parallel automatically. Communication takes place either implicitly, where there are references to non-local data, or explicitly where language communication intrinsics are called. Each processor performs the same operation on different data and, from the programmer's point of view, there is a single thread of control, a global name space and [loosely] synchronous execution.

This programming paradigm is most closely associated with, but no longer restricted to, SIMD architectures, particularly in the form of array processors. On a synchronous SIMD machine, the instructions in a data parallel program are naturally synchronous, but data parallel compilers are also available for MIMD machines, where synchronisation is typically only imposed by some program constructs, such as the completion of a loop.

The most characteristic feature of data parallel programming is its ability to treat whole arrays as single entities. For example, consider a simple inner product of two conformable vectors, `A` and `B`. In a traditional serial language we would sum the product `A(i)*B(i)` in a loop over `i`. A data parallel equivalent might be `SUM(A*B)`, where the product `A*B` is carried out element-wise over the *whole* of the two vectors, and the resulting vector summed by the `SUM` intrinsic. When executed on a parallel platform, each processor may perform the array element product concurrently for its local data (we assume each PE is responsible for a portion of the overall distributed vectors `A` and `B`). The summation of these products, and the communication it entails, proceeds transparently to the user. Intrinsics for common array operations, such as matrix multiplication, sum, transpose and so forth are usually provided[2].

[2] A inner product intrinsic would almost certainly exist also, but we wish to use the previous example to illustrate message passing in the next section, so did not use one here.

While the compiler can ensure concurrency in such operations with little difficulty, it generally requires additional information from the programmer to specify how data is distributed over the available processors of a parallel machine for maximum efficiency. This is often done by means of *compiler directives*, which are not executable in themselves and generally appear as comments of an identifiable format in the code. Typical distributions are simple *block* or *cyclic* layouts based on array index[3].

If an array is distributed in such a way, any reference to an array element not local to the processor wishing to perform calculation upon it will result in communication occurring; this is transparent to the programmer except in so far as performance is concerned. Communication also occurs during execution of intrinsics such as the `SUM` used above, or when explicit communication intrinsics are called. The functionality of the latter is usually limited to regular patterns of data motion. A good example of this would be a '`CSHIFT`' operation, where all the elements of an array are shifted in a circular manner along a specified dimension, equivalent to `A(i)=A(mod(i+s,n))` for an `n` dimensional array.

Historically data parallel languages have been vendor specific dialects of common languages, predominantly Fortran[4]. This hindered portability, although many concepts were common to these dialects. In an effort to standardise these dialects the High Performance Fortran (HPF) standard was drafted [Hig93] and has now gained wide acceptance with compilers available for most common platforms.

HPF is an extension of the Fortran 90 standard which itself extended Fortran 77 to include, amongst other things, operations on whole vectors and arrays (it may have been noticed that both the `A*B` syntax and the `SUM` and `CSHIFT` intrinsics are included in Fortran 90). HPF extends Fortran 90 by adding direct support for data parallel programming through compiler directives which determine data distribution, and provides added intrinsic and library functions to support specifically parallel operations.

Often, HPF compilers make use of lower level parallel software libraries such as threads or message passing systems; indeed, if the latter is implemented for a

[3]These correspond respectively to the lexicographic and cyclic partitioning methods we shall encounter in chapter 6 applied to array index. See [KLS+94] or the HPF standard itself [Hig93] for a more formal description.

[4]A data parallel C, know as C*, was available for the Connection Machine series of computers from Thinking Machines [Thi93a]; a Parallel Pascal was developed by NASA [Ree84]; parallel dialects of functional languages, notably Lisp and Prolog, also exist - [Brä93] contains a short survey of these.

MIMD architecture, the result is a good example of SPMD programming.

3.2.3 Message Passing

Unlike data parallel programming, *message passing* programming makes use of standard languages such as C and Fortran, but adds communication through calls to special library routines. The communications library forms the *message passing system* (MPS) and will typically implement a variety of *point-to-point* (i.e. single PE to single PE) and *collective* (multiple PE) communication operations.

The message passing model assumes that many separate programs are run concurrently, and that the only interaction between them takes place through the MPS, thus all variables are private within each program. While this is the only way a distributed memory MIMD machine may be programmed at the lowest level, message passing is also a useful paradigm for shared memory architectures, where it can be less cumbersome than multi-threaded programming, as it may avoid the need for locks and semaphores to coordinate memory access and ensure determinism[5].

Message passing is a much more flexible programming model than data parallel, as the programmer has full and unrestricted control of data distribution and communication. Of course, this flexibility passes on to the programmer many of the responsibilities that a data parallel compiler automates, but it also allows much more freedom to tailor code for maximum performance.

Message passing point-to-point communications usually occur in a send/receive pair, as we see from the following illustrative SPMD pseudocode fragment:

```
if (PE 1)
   mps_send(task, PE 2)
else if (PE 2)
   mps_receive(task, PE 1)
   if (task == 'A')
      execute instruction A
   else if (task == 'B')
      execute instruction B
   endif
endif
```

[5]Lack of synchronisation can result in race conditions occuring, leading in turn to unpredictable behaviour. We say programs that avoid this are *deterministic*, in that they always produce the same output given the same input; clearly a desirable property.

Here we see PE 1 sending the data in `task` to PE 2, which is awaiting the message with a corresponding receive. It then executes either `instruction A` or `B` according to its value. In this example the message sent is just a simple scalar flag, but the MPS will usually support messages which are entire arrays or even arbitrary data types.

Collective communications include operations such as all-to-all, gather/scatter or global data reduction functions. No only do these facilities save the programmer from re-inventing common communication operations, but they also allow the MPS library implementors to take advantage of the underlying hardware on a given platform, as the most efficient algorithm may be influenced by such factors as network topology.

An example of collective communication occurs if we consider again the implementation of a simple inner product, as we did in the previous section. In the message passing model, there is no concurrent equivalent of the array product `A*B`, as there is no assumption that arrays are the favoured data structure and neither is there any global name space. Thus, each processor must sum the product `A(i)*B(i)` in a loop over `i` for its local data (we assume each PE has a portion of the overall distributed vectors stored in the *local* vectors `A` and `B`). If this local sum is stored in some temporary scalar `T`, then each processor would subsequently call some MPS global reduction function, such as `mps_sum(T)`, in order to know the overall sum which is the complete inner product.

There are two prevalent MPS interfaces today:

PVM - Parallel Virtual Machine [SGDM94]

MPI - Message Passing Interface [For94]

PVM development began in 1989 at Oak Ridge National Lab in the US and was originally designed to operate on heterogeneous networks of workstations, and still contains important features for supporting applications in such environments. PVM implements communication operations of the type outlined above with interface bindings for Fortran 77, C and C++. In addition, it allows dynamic process creation and provides a standard method of configuring the parallel machine.

MPI is a software standard, defined and maintained by the MPI Forum, with its first specification completed in 1994. The rigor of the standards procedure has helped MPI reach a prime place amongst message passing systems, offering real

portability between HPC platforms. Many vendors of HPC systems now offer native MPI support on their machines, and many generic versions are available free of charge.

The 1994 standard (version 1.1) defined only the interface for Fortran 77 and C; it said nothing regarding process management and did not allow dynamic process creation, but otherwise included many of the features of PVM. In addition it added the concept of a 'virtual topology,' which allows the abstraction of the application communication topology from that of the underlying hardware. The recently published update to the standard (version 2.0, 1997) has C++ and Fortran 90 bindings, additional functionality for dynamic processes, one-sided communication (useful for shared memory architectures), and parallel I/O.

An important feature of MPI that was present from its first definition is that of communication context. Every communication occurs only within a specified context and this allows the development of third party parallel libraries whose communication is insulated from all other messages. In section 5.1.6 we shall encounter the PUL-sm library developed at EPCC for the runtime support of parallel unstructured mesh applications, which forms an example of such a parallel library; without the idea of communication context such a library could not have been robustly written.

Chapter 4

Decomposition

In order to make use of a parallel computer the problem at hand must be sub-divided in some way, either explicitly by the programmer or implicitly by the compiler (as is the case in the data parallel programming model). We refer to this sub-division as *decomposition* of the problem.

Where the problem is one of physical simulation, we will be simulating a finite region, the problem *domain*. In this case, the parts into which we have divided the domain, we refer to as *sub-domains*. A sub-domain is therefore that part of the simulation domain with which an individual processor is concerned.

The way in which this decomposition is done is crucial to the performance of a parallel program. In this chapter we will consider the factors which affect performance and see why this is so. Additionally, we will briefly consider the main types of decomposition in common use.

4.1 Efficient Parallel Computation

There will almost always be overheads involved in the use of a parallel computer which are not present in the use of a serial machine (the exception being *trivial parallelism*, see section 4.2.1). These overheads come from two sources; firstly, any time a processor spends idle while waiting on another is wasted, and, secondly, any time spent communicating between processors does not directly aid in the solution of the problem. These are *load balance* and *communication overheads*, respectively.

4.1.1 Load Balance

Load balance simply refers to ensuring that each processor has the same amount of work to do. If there is any imbalance in the load on the processors then some must be idle while waiting on others.

Although this is a simple concept and, indeed, in many instances is straightforward to ensure, it may be complicated in practice. An example of this would be a case where the load is determined by the data in such a way that it is not known in advance. Another would be where the program was running on a workstation cluster shared with other users or made up of a heterogeneous mixture of machines, in which case the load would have to be tailored to the computing resources available.

4.1.2 Communication

Communication overheads arise as a result of decomposition. The parts into which the problem has been sub-divided are not, in general, independent and so must communicate in order to cooperatively solve the problem.

The actual cost of communication will be determined by the platform on which the program is running. However, the main factors contributing to communication costs are:

- The volume of communication; that is the number of bytes of data sent.
- The frequency of communication; there is a start-up cost associated with each message sent.
- The 'distance' between communicating processors across the network; the time taken for a message to reach its destination may, for example, depend on the number of routers it has to pass through.
- Contention for network resources; one message may get held up while another is using the same part of the network, particularly if all processors are communicating at once, which is often the case.

The relative cost of communication and computation may also be an issue. For example, it may be more efficient for a processor to recalculate a result locally than for it to fetch it from another processor that already has that result.

4.1.3 Measures of Performance

We would like to be able to quantify the performance of a parallel program, both in relation to a good serial implementation (which may well contain algorithmic differences), and also in relation to the number of processors used to run the parallel code.

By comparing the execution times of serial and parallel codes, running on identical processors, and also by studying how the parallel execution time varies with number of processors, we can arrive at some measures of performance, as follows.

If we denote the serial and parallel execution times as t_{serial} and $t^k_{parallel}$, respectively, then we can define the *total speed-up* obtained by parallelisation as

$$S_{tot} = \frac{t_{serial}}{t^k_{parallel}}, \tag{4.1}$$

and the *total efficiency* as

$$E_{tot} = \frac{S_{tot}}{k} = \frac{t_{serial}}{kt^k_{parallel}}. \tag{4.2}$$

Where k is the number of processors used by the parallel implementation.

It is often the case that a good serial implementation is not available, cannot be run on a single processor of the parallel machine, or that we are only interested in examining features of the parallel algorithm, for example looking at how decomposition affects performance. It is therefore common to compare the execution time of the parallel code running on a single processor, $t^1_{parallel}$, with that of the parallel code running on k processors.

We define the *parallel speed-up* as

$$S_{par} = \frac{t^1_{parallel}}{t^k_{parallel}}, \tag{4.3}$$

and the *parallel efficiency* as

$$E_{par} = \frac{S_{par}}{k} = \frac{t^1_{parallel}}{kt^k_{parallel}}. \tag{4.4}$$

4.2 Types of Decomposition

4.2.1 Trivial

If a problem has a number of similar, independent parts, and each of these can be accommodated on a single processor, then these parts may be computed concurrently with none of the overheads previously discussed. Such a problem is sometimes termed *embarrassingly parallel.*

To illustrate this, consider a simple engineering code which performs a structural analysis. The user wishes to vary a parameter which describes a feature of the structure, say the thickness of a shell, and study how the strength of the structure varies with this parameter. If we have k processors available, then k parameter values may be studied concurrently in the time it would take us to perform one analysis on one processor.

Clearly, if the time taken by a single structural analysis is t_{sa} then the parallel speed-up is $kt_{sa}/t_{sa} = k$ and the parallel efficiency is 100%. As there are no overheads associated with this trivial parallelism, this is often taken as providing an upper limit on the speed-up that may be obtained by any parallel program. In practice other issues, such as cache utilisation, may invalidate this limit.

This example is typical of trivial decomposition; we are not so much sub-dividing a problem, as much as running several problems at once. As parallel computing is usually applied to problems that would be too large to fit on a single processor, trivial decomposition is of little interest.

4.2.2 Functional

Functional decomposition seeks to sub-divide the problem into a number of *tasks*, here akin to subroutines or code blocks, each of which can be run on a separate processor. The output of one task is communicated from the processor on which it is running to another, where it is used as the input for another task running there.

A typical structure for a parallel program which used functional decomposition would be a *pipeline*, as shown in figure 4.1. Here a sequence of data items are read in, processed by task A, passed to task B for further processing, and so on, through task C to output.

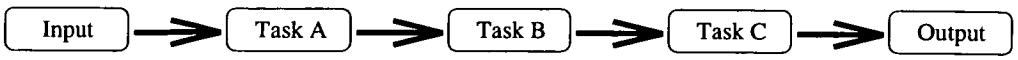


Figure 4.1: Functional parallelism, a pipelined approach.

In a serial implementation of this program the tasks would simply be executed in sequence, looping over data items. If we consider a item of data flowing through the pipeline then it is processed in sequence, just as in the serial case. This means that, if we are only processing a single data item, then the execution time of the serial and parallel codes (running on similar processors) will be the same, bar the additional communication costs for the parallel code.

The performance of a pipeline is, therefore, only seen if there are a large number of data items to be processed. In other words, there is a start-up cost associated with filling the pipeline at the start of execution and, similarly, a close-down cost for emptying it at the end. At both of these stages the pipeline is not full and some processors will be idle.

This is not a problem if there is sufficient data to make pipelining worthwhile. However, more serious issues can arise in the implementation of such a program.

Our example will perform well if there are five processors for it to run on, but if there are less it will not run at all, and if there are more they will remain idle and cannot be taken advantage of. Utilising all processors would involve altering the program to take the number of processors into account, which may well have to be done by hand.

Another issue is that of load balancing; any load imbalance will result in a bottleneck in the pipeline, with a corresponding decrease in performance. As the work involved in each of the tasks is a function of the code executed as part of that task, any alteration in that code will alter the load balance, making the code hard to maintain.

However, if these issues can be dealt with, for the problem in question, then functional parallelism can be very effective. This is particularly the case where a small number of very powerful processors are to be used.

4.2.3 Data

The majority of parallel programs decompose a problem by decomposing the corresponding data. Where identical operations are being performed on each

part of the data, the amount of data that a processor has to deal with will be a measure of the work it has to do; we term this a *balanced* problem. However, this is not always the case and the work required may be determined by the content of the data; we term this an *unbalanced* problem.

To categorise decomposition further is problematic as it is essentially problem dependent and has potentially as many solutions as there are applications. However, we can present some canonical examples which illustrate common approaches and will help to put later discussions of the decomposition of unstructured meshes (which we will cover in section 5.3) in context.

4.2.3.1 Regular Grid with Balanced Load

A simple balanced problem is the regular grid, such as would be found in a finite difference calculation, for example. Here we have an array of data, each element of which is treated similarly and only interacts with its immediate neighbours.

It is easy for us to decompose this problem so as to obtain perfect load balance. As data elements are all treated identically an even distribution of work is achieved by having an equal number of data elements on each processor.

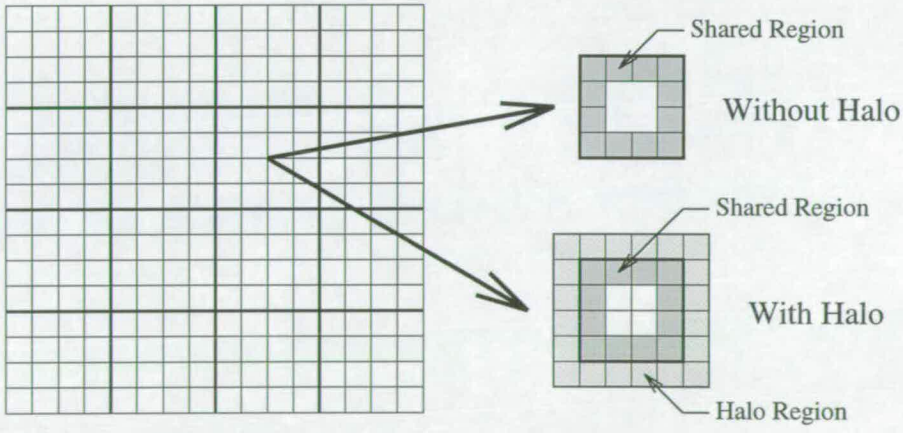


Figure 4.2: A regular 16x16 grid decomposed for 16 processors, with and without halos.

Looking at the regular grid in figure 4.2, we see that we could simply assign each row of the array to a different processor. This would certainly give load balance, but would result in an large amount of communication. Each row would have to communicate with its two neighbouring rows, as we know interactions are nearest neighbour. This is not a great problem for small arrays, like the one in figure 4.2, but as the arrays size grows so does the amount of communication.

The best decomposition is generally[1] that shown in the figure. Here each processor owns a square section of the array and will therefore only need to communicate at borders to neighbouring squares.

The distinction between these two decompositions show an important feature of parallel programming. The efficiency of the row decomposition decreases as the problem size increases, whereas the square decomposition does not. We therefore say that the latter is *scalable.*

It is clear from this discussion that what we are seeking to do is to maximise the ratio of useful calculation to communication, and that this is related to minimising the 'surface area' of the sub-domain assigned to each processor.

Two possible implementations of this decomposition are also shown in figure 4.2. To perform a calculation for a data element in the shared region will require knowledge of the neighbouring data elements and these are on another processor. We could fetch these elements, one at a time, as we need them (the 'without halo' option) but we would incur a cost for starting each separate communication. The usual solution is to add a *halo* around the local part of the array and swap data with its neighbours for the corresponding parts of the halo in one communication per neighbour.

An example of this type of problem can be found in [Boo96], which details the application of this type of decomposition to Quantum Chromodynamics.

4.2.3.2 Regular Grid with Unbalanced Load

If we consider a variation of the previous problem where not all data elements are treated identically then we have an unbalanced problem. Consider the problem of ocean modelling around a land mass. We superimpose a regular grid over the land mass, but now those data points which lie over land will be idle.

An example of this type of problem can be found in [Gwi95], which details the work of the OCCAM ocean modelling consortium.

The load imbalance that the presence of the land mass will cause, if we use the previous decomposition, is clear from examining the work load of the processor

[1] In some instances it may be preferable to have fewer larger messages, as we do for the row-based decomposition just described. The reasons for this will become clearer when we discuss communication *latency* in section 5.3.1.

marked 'A' in figure 4.3, which has only three active data elements, while several others have all sixteen data elements active.

If the shape of the land mass is known in advance then a good decomposition can be found where the sub-domain a processor owns is irregular, as shown. This in itself may be time consuming, but worthwhile if many calculations are to be performed.

If the shape is not known in advance, and it is considered unprofitable to find a decomposition in the manner just described, other options exist.

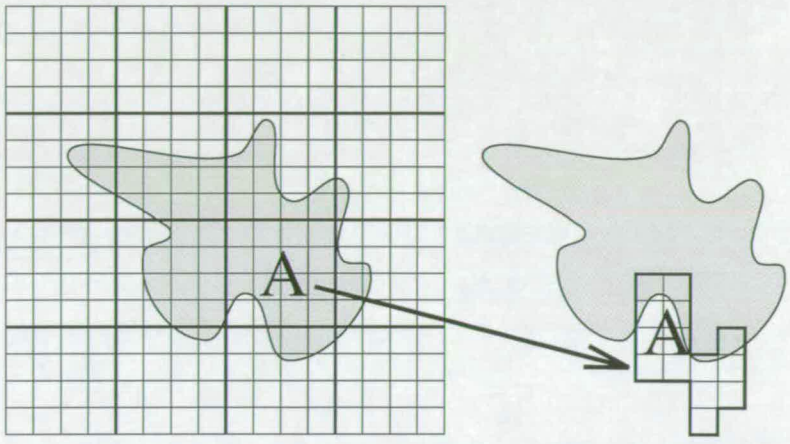


Figure 4.3: A regular grid with an inactive region.

One other option would be a *scattered* decomposition. Here we almost abandon the attempt to keep interacting (i.e. adjacent) data elements together and assign data elements to processors essentially at random (possibly *cyclically*) and trust that, on average, each has the same work load. The problem here is that this will tend to maximise communication costs. To overcome this deficiency we may group interacting elements together into *grains* of some size smaller than the total we wish to find in a sub-domain. By tuning this grain size we can trade off the higher communication cost of small grains against the improvement in load balance they bring.

An illustrative example of this approach occurs in N-body or molecular dynamics simulations, where a number of discrete particles interact according to the forces they exert on one another. If the space in which the particles move is discretised into a simple cartesian grid, then the number of particles in each grid cell at a given time is unknown. A scattered decomposition of the grid cells will, if there are enough cells per processor, even out the number of particles per processor, and hence ensure reasonable load balance. If the forces are of limited range (like

the Lennard-Jones potential, for example) then communication costs increase as we decrease cell size, as nearby particles are more likely to be resident in cells assigned to different processors. The trade off between load balance and communication costs evidently influences the optimal choice of cell (i.e. grain) size.

An example of this type of decomposition applied to very large scale cosmological simulations can be found in [MPPC97], where gravitational forces are represented, at short range, by particle-particle interactions, and, at long range, by particle-field interactions. Other examples occur in biological population modelling and LU factorisation of matrices [FJL+88, FWM94].

4.2.3.3 Task Farming

Task farming provides a means of *dynamic* load balancing, that is load balancing that must be carried out at run-time. If the problem can be sub-divided into grains which can be processed independently, but which have varying, and possibly unknown, amounts of work associated with them then it is a suitable candidate for task farming.

If we return to the example in section 4.2.1, of multiple, simple structural analyses being performed, we see that the perfect speed-up rests on the assumption that t_{sa} is a constant. If this is not the case, and t_{sa} is a function of the input parameters, then the parallel execution time will be that of the processor which finishes last and the speed-up inferior.

Now it may be the case that many more analyses are to be performed than there are processors available. In this case we may use a task farm, as shown in figure 4.4.

The task farm consists of a single source, several workers and a single sink (which is often also the source). Here a *task* consists of performing the analysis for a particular parameter set. The task farm then operates as follows:

- The sink handles input and initially hands out parameter values to the workers. Thereafter, it waits for a request from a worker for a new set of parameters.
- The workers perform the structural analyses for the parameter set they have. Their results they communicate to the sink and then request another

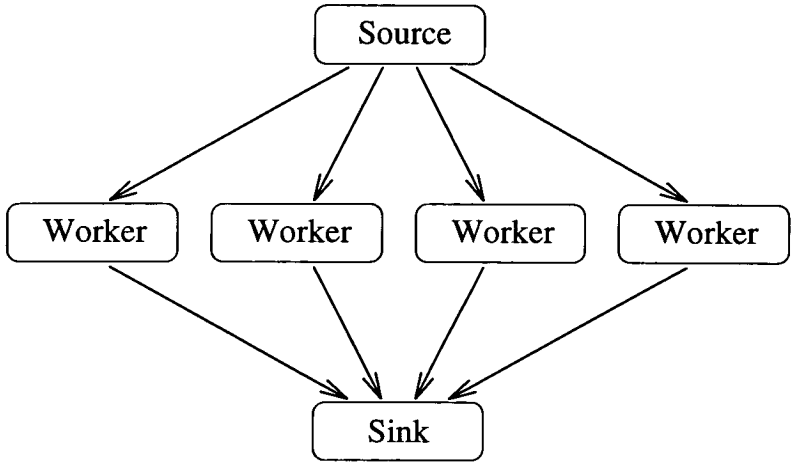


Figure 4.4: Task Farm

parameter set from the source.

- The sink receives results from the workers, performs any additional processing of results and handles output.

This process will tend to even out the work load between workers, as they only request more work once they have finished the work they have. Inefficiencies may arise if, at the end of execution, some workers are still calculating their last task long after many others have finished. If the work associated with a task (i.e. parameter set, in this example) is known, or may be estimated, then the source can give out the more intensive tasks first to alleviate this problem.

An example of just such an application of task farms for engineering design is to be found in [Boy94], which details their use for the design of underwater storage tanks.

Chapter 5

Parallel Unstructured Mesh Calculations

In this chapter we look at issues arising in the implementation of parallel unstructured mesh calculations. We look at two models of mesh distribution (*halos* and *shadow nodes*) and how communication between processors occurs. The implications this has are examined using the finite element method as an example.

We also look in detail at the decomposition of unstructured meshes and the related issues of *graph partitioning* and *mapping*. This necessitates an examination of platform and application communication, so that a model can be built which will allow evaluation of decomposition algorithms. This model, the *dual graph*, is also fundamental to many of the decomposition algorithms which will be introduced in later chapters.

The emphasis here, and indeed for the rest of this thesis, is on implementation in a message passing environment, for the reasons stated below. However, this should not be seen as restricting the scope of these discussions, as the problems that arise during implementation in that *or* the data parallel programming model are essentially the same; reducing communication costs while maximising useful computation.

5.1 Implementation in a Message Passing Environment

In order for an unstructured mesh calculation to be carried out in parallel the mesh must be decomposed into the sub-domains that will be resident on each processor. The discussions in chapter 4 regarding the decomposition of regular grids are just as relevant here. In the terms of that chapter, we are dealing with a data decomposition; balanced if all parts of the mesh are treated equally, unbalanced otherwise.

The question then arises as to what the basic unit of sub-division is to be. We have two choices; elements or nodes. Given that a considerable proportion of the computation involved in an unstructured mesh calculation is often associated with determining the properties of a mesh element, the element is the preferred unit.

If we consider a finite element calculation, for example, we see that each elemental stiffness matrix must be calculated and that these calculations are independent. If we distribute elements across processors then this part of the calculation is embarrassingly parallel. However, if we distribute nodes then communication will be necessary wherever the nodes of a given element are not all resident on the same processor. The former is clearly more efficient.

Having decided on the basic unit we then need to find a good decomposition. We will discuss how this is done in detail in section 5.3 and following chapters, but for the moment we assume we have some reasonable decomposition and that the mesh is static.

The issues that then arise are how to perform scalable input and output, how to communicate data between sub-domains, and what algorithmic differences are there between serial and parallel implementations. Scalable input and output are covered briefly in section 5.1.6, in the context of PUL-sm. Algorithmic differences are most visible in the context of solution procedures and are dealt with in section 5.1.5.

Communication between sub-domains and the associated data structures are of primary interest; here we present two models - *halos* and *shadow nodes* - and compare their merits.

5.1.1 Shadow Nodes and Halos

In the shadow node model each processor stores only information associated with those elements it is assigned ownership of. Referring to figure 5.1 this is the central, unshaded region of local mesh 3. The shadow nodes are then those which are duplicated on every processor which owns an element of which they are part. This duplication implies that the shadow nodes will require special treatment.

In the halo model additional storage is used for copies of those elements immediately adjacent to the local sub-domain. These elements are precisely those which the local elements interact (and therefore communicate) with. For some applications this can reduce communication costs relative to the shadow node model, at the expense of an increase in memory requirements (generally modest) and the introduction of some redundant computation.

The situation is similar to that discussed in section 4.2.3.1, regarding halos for regular grids. However, it is important to realize that the example used there was of a finite difference calculation, and that the decomposition was carried out at the node level, whereas here we are considering decomposition at the element level for finite element and volume calculations.

5.1.2 Relative Merits of the Two Models

The immediately evident disadvantage of the halo model is the increase in memory requirements. However, in practice this will not be great compared to the size of a sub-domain, which will typically contain a large number of elements, even before any halo might be added.

If we consider a compactly shaped sub-domain of size r and add thin halo layer around it of thickness δr, then, in two dimensions, the area of the sub-domain will be $O(r^2)$ while the area of the halo will be $O(r\delta r)$. Clearly the area of the halo is small in comparison, so long as $\delta r << r$. If we assume that mesh elements are all of similar size then the area will be approximately proportional to the number of elements and our claim that the added memory requirements of the halo model is modest is justified. If elements are not of a similar size then we could extend our argument by measuring area against a metric of length based on local element size and our claim would still hold true.

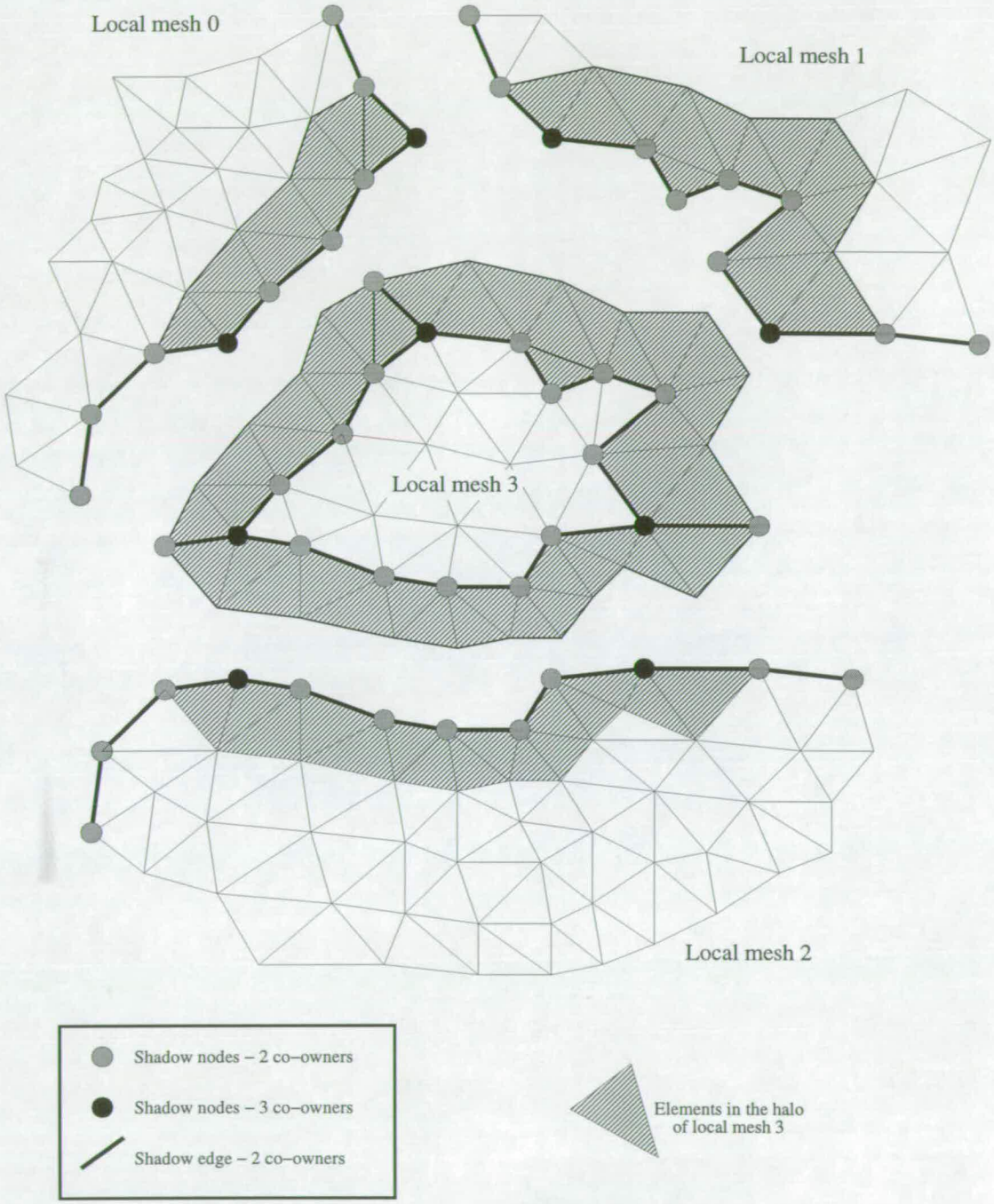


Figure 5.1: Mesh decomposition, showing both halos and shadow nodes.

Another disadvantage is that additional computation may have to be performed in the halo region which is in fact redundant.

These costs must be be justified in terms of increased performance and whether this is realized is application and platform dependent. There are two ways in which the addition of halos may allow increased performance.

The first is by allowing in-place communication; because data located at halo nodes is not calculated by the local processor, but updated by the processor with primary ownership of that node, the update can simply write the new value to that location. If the shadow node model is used then two or more processors share ownership of a shadow node and it will usually be the case that a combine operation (send-with-add, for example) must be used to determine the correct value at that node. This is generally a more expensive operation than an in-place send and may indeed require buffer space comparable to that required by a halo.

The second is by reducing the volume, and possibly number, of messages communicated. We illustrate why this may be so in the context of a typical explicit scheme and then go on to contrast this with the situation in a typical implicit scheme.

5.1.3 Implications for an Explicit Scheme

An explicit scheme is one in which new values of the problem unknowns are given as a function of those at a previous time step or iteration, as is the case for the Forward Euler scheme.

Here one might adopt an iterative numerical scheme whereby the iterate at a given point in the mesh depends upon the values of variables stored at neighbouring points. If, for instance, the governing equations were

$$\frac{\partial u}{\partial t} = F(u, \nabla u) + D(u, \nabla u)$$

for unknowns u, we might have the following numerical scheme:

$$\begin{aligned} \Delta t &= h(u_i \pm) \\ D' &= d(u_i \pm) \\ F' &= f(u_i \pm) \\ u_{i+1} &= u_i + (F' + D')\Delta t \end{aligned}$$

where Δt is a time-step determined adaptively by the function $h(u)$ to be as large as possible and still ensure convergence. D' and F' are discretised approximations to D and F, such that $D' \rightarrow D$ and $F' \rightarrow F$ as the mesh element size $\rightarrow 0$. We use $u_i\pm$ to denote a reference to values of u_i at nearest neighbour points, where u_i is our approximation to the true solution at time step i.

In the shadow node model, the above scheme would be parallelised like this:

$$\begin{aligned}
\Delta t &= f(u_i\pm), \quad \text{update boundary of } \Delta t \\
D' &= d(u_i\pm), \quad \text{update boundary of } D' \\
F' &= f(u_i\pm), \quad \text{update boundary of } F' \\
u_{i+1} &= u_i + (F' + D')\Delta t
\end{aligned}$$

If the mesh domains were to include a single overlapping level of halo elements the parallel scheme would become:

$$\begin{aligned}
\Delta t &= f(u_i\pm) \\
D' &= d(u_i\pm) \\
F' &= f(u_i\pm) \\
u_{i+1} &= u_i + (F' + D')\Delta t, \quad \text{update halo of } u_{i+1}
\end{aligned}$$

As can be seen, we have reduced the number of communication phases from three to one.

The gain, in terms of reduction of number of communication phases, is thus dependent on the number of calculation phases and therefore application dependent.

This sort of scheme arises in CFD and this example is motivated by experience gained while porting the British Aerospace FLITE3D code to the Cray T3D [BMT96]. FLITE3D provides explicit multi-grid Euler solution for modelling high speed airflow around complex geometries. It was parallelised using the shadow node model, but it was felt that the number of calculation phases (which is greater than in our example) may warrant the use of halos. Although the project did not extend to an investigation of this, similar investigations have been carried out [LL96] and it is upon this that many of the comments here are based.

5.1.4 Implications for an Implicit Scheme

An implicit scheme is one in which new values of the problem unknowns require the solution of a set of linear equations based on those at a previous time step or iteration, as is the case for the Backward Euler, Galerkin or Crank Nicolson schemes.

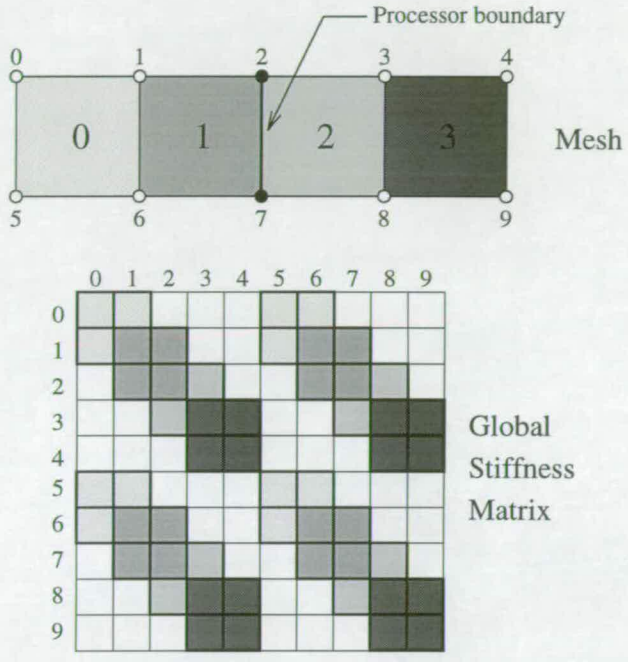


Figure 5.2: Four element mesh and corresponding global stiffness matrix.

This is also a requirement in the case of finite element analyses where there is no time-like coordinate. In a linear analysis of this type, a system of linear equations (independent of the unknowns) must be solved just once. In a non-linear analysis of this type, an iterative procedure is performed where, at each iteration, a set of linear equations (dependent on the unknowns at the previous step) must be solved.

We refer to this matrix on the left hand side of this system of linear equations as the *stiffness matrix*. It is a sparse, often symmetric positive definite, matrix.

In all cases the pattern of the stiffness matrix is determined solely by the connectivity of the mesh. Hence, the decomposition of the mesh has direct bearing on the structure of the parts of the stiffness matrix local to a processor. To illustrate this we first examine assembly of the stiffness matrix without reference to parallelisation, then move on to look at how the shadow and halo models of parallelisation change the local stiffness matrix.

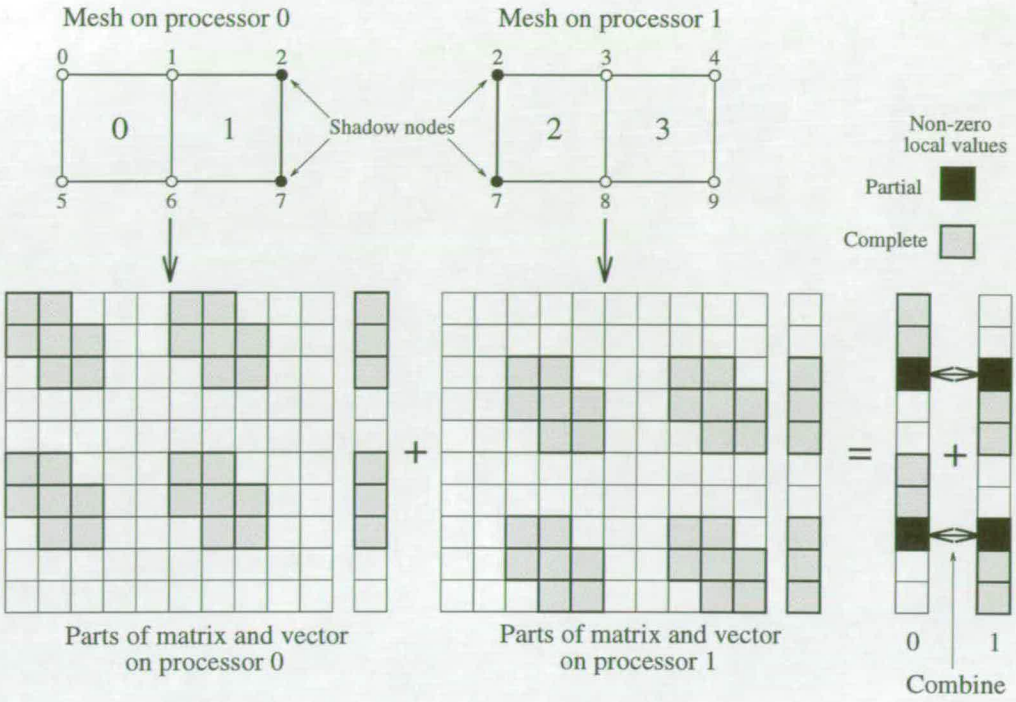


Figure 5.3: Mesh decomposed using the shadow node model and corresponding local stiffness matrices.

Figure 5.2 shows a simple four element mesh and corresponding global stiffness matrix, $\boldsymbol{K}$, with one degree of freedom per node. $\boldsymbol{K}$ is the sum of four elemental stiffness matrices, $\boldsymbol{K}^e$, each of which has (number of nodes per element)^2 entries identified by shading. These overlap and are summed wherever a node is part of more than one element, for example at (1,1), (1,6), (6,1) and (6,6). The interaction between nodes is clear from the non-zero entries on a row. Consider node 1; it is part of both elements 0 and 1, and therefore interacts with all the other nodes in those elements. Looking at row 1, we see entries for nodes 0, 1, 2, 5, 6, and 7, precisely those nodes in elements 0 and 1. The same is true of columns, as the pattern of sparsity of the matrix is symmetric.

If we now consider the shadow node model depicted in figure 5.3 we see that each processor now has stiffness matrix entries for only those elements in its sub-domain. Processor 0 has entries for $\boldsymbol{K}^0$ and $\boldsymbol{K}^1$, while processor 1 has entries for $\boldsymbol{K}^2$ and $\boldsymbol{K}^3$.

The halo model extends this to include entries for $\boldsymbol{K}^e$, where e is on another processor, but adjacent to the local sub-domain. Processor 0 now has entries for $\boldsymbol{K}^0$, $\boldsymbol{K}^1$ and $\boldsymbol{K}^2$, while processor 1 now has entries for $\boldsymbol{K}^1$, $\boldsymbol{K}^2$ and $\boldsymbol{K}^3$.

While it is evident that this difference may have implications for solving these

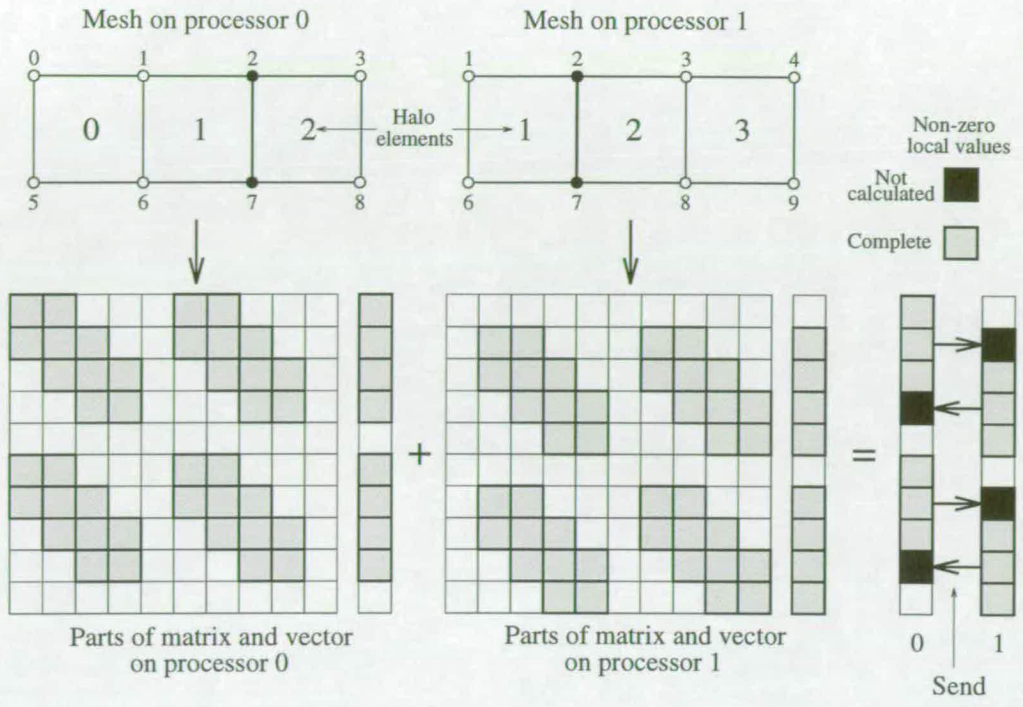


Figure 5.4: Mesh decomposed using the halo model and corresponding local stiffness matrices.

equations in parallel, it will not be clear in what way until we have examined how this procedure is carried out.

5.1.5 Parallel Solvers

Solution of systems of linear equations is a vast field and solution in parallel a large and rapidly developing part of that field. It is quite beyond the scope of this thesis to do this subject justice and we refer the reader to [JM92] and [BBC+94] for a good introduction to the subject.

Many serial unstructured mesh codes use elimination methods for solution, but these have several disadvantages, both for large problems and for parallel implementation.

If we are solving a system of equations with n_{dof} degrees of freedom using an elimination method, then the computation involved is $O(n_{dof}^3)$, for a dense system. For sparse systems, which we have seen arise in unstructured mesh applications, this may be reduced to $O(n_{dof}^2 b)$, where b is the bandwidth of the sparse matrix. Methods which make use of this sparsity, such as frontal or profile solvers, are inherently sequential, as are many methods which seek to reduce bandwidth

and so optimise solution. Additionally, elimination methods alter the pattern of sparsity of the matrix.

Iterative methods require $O(n_{dof}^2)$ operations per iteration and so have an advantage if convergence occurs in few enough iterations. Whether iterative or elimination methods are superior on a serial platform depends very much on the application, however on a parallel platform iterative methods have clear advantages. Firstly, they are easy to parallelise in a scalable manner, secondly they do not alter the structure of the matrix and thirdly we are going to be solving large systems, if the use of a parallel platform is to be justified at all, and it is here that iterative methods are most competitive, even in serial.

A common choice of parallel solver is the *conjugate gradient method* [GL89][JM92]. It is typical of parallel solvers and often the best choice of method, so we will detail it here.

We wish to solve

$$\boldsymbol{K}\boldsymbol{x} = \boldsymbol{l}. \tag{5.1}$$

Where $\boldsymbol{K} \in \mathbb{R}^{n_{dof} \times n_{dof}}$ and $\boldsymbol{x}, \boldsymbol{l} \in \mathbb{R}^{n_{dof}}$.

The solution procedure can be viewed as finding the position in $\mathbb{R}^{n_{dof}}$ which minimises an error function defined over that space. The *method of steepest descent* is the basic method in this class. From an initial vector $\boldsymbol{x}_0$, it produces a sequence of vectors $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_i$ each of which has a lower value of the error function than the last, that is, is a better solution than the last. $\boldsymbol{x}_{i+1}$ is generated from $\boldsymbol{x}_i$ by moving in the direction of maximum gradient of the error function, $\boldsymbol{p}_i$, at that point (hence *steepest descent*).

The conjugate gradient method is similar, but adds the constraint that the $\boldsymbol{p}_i$ be mutually conjugate with respect to $\boldsymbol{K}$, that is

$$\boldsymbol{p}_i^T \boldsymbol{K} \boldsymbol{p}_j = 0 \ \ \forall i \neq j$$

If we start with

$$\boldsymbol{p}_0 = \boldsymbol{r}_0 = \boldsymbol{l} - \boldsymbol{K}\boldsymbol{x}_0$$

The conjugate gradient method, at step $i+1$, is then

$$\begin{aligned} \boldsymbol{u}_i &= \boldsymbol{K}\boldsymbol{p}_i \\ \alpha_i &= \boldsymbol{r}_i^T\boldsymbol{r}_i / \boldsymbol{p}_i^T\boldsymbol{u}_i \\ \boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i \\ \boldsymbol{r}_{i+1} &= \boldsymbol{r}_i - \alpha_i\boldsymbol{u}_i \\ \beta_i &= \boldsymbol{r}_{i+1}^T\boldsymbol{r}_{i+1} / \boldsymbol{r}_i^T\boldsymbol{r}_i \\ \boldsymbol{p}_{i+1} &= \boldsymbol{r}_{i+1} + \beta_i\boldsymbol{p}_i \end{aligned} \tag{5.2}$$

In exact arithmetic the method will find the correct solution in n_{dof} iterations, but requires approximately six times as much computation as Gaussian elimination for a fully populated matrix. In finite precision arithmetic more iterations may be needed and it is only when viewed as an iterative method for solution of large, sparse, symmetric positive definite systems that it becomes favourable. This is, however, often exactly what we require (for non-symmetric systems the similar biconjugate [Fle76] or the generalised minimum residual (GMRES) [SS86] methods are available).

Examining 5.2 we see that the only array operations required to implement the method are:

- vector multiplied by scalar
- vector addition/subtraction
- inner product
- matrix vector product

Now if we assume each processor has all values that are stored on it correct to start with, including any halo data in that model, we can proceed as follows.

The vector multiplied by scalar and vector addition/subtraction operations are trivially parallel, in either the halo or shadow node model. Each processor loops over those array elements (i.e nodal degrees of freedom) local to it and performs the required operation, no communication being required. In the halo model additional calculation must be performed for halo nodes, over and above that in the shadow node model. If this is not done then halo node data will be incorrect

for use in later calculations. The alternative, introducing communication, will never be economic.

The inner product operator requires each processor to perform an inner product on the array elements local to it. All processors then sum their local contributions in a global reduction communication, at the end of which they all have the total.

The only complication with this is to avoid over-counting; if two processors share a node and the corresponding array elements in the inner product are multiplied together on both processors and then summed (first locally, then globally), clearly those contributions will be double the correct value. We must therefore ensure that only one processor in the group which shares a node performs this multiplication and addition. This is done by assigning *primary ownership* of those nodal degrees of freedom to one processor only; all others which share it have *secondary ownership*. Processors then only deal with data they have primary ownership of. The halo and shadow node models perform the same calculations in this case.

How the matrix vector product is carried out is illustrated in figure 5.3, for the shadow node model, and in figure 5.4, for the halo model.

In both cases a local matrix vector product is performed, the effects of neighbours are then resolved through a communication phase.

In the shadow node model, array elements associated with shadow nodes have only partial results after the local matrix vector product. The correct values for those elements are the sums of the local results on all processors which share the corresponding nodes. Note that the primary/secondary distinction is not required here. Communication therefore takes the form of a send-with-combine (i.e. send the value and add it to the local one).

In figure 5.3 we can see this for nodes 2 and 7. After the local matrix vector product processor 0 sends its contributions to processor 1, where they are added to the values there and processor 1 sends its contributions to processor 0, similarly.

In the halo model correct local values are obtained for the corresponding (boundary) nodes immediately after the local matrix vector product. Values at other nodes in the halo will be only partial results, however the full, correct values will be known on other processors. The communication phase can therefore be an in-place send operation, which is generally quicker than a send-with-combine, which

may require buffering as well as actually performing the combine. Of course, unwanted local values that are replaced by the send need not be calculated in the first place.

In figure 5.4 we can see this for nodes 3 and 8. Processor 0 has only partial results for these elements, whereas processor 1 has complete results and so can send them to processor 0. The situation is symmetric for nodes 1 and 6.

In summary, we have seen that iterative procedures for the solution of linear systems are generally favourable in terms of parallel computation, and that this implies that the efficient implementation of a distributed matrix vector product is of crucial importance to the overall efficiency of the parallel solution procedure. For both shadow node and halo models, this distributed matrix vector product proceeds as a local matrix vector product on every processor, followed by a communication phase. For the shadow node model the local matrix vector product involves less calculation compared to the halo model, but the communication phase must be a send-with-combine, which itself involves some further calculation. The total calculation in the shadow node model is, however, still less than that required in the halo model, as a simple study of the matrices in figures 5.3 and 5.4 shows[1]. Whether the halo model has anything to commend it over the shadow node model may then be an issue of the ease with which storage may be arranged for efficient in-place communication with the minimum of memory copies. In short, the relative merits of the two models are much less distinct for an implicit scheme than for an explicit scheme.

5.1.6 PUL-sm

PUL-sm [TB96, Tre95b] is EPCC's library for run-time support of parallel, static, unstructured mesh calculations. The library consists of a set of routines, with interfaces in both C and FORTRAN, which provide much of the functionality we have just discussed, as well as tackling several other issues which need to be addressed in the course of developing such an application or porting an existing serial code to a parallel platform. The library is built on top of the MPI message passing library and so can be compiled on a wide variety of platforms.

[1]For the matrices shown each processor performs 32 multiplications and 26 additions in the local matrix vector product for the halo model, compared to 28 multiplications and 22 additions for the shadow node model. The send-with-combine in the latter model incurs a further two additions.

It is closely associated with PUL-md [BDT96], the mesh decomposition library which acts as a serial preprocessor to PUL-sm. We will study PUL-md closely in chapter 7. For now it will suffice to say that it provides a decomposition of the mesh, is capable of preprocessing mesh structure and data files to allow fast distribution and determine any halo structure (if required).

PUL-sm (current PUL-sm-2-3 release) provides the following features:

- It provides support for static meshes; those whose structure does not change in the course of execution.
- It implements both halo and shadow node models.
- Decomposition is expected to be provided by the application (PUL-md may fill this role).
- It manages reading and distribution of mesh structure and data files.
- Scalable distribution of such files is provided where they have been suitably preprocessed by PUL-md (see section 5.1.6.1).
- It manages communication between processors via *boundary swap* library calls.
- Primary and secondary ownership is handled by the boundary swap operations transparently.
- Only a single mesh can exist in an instance of PUL-sm
- Multiple instances of PUL-sm can coexist, allowing multi-grid calculations.
- Migration of mesh elements for dynamic load balancing is provided (initial PUL-sm-1-0 release only).
- Scalable output is also provided.

5.1.6.1 Scalable Mesh Distribution

Experience with the FLITE3D project showed that i/o could be a serious bottle neck in a parallel unstructured mesh calculation. The naïve approach, in which one processor handles i/o of the mesh structure and data files and then communicates with all the remaining processors, is not found to be scalable [BMT96].

If it was only at the start and finish of execution that i/o occurred then, although this would ultimately be undesirable, we might be able to ignore the

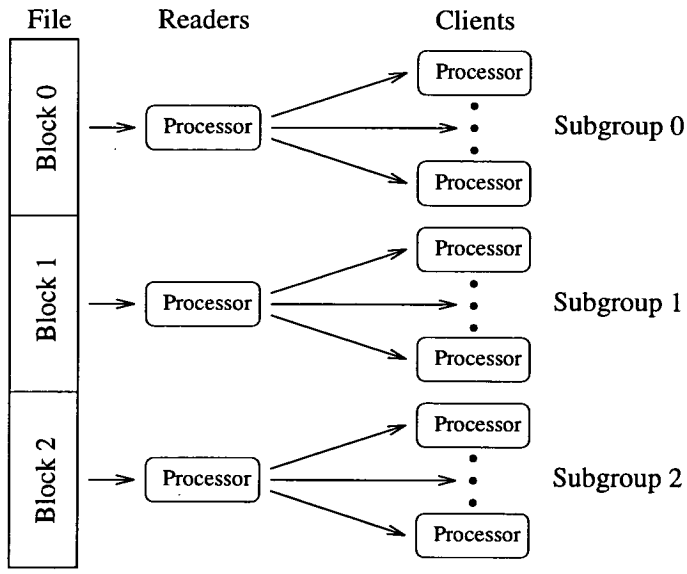


Figure 5.5: Processor blocked scalable input.

issue. However, problems large enough to exhibit this bottle neck will be likely to run for some time, and so will require *check-pointing* (writing out of intermediate results to allow restart in case of error). Initial, check-pointing and final i/o can together come to dominate the run-time of the application if the naïve approach is used.

The solution, implemented as part of PUL-sm, is for a number of i/o processors to read (or write) simultaneously from (or to) a file. Each of these i/o processors then communicates data with a subgroup of other processors for which it is solely responsible.

For efficient file access we need to arrange the file so that all the data for a given subgroup is contiguous in the file. This will ensure that there is no contention for file access and that each i/o processor may proceed sequentially through its portion of the file. We term this file format *processor blocked.* The necessary preprocessing for this is one of the functions of PUL-md.

Given a processor blocked mesh file, input proceeds as shown in figure 5.5. Output is simply the reverse. Typically eight processors are used in each subgroup, although this may be altered at compile time.

5.2 Implementation in a Data Parallel Environment

Unstructured mesh calculations do not lend themselves well to implementation in a data parallel environment. The strength of data parallel programming is in the ease with which it handles arrays and regular patterns of communication, making it ideal for the sort of regular grid calculation used as an example in section 4.2.3.1 of the earlier decomposition chapter. It is much less suitable for unstructured mesh calculations, due to the very irregular nature of their data structures and communication patterns.

This is not to discount the use of the data parallel architectures in this context entirely, as successful implementations have been made; [Mat92] describes a finite element three dimensional stress analysis using the Thinking Machines CM-2, [JMJH93] and [JMJH94] discuss communication strategies for finite element calculations on the CM-5 and [Ego92] compares six computational fluid dynamics applications which use this architecture, several of which use unstructured meshes.

To illustrate implementation in a data parallel environment we present a example code which shows how the programming environment impacts on a simple unstructured mesh application.

5.2.1 An Example Data Parallel Finite Element Code: LEASH

In the early stages of research for this thesis the current author parallelised a linear elastic shell analysis code to run in a data parallel environment on the CM-200 distributed memory SIMD architecture. The code, known as LEASH, was part of the FELASH suite of programs [TR89a, TR89b, Rot85] for axisymmetric shell analyses of various types, with particular application to silo structures.

The LEASH code employed a mixed harmonic analysis and finite element approach, with circumferential variation being described by Fourier series, while normal and meridional variations were accounted for by finite element analysis. This may be thought of as carrying out, once for each harmonic, a two-dimensional finite element analysis using simple line elements. This enables

both non-symmetric loads and branched axial sections to be studied. The latter implies that the element topology, although relatively simple, is nonetheless unstructured as an arbitrary number of elements may meet at a common node. A typical geometry is illustrated in figure 5.6, as is the corresponding element topology.

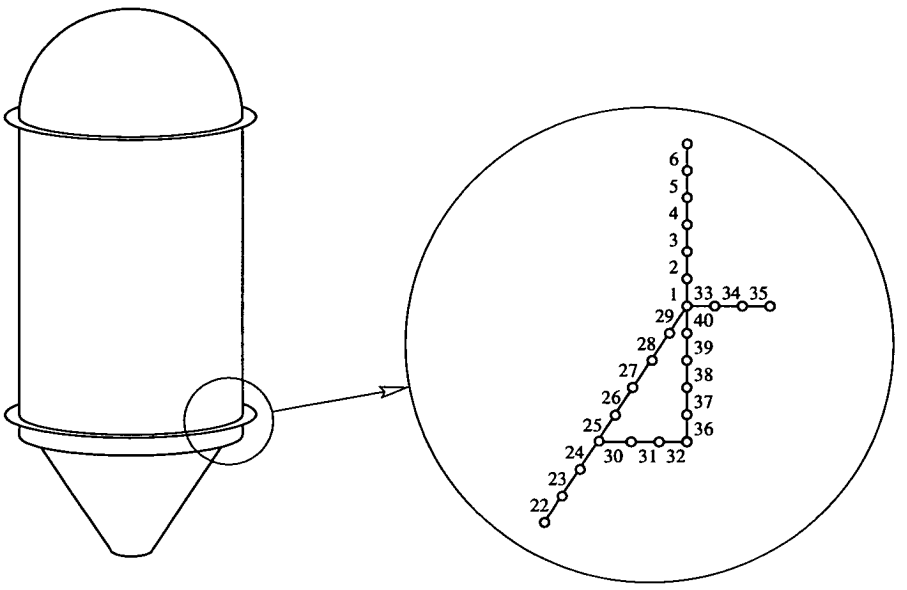


Figure 5.6: Typical axisymmetric shell geometry arising in the study of silo structures, together with detail showing element numbering of the multiple segments making up the ring-beam local geometry.

Parallelisation of the code, so far as individual harmonics were concerned, was trivial; they are independent for a linear analysis and therefore embarrassingly parallel. The situation as regards the finite element analysis was less straightforward, but also of more interest to us in the context of this thesis. Two options regarding data distribution were feasible; either to distribute node or element indexed arrays. To allow the calculation of element stiffness matrices without incurring communication, it was decided to distribute elements; a decision influenced by [MJ90a] and [MJ90b].

Given this data distribution, assignment of elements to CM-200 virtual processors was based on element numbering (a lexicographic decomposition), which the code's preprocessor ensured was contiguous except where branched axial sections met. In other words, the numbering within each 'segment' of a branched structure was contiguous, as we can see from figure 5.6. The communication pattern this imposed was an arbitrary one determined by the problem geometry. If there were no branches in the structure, then nearest neighbour communica-

tion is all that would be required, and simple data-parallel shift communication primitives could be efficiently employed. However, if this was not the case then interaction between elements at the end of each segment that joined another would require communication between an arbitrary set of processors. The irregular communication pattern this represents could be accounted for in this data parallel/SIMD environment by using shift primitives *within* each segment, and treating terminal elements as a special case with, say, an expensive all-to-all communication. This approach is clearly only efficient for the partially structured topology we consider here, and would not be suitable for more general finite element meshes, although random decompositions and all-to-all communication have occasionally been employed [Mat90, MJ92]. The most efficient approach was provided by the *communication compiler* routines which form part of the Connection Machine Scientific Software Library (CMSSL) for CM Fortran. These routines compute, store, load and use message delivery optimisations for basic data motion and combining operations. They are particularly appropriate for the type of repetitive, partially regular communication occasioned by the iterative solver employed in the parallel LEASH code. Given a pattern of communication, a *trace* is compiled, which described a near optimal schedule for best use of the machine's network given the restrictions imposed by the synchronous nature of the SIMD architecture. A variety of methods for compiling the trace were available, with the FastGraph method being most suitable in this instance [Thi93b, Dah90].

While this discussion illustrates some of the difficulties that arise in the implementation of unstructured mesh codes in a data parallel environment, it has been coloured by the restrictions of the fine-grained parallel, SIMD architecture of the CM-200. As we have seen, data-parallel languages such as HPF are available for MIMD architectures and only impose partial synchronisation, making irregular communications considerably easier to deal with. It may still, however, be the case that for efficient performance special treatment is required for this class of problems, as the data parallel compiler may not be sufficiently sophisticated to handle the complex patterns of communication that arise. This special treatment may resemble message passing to such a degree that that the apparent convenience of data parallel programming is eroded.

We close this discussion of the parallel LEASH code with an illustration of the results of a simple analysis. Figure 5.7 shows a cylindrical shell (not a branched structure) subject to a small patch load, with the resulting membrane shear depicted by surface colour.

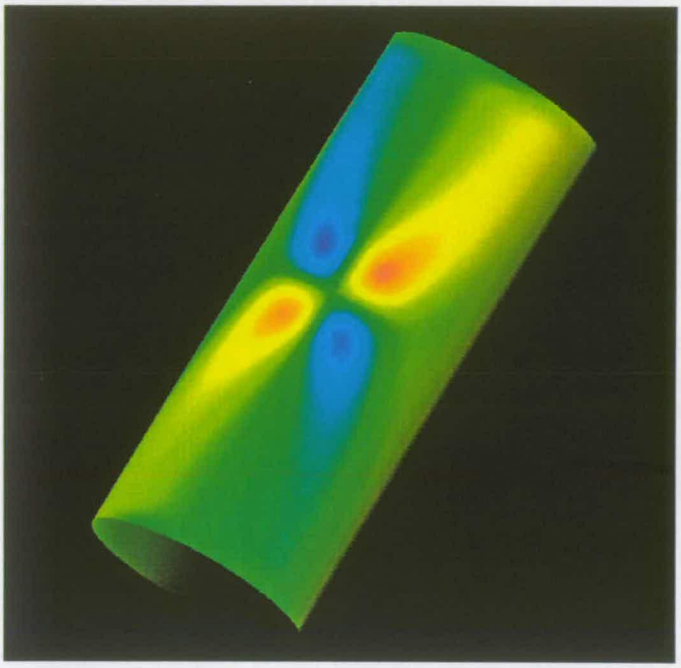


Figure 5.7: Stress analysis resulting from the parallel LEASH code of an axisymmetric shell subject to a small patch load.

5.3 Decomposition

Having seen how parallel unstructured mesh calculations are implemented, we now have some idea as to what we would consider a good decomposition. However, we have also seen that what constitutes 'good' is both application and platform dependent. As we clearly do not wish to develop decomposition methods that are tied to any one application or machine, we need to abstract the concepts we have introduced and thereby find methods which have general applicability.

The abstraction we use to model the application is the *dual graph* (see section 5.3.3). With this we can then model the run-time of the application if we have an understanding of the costs of communication and computation on a given platform.

One approach is to view this as an optimisation problem; we wish to find the

decomposition which minimises run-time over the space of all possible decompositions. We have seen that there is a trade off between communication and computation and may write an objective function which reflects this

$$H = H_{calc} + \mu H_{comm}. \tag{5.3}$$

Here H_{calc} represents load balance and is minimised when all processors have the same load, H_{comm} measures the cost of communication and μ the relative importance of the two.

Not all decomposition methods use this formulation directly (section 6.4 details those that do) but it is a useful starting point as it concisely summarises our requirements.

We now look at the terms in equation 5.3 in relation first to the platform and then the application.

5.3.1 Modelling the Platform

The platform on which the application is running influences H primarily through the second term. The relative efficiency of the processors compared to the network connecting them has a direct bearing on μ; on a machine with slow communication and fast processors we can accept some load imbalance, if we can reduce communication as a result, and so would favour H_{comm} by increasing μ; conversely, on a machine with fast communication and slow processors we would prefer stricter load balance and so would favour H_{calc} by decreasing μ.

The form of H_{comm} is a direct result of the performance of the platform's communication network. Section 4.1 first introduced communication overheads and we now examine them in more detail.

Communication on a parallel computer is often modelled, to a first approximation, by the *latency-bandwidth model.*

In this model the cost of a message, i.e. the time it takes to complete its communication is given by

$$H_{message} = t_{latency} + \frac{s_m}{\beta}. \tag{5.4}$$

In other words there is a cost associated with starting a message which is fixed, the *latency*, $t_{latency}$, and there is a cost associated with the size of the message, s_m, i.e. the number of bytes of data sent in message m. The network *bandwidth*, β, is a measure of how many bytes a second the network can transmit.

This is only a first approximation because it assumes communication between processors does not depend on their location in the network. If we consider a hypercube network, as described in section 3.1.3, it can be seen that this is not necessarily a good assumption. Processors in a hypercube, as we have seen, can be addressed by a binary number with as many digits as there are dimensions in the hypercube, d_{net}. Moreover, the number of network connections that a message needs to traverse is given by the number of bits which are different in the binary addresses of the communicating processors.

For example, consider the case where $d_{net} = 2$. We have four processors 0,1,2 and 3, with 3 at the opposite corner from 0. A message is sent from processor 0 (00); if it is sent to 1 (01) or 2 (10), it only travels along one side of the square and there is only one bit different between 00 and 01 or 10; if it is sent to 3 (11) then it travels along two sides of the square and there are two bits different between 00 and 11.

The number of *hops* a message makes, that is the number of network connection it traverses, was a major consideration for earlier parallel computers, where the time taken for a single message was proportional to the hops. This is less so with the advent of cut-through routing, where special routing hardware makes the time almost independent of the number of hops.

The notion of hops is still useful however, because the previous discussion neglects contention for network resources. In the sort of loosely synchronous applications we are considering communication takes place in bursts and contention is an issue. The hops metric is useful here because any connection a message uses cannot be used by another message at the same time; the further it has to go, the greater congestion it causes.

Typically we are dealing with relatively large messages; a processor will send all the relevant boundary data for its sub-domain to the processor holding the sub-domain on the other side of the boundary in one message, and so we can neglect latency costs.

If we do so, one model of the cost of the whole communication phase is then

$$H_{comm} = \sum_{m} s_m h_{ij} \tag{5.5}$$

where h_{ij} is the number of hops that a message m, communicating between processors i and j, makes. An empirical study that justifies equation 5.5 can be found in [Ham92].

5.3.2 Modelling the Application

The application has a direct bearing on equation 5.3 through μ, and an indirect one through H_{comm}.

μ, as well as being a function of the platform, is also a function of the application; if the application requires little communication then we would favour H_{calc}; if it requires much then we would favour H_{comm}. μ would be altered accordingly, as before.

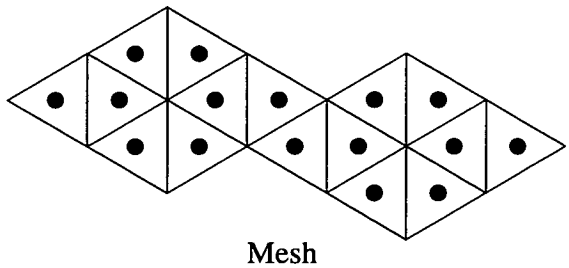
Of greater interest is the way in which the application determines the communication pattern. As we are dealing with unstructured mesh applications, where each mesh element only interacts with its neighbours, the communication pattern is a direct result of the mesh structure.

5.3.3 Dual Graphs

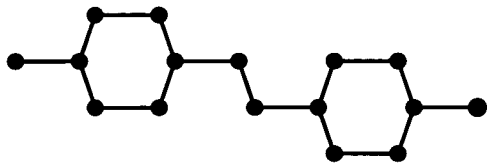
We have seen that the basic entity which we are assigning to processors is the mesh element, and that interactions between mesh elements occurs only between neighbours. We can therefore use a graph to represent this, namely the *dual graph* of the mesh.

Each mesh element, $\varepsilon_i \in \mathcal{E}$, corresponds to a dual graph *vertex* and each interaction, and hence communication, to an *edge* in the graph. The question then arises as to what constitutes a 'neighbour'. Mesh elements can be considered to be neighbours if they are connected by at least one node, at least one edge or (in three dimensions) at least one face. We refer to the corresponding types of dual graphs as *node based*, *edge based* or *face based*, respectively. This is illustrated for a small, two dimensional, triangulated mesh in figure 5.8.

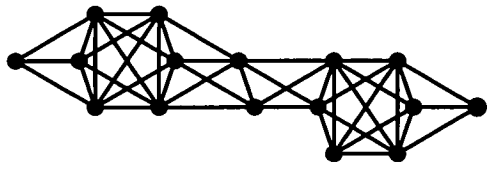
Which type of dual graph is appropriate to a given application is determined by



Mesh



Edge-based Dual Graph



Node-based Dual Graph

Figure 5.8: Mesh and corresponding dual graphs.

the way in which elements interact, and this will be a product of the formulation of the problem that the application solves.

It will be clear from the discussions in section 5.1.4 that for a finite element formulation, where the problem unknowns are associated with the mesh nodes, any two elements which share a node must interact. Thus communication must occur between them, and so we would choose a node based dual graph, where there will be an edge connecting the two corresponding vertices.

For a finite volume calculation, where the problem unknowns are fluxes through the mesh element faces (in three dimensions) or edges (in two dimensions), we would choose a face or edge based dual graph, respectively.

We stated at the beginning of section 5.1 that we could be dealing with either a balanced or unbalanced data decomposition. It may be the case that all mesh elements are not treated equally; examples of this include the imposition of boundary conditions or a differing physical model being used in some region (in CFD, say, we may want viscous flow in a boundary layer around an aircraft wing, while using inviscid equations outside this layer). If this results in a difference in the work associated with different elements we need to take this into account.

We can incorporate this into the dual graph by assigning a *vertex weight*, proportional to the work load of the corresponding mesh element, to each vertex.

Similarly, we can use an *edge weight* to incorporate variations in communication between elements into the dual graph.

If we consider a node based graph, then two vertices which represent a pair of mesh elements sharing a single node would have a lower weight associated with the edge joining them than would two vertices representing elements which share a face, for example. Thus the edge weight measures volume of communication between mesh elements.

Given a dual graph and a decomposition of that graph, we can define a *cut edge* in the graph as one where the vertices at either end of the edge lie on different processors. Communication therefore only occurs across cut edges.

The task of decomposing the mesh is therefore related to the task of *partitioning* the graph, that is assigning graph vertices to as many disjoint sub-sets as there are processors.

Let us now put this more formally.

5.3.4 The Partitioning Problem

We define the dual graph to consist of an undirected, weighted graph $G = (V, E)$, where V is the set of vertices and E the set of edges. Let $n_v = |V|$ be the number of vertices and $n_e = |E|$ be the number of edges.

We associate a vertex weight, $w_v(v_i) > 0$, with each $v_i \in V$. Similarly we associate a edge weight, $w_e(e_{ij}) > 0$, with each $e_{ij} \in E$, where $e_{ij} = \{v_i, v_j\}$ and $i \neq j$.

For a set of k processors, P, we define a partition, M_{part}, of G, as a mapping $M_{part} : V \rightarrow P$. The sub-domains that M_{part} defines are then the disjoint sub-sets $S_p = \{v_i \in V : M_{part}(v_i) = p\}$ where $p \in P$.

The set of cut edges is then $E_{cut} = \{e_{ij} \in E : v_i \in S_p \Rightarrow v_j \notin S_p\}$.

We define the total vertex weight $|A|_v$ of a set $A \subseteq V$ as

$$|A|_v = \sum_{v_i \in A} w_v(v_i)$$

and the total edge weight $|B|_e$ of a set $B \subseteq E$ as

$$|B|_e = \sum_{e_{ij} \in B} w_e(e_{ij}).$$

We may now formulate the graph **partitioning problem**:

Find a partition, M_{part}, of a graph, G, such that

$$(a) \qquad |S_p|_v \simeq \frac{|V|_v}{k} \quad \forall\, p \in P \tag{5.6}$$

and

$$(b) \qquad |E_{cut}|_e \text{ is minimised.}$$

Requirement (a) says that each sub-domain should have equal work, in so far as this is possible. Evidently if n_v is odd and all vertices have equal weight then exact load balance can not be achieved if k is even. Similarly, if vertex weights are not uniform it may not be possible to attain exact load balance. The '$\simeq$' in (a) reflects a difference of no more than the weight of the heaviest vertex.

Requirement (b) seeks to reduce the total volume of communication, but takes no notice of network distance or contention.

In other words, the partitioning problem is: enforce *strict load balance* and minimise total volume of communication subject to this constraint.

5.3.5 Partitioning and Mapping

If we have a solution to the partitioning problem we are still left with the problem of *mapping* the sub-domains to processors, as this mapping does not feature in the formulation of the problem. Indeed, given a solution to 5.6, it is possible to generate another which is just as good by swapping any pair of sub-domains between processors, as this does not affect the value of $|E_{cut}|_e$.

Hence, it is possible to treat this issue separately, keeping the assignment of vertices to sub-domains fixed, but changing assignment of sub-domains to processors. However, it is better to deal with both the partitioning and mapping problems together, as a larger space of possibilities can be explored. Of course,

this complicates the problem and many decomposition algorithms just attack the partitioning problem and ignore mapping entirely.

If we wish to deal with both partitioning and mapping together we can formulate the **generalised partitioning problem**:

Find a partition, M_{part}, of a graph, G, given an objective function $H(G, M_{part})$ which provides a model of the target platform, such that

$$H(G, M_{part}) \text{ is minimised.} \tag{5.7}$$

The generalised partitioning problem makes no assumptions as to the nature of $H(G, M_{part})$ and its solution is the province of optimisation algorithms, which we will turn to in section 6.4. The flexibility this gives is offset by the need to define $H(G, M_{part})$, which may involve the specification of many parameters which are hard to determine.

It is also possible to restrict the problem somewhat, while still maintaining some notion of network locality. If we enforce strict load balance then we can ignore H_{calc} in equation 5.3, μ in that equation becomes irrelevant and we may write $H(G, M_{part}) = H_{comm}(G, M_{part})$.

We then arrive at the **constrained partitioning problem**:

Find a partition, M_{part}, of a graph, G, given an objective function $H_{comm}(G, M_{part})$ which provides a model of communication on the target platform, such that

$$(a) \qquad |S_p|_v \simeq \frac{|V|_v}{k} \quad \forall\, p \in P \tag{5.8}$$

and

$$(b) \qquad H_{comm}(G, M_{part}) \text{ is minimised.}$$

In other words, the constrained partitioning problem is: enforce strict load balance and minimise total cost of communication subject to this constraint.

5.4 Problem Complexity

Having posed the problem we wish to solve, we now comment on the difficulty of finding an exact solution.

Of the three formulations of the problem, 5.6, 5.7 and 5.8, it is clear that 5.6, the partitioning problem, is the simplest. If we can not solve this in a reasonable time, then we will certainly not be able to solve the generalised or constrained problems in a reasonable time either.

Unfortunately, even the partitioning problem, 5.6, in its simplest case, where $k = 2$ and $v_i, e_{ij} = 1$, $\forall i, j$, has been shown to be *NP*-complete [GJS76]. This means that no *known* algorithm to provide an optimal solution for *large* problems in a reasonable time exists and it is highly unlikely that one will ever be found; exact solution of the problem is intractable.

The notion of tractability is defined in terms of how the time complexity of an algorithm depends on the size of the problem it solves. We say that an algorithm is *polynomial* if it runs in $O(n^a)$ time, where $a \geqslant 0$ is some constant and n is the size of the problem. Similarly, we say an algorithm is *exponential* if it runs in $O(b^n)$ time, where $b \geqslant 2$ is some constant.

The class P contains those problems for which there exists a polynomial time algorithm to solve the problem. The class *NP* contains those problems for which there exists a polynomial time algorithm to check the correctness of a solution. This says nothing about the time complexity of the solution itself. Additionally, a problem is *NP-complete* if any other problem in *NP* may be reduced to it in polynomial time.

Not only are there no known polynomial time algorithms to solve *NP*-complete problems (in fact all known algorithms are exponential) but if we were to find a polynomial time algorithm to solve any problem in that class, then we would be be able to solve all *NP*-complete problems in polynomial time (via the polynomial time reduction) and P would equal *NP*.

As the class of problems in *NP* but not in P is large, arising in almost all areas of computation, it seems unlikely that $P = NP$. *NP*-complete problems are therefore generally regarded as intractable, while those in P are considered tractable.

If we consider complete enumeration of all possible solutions to 5.7 then the exponential growth of the search space is clear, in that there are k^{n_v} possible

decompositions. Even parallel algorithms help only a little in mitigating this exponential growth in run time [Akl89]. As we are quite likely to be dealing with k in the hundreds and n_v in the millions, complete enumeration is out of the question.

As we can not realistically hope to find an exact solution to any of our problems, we must resort to *heuristic* approaches, where we seek only an approximate solution.

5.5 Summary

We have examined the problems that arise in implementing an unstructured mesh calculation in parallel and looked at the factors which affect the performance of the application. From this we have built up an abstract model of performance which has allowed us to formally define the problem we wish to solve, namely the graph partitioning problem (and its variants).

It has been seen that we must take a heuristic approach to the problem. We therefore now turn to examine the heuristics which have been developed for this purpose, which will be the subject of the following chapter.

Chapter 6

Decomposition Algorithms

A large body of work exists on the topic of mesh decomposition and graph partitioning. There has been research carried out in this area since before 1970 [Ker69] and there is still active research in the area today [WCE97, PD97, KK97] etc.

The survey of methods which is the subject of this chapter aims to be as exhaustive as possible; it covers all important methods together with many others which are deemed to be illustrative.

We first examine the general characteristics of decomposition algorithms. We then survey the algorithms that have been developed and comment on their merits. This commentary is qualitative in nature, presenting visual examples where possible. A quantitative examination will be presented in chapter 8.

Where an algorithm is one that is included in PUL-md, issues which are relevant to implementation are covered in greater depth in the following chapter, where we examine PUL-md in detail.

6.1 Characteristics of Algorithms

We classify decomposition algorithms as follows:

- Global methods
  - Direct k-way methods.
  - Recursive methods.

- Local refinement techniques.

We view global methods in contrast to local methods. The latter work with an initial decomposition and attempt to improve upon it locally, while maintaining its overall structure; the former decompose the mesh as a whole.

Similarly, of the global methods, we view k-way methods in contrast to recursive methods. The latter decompose the mesh into $l \ll k$ parts and then are recursively applied to each part, until k sub-domains are generated; the former do not have this recursive nature.

Broadly, the structure of this chapter reflects the approaches which characterise decomposition algorithms:

- Simple, direct k-way algorithms.
- Optimisation algorithms, where we seek to minimise $H(G, M_{part})$ and are concerned with the generalised partitioning problem 5.7.
- Geometry based recursive algorithms, which use additional geometric information associated with the graph vertices and are concerned with the partitioning problem 5.6.
- Graph based recursive algorithms which use only the graph structure and are concerned with the partitioning problem 5.6 or the constrained problem 5.8.
- Local refinement algorithms, as discussed above.
- Multi-level and hybrid variants, which use combinations of other methods and multi-level acceleration to reduce the run-time of the decomposition algorithm and/or increase partition quality.

6.2 The Example Mesh

In the course of this chapter we will use a small, triangulated, two dimensional mesh to illustrate various points (figures 6.1, 6.2, etc.) This mesh originates from the HEAT2D heat transfer code which is detailed in chapter 9 and is used in the evaluation of algorithms in chapter 8. We refer to this data set as the *Widget* data-set; it has 1746 elements; its dual graphs therefore have the same number

of vertices, while the number of edges in the graph is 10072 for a node-based graph and 2527 for an edge-based graph. These graphs are unweighted.

6.3 Simple, Direct *k*-way Algorithms

In this section we look at some naïve approaches (random, cyclic and lexicographic) primarily to see why we need better solutions, and then look at bandwidth reduction (which is an improved lexicographic algorithm) and the greedy algorithm which are simple, but not unreasonable, algorithms for the solution of the partitioning problem 5.6.

6.3.1 Random

The most naïve approach we could take would be to assign each vertex to a processor at random. Strict load balance would not be achieved, but, on average, we would expect it to be about right. It is simple to produce a random partition with strict load balance. Consider an unweighted graph; assign a random number to each vertex, sort the vertices on this key and assign the first $1/k$ vertices to the first processor, the second $1/k$ to the second processor, etc.

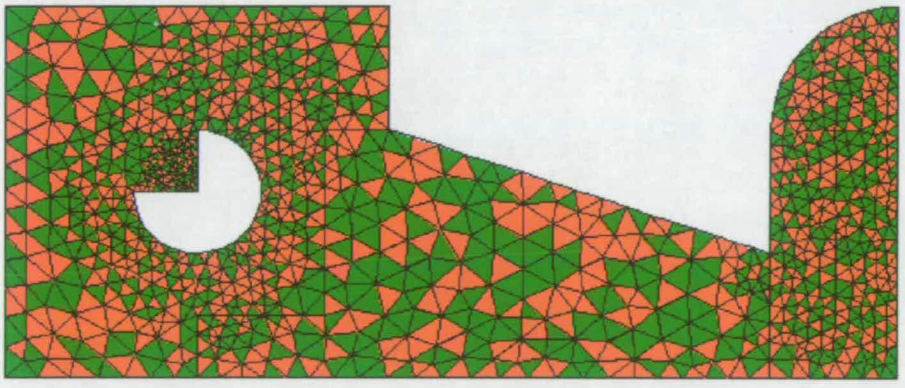


Figure 6.1: A random, strictly load balanced decomposition of Widget over 2 processors.

As no account is taken of connectivity, vertices adjacent in the graph will not, in general, tend to be in the same sub-domain and communication costs will be enormous. This is illustrated in figure 6.1; clearly this is the opposite of what we are after.

6.3.2 Cyclic

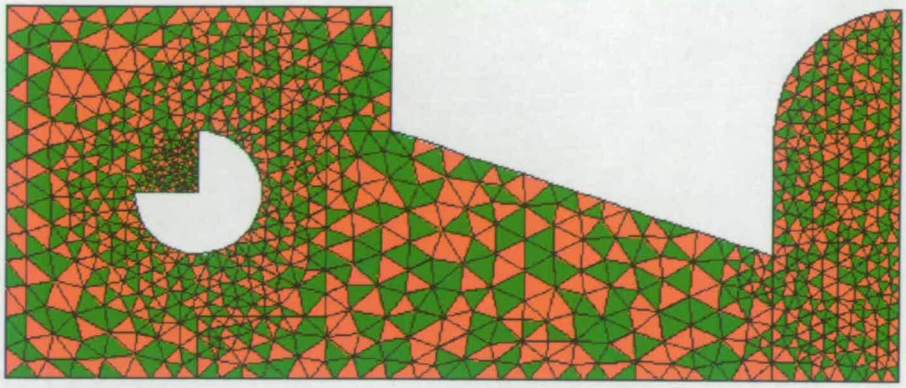


Figure 6.2: A cyclic decomposition of Widget over 2 processors.

A cyclic or scattered partition of a graph is obtained by placing vertex v_0 on processor 0, vertex v_1 on processor 1 and so on, until we reach v_k where we start at processor 0, once more. In other words, $M_{part}(v_i) = i \bmod k$.

Strict load balance will be achieved (for an unweighted graph), but, examining figure 6.2, we see that this is no better than the random approach so far as communication is concerned. In fact it is slightly worse; the random partition produces 5054 cut edges in the node-based dual graph of Widget and the cyclic 5303 (corresponding figures for an edge-based graph are 1269 and 1473, respectively).

6.3.3 Lexicographic

A lexicographic partition simply assigns vertices to processors in order of vertex numbering. The first n_v/k vertices go on processor 0, the second on processor 1, and so on. In other words, $M_{part}(v_i) = int(i/k)$ for an unweighted graph.

This often produces a not unreasonable decomposition, as is shown in figure 6.3. The reason for this is that the numbering of elements in the mesh (and hence vertices in the graph) is far from random; it is an artifact of the mesh generation process. Often, during mesh generation, each new element is created next to the previous one (for instance in the advancing front method [PVMZ87]) and the element numbering reflects this.

This also explains why a cyclic decomposition is generally worse, in terms of communication costs, than a random one. In a cyclic decomposition two consec-

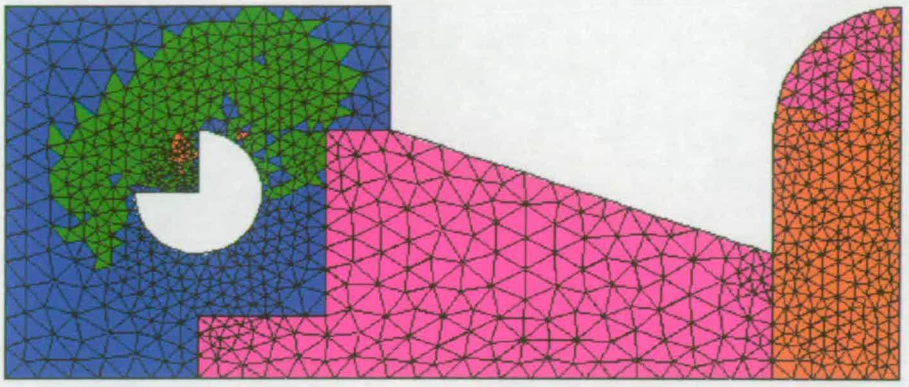


Figure 6.3: Lexicographic decomposition of Widget over 4 processors.

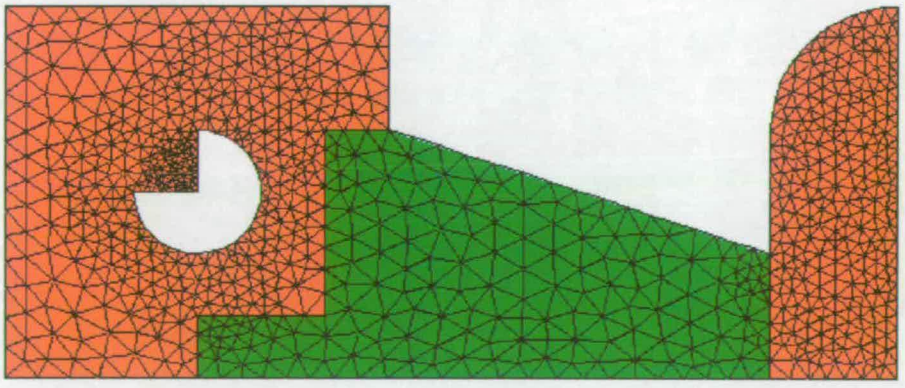


Figure 6.4: Material types for Widget.

utively numbered vertices are guaranteed to be on different processors; with a random decomposition there is at least a chance that they will share the same processor.

The influence of mesh generation on the decomposition is clear if we compare figure 6.3 with figure 6.4, which shows how the mesh is composed of two materials. The two different materials have different thermal properties and are meshed separately, one after the other, with visible effects on the element numbering and hence decomposition.

6.3.4 Bandwidth Reduction

While lexicographic decomposition is a vast improvement over the other methods we have looked at it is still far from optimal. We clearly can not accept being at the mercy of an arbitrary numbering scheme, which may or may not give reasonable results. It is, however, quite possible to renumber the mesh elements in such a way that a subsequent lexicographic decomposition produces better results.

This renumbering is referred to as *bandwidth reduction.* This procedure was developed to facilitate the solution of sparse systems of linear equations, regardless of whether the equations are obtained from an unstructured mesh problem or not.

We have seen in section 5.1.4 how the structure of the mesh is reflected in the stiffness matrix. If we consider which mesh nodes interact in figure 5.2 we see that they are precisely those with non-zero off-diagonal entries in the stiffness matrix. We can form a graph which represents this, which we will term the *dependency graph* for the purposes of this discussion, to avoid confusion with the dual graph, which is quite distinct. Vertices in the dependency graph represent mesh nodes (and hence variables) and edges connect pairs of mesh nodes with corresponding non-zero off-diagonal matrix entries. Even if we do not have a mesh corresponding to the matrix we can still form such a graph from the matrix.

We now define the half-bandwidth and profile of a symmetric matrix, $\boldsymbol{A}_{ij}$. Let b_i be the smallest number such that $\boldsymbol{A}_{ij} = 0 \quad \forall j > i + b_i$, the half-bandwidth is then $\max_i\ b_i$ and the profile $\sum_i b_i$.

If we wish to solve a sparse system using an elimination method which takes advantage of the sparsity (e.g. the profile method [JM92]. See also [DER86] for

direct methods in general or [Duf96] on frontal methods) we would like to reduce the bandwidth, as this will result in decrease in the amount of computation needed. Also, sparse matrices are usually stored in a packed form, where the storage required is determined by the bandwidth or profile of the matrix; reducing these reduces the memory required.

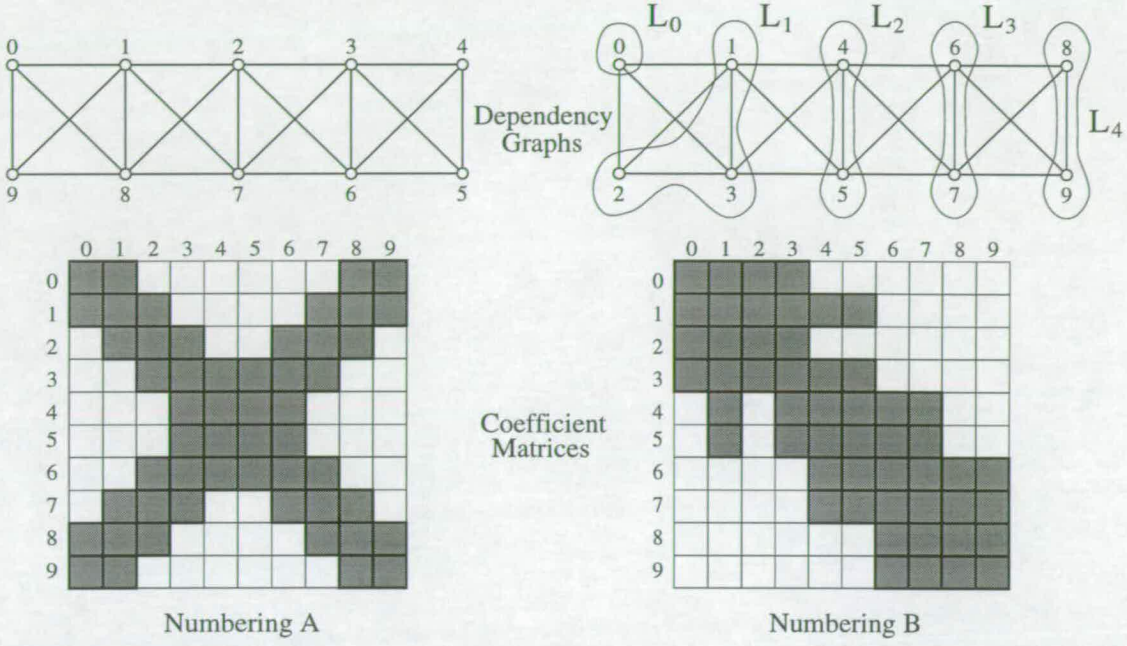


Figure 6.5: Two node numberings and corresponding matrices.

The required reduction may be obtained by renumbering the nodes, as is shown in 6.5. Here we see two different numberings of the dependency graph of the mesh we have seen before, in section 5.1.4. Numbering A has the maximum possible half-bandwidth of 9, while the numbering in B results in non-zero entries being much closer to the diagonal, with a half-bandwidth of 4 (figure 5.2 has a half-bandwidth of 6).

One algorithm for bandwidth reduction that has also been applied to mesh decomposition is the *Cuthill-McKee* algorithm [CM69], which is shown in the pseudocode of figure 6.6.

The algorithm proceeds by finding *level sets*, L_i, in successive layers around the seed point. The first level set is the seed itself, the second all its neighbours, the third all the neighbours of vertices in the second set that are not in any other set, and so on.

The sorting by vertex degree, on line 11 of the pseudocode, is often not used

Pseudo: Cuthill-McKee

1. Choose a seed vertex, v, preferable at an extremity of the graph and of low degree.
2. i=0
3. $L_i = \{v\}$
4. Number v as vertex 0.
5. **While** There are un-numbered vertices.
6. $i \mathrel{+}= 1$
7. $L_i = \emptyset$
8. **For** $\forall v \in L_{i-1}$
9. $L_i \uplus \{$un-numbered neighbours of $v\}$
10. **EndFor**
11. Sort L_i by vertex degree, keeping original order of addition where possible.
12. Number vertices in L_i in order.
13. **EndWhile**

EndPseudo

Figure 6.6: The Cuthill-McKee algorithm.

for partitioning, as we are more interested in the layer structure than numbering within layers.

The layer structure generated by the Cuthill-McKee algorithm starting from node 0 in the dependency graph which we have previously examined is also shown in the same figure (6.5).

When used for mesh decomposition the Cuthill-McKee algorithm is applied to the dual graph. The layer structure which results is shown for the Widget data set in figure 6.8, where a node based dual graph has been employed.

It is a feature of the Cuthill-McKee algorithm that the numbering for two adjacent vertices in the same layer, L_i, can not differ by more than $|L_i|$. Similarly, for two adjacent vertices in different layers, L_i and L_j, the difference may not exceed $|L_i| + |L_j|$. It is therefore advantageous to ensure that the maximum layer size is kept as small as possible. This may be done by finding two vertices which are maximally distant from each other, where the metric of graph distance is the number of layers separating the two.

If the last vertex numbered by the Cuthill-McKee algorithm is used as the seed for another application of the algorithm, and this procedure is iterated, it is typically found that within a very few iterations such a maximally separated pair is discovered. This is shown graphically in figure 6.7, with the corresponding layer structures in figures 6.8 to 6.10. The starting vertex is 0, the maximally distant vertex from this is vertex 930, and from 930 we find vertex 805 (all numberings given in original numbering). If we apply the procedure again then we return to

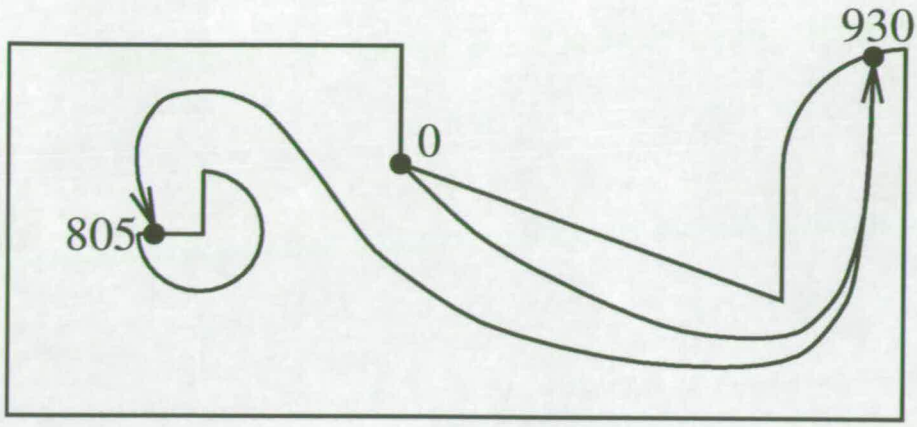


Figure 6.7: Successive seed vertices found by repeated application of the Cuthill-McKee algorithm.

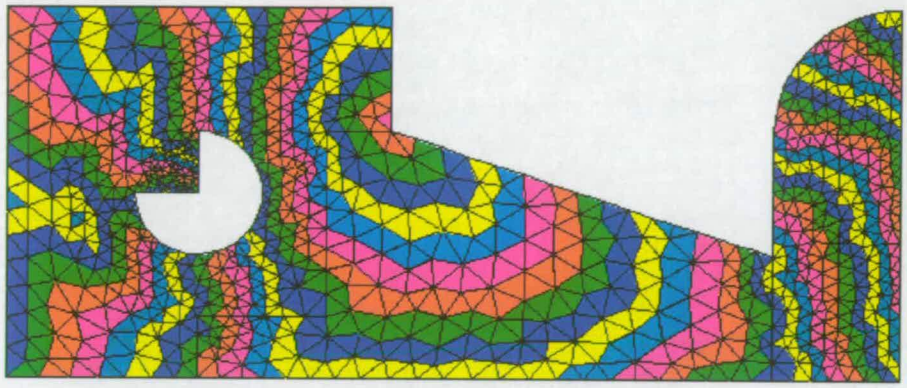


Figure 6.8: Layer structure for node based graph of Widget starting from vertex 0.

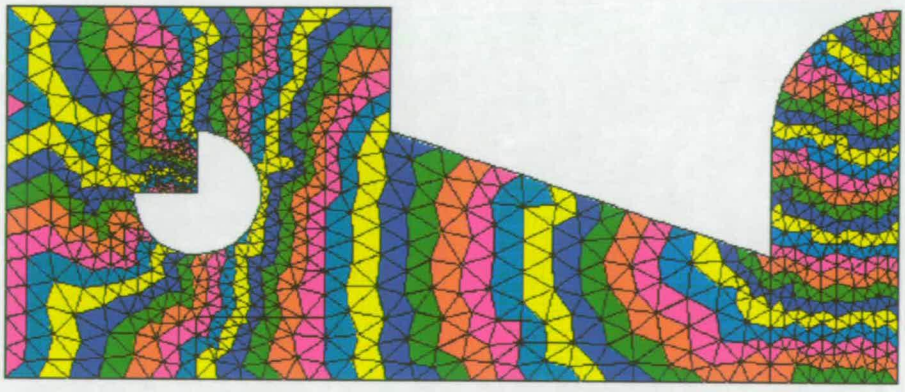


Figure 6.9: Layer structure for node based graph of Widget starting from vertex 930.

vertex 930.

A decomposition of Widget over eight processors is shown in figure 6.11. It is typical of the type of decomposition which results from the use of Cuthill-McKee, in that each sub-domain has few neighbours (which is desirable on machines with high $t_{latency}$), but $|E_{cut}|_e$ is quite large due to the elongated shape of many of the sub-domains. Clearly as we increase k the resulting sub-domains become more and more elongated and this effect is accentuated. It is therefore common to apply the method recursively; we shall return to this in section 6.7.1.

6.3.5 The Greedy Algorithm

The Farhat's greedy algorithm [Far88] makes similar use of the connectivity of the mesh to expand out in layers from a seed vertex, but it is distinct, in that it uses several seeds, one for each sub-domain.

The algorithm proceeds by expanding out from its initial seed, as before, but stops once sufficient vertices have been claimed to fill one sub-domain. It then chooses a favourable vertex from the boundary of the previous sub-domain as the seed of the next. This process is repeated until the required decomposition is obtained.

The version of the algorithm used by Farhat in [Far88] is not phrased in terms of the dual graph, although the algorithm may be cast in such a form (see section 7.4.5 for the way PUL-md does this). Rather, it works directly with the mesh, as follows.

A weight, ω_i, of a node $\eta_i \in \mathcal{N}$ is defined as $\omega_i = |\{\varepsilon_j \in \mathcal{E} : \eta_i \in \varepsilon_j\}|$. In other words, the weight of a node is the number of elements connected to that node. The *interior boundary* of a sub-domain, S_p, is then defined as $\Gamma_p = \{\eta_i \in S_p : \eta_i \in \varepsilon_j \notin S_p\}$. We number the sub-domains from 1 to k, and consider Γ_0 to be the external boundary of $\mathcal{M}$. Finally, we *mask* an element by adding it to the set $\mathcal{E}_{masked}$.

With this notation Farhat's greedy algorithm is then given by the pseudocode in figure 6.12.

As the algorithm progresses, a sub-domain, S_i, expands about its initial seed node, η_i, until it has the required number of elements, n_e/k, as given by line 11. Masking the elements during this process ensures that elements are not

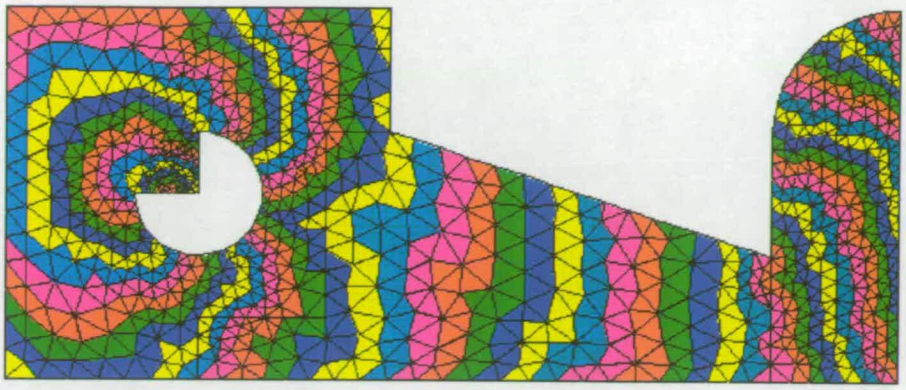


Figure 6.10: Layer structure for node based graph of Widget starting from vertex 805.

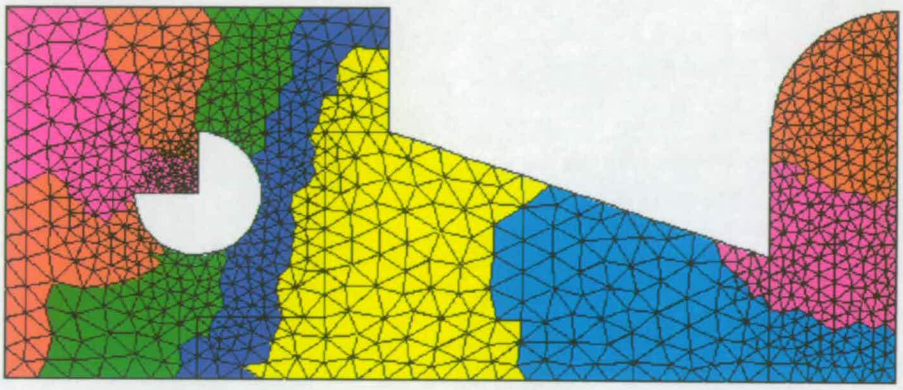


Figure 6.11: Lexicographic decomposition of Widget over eight processors after Cuthill-McKee renumbering.

Pseudo: Greedy

1. $\mathcal{E}_{masked} = \emptyset$
2. **For** $i = 1$ to k
3. Choose $\eta_i \in \Gamma_{i-1}$ with minimal weight ω_i.
4. $S_i = \{\varepsilon \notin \mathcal{E}_{masked} : \eta_i \in \varepsilon\}$
5. **For** $\varepsilon_j \in S_i$ **do recursively**
6. $\mathcal{E}_{masked} \uplus \varepsilon_j$
7. **For** $\eta_k \in \varepsilon_j$
8. ω_k -= 1
9. **EndFor**
10. $S_i \uplus \{\varepsilon \notin \mathcal{E}_{masked} : \exists \eta$ where $\eta \in \varepsilon_j$ and $\eta \in \varepsilon\}$
11. **Break** when $|S_i| = n_\varepsilon / k$
12. **EndFor**

EndPseudo

Figure 6.12: Farhat's greedy algorithm.

reconsidered, as we only select unmasked elements, $\varepsilon \notin \mathcal{E}_{masked}$ in lines 4 and 10. The decrement of the weight in line 8, ensures that the weight of a node not on the interior boundary of a sub-domain is zero, allowing Γ_{i-1} to be identified as $\{\eta_j \in S_{i-1} : \omega_j > 0\}$.

The seed node for sub-domain S_i is thus the node on the interior boundary of the previous sub-domain, S_{i-1}, which is a member of the minimum number of unmasked elements.

The way in which the loop 5 is carried out results in the algorithm appearing to take successive bites out of the mesh; hence the *greedy* algorithm. This can be clearly seen in figures 6.13 to 6.15, which show the first three sub-domains being formed.

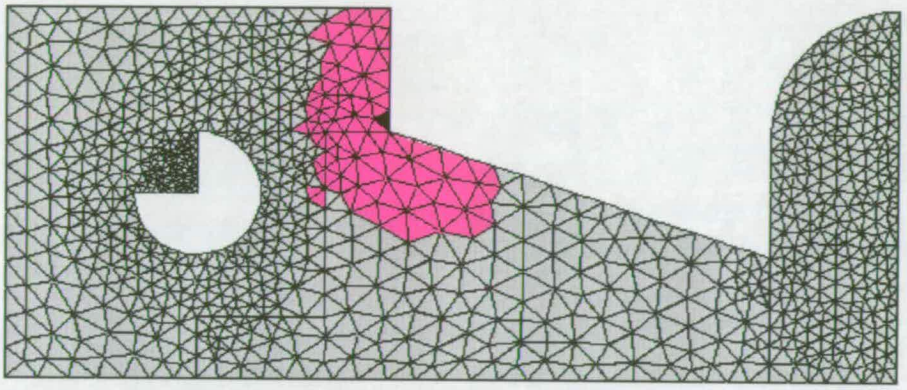


Figure 6.13: First sub-domain generated from Widget by the greedy algorithm.

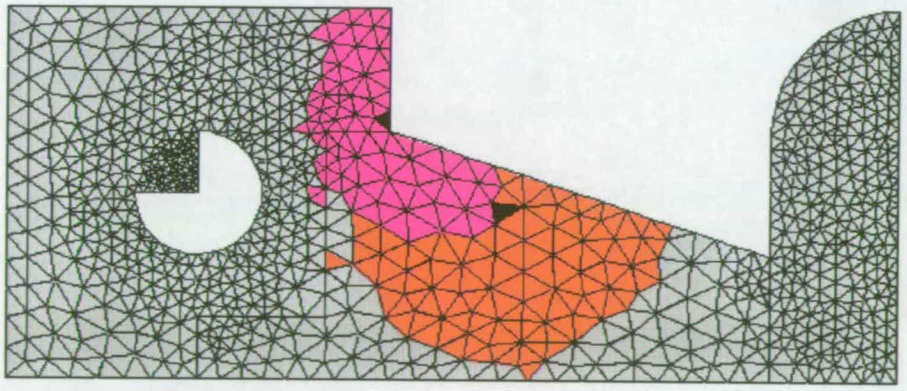


Figure 6.14: First two sub-domains generated from Widget by the greedy algorithm.

These figures are generate from the PUL-md implementation of the greedy algorithm which works with the dual of the mesh, as has already been mentioned.

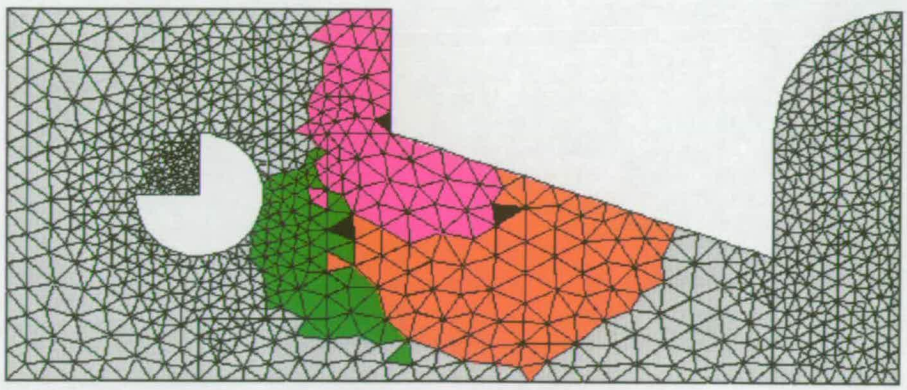


Figure 6.15: First three sub-domains generated from Widget by the greedy algorithm.

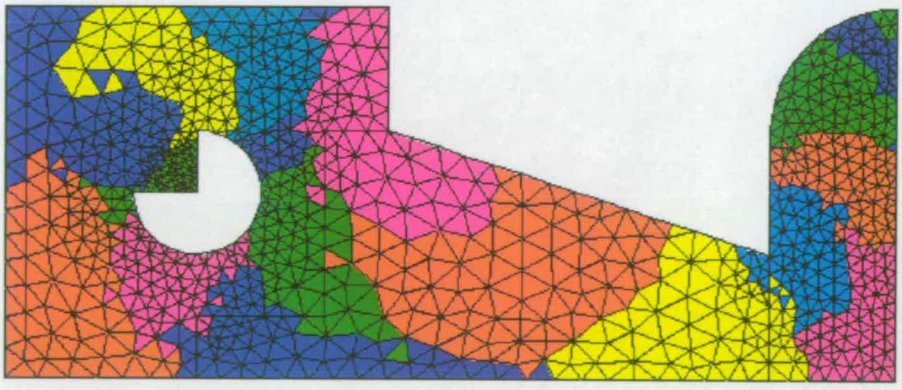


Figure 6.16: Decomposition of Widget over 16 processors by the greedy algorithm.

Here we start from a seed vertex in the graph, rather than a seed node in the mesh. The elements marked in black are these seed vertices.

A full decomposition of the mesh is shown in figure 6.16. It can be seen that most sub-domains have a quite compact shape and we have avoided the formation of elongated sub-domains which occurred during lexicographic decomposition. However, as more and more sub-domains are formed the remaining part of the mesh becomes increasingly convoluted. This results in less satisfactory sub-domain shapes, with increased boundary sizes, indeed, often the growth of a sub-domain will be trapped by surrounding sub-domains and a disconnected sub-domain results.

Nonetheless, the greedy algorithm is a fast and simple algorithm, which is independent of mesh numbering, and produces strict load balance and a generally acceptable (if not optimal) level of communication.

6.4 Optimisation Algorithms

Optimisation algorithms are quite general algorithms for the class of problems where we can write some objective function, $H(x)$, which measures the quality of a solution, x. This objective function maps from the space of all possible solutions of the problem to $\mathbb{R}$, and will be minimised for an exact solution. The approach is then to explore the solution space (in an intelligent manner) looking for a global minimum of H.

The objective function is sometimes referred to as a *Hamiltonian*, in that is it analogous to a physical energy; we may think of the exploration of the solution space as exploring a landscape, looking for the deepest valley. Another term for H is the *cost function*, in that it measures the cost of an inexact solution. Of course, we could equally well phrase this in terms of maximising a function; here we will always refer to minimisation.

We have seen in section 5.3 how we can model the run time of an unstructured mesh calculation as a function of the decomposition, and have arrived at the statement of the generalised partitioning problem 5.7. It is therefore clear that we can apply optimisation algorithms to the task at hand.

We will now survey the algorithms commonly applied to optimisation problems, keeping the discussion in quite general terms, and then look at how these al-

gorithms may be applied to decomposition.

6.4.1 Gradient Descent

If we use the analogy of exploring an energy landscape then a simple algorithm would be to head straight downhill from some arbitrary initial point. This is the *gradient descent* algorithm (also known as *hill climbing*, in the context of maximisation), and is shown in the pseudocode of figure 6.17.

Pseudo: Gradient Descent

1. **While** $\delta H > 0$
2. Evaluate H for all changes, $\delta(x_i)$, in some small neighbourhood of x_i.
3. Choose $\delta(x_i)$ such that $\delta H = H(x_i + \delta(x_i)) - H(x_i)$ is minimised.
4. $x_{i+1} = x_i + \delta(x_i)$.
5. **EndWhile**

EndPseudo

Figure 6.17: The gradient descent algorithm.

In other words, $\delta(x_i)$ is the direction of maximum negative gradient of H at x_i, and we move in that direction.

The drawback with this is that, although we will certainly reach a minimum, we have no guarantee that it is remotely near the global minimum. This is shown in figure 6.18. If we are starting from some point in the 'valley' around the global minimum, it is clear that we will converge to the desired solution using the gradient descent algorithm. However, if we start at *any* point outside this region, the process will become trapped in a local minimum.

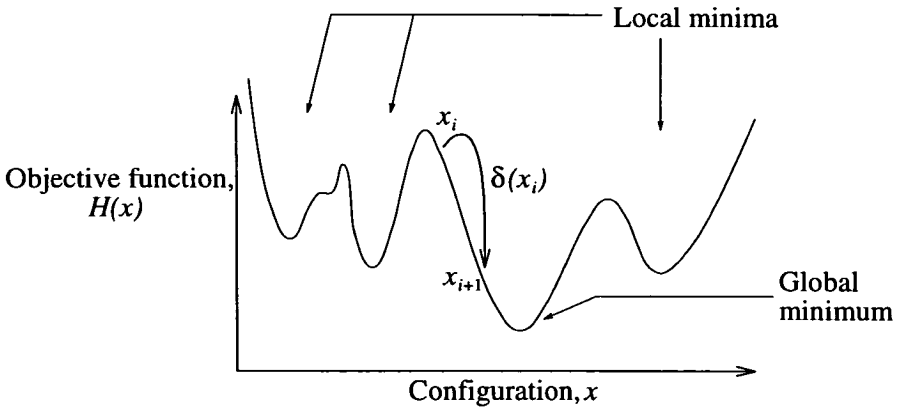


Figure 6.18: The energy landscape and an iterative change to the solution.

It is for this reason that the gradient descent algorithm is not commonly used, except for problems with exceptionally well behaved objective functions. This is the case with the method of steepest descent introduced in section 5.1.5, for the solution of systems of linear equations; because the method is based on an objective function derived from the quadratic form of a symmetric positive definite matrix, we can be assured that there is only one minimum, which must therefore be the global minimum [JM92]. In this case we may also evaluate the gradient of H, so there is no need to search for the direction of maximum gradient as we do in 6.17.

6.4.2 Simulated Annealing

Simulated annealing [KJV83] is similar to the gradient descent algorithm, in that it proceeds by iteratively proposing small changes to the solution, but avoids becoming trapped in local minima by also accepting $\delta(x_i)$ which increase H.

It does this in a stochastic manner, where the probability of accepting such a change is a function of a parameter which is analogous to the temperature of a physical system. This temperature starts off at a high value, indicating a high probability of accepting an increase in H, and is slowly reduced to zero, indicating that no such change will be accepted. It is for this reason that the algorithm is called simulated annealing, in that it mimics the formation of highly ordered states (with low energies) in metals as they are slowly cooled from a high temperature.

Given a small, *random* change to the solution, $\delta(x_i)$, we accept or reject it based on the *Metropolis criterion* [MRR+53]:

- If $\delta H \leq 0$ we accept unconditionally.
- If $\delta H > 0$ we accept with probability $e^{-\delta H/T_i}$.

Where T_i is the temperature and $\delta H = H(x_i + \delta(x_i)) - H(x_i)$, as before. The temperature at each iteration is given by the *annealing schedule*, which will monotonically decrease to zero.

The algorithm is shown in the pseudocode of figure 6.19.

If T_i is very large then any change to the solution is accepted and we simply move through the space randomly, with no regard to the objective function. On the

```
Pseudo: Simulated Annealing
1.  While T_i > 0 or δH > 0
2.      Determine T_i from the annealing schedule.
3.      Propose a random δ(x_i).
4.      δH = H(x_i + δ(x_i)) − H(x_i).
5.      Generate random number, r_i ∈ [0, 1]
6.      If δH ⩽ 0 or r_i < e^{−δH/T_i} Then
7.          x_{i+1} = x_i + δ(x_i).
8.      EndIf
9.  EndWhile
EndPseudo
```

Figure 6.19: The simulated annealing algorithm.

other hand, if $T_i = 0$ then only we only accept $\delta H \leq 0$ and we have a situation equivalent to the gradient descent algorithm; we may not go directly downhill, but we shall certainly arrive at the same minimum that the descent algorithm would find (given the same starting point). As before we will have no guarantee that this is the global minimum.

The annealing process bridges the gap between these two extremes, allowing that any part of the solution space may be explored, while eventually settling into a minimum; hopefully the global minimum. It can be shown that, if T_i is decreased sufficiently slowly ($1/\log i$, or better), then the probability of attaining the global minimum tends to certainty [Haj88].

The choice of annealing schedule is crucial to the success and efficiency of the algorithm; if it cools too fast the solution may become caught in a poor local minimum; if it cools too slowly then a large amount of computation is wasted. Making the right choice may require physical insight or trial and error experimentation.

6.4.3 Chained Local Optimisation

Chained local optimisation [MOF91] combines simulated annealing with a local search heuristic. As before, we propose a change $\delta(x_i)$ and accept or reject it based on the Metropolis criterion. However, the change now has two stages. First, a large random 'kick' is given to the solution, for example n of the changes that would have been used in the simulated annealing case, to arrive at an intermediate state. Then a local search heuristic is applied starting from this intermediate state, to seek out a local minimum.

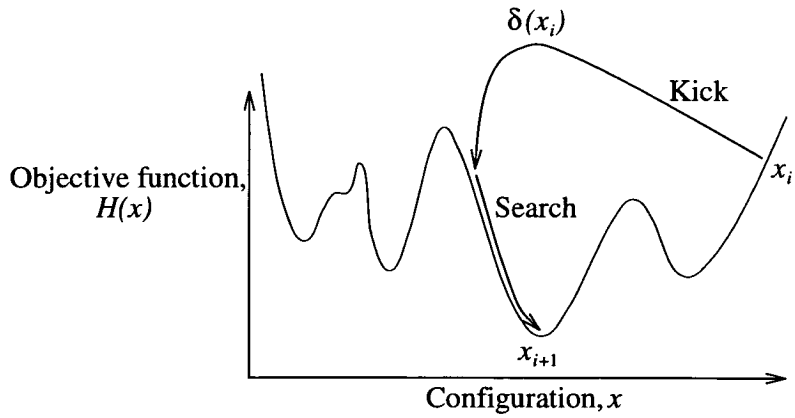


Figure 6.20: Chained local optimisation.

This is illustrated in figure 6.20. If a good local search heuristic is available this can be quite efficient; effectively smoothing out the energy landscape. Where simulated annealing would have to climb over the barriers formed by local maxima in a sequence of steps, chained local optimisation may cross them in one iteration.

6.4.4 Stochastic Evolution

Stochastic evolution is, again, similar to simulated annealing, but now $\delta(x_i)$ is *deterministic*. The accept/reject stage is still random, but the probability of accepting a given increase in H does not monotonically decline. Instead, the probability fluctuates, increasing if no better solution is found, decreasing thereafter. The best solution found during this process is recorded and is used as the final solution, once termination criteria have been met. We limit ourselves to this brief discussion, and refer the interested reader to [RS91] for further details.

6.4.5 Genetic Algorithms

Genetic algorithms [Mic96] take the ideas of biological evolution and apply them to optimisation. A *population* of solutions, which are known as *individuals*, *genotypes* or *chromosomes*, is evolved over successive *generations*. Evolutionary pressure towards the desired solution is introduced via a *fitness function*, which is simply the objective function, H.

Out of each generation parents are *selected*, favouring the fittest. Offspring are

then formed from the parents using a *recombination operator* with random *mutations* being introduced at this stage.

The offspring are then inserted back into the population, sometimes replacing their parents. Thus a new generation is formed and the process repeated. Once the termination criteria (often a set number of generations or a required value of fitness) have been met, the solution is then that encoded in the fittest individual in the population, or, if it has been stored, the fittest individual in any generation.

In contrast to the other methods we have looked at in this section, genetic algorithms examine several areas of the solution space simultaneously (as many as there are individuals in the population). If the recombination operator is correctly chosen it will allow the qualities of the two parents to be merged, rather than garbled, and distinct species of solution to emerge and compete for survival. However, a balance must be stuck between allowing the dominance of a single species (which is akin to becoming trapped in a local minimum) and excessive mutation. We wish to preserve diversity, yet allow some members of the population to approach the global optimum.

Of key importance is the *representation* of the problem, that is, how the possible solutions are encoded in the genotypes. A genotype consists of a set of *alleles*, which correspond to features of the solution. Interpreting the alleles of a genotype gives us the *phenotype*, which is the actual solution.

The representation used in Holland's influential early work on genetic algorithms [Hol75] is a binary one; each chromosome is a bit string of fixed length. Two parents, x_1 and x_2, are then recombined with (for instance) a *one-point crossover* operator, which splits both at the same (random) point:

$$\begin{aligned} x_1 &= (00000|01100000001000), \\ x_2 &= (10101|00011001101111). \end{aligned}$$

The two offspring are then

$$\begin{aligned} x_1' &= (10101|01100000001000), \\ x_2' &= (00000|00011001101111). \end{aligned}$$

In theory we can use this binary representation for any sort of computational optimisation problem. After all, all data is binary - the bit string may represent floating point numbers, characters; any data structure. The attraction of Hol-

land's approach is its generality. Once a binary representation has been chosen, standard techniques may be applied and a solution obtained. However, this does not permit us to take advantage of any features of the problem which may allow for more efficient solution. Another approach is then to use a *problem-specific* representation, with crossover and mutation operators tailored to the situation at hand.

6.4.6 Summary

We summarise these algorithms with a posting to comp.ai.neural-nets [Sar93] quoted in [Mic96]:

> *"Notice that in all [hill-climbing] methods discussed so far, the kangaroo can hope at best to find the top of a mountain close to where he starts. There's no guarantee that this mountain will be Everest, or even a very high mountain. Various methods are used to try to find the actual global optimum.*
>
> *In simulated annealing, the kangaroo is drunk and hops around randomly for a long time. However, he gradually sobers up and tends to hop up hill.*
>
> *In genetic algorithms, there are lots of kangaroos that are parachuted into the Himalayas (if the pilot didn't get lost) at random places. These kangaroos do not know that they are supposed to be looking for the top of Mt. Everest. However, every few years, you shoot the kangaroos at low altitudes and hope that the ones that are left will be fruitful and multiply."*

6.4.7 Applications to Mesh Decomposition

We now examine how optimisation algorithms may be applied to the decomposition of unstructured meshes.

In principle, optimisation algorithms allow for all the details of application and architecture to be taken into account, in that H may be as complex as desired, with no effect on the algorithm itself. H is simply a black box, as far as the algorithm is concerned. Attractive though this is, in practice the use of a complex model will involve the determination of many parameters which may not be readily ascertained. However, if a good model is available (in the form of H)

then optimisation algorithms can approach the task of decomposition in a holistic manner which none of the other algorithms we shall encounter are capable of.

All of the optimisation algorithms we have looked at, with the exception of genetic algorithms, explore the solution space in a step-by-step manner. This means that the point from which we start will have a large impact on the run-time of the optimisation algorithm. If we start near the global minimum then few iterations will (hopefully) be needed to reach it. Similarly, if we start far from it, it may take many iterations to reach it.

In the light of this, there are two ways we can view optimisation algorithms; either as global methods, where we start from a random decomposition and allow the optimisation algorithm to do all the work of finding a good decomposition; or as local refinement techniques, where we use some other algorithm to provide an initial decomposition, and subsequently use optimisation to improve upon it, essentially tidying up the details. The former approach is more likely to come closer to the global optimum, but may take prohibitively long. The latter will generally be quicker (assuming we are using a fast algorithm to generate the initial state) but may not be able to escape from a local minimum near a poor initial decomposition.

In either case, and for that matter with genetic algorithms too, it should be realised that it is unlikely to be efficient to run for sufficiently long to attain the global optimum. All we seek is an *acceptably* good decomposition.

6.4.7.1 Gradient Descent

Gradient descent is not much used for mesh decomposition, due to the problems it has with local minima. Although the energy landscapes commonly found in decomposition problems are relatively well behaved, compared with those found in the travelling salesman problem for instance [FWM94], they are still too convoluted for this algorithm to be applicable.

6.4.7.2 Simulated Annealing

Simulated annealing has been successfully applied to mesh decomposition and graph partitioning in several instances [Wil91, FLS93, JAMS89].

Of particular interest is the nature of $\delta(x_i)$. In line 3 the pseudocode of figure

6.19 we simply state that a random change be made but do not allude to what this might be.

Williams [Wil91] proposes several options:

1. Choose a vertex $v \in V$ at random. If, currently, $v \in S_p$ move it to a random S_q, where $p \neq q$.
2. Choose a vertex $v \in V$ at random. Move v to S_q, where $v_{neighbour} \in S_q$ for some random neighbour, $v_{neighbour}$, of v.
3. Choose a vertex $v \in V$ at random. If, currently, $v \in S_p$ move it to:
   - S_q, where $v_{neighbour} \in S_q$ for some random neighbour, $v_{neighbour}$, of v, with high probability.
   - a random S_q, where $p \neq q$, otherwise.
4. Choose a cluster of vertices $C \subset V$, by choosing a random vertex $v_{initial} \in V$ and adding neighbouring vertices, $v_{neighbour}$, to C with probability p if $v_{neighbour} \in S_p$ and $v_{initial} \in S_p$, never otherwise. C is complete once any $v_{neighbour}$ fails to be added. Use any of the previous methods to move C.

He finds that method 1 tends to produce fragmented sub-domains and is slow to converge, and so introduces method 2.

As method 2 only migrates vertices to neighbouring sub-domains, it effectively moves sub-domain boundaries and so is less prone to this problem. However, a crucial requirement of any $\delta(x_i)$ in simulated annealing is that it be *ergodic*, that is, that we may always reach any state from any other. Method 2 violates this principle, in that, if a sub-domain ceases to exist, or never existed in the initial state, then it can never be created or recreated.

Method 3 is ergodic and combines the features of methods 1 and 2. This is found to be the most favourable of the methods which move individual vertices.

Method 4 forms a cluster around an initial vertex within the sub-domain of which it is part. The whole cluster is then moved as a unit, thus further reducing fragmentation and speeding up the process.

Williams also examines a variant of simulated annealing, *collisional simulated annealing*, where several moves are made at once, and which may be implemented in parallel itself.

Williams discussed the form of H and makes an interesting point concerning H_{calc}, which we have not examined in detail before[1].

Previously, we merely stated that H_{calc} should be minimised when load balance is achieved. A simple statement of this would be

$$H_{calc} = \max_{p \in P} |S_p|_v.$$

As the whole calculation runs at the speed of the most heavily loaded processor, this would seem to be satisfactory. However, if we add a linear perturbation, for example one proportional to $|S_0|_v$, so that

$$H_{calc} = \max_{p \in P} |S_p|_v + \epsilon |S_0|_v,$$

then H_{calc} exhibits an undesirable, discontinuous behaviour. If $\epsilon < 1/(k-1)$ then the minimum of H_{calc} is achieved when $|S_p|_v = |V|_v/k \quad \forall p \in P$, but if $\epsilon > 1/(k-1)$ then the optimum becomes $|S_q|_v = 0$ and $|S_p|_v = |V|_v/(k-1) \quad \forall p \neq q$.

As we intend to add such a perturbation, in the form of H_{comm}, we prefer a sum of squares, so that

$$H_{calc} = \zeta \sum_p |S_p|_v^2,$$

where ζ is a scaling constant.

Williams uses a form of H_{comm} similar to equation 5.5, in that he assumes that the total communication cost is the sum of the individual costs and that $t_{latency} = 0$. He takes no account of network distance, so that $h_{ij} = 1 \quad \forall i \neq j$ and is zero otherwise, and arrives at

$$H_{comm} = \epsilon |E_{cut}|_e,$$

where ϵ is a scaling constant.

He then chooses the scaling constants so that the optimal H_{calc} and H_{comm} have approximately unit contributions from each processor, with

$$\zeta = \frac{k^2}{|V|_v^2} \quad \text{and} \quad \epsilon = \left(\frac{k}{|V|_v}\right)^{\frac{d-1}{d}},$$

where the form of ϵ incorporates the dimensionality of the mesh because the

[1]Williams studies only unweighted graphs; we use the notation for weighted graphs here, for consistency.

surface area of a compact shape shape in d dimensions varies as the $(d-1)$ power of the size, while the volume varies as the d power.

The final form of H is then

$$H = \frac{k^2}{|V|_v^2} \sum_p |S_p|_v^2 + \mu \left(\frac{k}{|V|_v} \right)^{\frac{d-1}{d}} |E_{cut}|_e. \tag{6.1}$$

Using 6.1 Williams compares simulated annealing, used as a global decomposition method, to the fast and simple recursive coordinate bisection algorithm and the slower, but more sophisticated recursive spectral bisection algorithm, both of which we will study later in this chapter (sections 6.6.1 and 6.7.2). He concludes that, for sufficiently slow cooling, simulated annealing produces the best results, both in terms of the value of H and actual application run-time. However, he finds the time taken for the decomposition to be far too long in comparison to the other methods, speculating that the numerous input parameters may not be optimally set.

When used as a local refinement technique simulated annealing has proved more useful. The software package TOP/DOMDEC [FLS93] for mesh decomposition successfully implements the algorithm in this this manner, offering the user a variety of terms which may be included in H.

6.4.7.3 Chained Local Optimisation

Chained local optimisation has been applied to mesh decomposition by Martin and Otto [MO95], having previously successfully applied the algorithm to the travelling salesman problem [MOF91].

The 'kick' used to find an intermediate state is an exchange of n vertices, with n random and not too small. Each set of n vertices is generated as a cluster in a manner similar to the clustering suggested in the previous section. The local search that follows this uses the Kernighan and Lin algorithm which we will examine in section 6.8.1.

They find that chained local optimisation is superior to simulated annealing, and also to coordinate bisection followed by Kernighan and Lin. However, they only examine the method for bisection of a graph.

6.4.7.4 Stochastic Evolution

TOP/DOMDEC uses stochastic evolution as a local refinement technique as in the same manner as simulated annealing. No comparison is given in [FLS93].

6.4.7.5 Genetic Algorithms

The problem of representation is very apparent when we try to apply genetic algorithms to mesh decomposition.

If we consider partitioning over $k = 2^{d_{net}}$ processors, as is common, we may use Holland's bit string representation where the length of the string is $d_{net}n_v$. The first d_{net} bits determine which processor v_0 is assigned to, the second d_{net} bits determine the processor for v_1, and so on.

The problem with this, is that maintaining a sufficiently large population will require a correspondingly large amount of memory and, although genetic algorithms can often out-perform simulated annealing, the method is still likely to be too slow. Even if $d_{net} = 1$, and we are simply bisecting the mesh (the method could, after all, be applied recursively; see section 6.5 which follows) this will still be the case, as n_v is the dominant factor.

Clearly, if genetic algorithms are to be competitive we need a more compact and problem-specific representation.

Using genetic algorithms to optimise the position of a line, in two dimensions, or a plane, in three dimension, which recursively bisects the mesh is one possibility. Another is to use a coarse approximation to the mesh (see section 6.9.1 which fully discusses multilevel methods) and partition this. Both these approaches have been used in the *sub-domain generation method* [KT93, ST97], where decomposition becomes part of the mesh generation process, the coarse mesh having yet to be finely meshed, and a neural network being employed to predict the number of elements that will be generated within each coarse element.

Another approach is to use a representation where the partition is determined by a set of seed vertices, one for each sub-domain. Sub-domains are then simultaneously grown out from the seeds, in a similar layered manner to the greedy algorithm. This allows a compact representation to be used, although it does restrict the solution to a sub-set of all possible solutions. However, this sub-set consists of those partitions where the sub-domains are connected and tend to be

compact in shape; precisely those we would favour. This approach was taken by an EPCC summer scholarship project [Wen96], and will be the subject of chapter 10.

6.5 Recursive Partitioning

We have seen in section 5.4 that even the simplest of the problems posed, namely the partitioning problem 5.6, is of considerable complexity. One way to reduce the complexity of the problem is to partition into $l \ll k$ parts, and then recursively partition each of these parts in the same manner, until the required k sub-domains have been generated. This, we refer to as *recursive* partitioning, which is illustrated for a simple dual graph with $l = 2$ in figure 6.21 (the graph shown is the same as figure 5.8).

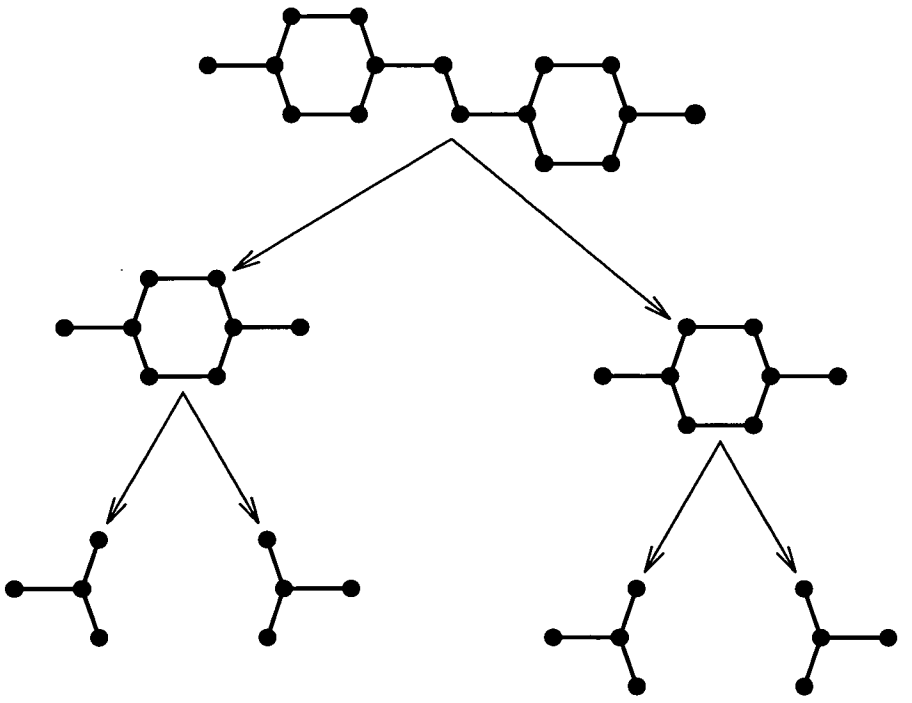


Figure 6.21: Recursive bisection of a dual graph.

Commonly, recursive *bisection* is employed ($l = 2$), although methods for *quadrisection* ($l = 4$) and *octasection* ($l = 8$) are also known. We refer to methods where $l > 2$ as *multi-dimensional.*

In general, recursive methods attack either the partitioning problem 5.6 or the constrained partitioning problem 5.8 where strict load balance is required[2].

6.5.1 Limitations

Clearly, the more we restrict the range of possible solutions an algorithm may produce the more likely we are to miss other, better solutions.

A theoretical study [ST93] indicates that, for the type of graphs which derive from finite element and volume meshes, recursive bisection is normally within a constant factor of the optimum, using cut edges as a metric of quality. This study also indicates that, by relaxing the requirement of strict load balance (requirement (a) in 5.6 and 5.8), a better partition may be found. The edge cuts of a strictly load balanced partition obtained by bisection are found to be $O(k^{1/d} n_v^{1-1/d})$ and the edge cuts of an approximately balanced partition, where $|S_p|_v < (1+\epsilon)|V|_v/k \quad \forall p \in P$, for some small ϵ, are within a factor of $O(\log k)$ of the optimum.

A particular deficiency of bisection is that it can not take network distance into account. Even if we are solving the constrained partitioning problem 5.8, any terms in H_{comm} which model network distance are irrelevant. However, if we use a multi-dimensional method this is not the case.

This is clear if we consider the hops metric of network distance introduced in equation 5.5. If we are recursively partitioning into l parts, then we can use the hops metric at each stage, as if $2^{d_{rec}} = l$, but if $l = 2$ then $d_{rec} = 1$ and it makes no difference which of the two 'sub-domains' are on which of the two 'processors' as $h_{01} = h_{10}$ (for any sane architecture).

6.5.2 Separator Fields

After [Wil91] we introduce the concept of a *separator field.* This is simply a real number associated with each graph vertex, in other words a vector $\boldsymbol{f} \in \mathbb{R}^{n_v}$.

Given a separator field we may partition a graph into l parts, S_j, with the simple algorithm given in the pseudocode of figure 6.22.

[2]Note that even the partitioning problem 5.6 for bisection is still NP-complete.

```
Pseudo: Separator
 1. i = 0
 2. Sort the vertices on the key f.
 3. For j = 1 to l
 4.    S_j = ∅
 5.    While |S_j|_v ⩽ |V|_v/l
 6.       S_j ⊎ v_i
 7.       i += 1
 8.    EndWhile
 9. EndFor
EndPseudo
```

Figure 6.22: Partitioning with a separator field.

If we are bisecting an unweighted graph this is akin to finding the median of $\boldsymbol{f}$, call it m_f, and partitioning into $S_0 = \{v_i \in V : f_i \leqslant m_f\}$ and $S_1 = \{v_i \in V : f_i > m_f\}$.

Once we examine the nature of $\boldsymbol{f}$ it will become clear that, if $l > 2$ is used, then the mesh will be divided into parts with increasingly poor aspect ratios. Indeed, there is nothing to stop us using $l = k$, however, this would result in very elongated sub-domains and a correspondingly large volume of communication. In this instance, each sub-domain would be expected to have very few neighbours, so that if $t_{latency}$ were *very* high this may be beneficial. This not usually the case, and recursive bisection is typically usually used for separator field based techniques.

Multi-dimensional methods use more than one separator (two for quadrisection, three for octasection) at each level of recursion [HL93a, HL92]; hence the choice of terminology.

6.6 Geometry Based Recursive Algorithms

If we have geometric information associated with a graph then we can use this to provide a separator field for partitioning. As unstructured meshes always have coordinate information associated with them, this is a useful approach.

Each mesh element, ε_i, consists of a set of nodes, $\{\eta_a, \eta_b, \ldots\}$, the coordinates of which will be known. If the coordinates are $\boldsymbol{\chi}_a, \boldsymbol{\chi}_b, \ldots \in \mathbb{R}^d$ then their mean may be used as a position for the corresponding graph vertex v_i. If that position is $\boldsymbol{x}_i$ then $\boldsymbol{x}_i = (\sum_{\eta_j \in \varepsilon_i} \boldsymbol{\chi}_j)/|\varepsilon_i|$.

6.6.1 Coordinate Partitioning

Given this geometric information for the graph we can simply use the component of the vertex coordinate in the direction of one of the coordinate axes as a separator field. If $\boldsymbol{d}_j$ is the unit vector in the direction of the j-axis the separator is then $f_i = \boldsymbol{x}_i.\boldsymbol{d}_j$. This will result in the graph being partitioned with a plane (or planes, if $l > 2$) orthogonal to $\boldsymbol{d}_j$.

The question is then; which axis do we choose?

If the graph vertices are distributed evenly in space then the volume of communication resulting from cutting through the mesh with a plane (or line in two dimensions) is proportional to the size of the intersection of the plane with the simulation domain, and clearly should be minimised.

If the simulation domain is particularly extended in one direction then choosing that direction is likely to provide the best choice, as shown in figure 6.23.

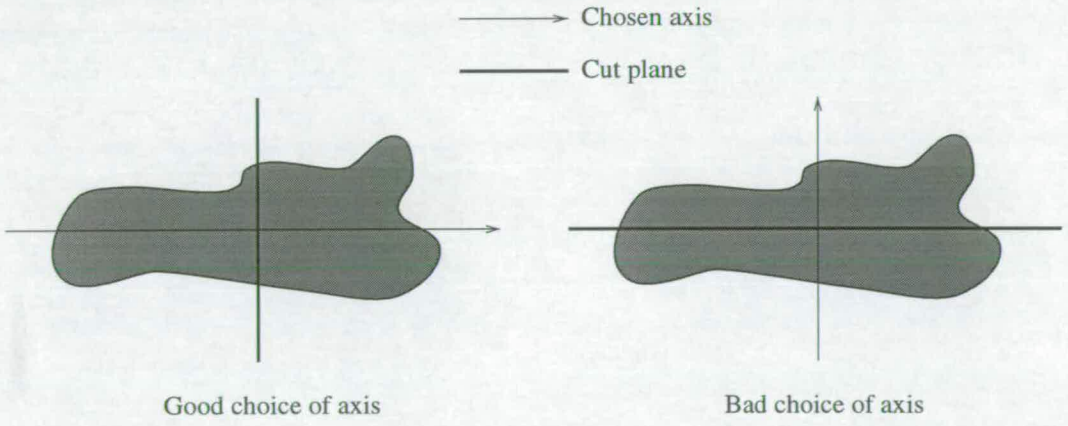


Figure 6.23: Good and bad choices of axis for coordinate partitioning.

Having made this observation there are several ways we may apply coordinate partitioning:

1. Find the direction of maximum extent, determine the separator *once* for this direction, and partition using $l = k$.
2. Find the direction of maximum extent, determine the separator for this direction, bisect the graph ($l = 2$) and repeat the procedure recursively.
3. Ignore this observation and cycle through the dimensions, recursively bisecting, as before.

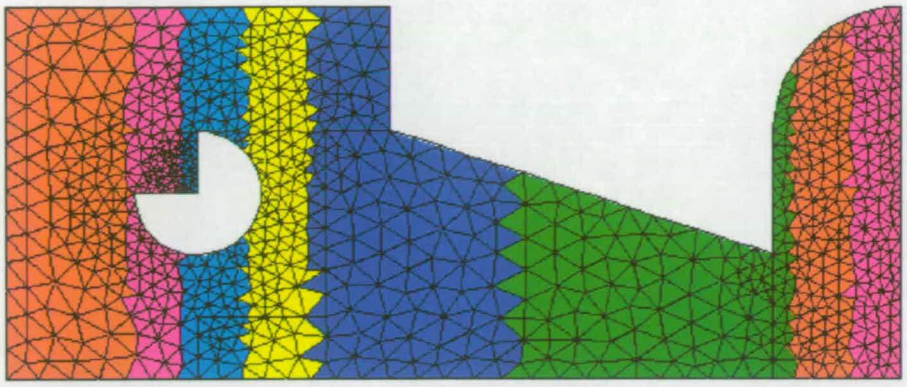


Figure 6.24: Direct coordinate partition of Widget over 8 processors.

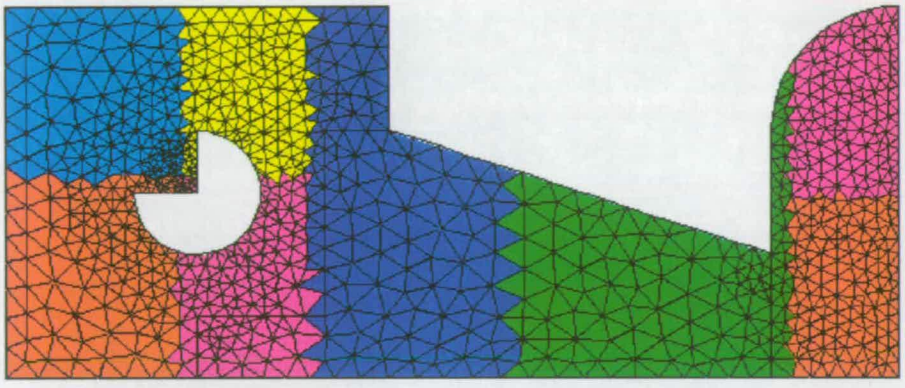


Figure 6.25: Recursive coordinate bisection of Widget over 8 processors.

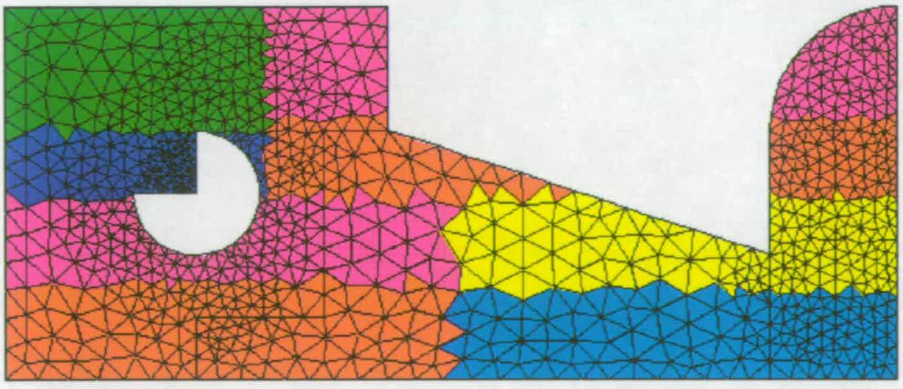


Figure 6.26: Orthogonal recursive bisection of Widget over 8 processors.

The first option, which we term *direct coordinate partitioning*, suffers from the advantages and disadvantages mentioned in section 6.5.2, concerning elongated sub-domains. This is illustrated in figure 6.24.

The second is generally preferable (in terms of $|E_{cut}|_e$), forming much more compact sub-domains, as shown in figure 6.25. This method was termed *recursive coordinate bisection* by Simon [Sim91].

The last option saves computation by not evaluating which direction is preferable in the hope that, on average, it will make little difference. This method was termed *orthogonal recursive bisection* by Williams [Wil91]. In practice, evaluating the preferred direction is of little cost, and the results of not doing so are often significantly worse that the previous option, as is illustrated in 6.26.

Recursive coordinate bisection[3] (RCB) is a very fast method, which is easy to implement and can often produce acceptable results. However, it takes no account of graph connectivity and the assumption that graph vertices are evenly distributed is usually not the case. Its fundamental flaw is that it relies on the mesh being strongly aligned with the coordinate axes.

6.6.2 Inertial Partitioning

In view of the flaw of coordinate partitioning which we have just observed, inertial partitioning was developed. Rather than rely on the alignment of the mesh with the coordinate axes, inertial partitioning seeks to determine the direction of maximum extent directly from the vertex coordinates.

Based on the observation that a rotating object has minimal moment of inertia when rotating about its long axis (if it is reasonably compact), inertial partitioning treats each vertex as a point mass and calculates this direction, $\boldsymbol{d}_{inertial}$. This is then used to give the separator as before, with $f_i = \boldsymbol{x}_i.\boldsymbol{d}_{inertial}$. This is illustrated in figure 6.27, in contrast to coordinate partitioning.

The calculation of $\boldsymbol{d}_{inertial}$ is not a computationally intensive one; the moment of inertia tensor, $\boldsymbol{I}$ is formed by accumulating the contributions from each vertex, this is then solved for its smallest eigenvalue and the corresponding eigenvector is taken as $\boldsymbol{d}_{inertial}$. As $\boldsymbol{I} \in \mathbb{R}^{d \times d}$ the eigensolution is trivial and may be carried

[3]When the RCB algorithm is selected in PUL-md all three of the options mentioned here are available via tunable parameters. So, in that context, RCB may mean any of these variants. In general, if we use the term RCB we mean option 2, unless stated otherwise.

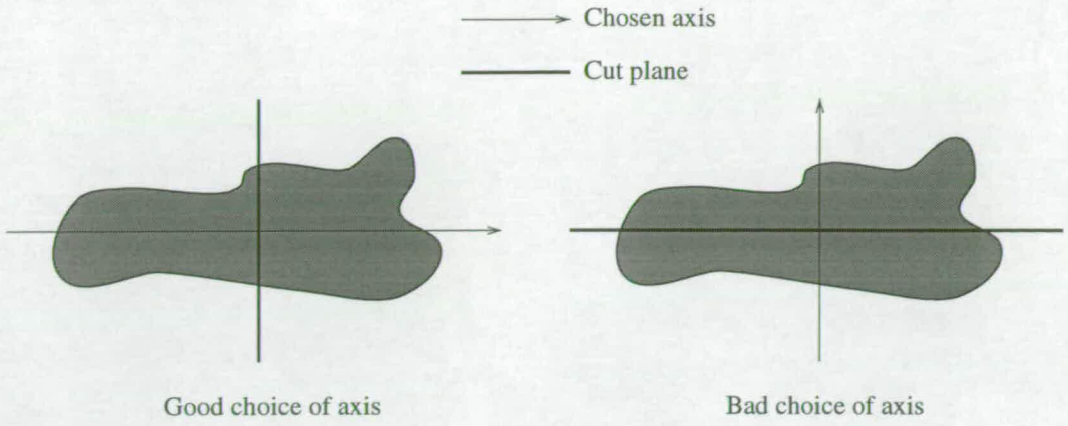


Figure 6.27: Coordinate versus inertial partitioning.

out with any number of standard methods.

The contribution of vertex v_i to the moment of inertia tensor[4] is, in three dimensions,

$$\boldsymbol{I}_i = \begin{pmatrix} \hat{y}_i^2 + \hat{z}_i^2 & -\hat{x}_i\hat{y}_i & -\hat{x}_i\hat{z}_i \\ -\hat{y}_i\hat{x}_i & \hat{x}_i^2 + \hat{z}_i^2 & -\hat{y}_i\hat{z}_i \\ -\hat{z}_i\hat{x}_i & -\hat{z}_i\hat{y}_i & \hat{x}_i^2 + \hat{y}_i^2 \end{pmatrix}$$

where $\hat{\boldsymbol{x}}_i \equiv (\hat{x}_i, \hat{y}_i, \hat{z}_i) = \boldsymbol{x}_i - \boldsymbol{x}_{cg}$ and $\boldsymbol{x}_{cg}$ is the centre of mass given by $\boldsymbol{x}_{cg} = (\sum_{v_i \in V} \boldsymbol{x}_i)/n_v$. In other words, $\hat{\boldsymbol{x}}_i$ is the coordinate of v_i relative to the centre of mass.

$\boldsymbol{I}$ is then the sum of the contributions of all the vertices

$$\boldsymbol{I} = \sum_{v_i \in V} \boldsymbol{I}_i.$$

We then solve $\lambda \boldsymbol{i} = \boldsymbol{I}\boldsymbol{i}$ for the eigenvalues $\lambda_1 \leqslant \lambda_2 \leqslant \lambda_3$ and corresponding eigenvectors $\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{i}_3,$. The required direction is then $\boldsymbol{d}_{inertial} = \boldsymbol{i}_1$.

Having arrived at $\boldsymbol{d}_{inertial}$ we may then either proceed as in option 1 of the previous discussion of coordinate partitioning, with $l = k$, which we would term *direct inertial partitioning*, with the comments previously made still applicable, or we may use bisection, in which case the method is widely known as *recursive inertial bisection* (RIB), illustrated in figure 6.28.

Although RIB makes no assumptions as to the orientation of the simulation

[4]Note that vertex weights do not figure in this formulation.

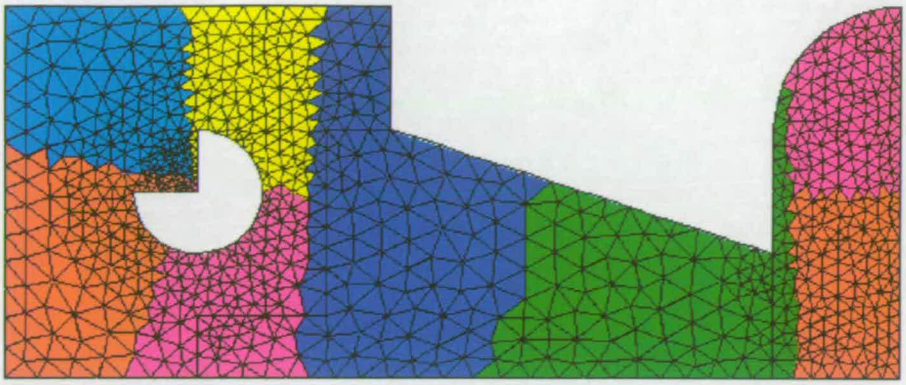


Figure 6.28: Recursive inertial bisection of Widget over 8 processors.

domain, it still has the deficiency that it does not take into account graph connectivity in common with RCB. Nonetheless, it is a fast, robust, easily implemented method for obtaining partitions of reasonable quality and will almost always produce partitions of as good or better quality than RCB. It has therefore been widely employed [FL93, FLS93, HL95, HL94, NORL86, KR92, FR94] and is an excellent choice of algorithm for use in conjunction with local refinement techniques.

6.7 Graph Based Recursive Algorithms

We have seen that coordinate and inertial partitioning do not take graph connectivity information into account, indeed both always cut the mesh with a line or plane, which clearly limits the quality of solutions they may produce, as there is no reason, in general, to suppose the optimal decomposition is one where the sub-domains are delimited in such a regular manner.

An additional consideration is whether we have geometric information at all. In the case of unstructured mesh problems we will, but if we are dealing with a circuit placement problem then we will not.

We now turn our attention to recursive algorithms, which work solely with the dual graph structure, and so avoid these deficiencies (although this is not to say that they are *necessarily* superior).

6.7.1 Layered Partitioning

Layered partitioning is essentially already familiar to us from the discussion concerning bandwidth reduction in section 6.3.4. We stated there that sub-domains of better aspect ratio (if we may use such a geometric term in this context), and hence lesser $|E_{cut}|_e$, are formed if we apply Cuthill-McKee renumbering followed by lexicographic partitioning recursively; this is precisely what we mean by layered partitioning, due to the layer structure induced by Cuthill-McKee.

If we are to take this approach, the best results occur when we bisect the graph at each stage of recursion, as we have seen for coordinate and inertial partitioning. When we do this, we refer to the method as *recursive layered bisection* (RLB).

To review the discussion of bandwidth reduction, as it is applied in this instance, what we are essentially doing is taking a seed vertex in the graph and expanding out in layers around it. In the case of RLB we then bisect the graph into two balanced halves, so that one part consists of those vertices closest to the seed and the the other part is the compliment of this. The boundary between the two will therefore tend to fall in the region of a layer, call it L_{cut}, and the volume of communication across the boundary will be approximately proportional to its size, $|L_{cut}|$. This is illustrated in figure 6.29, where we see the layer structure within the half of the mesh surrounding the seed vertex, the other half of the bisection being of uniform colour.

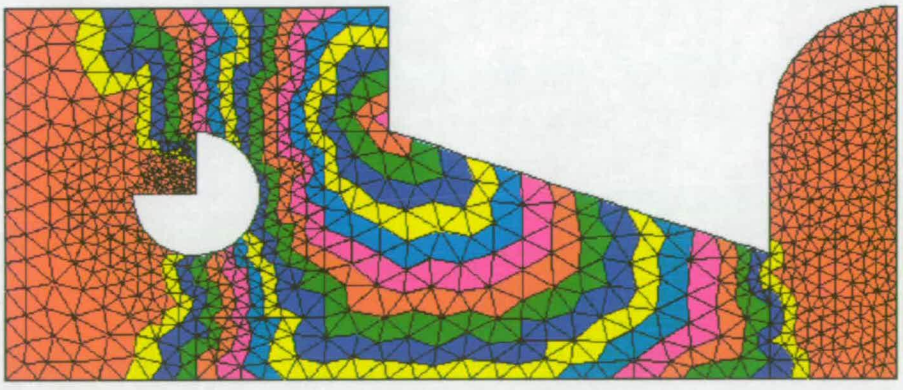


Figure 6.29: Layered bisection for node based graph of Widget starting from vertex 0.

As we discussed in section 6.3.4, $|L_{cut}|$ may be reduced by repeatedly applying Cuthill-McKee to find a pair of maximally separated vertices and using one of these as the final seed. In that section we stated that we are often not concerned with numbering within layers when using Cuthill-McKee for partitioning, and it

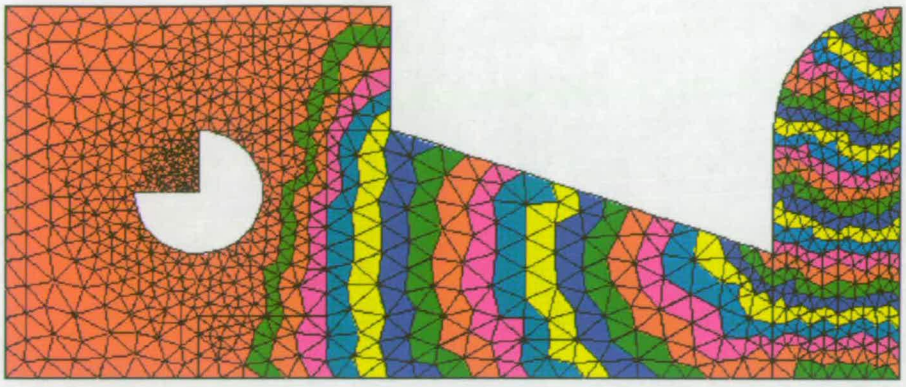


Figure 6.30: Layered bisection for node based graph of Widget starting from vertex 930.

is now clear why, in that numbering within layers has little bearing on RLB. It is possible to view the numbering as a separator field, or, indeed the method as allied to the Greedy algorithm, to which it has clear similarities, but we feel that it is most closely allied to bandwidth reduction, in that Cuthill-McKee is almost always used.

We also saw in section 6.3.4 that the pair of vertices which are maximally separated are 805 and 930 for the Widget data set; if we choose one of this pair then we clearly reduce $|L_{cut}|$ with a commensurate reduction in cut edges. In figure 6.29 we arbitrarily choose vertex 0 as the seed, with $|E_{cut}|_e = 596$, while starting from one of the maximally separated pair (vertex 930, as shown in figure 6.30) is superior, with $|E_{cut}|_e = 388$.

As the computational cost of RLB is proportional to the number of times the Cuthill-McKee algorithm is applied, the runtime of the algorithm increases correspondingly when we use Cuthill-McKee repeatedly to find the maximally separated pair. Fortunately, as we previously observed, Cuthill-McKee usually finds this pair in a very few iterations. As the improvement, compared to starting from an arbitrary seed, is large and the cost of a single iteration is small, this is almost always favourable.

In the form we have described (with Cuthill-McKee finding the maximally separated pair) the method occurs in [Sim91], where it is known as *recursive graph bisection*, and in [FLS93], where it is known as the *recursive reverse Cuthill-McKee algorithm*.

Recursive layered bisection is usually superior to coordinate partitioning and on a par with recursive inertial bisection. Figure 6.31 illustrates this for the Widget

data set partitioned over the same number of processors used in the inertial example (figure 6.28).

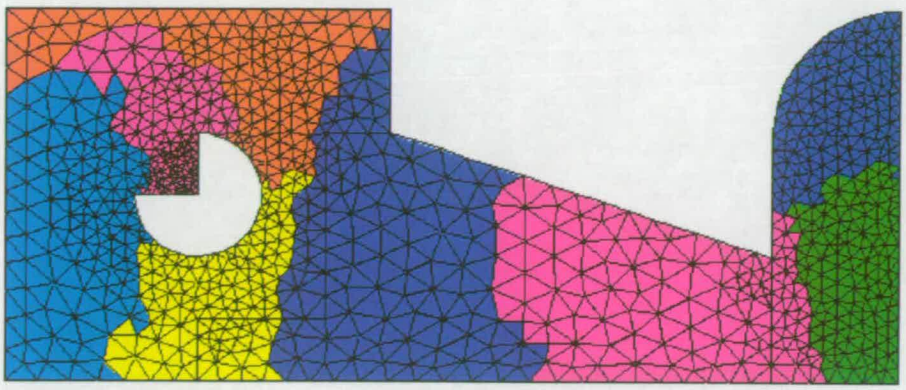


Figure 6.31: Recursive layered bisection of Widget over 8 processors.

6.7.2 Spectral Partitioning

Spectral techniques were first explored in the context of graph related problems by Donath and Hoffman [DH73], Fiedler [Fie75, Fie73] and Barnes [Bar82], but it was the work of Pothen *et al.*, on the factorisation of sparse, symmetric matrices [PSL90] which lead to the application of these techniques to the decomposition of unstructured meshes.

Simon [Sim91] and Williams [Wil91] concurrently used this prior work on factorisation to develop similar recursive bisection algorithms, which they termed *Recursive Spectral Bisection* and *Eigenvector Recursive Bisection*, respectively[5].

The approach taken is to form a matrix whose structure is closely associated with that of the dual graph, namely the *Laplacian* matrix, $\boldsymbol{L} \in \mathbb{R}^{n_v \times n_v}$. The eigenspectrum of $\boldsymbol{L}$ is then examined, hence the term *spectral partitioning*, and, as we shall see, one particular eigenvector of this matrix may be used as a separator field, in exactly the same way that we have encountered in coordinate and inertial partitioning.

We will now examine the mathematical background of spectral partitioning in some detail. Starting with a description of the partitioning problem in a discrete space, we will then see how a continuous approximation may be made, and how a solution of the continuous problem may be found by the eigensolution just described.

[5]We shall follow Simon and use the term Recursive Spectral Bisection, or RSB for short.

6.7.2.1 The Discrete Problem

If we consider recursive graph bisection ($l = 2$), where we attempt to solve the partitioning problem 5.6, to find two balanced sub-sets with minimal cut edges between them at each stage of the recursion (as we discussed in section 6.5) then we may rephrase the problem as follows.

A solution to the partitioning problem 5.6, is the mapping, M_{part}, of graph vertices, v_i, to the set of processors. If we are only concerned with bisection then we can define any such mapping with an *indicator* vector in a discrete space, $\boldsymbol{m} \in \{-1, +1\}^{n_v}$, such that v_i is placed in one half of the bisection if $m_i = -1$ and in the other if $m_i = +1$, and will denote these two sub-sets S_- and S_+, respectively.

We may now reformulate the partitioning problem 5.6 for bisection as the **discrete bisection problem**:

Given a graph, G, find an indicator vector, $\boldsymbol{m} \in \{-1, +1\}^{n_v}$, *such that*

$$(a) \qquad \sum_{v_i \in V} w_v(v_i) m_i \simeq 0 \tag{6.2}$$

and

$$(b) \qquad \frac{1}{4} \sum_{e_{ij} \in E} w_e(e_{ij})(m_i - m_j)^2 \text{ is minimised.}$$

Where the approximation in equality (a) is no larger than the weight of the heaviest vertex, as before.

As all vertex weights are positive (a) clearly states that the sum of the vertex weights in each half of the bisection should be equal, in so far as that is possible (i.e. that $|S_-|_v \simeq |S_+|_v$) and therefore still expresses the load balance requirement of (a) in 5.6.

The second requirement can be seen to be equivalent to that in (b) of 5.6 by noting that $(m_i - m_j)^2$ takes either the value zero or four, depending on whether v_i and v_j are in the same sub-set or not, and therefore whether the edge between them is cut.

Now that we have reformulated the problem we can express it in matrix form, as follows.

Firstly, we expand the sum in (b)

$$\sum_{e_{ij}\in E} w_e(e_{ij})(m_i - m_j)^2 = \sum_{e_{ij}\in E} w_e(e_{ij})(m_i^2 + m_j^2) - \sum_{e_{ij}\in E} 2w_e(e_{ij})m_i m_j. \quad (6.3)$$

We note that the first term on the right of 6.3 only contributes a constant factor, namely

$$\sum_{e_{ij}\in E} w_e(e_{ij})(m_i^2 + m_j^2) = 2|E|_e,$$

as $m_i, m_j \in \{-1, +1\}$.

Any constant factor is irrelevant so far as the minimisation of (b) is concerned, so we could ignore this term or indeed replace it with any other constant term. Such a constant term may be written in the form

$$\boldsymbol{m}^T\boldsymbol{D}\boldsymbol{m} = \sum_{v_i\in V} m_i^2 d_i = \sum_{v_i\in V} d_i = Const.$$

where $\boldsymbol{D} = Diag(\boldsymbol{d}) \in \mathbb{R}^{n_v \times n_v}$ is some diagonal matrix.

However, it will prove convenient, once the continuous approximation to the problem is made, to replace the first term in 6.3 with one that is equivalent. We therefore choose $d_i = |\{e_{ij} : v_i \in e_{ij}\}|_e$, that is the sum of the weights of all edges incident on vertex v_i, so that

$$\boldsymbol{m}^T\boldsymbol{D}\boldsymbol{m} = 2|E|_e,$$

also.

If we now define the *weighted adjacency matrix*, $\boldsymbol{A}$, of the graph as

$$A_{ij} = \begin{cases} w_e(e_{ij}) & \text{if } e_{ij} \in E \\ 0 & \text{otherwise,} \end{cases}$$

then we may treat the second term in 6.3 similarly, so that

$$\sum_{e_{ij} \in E} 2w_e(e_{ij}) m_i m_j = \sum_{v_i \in V} \sum_{v_j \in V} m_i A_{ij} m_j = \boldsymbol{m}^T \boldsymbol{A} \boldsymbol{m}.$$

Defining the matrix $\boldsymbol{L} \equiv \boldsymbol{D} - \boldsymbol{A}$, we therefore see that (b) in the discrete bisection problem 6.2 becomes

$$(b) \qquad \frac{1}{4} \boldsymbol{m}^T \boldsymbol{L} \boldsymbol{m} \text{ is minimised.} \tag{6.4}$$

If the graph is not weighted, so that $w_e(e_{ij}) = 1 \quad \forall e_{ij} \in E$, then $\boldsymbol{L}$ is known as the *Laplacian* matrix of the graph, where the diagonal elements are equal to the degree (number of incident edges) of the corresponding vertices, and off diagonal elements are zero, except where there is an edge between the corresponding two vertices, where they take the value -1. The Laplacian has a number of interesting properties that will make solution of the continuous problem tractable and give some guarantees as to the quality of that solution [Fie75, Fie73].

It is also interesting to recall our previous discussions in section 5.1.4, concerning the structure of the stiffness matrix, and also in section 6.3.4, concerning the relation of a matrix to its dependency graph, as the pattern of the sparsity of $\boldsymbol{L}$ is clearly related.

So far, all we have achieved is to rewrite the problem in an equivalent form. The problem, no matter how it is formulated, remains NP-complete, and we can therefore still not expect to find an exact solution. However, now that we have the problem in matrix form we can exploit this to find an approximate solution by relaxing the constraint that $\boldsymbol{m}$ take only discrete values.

6.7.2.2 The Continuous Problem

In the discrete problem we only allow $\boldsymbol{m} \in \{-1, +1\}^{n_v}$ but, in order to exploit the matrix form of 6.4, we would like to explore more of $\mathbb{R}^{n_v}$ while still remaining in the vicinity of the discrete solution space. Noting $\boldsymbol{m}^T \boldsymbol{m} = n_v$, we therefore look for minimisers of 6.4 in $\{\boldsymbol{f} \in \mathbb{R}^{n_v} : \boldsymbol{f}^T \boldsymbol{f} = n_v\}$.

We can interpret this geometrically, as shown if figure 6.32. The discrete solutions lie at the corners of a hypercube centred on the origin in $\mathbb{R}^{n_v}$, while the continuous

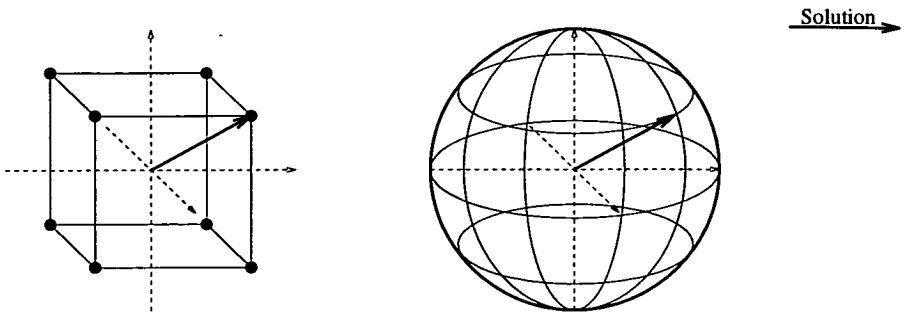


Figure 6.32: Discrete and continuous solution spaces.

solutions lie on the surface of a hypersphere which passes through the cube's corners[6].

If we can find $\boldsymbol{f}$ such that $\frac{1}{4}\boldsymbol{f}^T\boldsymbol{L}\boldsymbol{f}$ is minimised, it is then our hope that we may find some $\boldsymbol{m}$, close to $\boldsymbol{f}$, which will be a good minimiser of $\frac{1}{4}\boldsymbol{m}^T\boldsymbol{L}\boldsymbol{m}$.

Supposing we can indeed find such an $\boldsymbol{f}$, how do we then map it to a 'nearby' $\boldsymbol{m}$? Fortunately, we have already encountered a mechanism to perform just this task; we simply treat $\boldsymbol{f}$ as a separator field, just as we saw in section 6.5.2, and thereby partition the graph into S_+ and S_-, which is equivalent to specifying the values of m_i.

We now define $\boldsymbol{W} = Diag(\boldsymbol{w})$, where $w_i = \sqrt{w_v(v_i)}$, and $\overline{\boldsymbol{W}} = Diag(\overline{\boldsymbol{w}})$, where $\overline{w}_i = 1/w_i$, noting that the vertex weights are all strictly positive.

The only part of the discrete bisection problem 6.2 that we have not yet studied is the load balance constraint (a). Replacing $\boldsymbol{m}$ with $\boldsymbol{f}$, we may now write this as

$$\sum_{v_i \in V} w_v(v_i) f_i = \boldsymbol{w}^T \boldsymbol{W} \boldsymbol{f} \simeq 0.$$

If we apply the change of variables $\boldsymbol{f} = \overline{\boldsymbol{W}}\boldsymbol{g}$, this becomes

[6] A very similar analysis to the one presented here may also be applied to geometric bisection [Wil94]. By making a statistical approximation to the distribution of nodes with a continuous *density of nodes* function, and to the connectivity of the graph with a continuous *pair distribution* function, we may arrive at a solvable continuous problem. If it is assumed that the pair distribution function is sufficiently short range and linear solutions are sought, the eigenvalue problem $\boldsymbol{I}\boldsymbol{f} = \lambda \boldsymbol{P}\boldsymbol{f}$ arises, where $\boldsymbol{P}$ is the pair distribution moment matrix and $\boldsymbol{I}$ the moment of inertia matrix. If the pair distribution is isotropic, then $\boldsymbol{P}$ is proportional to the unit matrix, and we recognise this as inertial bisection.

$$\boldsymbol{w}^T\boldsymbol{g} \simeq 0,$$

as clearly $\overline{\boldsymbol{W}} = \boldsymbol{W}^{-1}$.

Writing $\boldsymbol{L}^w \equiv \overline{\boldsymbol{W}}^T\boldsymbol{L}\overline{\boldsymbol{W}}$, equation 6.4 now becomes

$$\frac{1}{4}\boldsymbol{g}^T\overline{\boldsymbol{W}}^T\boldsymbol{L}\overline{\boldsymbol{W}}\boldsymbol{g} = \frac{1}{4}\boldsymbol{g}^T\boldsymbol{L}^w\boldsymbol{g}.$$

Correspondingly, the normalisation $\boldsymbol{f}^T\boldsymbol{f} = n_v$ must now be $\boldsymbol{g}^T\overline{\boldsymbol{W}}^T\overline{\boldsymbol{W}}\boldsymbol{g} = n_v$. However, to facilitate later analysis, we follow Hendrickson and Leyland [HL92] and approximate the individual vertex weights with their average, $< w_v(v_i) >$, as follows

$$\boldsymbol{g}^T\overline{\boldsymbol{W}}^T\overline{\boldsymbol{W}}\boldsymbol{g} = \sum_{v_i \in V} \frac{g_i^2}{w_v(v_i)} \simeq \frac{\boldsymbol{g}^T\boldsymbol{g}}{< w_v(v_i) >},$$

so that the normalisation may be written[7]

$$\boldsymbol{g}^T\boldsymbol{g} \simeq < w_v(v_i) > n_v = |V|_v.$$

This approximation is reasonable given that the variation in vertex weights is likely to be small, and certainly inconsequential compared to the sum of all vertex weights, as n_v is typically large. Also, there is an error associated with moving from the discrete to continuous which will already be of a similar order.

[7]Hendrickson and Leyland do not, in fact, state this as an approximation, but a strict equality.

We are now in a position to pose the **continuous bisection problem**:

Given a graph, G, find a vector in $\{\boldsymbol{g} \in \mathbb{R}^{n_v} : \boldsymbol{g}^T\boldsymbol{g} = |V|_v\}$, such that

$$(a) \qquad \boldsymbol{w}^T\boldsymbol{g} \simeq 0 \tag{6.5}$$

and

$$(b) \qquad \frac{1}{4}\boldsymbol{g}^T\boldsymbol{L}^w\boldsymbol{g} \text{ is minimised.}$$

We now make some important observations about the matrix $\boldsymbol{L}^w$ that will facilitate the solution of this problem. We base these observations on related observations for $\boldsymbol{L}$, as follows, where we employ the notation $\mathbf{1} \equiv (1, 1, 1, ...)^T \in \mathbb{R}^{n_v}$.

Theorem 6.1 *The matrix $\boldsymbol{L}$ has the following properties:*

1. *$\boldsymbol{L}$ is symmetric and positive semi-definite.*
2. *The vector $\mathbf{1}$ is an eigenvector of $\boldsymbol{L}$ with eigenvalue zero.*
3. *If the graph is connected, then $\mathbf{1}$ is the only eigenvector with eigenvalue zero.*

Proof: $\boldsymbol{L}$ is clearly symmetric, given that it is the sum of $\boldsymbol{D}$, which is diagonal, and $\boldsymbol{A}$, which is symmetric by definition. As $L_{ii} = -\sum_{j \neq i} L_{ij} \quad \forall i$, the row sum of $\boldsymbol{L}$ is zero for all rows. It follows that $\boldsymbol{L}\mathbf{1} = 0$, and hence *2* is proved.

Gerschgorin's theorem tells us that, for a real symmetric matrix, the minimum eigenvalue is $(L_{ii} - \sum_{j \neq i} |L_{ij}|)_{min}$, which is zero by the same argument. All eigenvalues must therefore be zero or positive, and hence $\boldsymbol{L}$ is positive semi-definite and *1* is proved.

If we consider the discrete case, where $\boldsymbol{m} \in \{-1, +1\}^{n_v}$, then we can see that $\boldsymbol{m}^T\boldsymbol{L}\boldsymbol{m} = 0$ iff $\boldsymbol{m} = \mathbf{1}$ (or allowed multiple thereof) for a connected graph. The only way to partition a connected graph into two sub-sets without cutting an edge is if one set is $\emptyset$ and the other is V. This is precisely the partition described by $\boldsymbol{m} = \mathbf{1}$ and, as the quadratic form counts cut edges, the equivalence follows. Now, for a symmetric positive semi-definite matrix, it is easy to show that $\boldsymbol{x}^T\boldsymbol{L}\boldsymbol{x} = 0$ iff $\boldsymbol{L}\boldsymbol{x} = 0$, so there is a direct correspondence between vectors which make the quadratic form zero and those which are eigenvectors of eigenvalue zero. Taking these two observation together would lead us to suspect that *3* is indeed correct, but we refer the reader to [Moh88] for formal proof. □

It is now clear why the choice of $\boldsymbol{D}$ (as the sum of edge weights for edges incident on a vertex) was made as it was; it is this that results in the row sum of $\boldsymbol{L}$ being zero. Without this we could not prove Theorem 6.1 as we have.

Lemma 6.2 *The matrix $\boldsymbol{L}^w$ has the following properties:*

1. *$\boldsymbol{L}^w$ is symmetric and positive semi-definite.*
2. *The eigenvectors of $\boldsymbol{L}^w$ span $\mathbb{R}^{n_v}$ and are orthogonal.*
3. *The vector $\boldsymbol{w}$ is an eigenvector of $\boldsymbol{L}^w$ with eigenvalue zero.*
4. *If the graph is connected, then $\boldsymbol{w}$ is the only eigenvector with eigenvalue zero.*

Proof: From property *1* of theorem 6.1, and the definition of $\boldsymbol{L}^w$ as $\overline{\boldsymbol{W}}^T\boldsymbol{L}\overline{\boldsymbol{W}}$, *1* immediately follows.

The eigenvectors of any symmetric matrix may always be chosen to be pairwise orthogonal, even in the presence of multiple eigenvalues, and will therefore span the space. Thus, the symmetry of $\boldsymbol{L}^w$ implies *2*.

Although the transformation of of $\boldsymbol{L}^w$ to $\overline{\boldsymbol{W}}^T\boldsymbol{L}\overline{\boldsymbol{W}}$ is not a similarity transformation, as $\overline{\boldsymbol{W}}$ is not orthogonal, it does preserve eigenvalues which are zero. If we consider $\boldsymbol{x}$, such that $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{0}$, then clearly $\boldsymbol{L}^w\boldsymbol{W}\boldsymbol{x} = \boldsymbol{0}$. Hence, if $\boldsymbol{x}$ is an eigenvector of $\boldsymbol{L}$ with eigenvalue zero, then $\boldsymbol{W}\boldsymbol{x}$ is an eigenvector of $\boldsymbol{L}^w$ with eigenvalue zero, also. As $\boldsymbol{W}\boldsymbol{1} = \boldsymbol{w}$, properties *3* and *4* follow from properties *2* and *3* of theorem 6.1, respectively. □

We are now in a position to try to find a solution of the continuous bisection problem 6.5.

6.7.2.3 Solution of the Continuous Problem

If we take the eigenvalues of $\boldsymbol{L}^w$ to be ordered such that $\lambda_1 \leqslant \lambda_2 \leqslant \ldots \leqslant \lambda_{n_v}$, with corresponding orthonormal eigenvectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_{n_v}$, then the solution is given by the following theorem.

Theorem 6.3 *The solution of the continuous bisection problem 6.5 is given by* $\boldsymbol{g} = \sqrt{|V|_v}\boldsymbol{e}_2$.

Proof: Property *2* of Lemma 6.2, shows that we may express any potential solution in terms of the eigenvectors of $\boldsymbol{L}^w$.

Let $\boldsymbol{E}$ be a matrix whose columns are the orthonormal eigenvectors and $\boldsymbol{\Lambda}$ the diagonal matrix of eigenvalues. We may then express a solution as $\boldsymbol{g} = \boldsymbol{E}\boldsymbol{c}$, where $\boldsymbol{c}$ is a vector of real coefficients.

We may ensure that $\{\boldsymbol{g} \in \mathbb{R}^{n_v} : \boldsymbol{g}^T\boldsymbol{g} = |V|_v\}$ by taking $\boldsymbol{c}^T\boldsymbol{c} = |V|_v$, as clearly $\boldsymbol{g}^T\boldsymbol{g} = \boldsymbol{c}^T\boldsymbol{c}$ by the orthogonality of $\boldsymbol{E}$ given in property *2* of Lemma 6.2.

We know from property *3* of Lemma 6.2 that $\boldsymbol{e}_1 = \boldsymbol{w}$. Making use of the orthogonality of $\boldsymbol{E}$ once more, we see that $\boldsymbol{g}$ will only satisfy requirement (a) of the problem if $c_1 = 0$.

We need now only satisfy the minimisation requirement (b), which may be written as

$$\frac{1}{4}\boldsymbol{g}^T\boldsymbol{L}^w\boldsymbol{g} = \frac{1}{4}\boldsymbol{c}^T\boldsymbol{E}^T\boldsymbol{L}^w\boldsymbol{E}\boldsymbol{c} = \frac{1}{4}\boldsymbol{c}^T\boldsymbol{\Lambda}\boldsymbol{c} = \frac{1}{4}\sum_{i=2,n_v} c_i^2\lambda_i.$$

We note that the lower limit of $\frac{1}{4}\boldsymbol{g}^T\boldsymbol{L}^w\boldsymbol{g}$ is

$$\frac{1}{4}\lambda_2|V|_v = \frac{1}{4}\lambda_2\boldsymbol{c}^T\boldsymbol{c} \leq \frac{1}{4}\sum_{i=2,n_v} c_i^2\lambda_i.$$

Here we rely on property *1* of Lemma 6.2 to ensure that the eigenvalues are non-negative, the ordering we have imposed on the eigenvalues and the properties of $\boldsymbol{c}$ we have so far established.

Clearly, if we choose $c_i = 0 \quad \forall i \neq 2$ this lower limit may be achieved. The normalisation, $\boldsymbol{c}^T\boldsymbol{c} = |V|_v$, then gives $c_2 = \sqrt{|V|_v}$ and the required solution is therefore $\sqrt{|V|_v}\boldsymbol{e}_2$. □

Further, we note that if the graph is connected then property *4* of Lemma 6.2 tells us that this is a non-trivial solution. Additionally, if $\lambda_2 \neq \lambda_3$ this solution is unique.

A theorem due to Fiedler [Fie75, Fie73] provides the guarantee of solution quality alluded to in section 6.7.2.1. This theorem implies that if the graph from which $\boldsymbol{L}$ is derived is bisected using the second eigenvector as a separator field then at least one of the two sub-sets that result will be connected. Due to the theoretical work carried out by Fiedler in relation to the eigenspectrum of the Laplacian and its relation to the connectivity of graphs, the second eigenvector, $\boldsymbol{e}_2$, is widely referred to as the *Fiedler vector.*

6.7.2.4 Multi-Dimensional Variants

When we introduced recursive methods, at the start of section 6.5, we also considered the possibility of partitioning at each stage of recursion into $l > 2$ sub-sets, terming this multi-dimensional partitioning [HL93a, HL92]. We also observed in section 6.5.1 that this would be necessary if we were to try to incorporate the hops (or indeed any other) metric of network distance into the algorithm. We shall now extend the spectral approach to partitioning for $l = 4$, and subsequently indicate how it may be extended to higher dimensional partitioning.

Quadrisection

If we are to use the hops metric then we must be partitioning such that $l = 2^{d_{rec}}$ at each level of recursion. The simplest non-trivial case will be $d_{rec} = 2$ as the hops metric can have no bearing on $d_{rec} = 1$, as previously observed. The $d_{rec} = 2$ case is quadrisection (as $l = 4$) and we may incorporate this into the spectral partitioning as follows.

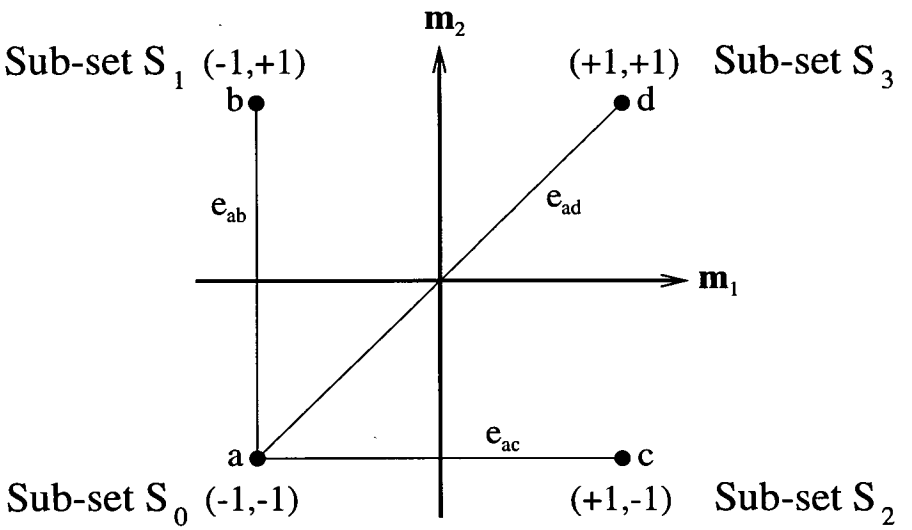


Figure 6.33: The hops metric in two dimensions.

The mapping of vertices to sub-sets may now be described by the use of two indicator vectors, $\boldsymbol{m}_1, \boldsymbol{m}_2 \in \{-1, +1\}^{n_v}$. We interpret this as a binary number for each vertex, so that $-1 \mapsto 0$ and $+1 \mapsto 1$. Thus $(+1, -1) \mapsto 10$ (binary) $\mapsto$ sub-set S_2 (decimal), for example. The hops metric then weights the cost of a cut edge, just as it did for message size in section 5.3.1.

We consider, without loss of generality, a vertex 'a' in sub-set S_0 of a quadrisection, and examine the edges connecting it to vertices 'b', 'c' and 'd' in each of the other sub-sets, as shown in figure 6.33. The number of hops from sub-set S_0 to the other sub-sets are $h_{01} = h_{02} = 1$ and $h_{03} = 2$, for sub-sets S_1, S_2 and S_3 respectively. The cost of the cut-edge e_{ab} is therefore $h_{01}w_e(e_{ab}) = w_e(e_{ab})$ and similarly the cost of the edge e_{ac} is $h_{02}w_e(e_{ac}) = w_e(e_{ac})$. However, the cost of the cut edge to 'd' is $h_{03}w_e(e_{ad}) = 2w_e(e_{ad})$.

This weighted sum of edge-cuts may be expressed as

$$\frac{1}{4}(\boldsymbol{m}_1^T \boldsymbol{L}\boldsymbol{m}_1 + \boldsymbol{m}_2^T \boldsymbol{L}\boldsymbol{m}_2). \tag{6.6}$$

If we consider the edges in figure 6.33 in relation to their contribution to this function, we see that e_{ab} contributes $w_e(e_{ab})$ through the $\boldsymbol{m}_2$ term, and nothing through the $\boldsymbol{m}_1$ term. The edge e_{ac} evidently does the reverse, only contributing it weight through the $\boldsymbol{m}_1$ term, but the edge e_{ad} contributes through *both* $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ terms, giving a net contribution of $2w_e(e_{ab})$. Hence, minimising 6.6 minimises the sum of the cut-edges, weighted by the hops metric.

The load balance constraint used in the bisection case may be generalised to quadrisection. The naïve generalisation would be to only require 6.2 (a) for $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, but this will only ensure balance between the pairs of sub-sets given below.

$$\sum_{v_i \in V} w_v(v_i) m_{1(i)} \simeq 0 \Rightarrow \text{balance between } S_0 \cup S_1 \text{ and } S_2 \cup S_3 \tag{6.7}$$

$$\sum_{v_i \in V} w_v(v_i) m_{2(i)} \simeq 0 \Rightarrow \text{balance between } S_0 \cup S_2 \text{ and } S_1 \cup S_3 \tag{6.8}$$

These two constraints are insufficient alone, as they do not preclude partitions such that $|S_0|_v = |S_3|_v = |V|_v/2$ and $S_1 = S_2 = \emptyset$, or $|S_1|_v = |S_2|_v = |V|_v/2$ and $S_0 = S_3 = \emptyset$. However, if we add the following constraint this problem may be avoided.

$$\sum_{v_i \in V} w_v(v_i) m_{1(i)} m_{2(i)} \simeq 0 \Rightarrow \text{balance between } S_0 \cup S_3 \text{ and } S_1 \cup S_2 \tag{6.9}$$

Now, if we apply constraints 6.7, 6.8 *and* 6.9 only partitions such that $|S_0|_v =$

$|S_1|_v = |S_2|_v = |S_3|_v = |V|_v/4$ are permitted, which is exactly the load balance between the four sub-sets that we require.

Moving from the discrete to the continuous case proceeds exactly as for bisection, allowing us to formulate the **continuous quadrisection problem**:

Given a graph, G, find vectors $\boldsymbol{g}_1$ *and* $\boldsymbol{g}_2$ *in* $\{\boldsymbol{g} \in \mathbb{R}^{n_v} : \boldsymbol{g}^T\boldsymbol{g} = |V|_v\}$, *such that*

$$(a) \qquad \boldsymbol{w}^T\boldsymbol{g}_1 \simeq \boldsymbol{w}^T\boldsymbol{g}_2 \simeq \boldsymbol{g}_1^T\boldsymbol{g}_2 \simeq 0 \tag{6.10}$$

and

$$(b) \qquad \frac{1}{4}(\boldsymbol{g}_1^T\boldsymbol{L}^w\boldsymbol{g}_1 + \boldsymbol{g}_2^T\boldsymbol{L}^w\boldsymbol{g}_2) \text{ is minimised.}$$

Given the similarity of 6.10 to the continuous bisection problem 6.5, it comes as no surprise that the solution is as stated in the following theorem.

Theorem 6.4 *A solution of the continuous quadrisection problem 6.5 is given by* $\boldsymbol{g}_1 = \sqrt{|V|_v}\boldsymbol{e}_2$ *and* $\boldsymbol{g}_2 = \sqrt{|V|_v}\boldsymbol{e}_3$.

Proof: A trivial extension of the proof for Theorem 6.3 is sufficient to prove 6.4. See [HL92]. □

This solution is no longer unique; in fact, we may choose any orthogonal pair of appropriately normalised vectors in $span\{\boldsymbol{e}_2, \boldsymbol{e}_3\}$, as all such pairs yield the same value of 6.10 (b) and still satisfy the constraints of (a).

Such pairs will be given by

$$\boldsymbol{g}_1 = \sqrt{|V|_v}(\cos\theta\boldsymbol{e}_2 + \sin\theta\boldsymbol{e}_3),$$

and

$$\boldsymbol{g}_2 = \sqrt{|V|_v}(-\sin\theta\boldsymbol{e}_2 + \cos\theta\boldsymbol{e}_3).$$

The rotational degree of freedom, θ, that this family of solutions gives us may be exploited to recover some of the accuracy lost in moving from the discrete to the continuous. If we reverse our change of variables, so that $\boldsymbol{f}_1 = \overline{\boldsymbol{W}}\boldsymbol{g}_1$ and $\boldsymbol{f}_2 = \overline{\boldsymbol{W}}\boldsymbol{g}_2$, we may then look for $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ as close to $\{-1, +1\}^{n_v}$ as possible.

This may be expressed as minimising

$$\sum_{v_i \in V} (1 - f_{1(i)}^2)^2 + (1 - f_{2(i)}^2)^2$$

with respect to θ. Substitution of the trigonometric expressions for $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ into this function gives a constant coefficient quartic equation in sines and cosines of θ. This may be solved by a short sequence of local minimisations from random starting points [DS83].

There remains the problem of mapping from the continuous to the discrete solution. The separator method is no longer applicable, but the approach of looking for the 'closest' solution in $\{-1,+1\}^{n_v}$ may be applied here also. If a metric of distance is defined from $(f_{1(i)}, f_{2(i)})$ to $(\pm 1, \pm 1)$ (the square of the Euclidean norm is used in [HL92]) then, within the constraints of load balance, we may attempt to minimise the sum of these distances. This type of minimisation is known as a *minimum cost assignment* problem, for which standard methods are known [TN91].

Taking all this together, we have described *Recursive Spectral Quadrisection*, or RSQ for short.

Higher Partitioning Dimensions

It may be thought that this approach might be extended for higher values of d_{rec}, and this is indeed true up to a point.

Consider the **continuous (unconstrained) multisection problem**:

Given a graph, G, find vectors $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_{d_{rec}}$ *in* $\{\boldsymbol{g} \in \mathbb{R}^{n_v} : \boldsymbol{g}^T\boldsymbol{g} = |V|_v\}$, *such that*

$$\begin{aligned} (a) \qquad \boldsymbol{w}^T\boldsymbol{g}_i &\simeq 0 \quad \forall i \in \{1, \ldots, d_{rec}\} \\ \boldsymbol{g}_i^T\boldsymbol{g}_j &\simeq 0 \quad \forall i \neq j \in \{1, \ldots, d_{rec}\} \end{aligned} \tag{6.11}$$

and

$$(b) \qquad \frac{1}{4}\sum_{i=1,d_{rec}} \boldsymbol{g}_i^T\boldsymbol{L}^w\boldsymbol{g}_i \text{ is minimised.}$$

Clearly for $d_{rec} = 1$ this is equivalent to the bisection problem 6.5 and for $d_{rec} = 2$ to the quadrisection problem 6.10. As we would therefore expect the solutions to this problem are as given in the following theorem.

Theorem 6.5 *A solution of the continuous (unconstrained) multisection problem 6.11 is given by* $\boldsymbol{g}_i = \sqrt{|V|_v}\boldsymbol{e}_{i+1}, \quad i = 1, d_{rec}$.

Proof: As with Theorem 6.4, a trivial extension of the proof for Theorem 6.3 is sufficient to prove 6.5. See [HL92]. □

Again this solution is not unique; any orthogonal set of appropriately normalised vectors in $span\{e_2, \ldots, e_{d_{rec}+1}\}$ will do just as well.

However, for $d_{rec} = 3$ an additional load balance constraint must be satisfied, which 6.11 does not reflect, namely

$$\sum_{v_i \in V} w_v(v_i) m_{1(i)} m_{2(i)} m_{3(i)} \simeq 0.$$

This occurrence of additional constraints for $d_{rec} \geqslant 3$ is the reason that we have termed 6.11 'unconstrained.'

For $d_{rec} = 3$ we have three rotational degrees of freedom in looking for a good solution in $span\{e_2, e_3, e_4\}$, and this allows us to ignore this new load balance constraint until we come to fix these degrees of freedom.

The minimisation with respect to rotational degrees of freedom now becomes minimise

$$\sum_{v_i \in V} \sum_{d=1,3} (1 - f_{d(i)}^2)^2$$

subject to

$$\sum_{v_i \in V} w_v(v_i) f_{1(i)} f_{2(i)} f_{3(i)} \simeq 0.$$

Substitution of the appropriate trigonometric combination of e_2, e_3 and e_4 now gives a constant coefficient polynomial in sines and cosines of the rotational degrees of freedom. This optimisation problem may be solved exactly as in the quadrisection case and the solution then mapped to $\{-1, +1\}^{n_v}$ using minimum cost assignment as before.

We have therefore extended the multi-dimensional approach to octasection ($d_{rec} = 3$, $l = 8$), which we term *Recursive Spectral Octasection* or RSO for short.

However, as we move to higher values of d_{rec} we find that the the number of new constraints we must add to ensure load balance grows faster than the rotational degrees of freedom we gain. For $d_{rec} = 4$ we find that we have five constraints to satisfy and six degrees of freedom, so it is still possible to find a solution, although the constraints include three cubic equations and one quartic, so solution would prove challenging. When we reach $d_{rec} = 5$ it is no longer generally possible to

find a balanced solution by this method, as there are sixteen constraints but only ten degrees of freedom; for higher values of d_{rec} the situation is correspondingly worse [HL92]. We conclude that $d_{rec} = 4$ is the maximum number of partitioning dimensions to which this method may be extended.

6.7.3 Summary

In this study of spectral partitioning we have seen how, by the use of a continuous approximation to an essentially discrete problem, we have reduced the problem to that of eigensolution of the Laplacian matrix, after which the Fiedler vector may be used as a separator field.

We have not discussed how this eigensolution may be carried out, and it is clear that the efficiency of the algorithm depends almost entirely on the efficiency of the eigensolution. We will not detail how this may be done here, but rather defer discussion until we come to look at implementation details in section 7.4.8. For now, we will merely state that an efficient algorithm for the calculation of the extreme eigenvectors of a large sparse symmetric matrix, such as the Laplacian, exists in the form of the *Lanczos algorithm.*

Spectral partitioning is one of the most complex algorithms we have discussed, but it is found that, in practice, it produces partitions of very high quality. This quality comes at the price of increased computational complexity, which leads to a correspondingly greater time taken to calculate a partition by this method, relative to any of the other recursive algorithms we have thus far encountered.

To illustrate the quality of results, we show a partition of the Widget data set over eight processors by recursive spectral bisection in figure 6.34. If we visually compare the result for RSB with the results obtained by the other recursive bisection methods we have studied, it may clearly be seen to be superior. Comparable figures are 6.25, 6.28 and 6.31, which show the decompositions for the same data set and number of processors provided by RCB, RIB and RLB, respectively.

Moreover, spectral partitioning is totally independent of coordinate information, and can therefore be applied to graph partitioning problems unrelated to mesh decomposition, where such information may not be available. It should also be noted that it is independent of graph numbering.

If vertex or edge weights are specified for the graph, then *both* of these are represented in the spectral approach. While any separator based technique may

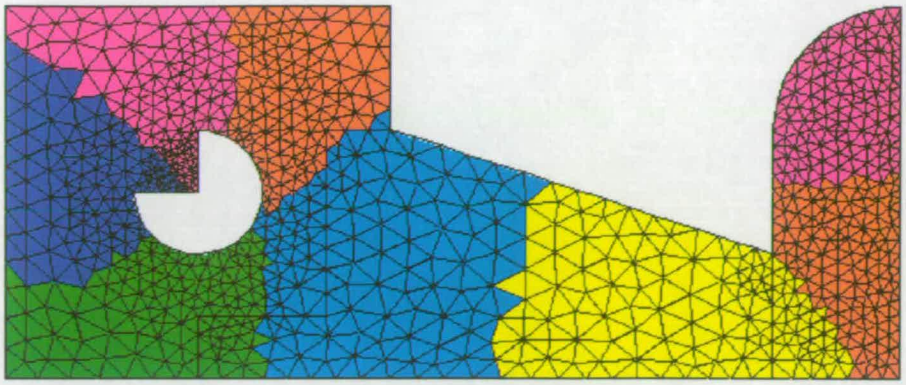


Figure 6.34: Recursive spectral bisection of Widget over 8 processors.

always take into account vertex weights, no other will account for edge weights.

The higher dimensional variants we have discussed allow spectral techniques to deal with the assignment of vertices, and therefore sub-domains, to processors in such a way that network topology may be taken into account. In this respect, they may approach the constrained partitioning problem 5.8 where H_{comm} models hypercube hops, in a way no other recursive algorithm may do without resorting to optimisation techniques.

For all these reasons, spectral partitioning is widely employed where high quality decompositions are sought, and where the cost of arriving at that decomposition is less of an issue. It is also particularly attractive, in that it produces high quality decompositions *reliably*, without the need for the user to tune many parameters relating to the algorithm.

When we later come to look at multilevel algorithms in section 6.9, we will see that they may be used to lessen the cost of spectral partitioning without significantly degrading the quality of results.

6.8 Local Refinement Algorithms

In our previous discussions concerning the application of optimisation algorithms to mesh decomposition (section 6.4.7), we made the point that whether we should regard those algorithms as global methods or local refinement techniques was determined by the initial state from which they explore the search space. If an essentially random start was given we would say that the algorithm was global, while if a reasonable initial partition was provided then we would call them local.

Conversely, what we present here as local refinement techniques may be given a random partition as their initial state, and so may also be employed as global methods; whether this is advisable is a question of efficiency and the resulting quality of decomposition.

In this section we shall examine the following algorithms; Kernighan and Lin, Mob and the Jostle heuristic. The former, Kernighan and Lin, is almost always used to refine an existing reasonable partition, for it is most efficient and reliable when so employed. The same may be said concerning the efficiency of Mob, although this is perhaps more arguable and, indeed, the algorithm was presented initially as a global method. The Jostle heuristic, however, has the concept of locality built-in and so may run into difficulties if not provided with a reasonable partition to start with[8]. All of these algorithms are based on the dual graph of the mesh.

Bearing in mind that the distinction between global and local is largely one of usage, we will now turn our attention to a detailed examination of these algorithms.

6.8.1 Kernighan and Lin

We will begin our discussion of Kernighan and Lin refinement (KL) by presenting the algorithm as applied to an existing bisection of a graph with weighted edges, but no vertex weights [KL70]. We will then indicate how it may be extended to higher partitioning dimensions ($l > 2$) and graphs with vertex weights. Further discussions may be found in the section on the implementation of KL in PUL-md, section 7.5.2.

6.8.1.1 KL for Bisection

Suppose we have a bisection of a graph into the two sub-sets S_0 and S_1, and also suppose that this is a balanced partition, i.e. $|S_0|_v = |S_1|_v = |V|_v/2$, which may not be optimal as far as communication is concerned. The KL algorithm seeks to find a better solution to the partitioning problem 5.6 by moving vertices between S_0 and S_1 in an effort to reduce $|E_{cut}|_e$.

[8]More recent developments of Jostle can begin from a random start, but only the basic algorithm will be presented here.

The algorithm selects which vertices to move by associating a *gain* value with each vertex and preferentially moving those with the highest gain. The gain is simply defined as the reduction in total cut edge weight that would result from moving the vertex from the sub-set it is currently in to the other.

The gain g_i of a vertex v_i may be written as

$$g_i = \sum_{e_{ij} \in E} \begin{cases} +w_e(e_{ij}) & \text{if } M_2(v_i) \neq M_2(v_j) \\ -w_e(e_{ij}) & \text{if } M_2(v_i) = M_2(v_j), \end{cases} \tag{6.12}$$

where $M_2(v) = 0$ if $v \in S_0$ and $M_2(v) = 1$ if $v \in S_1$.

In other words, g_i is the sum over edges incident on the vertex, counting the weights of those edges which connect it to another vertex in the same sub-set as negative and those which leave the sub-set as positive. With this definition of g_i we see that, if vertex v_i is is moved from its current sub-set to the other sub-set, then the new total cut edge weight is $|E_{cut}|_e - g_i$.

Now, given that the partition is balanced to start with, load balance may be maintained for a graph with uniform vertex weights, $w_v(v_i) = Const.\ \forall i$, by simply swapping pairs of vertices. One way to proceed would be to swap a pair with positive gains, update the gains for the neighbours of the pair, then swap another such pair, and so on. This is essentially a gradient descent procedure, and very soon would become trapped in a local optimum where there are no more vertices with positive gains to be moved.

While this will have produced some improvement in the partition, it is likely that we would be able to produce a greater improvement if we considered some moves with negative gains along the way, so long as this yields a net benefit overall. This is precisely what KL does and is the heart of its strength. The way it does so is as shown in the pseudocode of figure 6.35.

We see from the figure that the algorithm consists of two nested loops. A single iteration of the outer loop we term a *pass*. In preferential order, a pass moves pairs of vertices in turn, regarding all those already moved as 'taboo,' until there are no more pairs to be moved.

The outer loop applies successive passes to the partition, each time using the best configuration found on the previous pass as the starting point for the next.

There are several things to note about line 6 of the pseudocode, where the next

```
Pseudo: Kernighan and Lin
 1. M_best = M_current = initial partition.
 2. Repeat
 3.    Compute g_i for M_current.
 4.    Mark all vertices as 'unmoved.'
 5.    While An unmoved pair of vertices remains.
 6.       Choose unmoved v_i ∈ S_0 and v_j ∈ S_1,
           such that g_i + g_j − 2w_e(e_ij) is at a maximum.
 7.       Move v_i to S_1 and v_j to S_0.
 8.       Update gains for neighbours of v_i and v_j.
 9.       If |E_cut|_e for M_current < |E_cut|_e for M_best Then
10.          M_best = M_current
11.       EndIf
12.    EndWhile
13.    M_current = M_best
14. Until No better partition found.
EndPseudo
```

Figure 6.35: The Kernighan and Lin algorithm.

pair of vertices to be moved is determined.

Firstly, we note that when a pair is interchanged the decrease in $|E_{cut}|_e$ is not $g_i + g_j$, but $g_i + g_j - 2w_e(e_{ij})$ if there is an edge between v_i and v_j, as illustrated in figure 6.36.

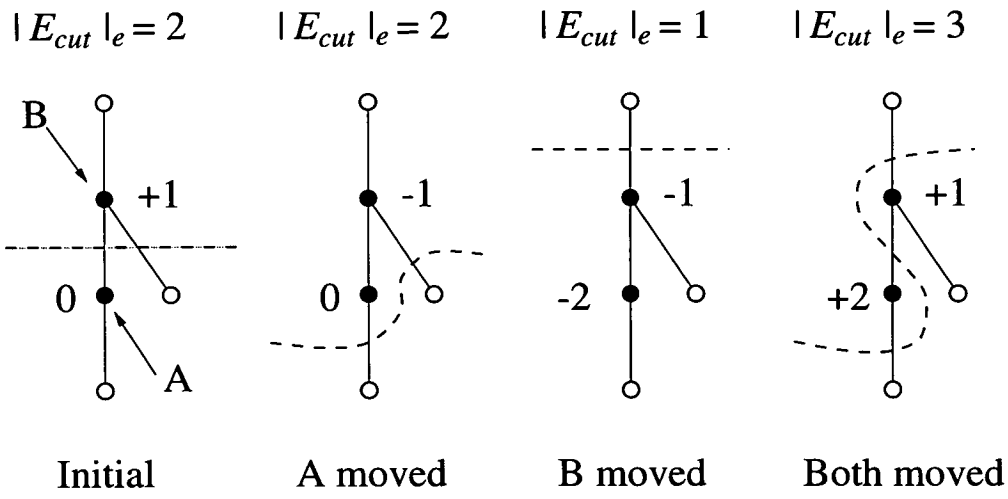


Figure 6.36: KL gains for a simple graph with unit edge weights. The dotted line indicates the bisection boundary.

Secondly, note that we specify the *maximum* of $g_i + g_j - 2w_e(e_{ij})$ which in no way precludes an increase in $|E_{cut}|_e$, as the maximum may well be negative. While there are pairs whose interchange produces a net reduction in $|E_{cut}|_e$ they will be chosen, but if there are no such pairs then the least damaging moves will be made.

Thirdly, and of key importance to the run time of the algorithm, is the question of exactly how the most favourable (least unfavourable) pair is found. Kernighan and Lin propose the following options:

1. Examine all possible combinations of pairs of vertices and choose the most favourable. This makes each pass an $O(n_v^{3/2} 4_v^n)$ procedure.

2. Sort the vertices in each half of the bisection on the key g_i. If matching pairs from each sorted set of vertices are considered in descending order of g_i, then a cut off point, past which it can be shown that the most favourable pair will not be found, may be established. Only combinations of vertices found before this cut off point need then be considered. If this set is small (experience indicates it is) and the sorting is carried out in $O(n_v \log(n_v))$ time, each pass is therefore an $O(n_v^2 \log(n_v))$ procedure.

3. Scan through the g_i and choose the the pair with maximum *individual* gain. This is equivalent to maximising $g_i + g_j$, but will be a reasonable approximation to maximising $g_i + g_j - 2w_e(e_{ij})$ if the probability that $\exists e_{ij}$ is small. A simple extension is to scan for two or three vertices from each half of the bisection with largest individual gain and choose the most favourable pair from this small set. In either case, the selection may be carried out in linear time, so that each pass is an $O(n_v^2)$ procedure.

Of these options, the exponential time of 1 clearly rules it out a a viable mechanism. Whether 2 or 3 is favourable depend largely on the connectivity of the graph.

Kernighan and Lin were considering arbitrary graphs in [KL70], but for dual graphs derived from unstructured meshes we can be sure that connections will occur only between vertices representing geometrically local mesh elements. The number of edges incident on a given vertex will therefore be bounded by some constant $n_e^{max} \ll (n_v - 1)$, and the probability that there will be an edge between an arbitrary pair of vertices will be small. This would seem to indicate that 3 is a reasonable choice for unstructured meshes. However, the fact that the vertices with highest gain will most likely be on or near the bisection boundary is also a factor and will increase the likelihood of a connection between the two vertices with highest individual gain.

In practice, maximising $g_i + g_j$ rather than $g_i + g_j - 2w_e(e_{ij})$ proves quite acceptable for unstructured mesh dual graphs. Moreover, an important extension of KL

due to Fiduccia and Mattheyses [FM82] allows this to be done in constant time, making a pass an $O(n_v)$ procedure. The Fiduccia and Mattheyses implementation (FM) provides by far the best performance and is almost always employed. As this implementation is the one used by PUL-md, we will defer discussion of the differences between it and the original KL algorithm until section 7.5.2.

This discussion has focused on line 6 of the pseudocode as the dominant factor in the runtime of a pass, and implied that this is also the dominant factor for the algorithm as a whole. This will only be the case if no other part of the algorithm has worse time complexity. We will now show this to be the case.

Of the other actions involved in a pass, the interchange of vertices (line 7) and keeping track of the best partition so far encountered (lines 9-11) only contribute a constant factor per iteration of the inner loop. Updating the gains (line 8) deserves some discussion, however.

Clearly, only the gains of the neighbours of each moved vertex are affected by the move and, as there are at most n_e^{max} of these per vertex, this too only contributes a constant factor. If v_j is a neighbour of a moved vertex v_i, then the gain of the neighbour may be updated according to

$$g_j \mathrel{+}= \begin{cases} +2w_e(e_{ij}) & \text{if } M_2(v_i) \neq M_2(v_j) \\ -2w_e(e_{ij}) & \text{if } M_2(v_i) = M_2(v_j), \end{cases} \tag{6.13}$$

where $M_2(v_i)$ indicates the *new location* of v_i.

Turning to our attention to the outer loop, we see that the computation of the gains (line 3) is the only expensive operation. For a densely connected graph this is an $O(n_v^2)$ operation, but given our observations regarding the connectivity of the graphs we are likely to encounter in mesh decomposition, we can expect this to be $O(n_v)$.

The question now arises as to how many iterations of the outer loops there are to be, that is; how many passes will the algorithm make? The answer is that it generally requires very few, typically 5 or 6 at most. Kernighan and Lin did not find any strong dependence of the number of passes on n_v, although they did not look at graphs as large as might be found in present day unstructured mesh applications, but more recent observations support this [HL93a].

To summarise, there are two significant costs associated with the KL algorithm; initial calculation of the gain values and the cost of a subsequent pass, the latter

being most strongly influenced by the cost of selecting a favourable pair of vertices to swap.

The progress of the KL algorithm is illustrated in figure 6.37. This figure shows $|E_{cut}|_e$ through the course of the algorithm; the horizontal axis indicates vertex moves made by KL, with the start of each new pass shifted to line up with M_{best} as found by the previous pass. The dark line shows actual changes made to the partition, while the lighter lines show explorations made by the algorithm that did not yield a better configuration.

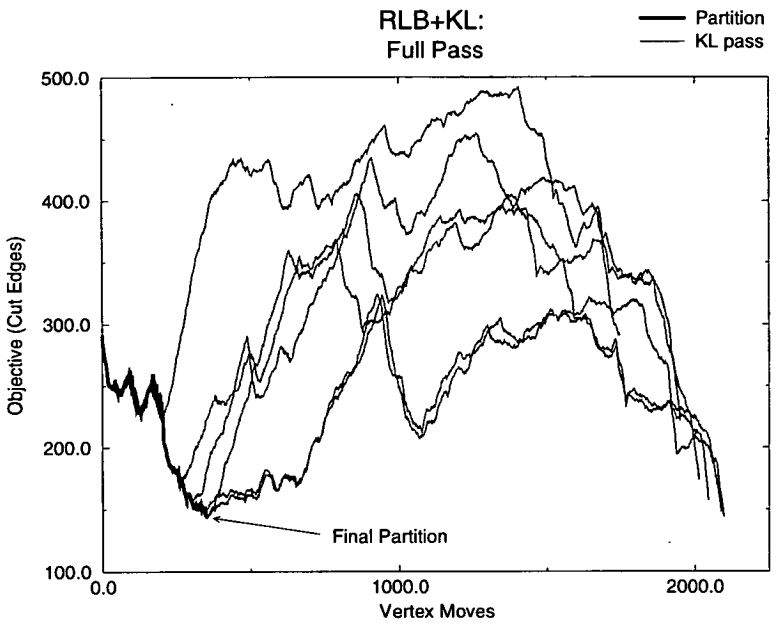


Figure 6.37: Progress of the KL algorithm from an initial layered (RLB) bisection.

The initial partition used in figure 6.37 was provided by a layered (RLB, without Cuthill-McKee) bisection of the Widget data set. This partition before and after application of KL refinement is shown in figures 6.38 and 6.39, where the improvement in partition quality is clearly visible. In this case cut edges were reduced by approximately 50% but it should be noted that the initial partition was rather poor.

In general KL is most profitably employed in combination with a fast initial decomposition algorithm that gives a starting state which is reasonable overall but may be poor in terms of local detail. When used with a random initial partition it is found to give erratic results little better than RCB [MO95] and taking far longer.

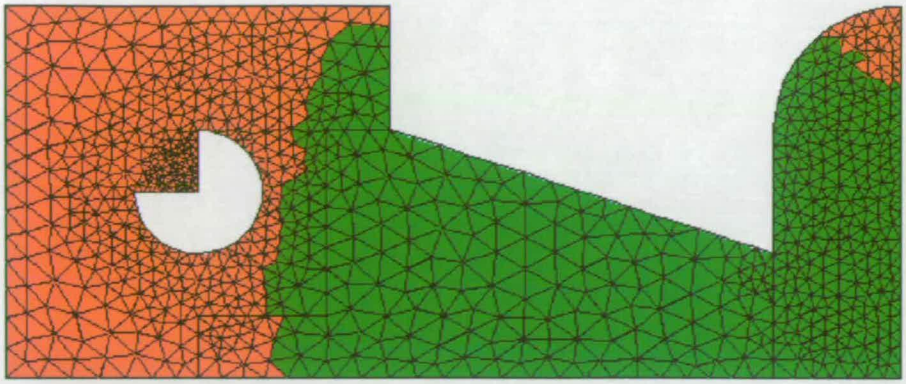


Figure 6.38: The initial layered (RLB) bisection of Widget.

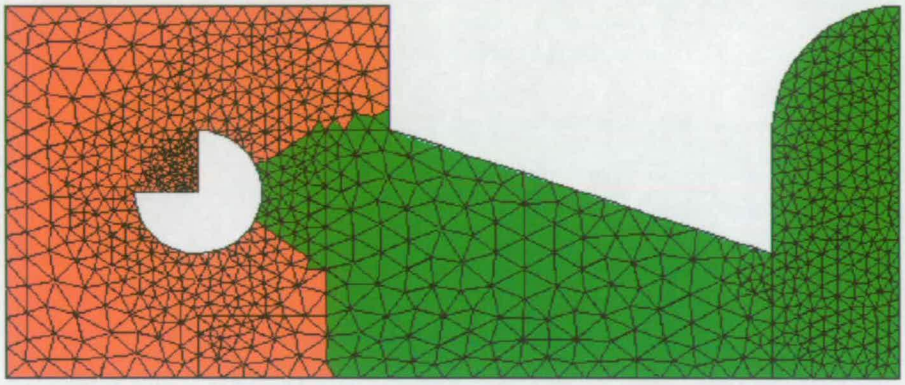


Figure 6.39: The bisection of Widget after KL refinement.

In combination with RIB, refinement with KL may produce a final partition of equivalent quality to RSB and may (depending on the relative efficiency of implementations) be as fast or even faster. Of course, KL may be used with RSB just as easily, resulting in a partition of higher quality still, but at the cost of increased runtime. KL is also employed as the local search heuristic in chained local optimisation [MO94, MO95], as was mentioned in section 6.4.7.3.

6.8.1.2 Extensions to the Basic KL Algorithm

There are several extensions to the basic KL algorithm and we will now review the most important of these.

The basic algorithm we have just described does not take into account non-uniform vertex weights, neither will it improve the load balance of an unbalanced initial partition. Kernighan and Lin [KL70] suggest that integer vertex weights might be incorporated by representing a vertex with $w_v(v_i) > 1$ as a fictitious cluster of $w_v(v_i)$ vertices of unit weight bound together with edges of very high

weight to ensure that the algorithm does not separate them. This is really a redefinition of the problem rather than a change to the algorithm.

A simpler approach, proposed by Hendrickson and Leyland [HL93a], is to no longer swap pairs of vertices, but rather move one vertex at a time, only considering moves from a sub-set with greater than average total vertex weight to those that are at or below average size to be valid. They also extend KL for higher partitioning dimensions ($l > 2$) which motivates this definition of a valid move; for bisection this definition is equivalent to simply moving vertices from the largest sub-set to the smallest, after all.

The extension of the algorithm in [HL93a] to higher partitioning dimensions is also distinct from that originally proposed in [KL70], which suggested that the standard algorithm for bisection be successively applied in a pairwise manner (between selected pairs of the k sub-domains, that is) until no further improvement occurs. Hendrickson and Leyland take the distinct approach of altering the selection and movement of a vertex in the innermost loop of KL.

If the algorithm is refining an initial partition into l sub-sets then for the vertices in a given sub-set there are $l - 1$ other sub-sets to which those vertices may be moved. Consequently, rather than associate a single gain with each vertex, they consider $l - 1$ gains per vertex and choose the most favourable of the vertices based on all these gains, subject to the move being valid as previously defined.

If this approach is implemented as an extension to the FM version of KL, then the resulting algorithm has $O(l(l-1)n_v)$ time complexity for move selection over a single pass, as there are $l(l-1)$ types of moves to be considered. Further, the memory required for the gains is increased to $O((l-1)n_v)$, compared to $O(n_v)$ for the basic algorithm. Both these considerations make this approach uneconomic for use with $l = k$ when there are many sub-domains, but neither make the time and memory costs of the multi-dimensional algorithm prohibitive for small l, say $l \leq 8$. The extended algorithm may then be used in a recursive manner, for instance in conjunction with multi-dimensional spectral partitioning, and may also take into account network distance by biasing the vertex gains by some inter-set cost metric, say hypercube hops.

Other extensions to the basic algorithm include early termination of a pass when further improvement seems unlikely (a glance at figure 6.37 is sufficient to show that most of the work of a full pass is wasted), and the addition of a certain amount of randomness to move selection which may allow the algorithm to escape

from a local minimum it might otherwise become trapped in. These extensions will be discussed in section 7.5.2, when we come to look at the implementation of KL in PUL-md.

The final extension we will discuss is an optimisation to avoid computing the gains for all of the vertices in the graph. If KL is working with a reasonable initial partition then it is likely that most of the moves it will actually make will be of vertices in the vicinity of the initial sub-set boundaries. If these vertices can be identified then only their gains need be computed, with a corresponding increase in performance and decrease in memory requirements. We will later see how this identification may be made in conjunction with a separator field (again in section 7.5.2); as part of a multi-level scheme in section 6.9.2 of this chapter; and also in section 6.8.3, when we come to discuss the Jostle refinement algorithm.

6.8.2 Mob

The Mob [SW91] algorithm seeks to refine an initial bisection in a way which resembles KL in many respects.

It is based on a similar process of swapping vertices between the two halves of the bisection in order to maintain load balance while reducing cut edges. The algorithm chooses which vertices to swap based on exactly the same gain function that KL uses, but is distinct in that it does not use the exchange of a pair of vertices as its basic change of state, but rather exchanges whole groups, or *mobs*, of vertices at once.

The way it does this is as shown in the pseudocode of figure 6.40. As with the description of basic KL, it is assumed that the partition is already balanced, and that vertex weights are uniform.

The mob schedule is of fundamental importance to the algorithm; it is a monotonically decreasing sequence of $len^{schedule}$ integers drawn from the range $[1, n_v/2]$. The first of these, $MOB_1^{schedule}$, is typically given as some moderately large percentage of $n_v/2$, say 10%, and the last as $MOB_{len^{schedule}}^{schedule} = 1$, with the intermediate values linearly decreasing in between, as suggested by the algorithm originators in [SW91].

The algorithm uses the mob schedule to specify the size of the two sets of vertices to be swapped between the two halves of the bisection, namely MOB_0 and MOB_1

Pseudo: MOB

1. Create the mob schedule, $MOB_s^{schedule}$, where $s = 1, len^{schedule}$.
2. **For** required number of iterations
3. $s = 1$
4. **For** $t = 1$ to $len^{schedule}$
5. $e = |E_{cut}|_e$
6. $MOB_0 = chooseMOB(MOB_s^{schedule}, S_0)$
7. $MOB_1 = chooseMOB(MOB_s^{schedule}, S_1)$
8. Move MOB_0 to S_1 and MOB_1 to S_0.
9. Update gains.
10. **If** $|E_{cut}|_e > e$ **Then**
11. s += 1
12. **EndIf**
13. **EndFor**
14. **EndFor**

EndPseudo

Function: $chooseMOB(size, S)$

1. Find maximal g_{min} such that $|\{v_i \in V : g_i \geq g_{min}\}| \geq size$
2. Choose the Pre-Mob, $MOB^{pre} = \{v_i \in V : g_i \geq g_{min}\}$.
3. Choose a random sub-set, MOB, from MOB^{pre} such that $|MOB| = size$.
4. **Return** MOB

EndFunction

Figure 6.40: The Mob algorithm.

in the pseudocode (see lines 6 and 7 of the main pseudocode). The vertices that make up these mobs are chosen on the basis of their gain.

First a Pre-Mob is chosen from each half, consisting of all those vertices with gain above a value, g_{min} (line 2 of function $chooseMOB$), where g_{min} is just low enough that a mob of the specified size may be chosen from the resulting Pre-Mob (line 1 of function $chooseMOB$). Once the Pre-Mob is determined, the actual mob is chosen from it as a random sub-set of size $MOB_s^{schedule}$, where s is the current index into the schedule. This index is incremented only when the algorithm fails to find a better partition with the current mob size (line 10 in the main pseudocode).

Thus the algorithm repeatedly swaps mobs of vertices that are a randomised approximation of the best candidates for moving, decreasing the size of mob whenever this fails to produce an improvement. As $|MOB_0| = |MOB_1| = MOB_s^{schedule}$ for each swap, load balance is unaltered by the algorithm.

The selection of mobs has $O(n_v)$ time complexity, as a loop through all vertices

in the partition is required to determine the Pre-Mobs, but no sorting of these is required due to the random nature of choosing vertices from Pre-Mobs to form the actual mob. This makes the algorithm as a whole $O(n_v)$ if the number of swaps is limited by some constant, although the real cost of each swap is rather high.

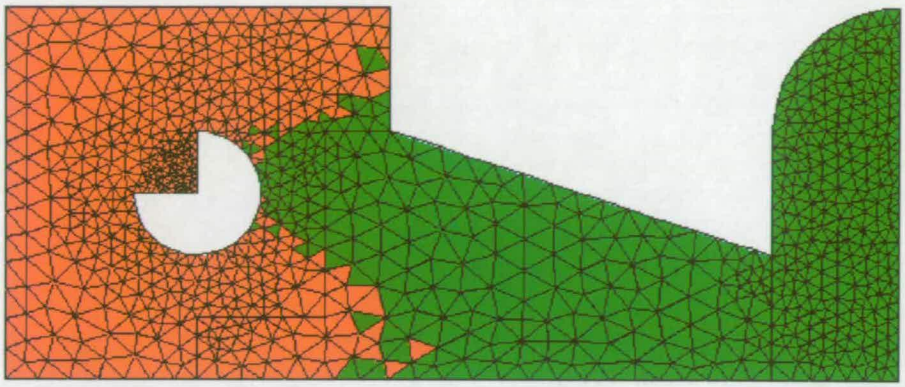


Figure 6.41: The bisection of Widget after Mob refinement.

The effect of Mob refinement is illustrated in figure 6.41, where the initial partition was the same as used to demonstrate the application of KL (figure 6.38). Here 50 iterations of the outer loop in the pseudocode of figure 6.40 were applied, with $len^{schedule} = 40$ and the schedule starting at 10% of sub-set size and linearly decreasing, as previously discussed[9]. It can be seen that the overall shape of the sub-set boundary has been improved and in particular the disconnected set in the top right of figure 6.38 has been removed. However, the fine detail of the border remains poor and in fact Mob has produced no improvement in $|E_{cut}|_e$ for this case.

In general Mob is found to be rather sensitive to the particular settings of the various parameters that control its behaviour. Also, as the algorithm does not keep track of the best partition it has encountered in the way KL does, there is nothing to prevent a worse partition than was initially provided resulting from Mob refinement[10]. We will examine this behaviour in more detail in chapter 8.

[9]Additional PUL-md features were used here to remove isolated individual vertices, and to inhibit the swapping of vertices not on sub-set borders; the decomposition is worse without use of these features (see section 7.5.1).

[10]In the course of writing, correspondence with the authors of the Mob algorithm has made clear that their implementation differs slightly from that described here. Firstly, the best partition found *is* recorded and returned by the algorithm, so that a worse partition never results. Secondly, the loop structure differs in such a way that a specified number of mobs are exchanged and, as this number is generally much larger then the schedule length, the end of the schedule is passed through at least once (probably several times). Hence, although the

The algorithm as presented here and in [SW91] was subsequently extended to take into account the mapping of sub-domains to processors and the topology of the processor network in [SW93]. As network topology may also be represented as a graph, the problem is referred to as *graph embedding* in [SW93] (the dual graph being *embedded* in the network graph). The basic structure of the algorithm remains unaltered, but the selection of vertices to form the Pre-Mobs is now restricted in order to take into account the network topology. Embeddings for 2-dimensional grids (*grid embedding*) and hypercube networks (*hypercube embedding*) are considered, with the emphasis on VLSI design rather than mesh partitioning.

The latter publication also contains the following intuitive explanation of the algorithm, which provides a interesting interpretation of the function of reducing the mob size:

> *"An analogy can be drawn between the movement of mobs and the rolling of a ball in a solution space where the goal is to find a global minimum. If the space is structured properly, a large ball rolling downhill will quickly find the region containing the global minimum but, due to its large size, will not find the global minimum itself. Replacing the ball with a ball of smaller size allows the algorithm to get closer to the global minimum."*

6.8.3 Jostle

The Jostle algorithm [WCE95b, WCE95a] works directly with an existing k-way graph partition, and seeks to compact the overall shape of each sub-domain as well as improving the fine detail of the sub-domain boundaries. It attempts to do this in a local manner, as it it designed with parallel implementation in mind (see section 6.10). The analogy the algorithm's authors use to justify that a local method may still approach a global optimum is that of the regularity of formation of bubbles in a foam. Each bubble seeks to minimise its surface energy without any global knowledge, except through contact with its immediate neighbours, but the resulting regular pattern has very low energy overall.

Jostle is a three stage process, and we will focus particularly on the first stage

partition generated after the last mob exchange is essentially arbitrary, as the algorithm does not necessarily terminate at the end of the schedule, an improved partition results. In many ways the 'Mob Complete' algorithm detailed in the pseudocode of figure 7.3 in section 7.5.1 is closer to the spirit of the original authors' implementation.

where the most major changes are made to the partition. These stages, in order of application, are:

- The *sub-domain heuristic* which attempts to compact the overall shape of each sub-domain, but which may result in some imbalance in sub-domain size.
- The *load balancing heuristic* to adjust any imbalance resulting from the sub-domain heuristic or already present in the initial decomposition.
- The *localised refinement heuristic* which attempts to tidy up the fine detail of the result of the previous stages.

All of these stages migrate *border vertices* between *adjacent* sub-domains only, and all make use of a gain function to do so, although the gain used by the sub-domain heuristic differs from that used in the subsequent two stages.

6.8.3.1 Sub-Domain Heuristic

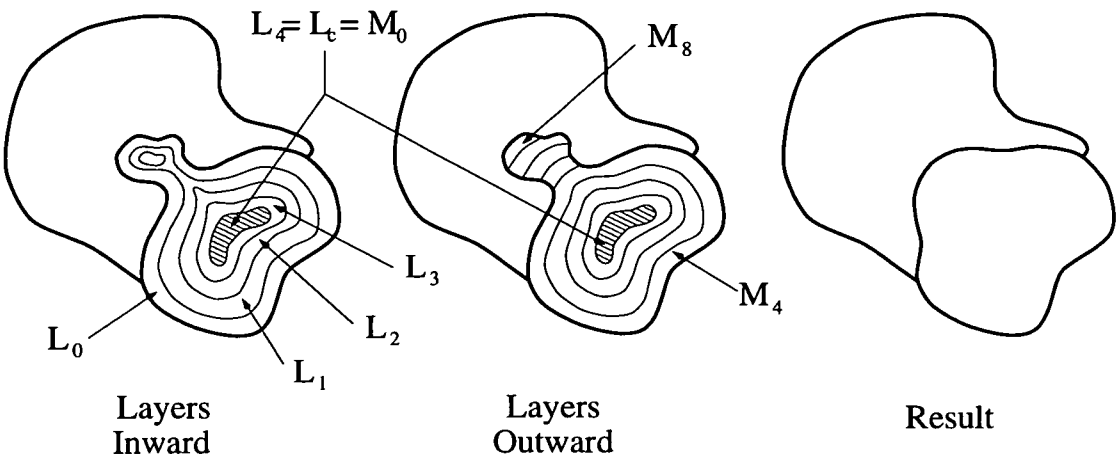


Figure 6.42: Progress of the Jostle sub-domain heuristic.

The sub-domain heuristic explores the level structure within each sub-domain in order to find its notional 'centre,' as shown in figure 6.42. To do this it starts with the boundary of the sub-domain, L_0, and then proceeds to find successive level sets *inwards* from the boundary. These level sets are found in exactly the same way we have already encountered in the the context of Cuthill-McKee bandwidth reduction, the greedy algorithm (when it is implemented on the dual graph of a mesh) and layered partitioning; it may therefore be useful to look again at figure

6.5, the only difference here is that L_0 is now the set of vertices on the boundary, rather than an individual seed vertex alone.

As successive level sets are found inward, a point will be reached where the next next level set is empty (i.e. the whole sub-domain has been explored) and the set immediately prior to this is taken to be the centre set, as shown in the left-most diagram of figure 6.42. We denote the centre set with index c. This set may not be connected, but this does not impede the following steps.

Having found the centre set, L_c, the process is reversed taking L_c as M_0, the first level set in an *outward* expansion. The level set, M_l, in which a particular vertex resides may now be used as a metric of distance relative to the sub-domain centre. This is then used to define a distance, d_i, relative to some 'ideal' sub-domain border, so that for a vertex $v_i \in M_l$ we define $d_i \equiv c - l$.

As vertices not in any M_l will be in regions disconnected from the main part of the sub-domain these are marked as such for later migration to other sub-domains.

The distance d_i may now be used to define a gain function such that those vertices furthest from the centre, for example those in the set M_8 in the middle diagram, are preferentially transferred to a better location. However, it would be inefficient to do this if the later load balancing and localised refinement stages undo the changes made by the sub-domain heuristic, and so some additional information is encoded in the gain to avoid this.

An empirical formulation which embodies the amount of vertex weight a sub-domain, S_p, can hope to gain or lose according to the size of its border and the average connectivity of the graph is given by

$$\delta_p = 20\frac{(|V|_v/k - |S_p|_v)}{|L_0|_v} \times \frac{|V|_v}{|E|_e}.$$

This is used to give an adjusted distance

$$\phi_i = d_i + \delta_p,$$

and thus to define the gains for a vertex on the sub-domain border, $v_i \in L_0 \subset S_p$, as

$$g_i^p = \phi_i + \sum_{\{v_j \in S_p : \exists e_{ij}\}} \phi_j + \theta|e_{ij}|_e$$

$$g_i^q = \sum_{\{v_j \in S_q : \exists e_{ij}\}} \phi_j + \theta |e_{ij}|_e \quad p \neq q$$

Here θ indicates the relative importance of distance and edge weight, and is generally incremented through the course of the run of the heuristic.

We now have a gain associated with leaving a border vertex where it is, g_i^p, and a gain for moving to any *adjacent* sub-domain q, g_i^q.

Where the gains indicate that it is beneficial to do so, vertices are moved from the sub-domain border to an adjacent sub-domain determined by whichever destination, S_q, yields the maximum g_i^q. This destination is termed the *preference*. This process is applied iteratively to the sub-domain borders, which are the only regions for which gains are calculated. A figurative representation of the result is shown in the right-most diagram of figure 6.42.

Additionally, disconnected sets are migrated to the sub-domains for which they have preference in their entirety (the only time internal vertices are moved).

6.8.3.2 Load Balancing Heuristic

Having applied the sub-domain heuristic the definition of gain is now changed to be the reduction in cut edges given by equation 6.12, just as for KL and Mob. This is then used to choose which vertices should be moved in order to re-establish load balance. The gain is calculated for any adjacent sub-domain (not *all* other sub-domains, as it is in the extended implementations of KL we previously discussed), and the preference defined as before.

Firstly, the number of nodes to be migrated is determined. In [WCE95b] an approximate transfer policy algorithm due to Song [Son94] is used, although a diffusion type scheme [Cyb89] could just as easily be employed, and indeed is used by the originators of Jostle in more recent work focusing on dynamic partitioning [WCE97].

Once this migration schedule has been determined, border vertices are sorted by gain and the load specified by the schedule moved to the sub-domain for which they have preference (never the current sub-domain). This process is iterated until the migration schedule is satisfied.

6.8.3.3 Localised Refinement Heuristic

This final stage attempts to minimise cut-edges while maintaining load balance, in a manner inspired by KL.

Gains are only calculated for vertices on the sub-domain borders and those that come into the border during the course of this refinement stage, a process very similar to the *lazy evaluation* of gains which we will encounter in section 6.9.2. Again migration only occurs between adjacent sub-domains, this time in a pair-wise exchange of vertices.

Each sub-domain considers the interfaces it has with its neighbours, and builds a list for each interface of local vertices whose preference is the neighbouring sub-domain at that interface. These lists are then sorted by gain and each interface refined in turn. First, the sub-domain on one side of the interface migrates the vertex at the head of its sorted list to the sub-domain on the other side, then the other sub-domain reciprocates by migrating the vertex at the head of its list in return. This process continues while the sum of the gains of the two nodes selected for exchange is positive.

6.8.4 Summary

The combination of the three stages of the algorithm yield respectively:

- Good overall sub-domain shape and internal connectivity, leading to reduced sub-domain borders and hopefully reduced cut edges.
- Good load balance, although not exact as the migration schedule is only approximate.
- Further reduced cut edges, although the exchange of vertices may lead to some small load imbalance for weighted graphs.

In the next section we will examine multilevel schemes and see that Jostle may be implemented in that manner. When this is done, it is found to be at least on a par with the other sophisticated multilevel algorithms we shall encounter. Later, in section 6.10, we shall see how it is also very amenable to parallel implementation. Overall, Jostle appears to be one of the more competitive decomposition algorithms available, particularly if a small (generally very small) load imbalance is acceptable.

6.9 Multilevel and Hybrid Variants

In this section we will look at how a multilevel approach not only accelerates the decomposition process, but also allows the development of hybrid techniques that may be more powerful than the sum of their parts.

The multilevel approach will be familiar to anyone acquainted with multigrid acceleration of CFD and related simulations [BMT96]. Here several levels of grid are employed, each of a different resolution, those with lower resolution generally being a coarse approximation to the finest grid. This allows the propagation of information across the simulation domain to occur at a variety of length scales, which accelerates convergence and may add numerical stability.

Typically a multigrid algorithm operates in a cycle, for example a 'V' cycle of fine-coarse-fine, or 'W' indicating fine-intermediate-coarse-intermediate-coarse-intermediate-fine. Data is interpolated between levels by *restriction* when moving from fine to coarse, and by *prolongation* when moving from coarse to fine.

The similar multilevel approach to decomposition involves the construction of a sequence of graphs, each a coarser approximation to the previous level, until the problem is reduced to a manageable size; we call this *graph contraction*, although it is also referred to as *reduction*. This smaller graph is then partitioned and the resulting decomposition interpolated back through the levels in a process analogous to multigrid prolongation.

We will now look at some of the ways in which this may be done, and then examine the algorithms that make use of this approach.

6.9.1 Graph Contraction

There are two commonly used techniques for graph contraction, with little to choose between them. The main distinction is in how the technique effects interpolation of the decomposition, rather than any more fundamental interaction with the decomposition algorithm. Thus, we will first look at these two techniques in isolation, then see how they are actually employed.

Both of these techniques reduce the graph to a smaller graph which retains the characteristics of the larger in some sense. The sequence of graphs in the multilevel scheme is simply constructed by iteratively applying whichever contraction

technique is desired. If the contraction is given by a mapping $C(G)$ such that $|C(G)|_v < |G|_v$, then the sequence of graphs $G^0, G^1, \ldots, G^{n_g}$ is defined by $G^0 = G$ and $G^{l+1} = C(G^l)$. Clearly, $|G^0|_v > |G^1|_v > \ldots > |G^{n_g}|_v$.

6.9.1.1 Edge Contraction

Edge contraction was first proposed in the context of multilevel Kernighan and Lin partitioning [HL93b]. The principal operation involved is the reduction of the two graph vertices that make up an edge to form a single vertex in the contracted graph.

If e_{ij} is contracted, the weight of the new vertex, v_c, formed from its components is simply $w_v(v_c) \equiv w_v(v_i) + w_v(v_j)$. Similarly, if there is a vertex, v_k, such that $\exists e_{ik}$ and $\exists e_{jk}$ then the weights of those two edges are combined so that $w_e(e_{ck}) \equiv w_e(e_{ik}) + w_e(e_{jk})$.

It is desirable that the edges selected for contraction be well dispersed through the graph and so a *maximal matching* set, E^{max}, is chosen from E. This consists of a maximal set of edges such that no two are incident on the same vertex, so that $e_{ij} \in E^{max} \Rightarrow e_{im} \notin E^{max}$ and $e_{nj} \notin E^{max}$. Such a set is not unique, but may be easily generated in a simple randomised manner. In general, there will always be vertices not members of an edge in the maximal matching, $\{v_i \in V : e_{ij} \notin E^{max}\}$, and edges that are also unaffected, $\{e_{ij} \in E : e_{ij} \notin E^{max}\}$. These vertices and edges are inherited unaltered by the contacted graph.

The technique has the useful property that there is a direct correspondence between a good partition of the coarsest graph in the sequence of levels and a good partition of the finest, initial graph if a simple method of interpolation of sub-domains is employed. This is because edge and vertex weights are preserved by the contraction process.

Consider a vertex, $v_c \in G^{l+1}$, derived from a contracted edge, $e_{ij} \in G^l$. A sub-domain, S_p, of a partition may simply be projected back through the levels so that $v_c \in S_p^{l+1} \Rightarrow v_i, v_j \in S_p^l$, where the superscript on S_p indicates the graph level to which it applies. Likewise, if a vertex in S_p^{l+1} was not derived from a contracted edge in G^l, then its assignment to the sub-domain is projected unaltered, so that it is taken to be in S_p^l also.

We will call this method of interpolation *direct projection.* If the sub-domain assignment of vertices is inherited by direct projection, then the load balance

and total cut edge weight are the same for all levels of graph.

6.9.1.2 Vertex Clustering

In the same way that we seek a well dispersed set of edges for edge contraction, we seek a well dispersed set of vertices for this clustering technique.

The technique, as presented in [BS92], proceeds by taking a *maximal independent set*, V_{max}, of vertices from V to be this well dispersed set. A set of vertices is said to be *independent* if there are no edges connecting vertices in the set, that is $v_i \in V_{max}$ and $e_{ij} \in E \Rightarrow v_j \notin V_{max}$. Further, the set is *maximal* if the inclusion of any additional vertex would render it no longer independent.

The vertices in the maximal independent set are taken as vertices in the contracted graph. The connectivity of the contracted graph is determined by growing small domains, or *clusters*, out in level sets from the vertices in V_{max} and adding an edge wherever these clusters intersect.

The resulting clusters will be quite small and not dissimilar to the sub-domains that would result from applying the greedy algorithm for very high k, except that here the clusters will always be internally connected and of varying sizes.

As it appears in [BS92], no account is taken of edge or vertex weight, but the technique is clearly amenable to extension to include these features. This has been done in various subsequent implementations, for example [WCE95b] and [DR95]. Taking a vertex in the contracted graph G^{l+1} to represent all the vertices in the cluster in G^l from which is was derived, a similar method of direct projection to that described for edge contraction may be employed to interpolate a partition back through the levels. If vertex and edge weights are accounted for also, the same comments made regarding the preservation of the quality of decomposition by direct projection still apply. However, this contraction technique was developed in the context of spectral partitioning and there a less intuitive method of interpolation is employed, as we shall see in section 6.9.3.

6.9.2 Multilevel Kernighan and Lin Partitioning

Having described graph contraction it is now straight-forward to describe multilevel Kernighan and Lin partitioning. We follow [HL93b], where edge contraction is used and full account is taken of vertex and edge weights. Although a similar

algorithm is described in [BJ93, Jon92], it does not account for these important features of the contracted graphs.

Firstly, the sequence of contracted graphs, $G^0, G^1, \ldots, G^{n_g}$, is constructed from the initial graph to be partitioned. The coarsest of these, G^{n_g}, is then partitioned and the algorithm proceeds as follows:

- Interpolate from G^{l+1} to G^l by direct projection.
- Refine the partition of G^l with the k-way implementation of KL described in section 6.8.1.2.

As n_g is typically chosen to be large enough that $|G^{n_g}|$ is of the order of a few hundred vertices, the cost of using a very sophisticated algorithm to partition G^{n_g} is not a consideration. Therefore in [HL93b] multidimensional spectral partitioning, as described in section 6.7.2.4 and references [HL93a, HL92], is used. This is also followed by additional KL refinement before the recursive steps back through the levels described above are begun.

While it is obviously helpful to use a sophisticated algorithm to partition G^{n_g}, it is found not to be critical to the quality of the resulting final partition of G_0 [HL95]. This is because the significant operation in the algorithm is not the partitioning of G^{n_g}, but rather the subsequent refinement of the intermediate G^l. As KL is applied to every intermediate G^l, and the vertices in intermediate G^l represent a larger and larger number of vertices in the original G^0 as l increases, the refinement is operating on a correspondingly varying length scale at each level; it is from this that the algorithm derives its success.

In practice KL is applied only periodically, say every couple of levels, to increase the speed of the algorithm. A further optimisation is to make use of the knowledge of the locations of sub-domain borders that may be extracted during the interpolation of the partition from level to level at no extra cost. If the borders are known then the KL gains and associated data structures need only be set up initially for border vertices.

Of course, as KL progresses, the sub-domain borders will change and vertices not on the initial borders will almost certainly also migrate. To deal with this a *lazy evaluation* scheme is used. Whenever a vertex is moved by KL, the gains of it neighbours are updated. If however a neighbour has not yet had its gain initialised, then storage is allocated and its gain calculated on an as-needed basis. This not only reduces the runtime of the algorithm, but also its memory

requirements. It also offsets the increased cost of using the k-way implementation of KL, where both runtime and memory costs are increased compared to the basic bisection refinement algorithm.

The resulting hybrid algorithm, which is termed *multilevel-KL*, is found to be not as fast as recursive inertial bisection followed by Kernighan and Lin refinement, but produces much better partitions for the reasons described above. In most cases it produces partitions at least as good, if not better, than recursive spectral bisection followed by Kernighan and Lin refinement. It is the fast runtime of the algorithm that makes it particularly attractive, as it is significantly faster than any spectral method, and so is particularly suited to partitioning very large graphs. If it has any disadvantage it is the memory required to store all of the intermediate G^l.

6.9.3 Multilevel Spectral Partitioning

Multilevel spectral partitioning was initially formulated using vertex clustering [BS92], but it has also been successfully implemented using edge contraction [HL95]. We will follow the original formulation to describe the algorithm here.

While multilevel-KL, makes use of the synergy between KL and the multilevel process, multilevel spectral partitioning is essentially only an acceleration of the eigensolution of the Laplacian required by the spectral method. As such, it is unlikely to ever yield a better quality of partition than standard spectral algorithm.

As before, the contracted sequence of graphs is constructed, but now G^{n_g} is not immediately partitioned. Rather, the Fiedler vector is calculated using the Lanczos method (the eigensolution method that was introduced in section 6.7.2, and that we shall examine in depth in section 7.4.8) and this vector interpolated back between levels. Once the Fiedler vector has been interpolated all the way back to G^0, that graph is bisected using the Fiedler vector as a separator field in the normal manner. The whole process (*including* graph contraction) is then repeated recursively on each half of the resulting bisection until the required number of sub-domains are generated.

It is in the interpolation of the Fiedler vector that the algorithm departs most significantly from methods we have already encountered.

For the purposes of this discussion we will denote the Fiedler vector, (which

was previously known as $\boldsymbol{e_2}$) for graph level l as $\boldsymbol{e}^l$, where we have omitted the subscript so that it will not be confused with the subscript indicating the scalar components of $\boldsymbol{e}^l$, e_i^l.

Now, for the vertex clustering method of graph contraction there is a one-to-one correspondence between the vertices in $V_{max}^l \subset G^l$ and vertices in G^{l+1}. Thus, if $v_j^{l+1} \in G^{l+1}$ corresponds to $v_i^l \in V_{max}^l$, then we may set those components of $\boldsymbol{e}^l$ accordingly, so that $e_i^l = e_j^{l+1}$. This step is termed *injection* and has only determined components of $\boldsymbol{e}^l$ for $v_i^l \in V_{max}^l$. The subsequent step, *averaging*, sets the remaining components relating to $v_i^l \notin V_{max}^l$. These remaining components are taken as the mean value of all the components relating to neighbouring vertices. As V_{max}^l is a maximal independent sub-set of G^l we know that all neighbours of $v_i^l \notin V_{max}^l$ will be in V_{max}^l and therefore will already have been set by injection.

Assuming $\boldsymbol{e}^{l+1}$ was a good approximation to the Fiedler vector for level $l+1$, by injection and averaging we now have a rough approximation for $\boldsymbol{e}^l$. This rough approximation may now be improved before it is interpolated on to the next finest level. As we can expect $\boldsymbol{e}^l$ to be close to the true Fiedler vector, Rayleigh quotient iteration (RQI) is well known to be a good choice as a method to improve upon this [GL89, Par92]. RQI has the property that, if its iterations start from a vector which is close to an eigenvector, then the components of the iterate in that direction are magnified, which results in the process converging to the nearby eigenvector.

The Rayleigh quotient of any arbitrary $\boldsymbol{x}$, for the matrix $\boldsymbol{A}$, is defined as

$$\rho(\boldsymbol{x}) \equiv \frac{\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}}{\boldsymbol{x}^T \boldsymbol{x}}.$$

For a symmetric matrix $\rho(\boldsymbol{x})$ may be shown to be the best estimate for an eigenvalue, λ, if $\boldsymbol{x}$ is considered to be an approximation to an eigenvector of $\boldsymbol{A}$. Further, inverse iteration theory tells us that if we solve $(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{b}$ for $\boldsymbol{x}$ then this will be a good approximation to the eigenvector.

RQI combines these two observations to yield the following algorithm:

$$\begin{aligned} \lambda_i &= \rho(\boldsymbol{x}_i) \\ \text{Solve } (\boldsymbol{A} - \lambda_i \boldsymbol{I})\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i \\ \boldsymbol{x}_{i+1} &= \boldsymbol{x}_{i+1}/|\boldsymbol{x}_{i+1}|_2, \end{aligned}$$

where it is assumed that $|\boldsymbol{x}_0|_2 = 1$.

RQI requires the solution of $\boldsymbol{A} - \lambda_i \boldsymbol{I}$ and, as in our application $\boldsymbol{A}$ will be the Laplacian of the graph at the current level, we need an efficient algorithm for solving large, sparse symmetric systems of equations. Unfortunately, $\boldsymbol{A} - \lambda_i \boldsymbol{I}$ may clearly be an indefinite system, so the conjugate gradient method can not be employed. Fortunately, a variant of CG known as SYMMLQ [PS74] has been developed for systems which are symmetric but indefinite, and which is well suited for large sparse systems too.

SYMMLQ derives the 'LQ' part of its name from the fact that it uses a LQ factorisation to solve a tridiagonal system, where the 'L' indicates a lower triangular factor, and the 'Q' an orthogonal factor. The normal CG algorithm effectively computes the Cholesky factorisation of a tridiagonal system. This will be poorly determined numerically if the tridiagonal is nearly singular, which is a situation that will arise if $\boldsymbol{A} - \lambda_i \boldsymbol{I}$ is close to indefinite. Altering CG to make use of LQ factorisation in place of Cholesky avoids this problem. While it is far from immediately obvious, and a detailed study is far outside the scope of this thesis, the CG and Lanczos methods are closely related. When we come to look in detail at Lanczos tridiagonalisation in section 7.4.8, the connection may become clearer to those already familiar with the derivation of CG; indeed, the similarity can be seen in the structure of the algorithms as described by the defining sets of equations, 5.2 and 7.3.

The steps we have described - injection, averaging, improving the Fiedler vector with RQI and using SYMMLQ to solve the resulting system of equations - defines the multilevel spectral algorithm. As we stated at the start of this section, this is a simply a method of accelerating eigensolution of the Laplacian, and so the question then arises as to how much faster is multilevel spectral compared to the usual Lanczos approach? In [BS92] an order of magnitude increase in performance is observed, while the implementation in [HL95] is found to be 'several times' faster than Lanczos with selective orthogonalisation (see section 7.4.8) which is one of the most efficient Lanczos variants. It avoids the large

memory requirements imposed by the use of Lanczos on the full graph, but offsets this against the memory requirements of the multi-level scheme itself. While the multilevel spectral algorithm is more prone to misconvergence than Lanczos, experience shows that eigenvectors other than the Fiedler vector may still result in good partitions.

6.9.4 Multilevel Jostle

The Jostle algorithm has been implemented using both vertex clustering [WCE95b] and edge contraction [WCE97]. In both cases, vertex and edge weights are accounted for and the algorithm is basically unchanged; it is therefore straightforward to describe its operation.

In [WCE95b], two variants are detailed. The first is 'JOSTLE/fast reduction,' which uses the full three-stage algorithm on G^{n_g} but only employs the load balancing and localised refinement heuristics on G^0. The second is 'JOSTLE/reduction,' which uses the sub-domain heuristic at both graph levels, although a limit of five iterations on G^0 is imposed. It is unclear whether either variant operates on the intermediate graph levels, or indeed by how much the graph is coarsened by each contraction, but the standard algorithm and the two multilevel variants all produce equivalent results in terms of partition quality. JOSTLE/reduction is found to be almost twice as fast as the standard algorithm, while JOSTLE/fast reduction is almost three times as fast. In comparison with the multilevel spectral algorithm, the results from the multilevel Jostle algorithms are superior and consistently several times faster at least. In particular, the runtimes for multilevel Jostle increase only very gradually with k, while those for multilevel spectral increase dramatically.

The edge contraction implementation in [WCE97] is studied primarily in the context of dynamic partitioning and so comparison is not so easy, but performance appears to be broadly similar.

6.10 Parallel Decomposition Algorithms and Dynamic Partitioning

Thus far we have detailed the available algorithms for mesh decomposition and graph partitioning, but have said little about their implementation. In the following chapter we will look at the implementation of those algorithms which are included in PUL-md, which is a serial utility, but first we will examine research in the area of parallel decomposition algorithms and how it relates to adaptive problems.

PUL-md is not alone in being a serial utility, as the same is true of most currently available decomposition packages; for example, Chaco [HL95], METIS[11] [KK95a], TOP/DOMDEC [FLS93], DDT [FR94] and Party [PD97] all fall into this category.

There are several reason for this, the most obvious being that parallel implementation is more challenging. Another is the success of the multilevel approach, which allows fairly large meshes to be handled on workstations of moderate specification in a reasonable time, but there are other considerations too, as we shall see.

Although the use of parallel platforms for unstructured mesh applications such as finite element or volume calculations is now commonplace, pre- and post-processing is generally still the province of serial workstations. Mesh generation is often part of a serial pre-processing stage, and so it is unsurprising that decomposition should often be viewed as part of this stage also. There are obvious drawbacks to this approach, in that the runtime of the the pre-processing stage may form an unacceptable bottle-neck and, moreover, that the memory required to store and pre-process the whole mesh may be greater than is available on the serial platform. Although the multilevel approach may go some way to alleviating these problems, it is of no help if even the initial mesh can not be accommodated.

These drawbacks apply to mesh generation as much as decomposition, and we would therefore like to be able to run *all* stages of processing in parallel if possible. While a discussion of parallel mesh generation is certainly inappropriate here, we will note in passing that it is an area of much current research and is far from

[11] In the course of writing, an alpha test parallel library has been released: ParMETIS Alpha 0.3 (May 1997) [KK97].

a solved problem; indeed, the same may be said of parallel partitioning.

Clearly, if the application in question uses a static mesh generated by a serial pre-processor we have the choice of partitioning either in serial or in parallel. If, however, the mesh was generated by a parallel pre-processor or the application is adaptive, where dynamic re-meshing changes the mesh at run-time, then parallel partitioning is the only reasonable option.

It is important to note the difference between parallel partitioning for a static mesh application, so called *static partitioning*, and for an adaptive one; *dynamic partitioning*. While the basic aims of static and dynamic partitioning are the same - to assign parts of the simulation to processors so that load balance is even and resulting communication is minimised - dynamic partitioning has other factors to take into account which make it a quite distinct, if related, problem. Static partitioning need only be carried out once, at the start of the calculation, while dynamic partitioning must be carried out several times during the course of solution. Thus dynamic partitioning forms an integral part of the overhead of using a parallel platform, in that it does not, in itself, contribute to the solution of the calculation. This tight coupling with the calculation also makes it harder to produce libraries of wide applicability, such as have been developed for static partitioning, for here it is much harder to abstract the problem away from the application.

A dynamic partitioning scheme must fulfil the following criteria:

- Above all, it must be fast and therefore:
  - Perform minimal changes to the partition.
  - Preserve locality of mesh elements as much as possible.
- It must be storage efficient.
- It must be able to correct any load imbalance that has arisen.
- The scheme must be able to evaluate whether it is worth re-partitioning after re-meshing at all.

The last point is a recognition that the cost of re-partitioning may out-weigh the benefit gained in terms of the increased performance of the subsequent calculation on the new mesh. While several of these points are just as valid for static partitioning, it is the need to take into account the existing partition and the increased emphasis on correcting load balance that are the most notable

differences. The former leads dynamic partitioning algorithms to most closely resemble the refinement techniques we have encountered in section 6.8. The techniques we have looked at mostly assume that load balance is already correct. In a dynamic setting an additional stage which first tackles load balance is required if such techniques are to be used; diffusion algorithms typically fill this role.

The idea behind diffusion [Cyb89] is to form a graph of the communication topology induced by the current partition; the *weighted processor communication graph*, G^{proc}. There are k vertices in G^{proc} which represent the processors and these are given weight $|S_p|_v$, which is the load upon them. Edges in G^{proc} are then added between processors which own adjacent sub-domains and are given a weight which is the number of cut edges in the dual graph of the mesh which cross that boundary.

G^{proc} is then considered as a physical thermal network, where the weight on its vertices corresponds to temperature and the weight on its edges to the conductivity of the connections between vertices. If this system is simulated and allowed to reach equilibrium, then each vertex will attain the same temperature (computational load) and the flow of thermal energy will correspond to the total vertex weight in the dual graph which needs to be exchanged. This is exactly the sort of algorithm that was introduced during our discussion of Jostle in section 6.8.3, and as we said there does not determine *which* elements need to migrate to attain load balance.

We choose to discuss diffusion here, not because we wish to present a complete study of dynamic partitioning (for which we refer the interested reader to [Jim97]) but rather to provide an illustrative distinction with static partitioning, as diffusion rarely finds place in the static context; indeed the one algorithm which we have encountered that makes use of it, Jostle, has been designed very much with dynamic partitioning in mind [WCE97].

Whether it is the static or dynamic partitioning problem for which we wish to develop a parallel method, employing one of the standard algorithms that works well in serial without regard to its suitability for parallel implementation is inadvisable. A good illustration of this is the Kernighan and Lin algorithm. As we have seen, the algorithm produces very good results and may be efficiently implemented in serial, but in parallel its performance is very poor. Not only that, but it is possible to formally demonstrate that it is fundamentally unsuitable for parallel implementation.

In the same way that we have seen that computational problems may be classified into categories such as '*NP*-complete,' a similar classification may be defined relating to concurrency. The class *NC* contains those problems which can be solved on a parallel machine with polynomially many processors in polylogarithmic time, and so algorithms in *NC* may be regarded amenable to parallel implementation. However, the Kernighan and Lin algorithm has been shown to be *P*-complete under log-space reductions [SW91]. The proof of this involves the reduction of Kernighan and Lin to the canonical *P*-complete problem; the Boolean circuit value problem. As the definition of *P*-complete is such that finding one instance of a *P*-complete problem that is also in *NC* would mean that *all* problem in *P* are in *NC* (a highly unlikely result) we may therefore consider Kernighan and Lin to be effectively outside *NC*. Additionally, the zero temperature version of simulated annealing has been shown, again in [SW91], to be *P*-hard, which is at least as difficult as *P*-complete[12].

These considerations lead the authors of [SW91] to develop the Mob algorithm, which they deemed to be more suitable to parallel implementation than either Kernighan and Lin or simulated annealing. They produced a data parallel implementation which ran on the Connection Machine CM-2 system, but the work does not seem to have been subsequently followed up past the extension to the embedding problem detailed in [SW93] or widely employed by other groups.

Despite the formal proofs in [SW91], parallel versions of both Kernighan and Lin [GZ87] and simulated annealing [Wil91] have been implemented, but with at best mixed results.

The parallel KL in [GZ87] was applied to sparse matrix factorisation with some degree of success on the Intel hypercube. However, it does not make particularly good used of parallelism, as its recursive structure is such that only two processors perform computation for each bisection. It is unclear how well this implementation would scale to very large problems as results are only detailed for small (less than 2000 variables) matrices.

We have already examined some of the variants of basic simulated annealing presented in [Wil91] in section 6.4.7.2, and indicated there that a parallel variant, *collisional simulated annealing*, had been developed. Collisional simulated annealing is not true simulated annealing, in that several moves are made concurrently and the resulting δH is not the sum of the individual δH values if the

[12]For a more detailed and formal discussion to these classes see [JaJ92].

changes were made one at a time. *Parallel collisions* arise, where two or more changes are made that may be individually beneficial but when taken together are not. The result is an algorithm that is highly parallel, but not particularly efficient. Some suggestions are made to tackle this problem, but the overheads of employing them make them unattractive and tend to erode the inherent parallelism of the approach.

[Wil91] compares collisional simulated annealing with parallel implementations of RCB and RSB on the NCUBE machine within the DIME[13] programming environment [Wil90]. Both recursive bisection implementations use the *divide and conquer* approach in which there is only limited parallelism. The approach is to perform each bisection sequentially, but to perform all the bisections at a given level of recursion concurrently, effectively reducing the number of stages from $k-1$ to $\log_2 k$. Thus, the first bisection occurs sequentially, then the two halves which result are bisected in two concurrent operations (each sequential in themselves), the four quarters which result are then bisected in four concurrent operations, and so on. As the cost of each bisection is dependent on the size of the sub-problem, this is a rather wasteful approach, particularly if we consider the fact that the dominant cost in RSB is likely to be the the initial bisection. Despite this, the divide and conquer RSB is still seen to be the best compromise, giving better results than RCB and being both faster and more reliable than collisional simulated annealing.

If a bisection algorithm is to be used efficiently in parallel then there must be concurrency in *both* the recursion *and* the individual bisections. Clearly, similar comments apply to any recursive multisection algorithm, for that matter. This approach has been taken in at least two parallel implementations of RSB, as follows.

Although we have yet to look at Lanczos eigensolution in detail, a glance ahead to the algorithm as detailed in equations 7.3 of section 7.4.8 is enough to show that the building blocks for a parallel implementation are concurrent sparse matrix-vector and inner products, neither of which present any great problem in themselves. We have previously noted the similarity between Lanczos and the conjugate gradient method, and used the latter as an example of the sort of solution procedure that is typical of many unstructured mesh applications, and now note that very much the same parallel linear algebra is required by the two methods. It is therefore evident that the efficiency of a parallel Lanczos eigensolution for

[13]DIME is the Distributed Irregular Mesh Environment developed at Caltech.

RSB will be influenced by how the mesh is distributed initially.

In a dynamic setting this may already be a reasonable distribution if the mesh has not changed radically, but in a static setting this will either be an arbitrary distribution (resulting, say, from parallel mesh generation) or else the mesh may not yet be resident on the parallel machine (if a serial pre-processor was used). In the latter case we need to use a 'cheap and dirty' decomposition algorithm to get the mesh onto the parallel platform as fast as possible and then use RSB to generate the actual decomposition that will be used by the application; good choices here would be either lexicographic, if we expect some coherence in element numbering, in which case we can probably just distribute the mesh in the order in which it is read from file, or a simple direct partitioning by the greedy algorithm.

Further, how the distribution of the mesh is inherited from previous levels of bisection is an issue. If the first bisection is calculated on all k processors (we assume the same number of sub-domains are required) then the two resulting sub-domains will not, in general, be resident on two disjoint sub-sets of processors. It may therefore be necessary to perform some redistribution of the mesh to take this into account, if the recursion is to proceed independently on two disjoint sub-sets of $k/2$ processors, which adds an undesirable software overhead to the use of a parallel platform.

A parallel implementation that follows broadly these lines, but side-steps the latter issue, has been implemented for the Connection Machine CM-5 system [JMJH95] and is available as part of the Connection Machine Scientific Software Library, CMSSL [Thi94]. Here a individual processor may be involved in the calculation of more than one bisection, but this is to some extent hidden behind the data-parallel language in which the implementation is written (CM Fortran). The required array operations for the Lanczos method make use of a segmented scan operation, where an additional integer *delimiter* vector provides a flag to separate the concurrent bisections. In this manner, all processors cooperate on the first bisection, then all cooperate on the two bisections at the next level of recursion (which occur concurrently) and so on. However, it does not appear that this parallel implementation significantly out-performs some of the accelerated spectral methods such as multilevel spectral, but it at least allows the use of consistent hardware between decomposition and problem solution.

Another implementation, this time one that performs each concurrent bisection on a disjoint sub-set of processors, has been carried out for the Cray T3D

[BS95, Bar95]. Here this sort of parallel control structure is referred to as *Recursive Asynchronous Task Teams* or RATTs. The main controlling operation of the RATTs is to split the problem into sub-problems, each of which are independent and may therefore be processed asynchronously. This implementation also takes the next logical step in the development of parallel decomposition algorithm, namely to incorporate the multilevel approach as well. This parallel implementation builds on the work we have previously detailed in section 6.9.3 and reference [BS92], and uses the same RQI/SYMMLQ interpolation of the eigenvector, which requires much the same linear algebra as Lanczos. There are additional issues in efficiently parallelising the determination of the maximal independent set for the graph contraction and in the detection of connected components for the eigensolution, but these problems are solved by the use of parallel algorithms due to Luby [Lub86] and Gazit [Gaz93], respectively. The performance of the resulting implementation is very high, about a factor of 140 times faster on 256 T3D processors when compared both to the same algorithm running in serial and to the CMSSL implementation which have very similar performance. The implementation's speed-up is noticeably sub-linear, something that the authors put down to poor initial partitioning (lexicographic is used) and the RATTs overhead. This implementation makes heavy use of the high performance, but Cray-specific, shared memory communication primitives, and is not therefore portable to other platforms.

Another approach is to treat parallel decomposition as a multilevel refinement problem, where redistribution of the mesh is now integral to the algorithm. This is the approach taken by both the parallel multilevel Jostle implementation [WCE97] and the parallel multilevel k-way algorithm of [KK97], both of which are MPI based and therefore widely portable[14].

The Jostle algorithm has the advantage when implemented in parallel that a large part of the work involved, namely the determination of the level structure *within* a sub-domain, is local to the processor on which it resides. Further, a halo is added to each sub-domain to allow the local calculation of gains, although this does add some complexity in the maintenance of the halo as the decomposition is changed. As we have already looked at multilevel Jostle, there is little further to say here, other than to reiterate our comments at the end of section 6.9.4, where we noted that comparison with static partitioning methods such as the

[14]The algorithm in [KK97] is also available in an implementation that makes use of the Cray-specific shared memory communication primitives which offer higher performance on platforms where they are supported.

RSB implementations we have just discussed is difficult, but that it appears that parallel multilevel Jostle is highly competitive and possibly one of the fastest methods we have encountered.

The multilevel k-way algorithm of [KK97] is not one that we have previously encountered. It has a very similar structure to multilevel KL, but is more amenable to parallel implementation (bearing in mind our earlier comments concerning the unsuitability of KL for this purpose). It is based on the serial version of the algorithm available in the METIS decomposition package, as detailed in [KK95b]. The refinement technique used is termed *greedy refinement* (not to be confused with Farhat's greedy algorithm, which is unrelated) and is essentially a simplified version of KL, with similar gain functions determined by cut-edges.

It is an iterative process where, at each iteration, all vertices on the sub-domain borders are considered and are either:

- Moved to the neighbouring sub-domain that results in the maximum reduction in cut-edges, provided this is beneficial and does do not violate a load-balance criterion.
- If no such move exists, the vertex is moved so as to improve load-balance, provided this does not increase cut edges.
- Otherwise left in place.

While this process does not possess KL's ability to climb out of local minima, and so might be thought to suffer from the deficiencies typified by gradient descent, when it is coupled with the multilevel approach it nonetheless proves highly successful. Although no direct comparison is offered in [KK97], it would appear that performance is on a par with parallel multilevel Jostle and so is of a very high level.

An advantage of both these parallel refinement algorithms is their suitability for use in dynamic applications, as they implicitly take into account an existing mesh decomposition. An extension to spectral methods which can explicitly take into account an existing decomposition and the mapping of sub-domains to processors originates in *terminal propagation* [DK85]. Here additional information is inherited from one level of recursion to another, resulting in a *constrained* problem [DR94] which can be solved by the iterative application of the eigensolution methods we have previously discussed. If constrained spectral methods are to be used in conjunction with dynamic applications they must clearly run in parallel

themselves, but to date no parallel implementation for the constrained problem has been attempted to this author's knowledge.

We will close this section with a short discussion of the *sub-domain generation method* [KT93, ST97], which we previously introduced in section 6.4.7.5 in reference to its use of genetic algorithms. The sub-domain generation method (SGM) performs parallel mesh decomposition as part of the mesh generation process itself, which also runs in parallel. The mesh is initially defined only in terms of a very coarse mesh sufficient to define the problem geometry and allow a reasonable initial decomposition to be performed. This initial decomposition of the coarse mesh is based on a prediction of how many elements will be generated within each coarse mesh element by subsequent refinement. This prediction is provided by a back-propagation neural network (one where neurons are arranged in layers, with information propagating from the input layer, through a number of hidden layers, to the output layer [RHW86]) which has been previously trained with example data to enable it to make this prediction. Given this prediction, the coarse mesh is decomposed by using a genetic algorithm to optimise the position of a line in 2D [KT93], or a plane in 3D [ST97] which bisects the mesh. This bisection procedure is applied recursively in the usual manner to yield a decomposition of the coarse mesh. The coarse mesh is then distributed and the final mesh elements generated in parallel by refinement within the coarse mesh elements. If the prediction by the neural network of the structure of the final mesh was good then a reasonable final decomposition will result. This method is evidently tightly coupled to the application in question and does not appear to have found wide acceptance, perhaps for that reason. While it is inapplicable to generic graph partitioning, where there may be no geometric information and no notion of mesh generation, for example in problems arising from matrix factorisation or circuit placement, SGM does have the useful property that it may deal with load-balancing the parallel mesh generation process, which is not the case with any other method we have encountered.

We will return to the discussion of genetic algorithms in chapter 10, where we introduce a novel approach to their use for mesh decomposition and graph partitioning that was originated at EPCC, as previously mentioned in section 6.4.7.5.

6.11 Summary

We have reviewed the range of methods that have been developed for the related tasks of mesh decomposition and graph partitioning, presenting these methods according to the classification we introduced at the beginning of this chapter in section 6.1, which we shall not reiterate here. The comparison of algorithms presented in this chapter has been a qualitative one, based primarily on the consensus in the relevant literature and illustrated by example decompositions of the Widget data-set.

While we have seen that a number of very powerful techniques exist, no one method has been developed that is clearly the definitive choice of algorithm. It remains the case that the choice of algorithm is strongly influenced by the application, which has lead to the development of a number of public domain or commercially available packages that implement a variety of algorithms, such as those listed at the beginning of section 6.10 and indeed PUL-md itself.

Despite the volume of work carried out in the field, considerable room for further study and development exists. The most promising and also most challenging areas of development would seem to be parallel and multilevel techniques, or a combination of the two. Not only are these promising directions for the development of static partitioning algorithms, but parallel implementation is a prerequisite if we wish to develop algorithms that are also applicable to dynamic partitioning.

It is based on the qualitative comparison presented in this chapter that the algorithms implemented in the PUL-md library have been selected. In the following chapter we look in detail at this implementation, thereafter turning to a quantitative comparison of the implemented algorithms in chapter 8.

Chapter 7

Development of a Mesh Decomposition Library

The subject of this chapter will be the development of the mesh decomposition library, PUL-md. The decomposition library is closely associated with the runtime support library PUL-sm, which we introduced in section 5.1.6. As well as providing the mesh decomposition, PUL-md provides various pre-processing facilities that PUL-sm relies upon, for example setting up mesh halos and processor blocked files. We will only briefly touch on these facilities, and primarily focus on the implementation of the mesh decomposition and graph partitioning algorithms, as this is the main topic of this thesis.

We begin with an overview of the capabilities and top-level design of the library, then move on to look in detail at its data structures and the implementation of the decomposition algorithms it supports. This will form the bulk of this chapter, and will make reference to the previous chapter's survey of algorithms wherever possible, as many of the concepts will already be familiar from that study. Where PUL-md departs from the ideas presented in chapter 6, or where discussion has been postponed from the previous chapter to the current, we shall go into greater detail. In particular, the Lanczos method of eigensolution, which is used by our implementation of RSB, and the Fiduccia and Mattheyses linear time implementation of KL which we employ, will be presented here. Finally, we shall close with a short demonstration of the visualisation that PUL-md's interface to the AVS visualisation package allows.

7.1 Introduction

PUL-md, and its sister library PUL-sm, are together intended to support most of the various aspects of computation and communication that parallel unstructured mesh calculations have in common. In section 5.1.6 we detailed the features of PUL-sm that support this, and will now do the same for the PUL-md library.

PUL-md provided the following features:

- It is a serial pre-processor for PUL-sm, but may also be used as a stand-alone decomposition package.
- The input mesh may either be defined by a file in standard format, or from application arrays by means of a *registration* function.
- The input mesh may be of arbitrary element type (subject to a few minor restrictions).
- It will extract the dual graph of a mesh, based on either node, edge, or (in three-dimensions) face connectivity.
- It will partition the dual graph with a variety of global methods and subsequently refine the result with a variety of local techniques.
- Given a partition (which need not have been generated by PUL-md itself), it will decompose the mesh constituents accordingly, a process which we term *mesh constituent localisation.*
- It supports both the shadow node and halo models of mesh distribution.
- After mesh constituent localisation, halo structures may be generated if required.
- It writes all files required by PUL-sm for initial mesh and application data input at the start of parallel calculation, in processor blocked format.

Although it is implemented as a set of library functions, the PUL-md distribution also provides code to allow its use as a stand-alone decomposition package with no further user programming necessary. Its visualisation facilities are presented in a similar manner, and do not therefore form part of the library proper.

7.2 Application Program Interface

The features of PUL-md which we have just outlined are presented to the user through a set of functions, which form the library's application program interface (API). Although PUL-md is implemented entirely in ANSI-C, like PUL-sm its API supports both ANSI-C and FORTRAN-77 prototypes. As PUL-sm is also implemented entirely in ANSI-C and makes use of the widely implemented MPI standard for all message passing, the two libraries together allow PUL to support a wide variety unstructured mesh application on a wide variety of platforms.

The full PUL-md API for C applications is as follows:[1]

- Mesh Initialisation and Registration:
  - `md_init`
  - `md_register`
  - `md_dataSizes`
- Mesh Decomposition:
  - `md_decompose`
  - `md_tune`
  - `md_dualGraph`
- Mesh Constituent Localisation:
  - `md_localNodesElmts`
  - `md_localFaces`
  - `md_localEdges`
- Mesh Halo Generation:
  - `md_nodeElementHalos`
  - `md_faceHalos`
  - `md_edgeHalos`
- Mesh Input/Output:

[1] changing the 'md_' prefix to 'MD' yields the FORTRAN interface. In this thesis we restrict ourselves to presenting code fragments and examples in C only.

- `md_readDecomp`
- `md_writeMesh`
- `md_createDataFile`
- `md_writeProcHeader`
- `md_writeProcData`
- `md_writeGlobalHeader`
- `md_writeGlobalData`
- `md_closeDataFile`

- Miscellaneous Functions:
  - `md_reportError`

Of these, those functions concerned with initialisation, registration and decomposition we will have more to say about. We will say no more than we already have regarding mesh constituent localisation, halo generation and input/output functions. Concerning the single function in the miscellaneous category, `md_reportError`, we will simply note that it serves to interpret returned error codes in the event of an internal library error. It is not our intention to duplicate the complete specification of the API and file formats that is presented in the PUL-md User Guide, and so we refer the reader to that document [BDT96] for further details.

Our main concern will be with the implementation of the `md_decompose` function, which performs the actual mesh decomposition, as its name would suggest. An example of its usage is as follows:

```
#include "pul-md.h"

result = md_decompose("meshFile", MD_EDGES, MD_RSB, MD_REF_KL,
                      "resultFile", resultArray, 16)
```

Here we are decomposing the mesh given in `"meshFile"` into 16 sub-domains using RSB subsequently refined by KL. The dual graph on which RSB operates is based on edge connectivity in the mesh, and the resulting decomposition is written to `"resultFile"` and the array `resultArray`. All of the symbolic constants (`"MD_EDGES"`, etc.) and function prototypes are defined in the include file `"pul-md.h"`.

Most of the decomposition algorithms available have associated tunable parameters which can modify their default behaviour if desired. This may be done through the function `md_tune`, so that we might precede the call to `md_decompose` with the following if we wished to modify the eigensolution used by RSB, for example:

```
result = md_tune(MD_RSB_ORTHOG, MD_FALSE)
```

If we do not wish to read the mesh from `"meshFile"`, but rather from application arrays, then the functions `md_init` and `md_register` allow this. The former specifies global features of the mesh, such as number of spatial dimensions and so forth, while the latter reads the application arrays into an internal data structure. If this has been done, then we may substitute `NULL` for `"meshFile"` in the call to `md_decompose` and the decomposition function will then use the previously registered information. Similarly, output may be limited to either file or application array by substituting `NULL` for the unrequired argument.

The remaining initialisation function, `md_dataSizes`, is used by PUL-sm for the input of application data, and so need not concern us here.

The remaining decomposition related function, `md_dualGraph`, is used to extract the dual graph of the mesh without subsequent partitioning. It is useful if another, graph based, partitioning package is to be used in place of `md_decompose` and may take its mesh description either from file or through `md_register`, just as we have seen for `md_decompose` itself. If an external partitioning package has been used then `md_readDecomp` enables the resulting decomposition to be read prior to subsequent mesh constituent localisation, halo generation and output.

7.2.1 Stand-Alone Usage

The test program for PUL-md forms both an example of usage for the libraries and a stand alone decomposition tool. It reads a mesh structure file and outputs a file giving a processor assignment for each element, both in PUL standard formats.

The program is called `md_test` and takes the following command line arguments (in order):

- `baseFileName`, the mesh structure filename root.

- `format`; `A` (ASCII) or `B` (binary) - defining the format of the input file.
- `dual`; `NODES`, `EDGES` or `FACES` - defining the type of dual graph required.
- `algorithm`; `RIB`, `RSB`, etc. - defining the global decomposition algorithm.
- `refine`; `NONE`, `KL`, etc. - selecting any subsequent refinement.
- `nParts`, the number of sub-domains into which to decompose the mesh.

It looks for a standard mesh structure file in `baseFileName.mdesc` and places its output in `baseFileName.decomp`. Optionally, if the file `baseFileName.param` is present, then that file is read to adjust user tunable parameters. It also provides statistics giving metrics of decomposition quality.

Thus, to decompose the mesh in the ASCII file `widget.mdesc` using the same options as were used in the previous example, the appropriate command would be:

```
unixshell$  md_test widget A EDGES RSB KL 16
```

Where the resulting decomposition would be written to `widget.decomp`.

Similarly, if we wished to tune the same parameter as we did in the previous example, then the file `widget.param` would contain the line:

```
MD_RSB_ORTHOG = MD_FALSE
```

Again, full details are available in the User Guide.

7.3 Design and Data Structures

We will now turn our attention to examining the implementation of the decomposition function `md_decompose`.

The algorithms implemented in `md_decompose` fall into two categories; global methods and local refinement techniques. These are specified as two separate arguments to the decomposition function, as was illustrated in the example of usage.

The global methods available are:

- Simple random (SR), cyclic (SC) and lexicographic (SL) partitioning.
- GREEDY, Farhat's greedy algorithm.
- RLB, a recursive layered (graph) bisection.
- RSB, a recursive spectral bisection.
- RCB, a recursive coordinate bisection.
- RIB, a recursive inertial bisection.

and may be used in combination with either:

- No refinement.
- MOB, the Mob refinement algorithm.
- KL, the Kernighan and Lin refinement algorithm.

We will examine each of these algorithms in turn later, but first will look at the high level design of the code, then turn to the data structures and features that the individual algorithms have in common.

7.3.1 Top Level Design

The top level of the decomposition function, in somewhat abbreviated form, is as follows:

```
Partition *partn;

/* Timing point A */

CALL(__dual__(&partn, dual, alg, meshFile));

/* Timing point B */

switch (alg)
{
case MD_RLB:
    CALL(__rlb__(partn, nParts, refine));
    break;

case MD_RSB:
    CALL(__rsb__(partn, nParts, refine));
    break;
 .
 .
```

```
default:
    STOP(MD_NOTSUPPORTED);
    break;
}

/* Timing point C */

if (decompFile != NULL)
{
    CALL(write_decomp(partn, decompFile));
}

if (MS.decomp != NULL)
{
    CALL(store_decomp(partn, MS.decomp));
}

CALL(print_stats(partn, nParts));

dummy = __partn_destroy__(&partn);
```

The first action is to declare a pointer to the `Partition` data structure which is common to all PUL-md decomposition functions. The `Partition` data structure essentially embodies both the dual graph and its partition into sub-domains. It also includes addition information that the various decomposition algorithms require, as we shall see.

The `Partition` data structure is allocated and initialized by the function `__dual__`, which extracts the dual graph of the mesh. In PUL-md all `externed` functions that do not form part of the API are pre- and postfixed with "`__`" to avoid name-clashes with application functions or variables.

Next, the `case` statement chooses the global decomposition algorithm that is to be used, after which the decomposition is written to file and/or array (functions `write_decomp` and `store_decomp`), statistics of decomposition quality are printed (`print_stats`) and finally the `Partition` data structure is destroyed (`__partn_destroy__`).

It can be seen that there is no facility for the partitioning of arbitrary graphs not arising directly from a mesh through the call to `__dual__`. The code could easily be modified to accommodate this by simply substituting a new routine in its place which sets the `Partition` data structure directly from a user-supplied graph description.

7.3.2 The Partition Data Structure

All access to the `Partition` data structure is abstracted in the code, either through functions like `__partn_destroy__` or through CPP macros. We will not detail the partition manager here, except insofar as it has direct bearing on the coding of the decomposition algorithms, but further details may be found in the original PUL-md Design Description [Tre95a]. However, the `Partition` data structure itself is fundamental to the decomposition algorithms and has particular bearing on their primarily recursive structure. Hence we will need at least some understanding of it before proceeding to describe the implementation of the algorithms themselves.

The basic unit of the `Partition` data structure is the dual graph vertex, which is defined as:

```
typedef struct vertex Vertex;
struct vertex
{
short index;                    /* Index of sub-domain the vertex is in */
short position;                 /* MD_BOUNDARY or MD_INTERNAL (RLB) */
short gain;                     /* Gain in bWidth if vertex moved (MOB/KL) */
short nNeigh;                   /* Number of edges from this vertex */
short localnNeigh;              /* Number of edges from this vertex
                                 * which do not leave the partition (RSB) */
int   layer;                    /* Layer index within domain. (RLB) */
Vertex **neighbours;            /* Array of ptrs to connected vertices */
int   vecIndex;                 /* Position of this vertex in a partition's
                                 * array of vertex pointers (RSB) */
double *coords;                 /* Pointer to coords of this vertex (RCB/RIB) */
DItem *list_loc;                /* Pointer to KL vertex gain list entry */
};
```

The connectivity of the graph is defined by the list of neighbours stored in the array `neighbours`, which is of size `nNeigh`. Of the remaining fields, most are algorithm specific (as indicated in the comments) with the exception of `index`, which gives the sub-domain in which the vertex presently resides.

There is currently no provision in the `Vertex` data structure for either vertex or edge weights, although these could easily be accommodated as additional fields in the structure[2]. As we are primarily dealing with graphs arising from unstructured meshes, most of which are of uniform element type, this is not a great drawback. However, if general graphs are to be considered, or if elements are non-uniform

[2]This would result in the duplication of data for the edge weights on the two vertices that make up the edge, but is still the most economic option.

in the mesh case, then such an extension would clearly be necessary.

The `Partition` data structure itself is defined as:

```
typedef struct Partition
{
    int size;                   /* Total number of vertices */
    int p1;                     /* Index of one side of partition */
    int p2;                     /* Index of other side of partition */
    int bWidth;                 /* Bisection width of the partition */
    int maxGain;                /* Maximum gain value for the vertices */
    int dims;                   /* Number of dimensions for coords */
    int sep_vertex;             /* Array index of boundary vertex, in */
                                /* separator sorted order */
    Vertex **vertexPtrs;        /* Array of pointers to partition vertices */
    Vertex *vertices;           /* Array of all vertices */
    double *vertexCoords;       /* Array of all vertex coords */
    IntList **tmp;              /* Workspace used during initialization */
    SpaceMgr *space;            /* Storage management */
} Partition;
```

How this structure is used is illustrated in figure 7.1. This figure shows its application to a mesh, the dual graph of which is *not* shown. The dual of the mesh in the figure is partitioned according to two instances of the `Partition` data structure; one for the original partition and one for the current partition, which is a sub-division of the former. Each instance determines a bisection of the dual, and so we see the mesh split into three. We note that the fields in the declaration of `Partition` which specify the indices of the sub-domains it contains (`p1` and `p2`) are two in number, which biases its use towards bisection.

From the figure we see the array of vertices, each of type `Vertex`, is only directly referenced by the *original* partition. The current partition has its `vertices` array undefined, which will be the case for all instances of the `Partition` data structure except the initial instance created by `__dual__`. This allows efficient use of storage, so that information is not unnecessarily duplicated when the data structure is used in a recursive manner, as only the `vertexPtrs` array is used in all instances. We also see the reference to the `SpaceMgr` structure which is made through the `space` field; this data structure is used by the PUL-md memory manager which the partition manager employs. This is of particular importance in the initial specification of the dual graph in `__dual__`, where we have to take into account that each vertex has an unknown number of neighbours, and so the size of the required storage for the `neighbours` array in each instance of the `Vertex` data structure is not known in advance.

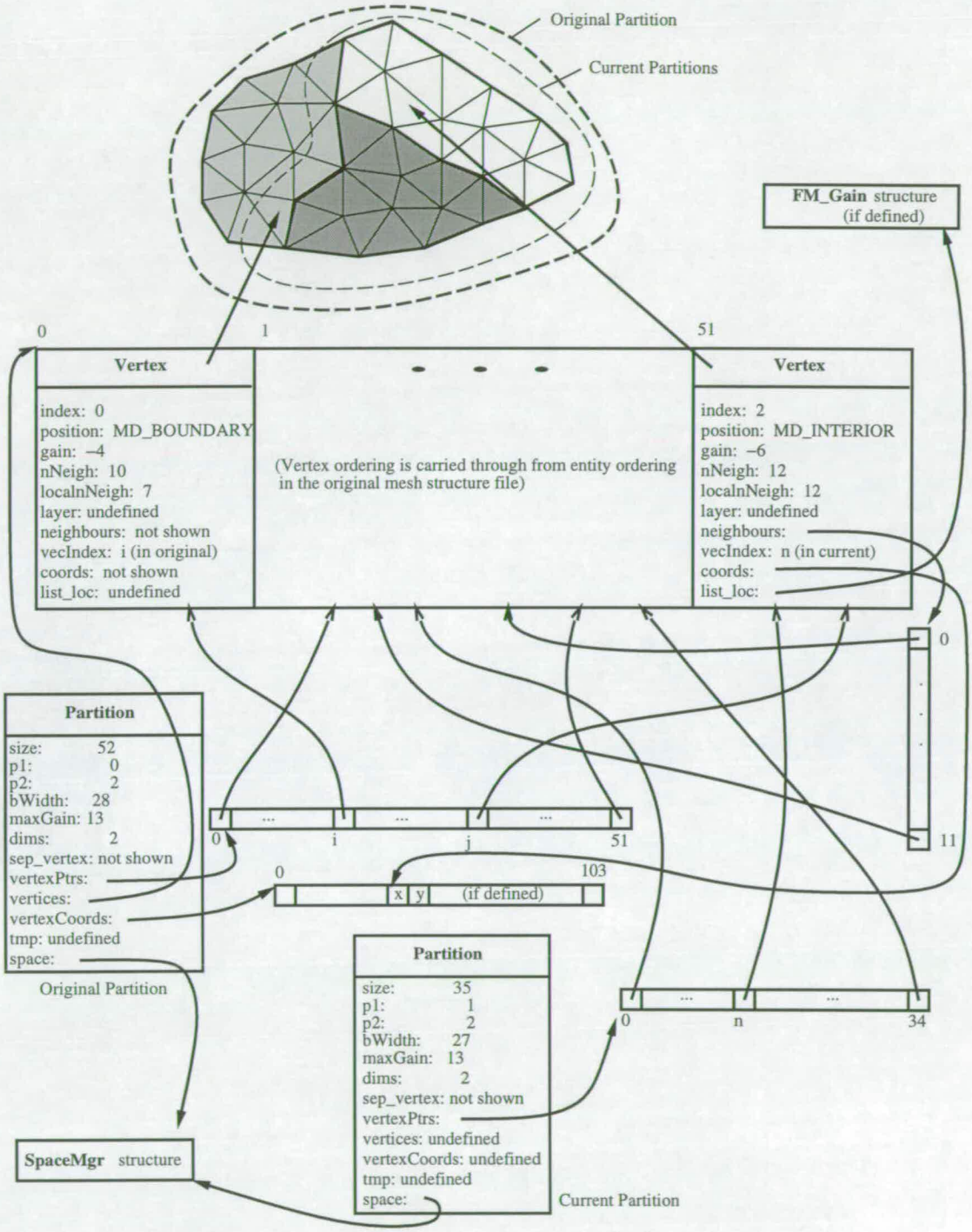


Figure 7.1: The partition data-structure.

7.3.3 Recursive Routines

How the `Partition` data structure relates to the recursive nature of most of the algorithms implemented in PUL-md is made clearer by examining a typical example. Below we show the code for RSB, again in somewhat abbreviated form; the code structure is similar for all the recursive algorithms, however.

```
extern int __rsb__(Partition *partn, int nParts, int refine)
{
    Partition *p1=NULL;

    lSideSize = (nParts / 2);
    rSideSize = (nParts / 2) + (nParts % 2);

    PARTN_BWIDTH(partn) = 0;
    PARTN_P2(partn) = PARTN_P1(partn) + lSideSize;

    CALL(spectral_bisection(partn));

    switch (refine)
    {
    case MD_REF_NONE:
        break;

    case MD_REF_KL:
        CALL(__kl__(partn, 2));
        break;

    case MD_REF_MOB:
        CALL(__mob__(partn, 0));
        break;

    default:
        STOP(MD_NOTSUPPORTED);
        break;
    }

    if ((splitPt = __partn_sort__(partn)) < 0) STOP(MD_ERROR);

    if (lSideSize > 1)
    {
        CALL(__partn_split__(partn, PARTN_P1(partn), splitPt, &p1));
        CALL(__rsb__(p1, lSideSize, refine));
        CALL(__partn_destroy__(&p1));
    }

    if (rSideSize > 1)
    {
        CALL(__partn_split__(partn, PARTN_P2(partn), splitPt, &p1));
        CALL(__rsb__(p1, rSideSize, refine));
        CALL(__partn_destroy__(&p1));
    }
```

```
    if (p1 != NULL) dummy = __partn_destroy__(&p1);
}
```

The first action, again, is to declare a pointer to an instance of the `Partition` data structure, this time one that will embody first one, then the other of the two sub-domains that result from the bisection. As the recursion occurs in a 'depth-first' manner, only one such structure is needed.

The argument, `nParts`, determines how many sub-domains are to ultimately be created from the current `Partition`. For bisection we can think of the two resulting sub-domains as the 'left' and 'right' sides, so we denote the number of sub-domains which must ultimately result down the two branches of recursion at this level as `lSideSize` and `rSideSize` accordingly. If we are partitioning into a power of two number of sub-domains in total, then `lSideSize` and `rSideSize` will be equal at each and every level of recursion. If this is not the case, then we must generate an unbalanced pair of sub-domains at some point in the recursion, so that the final sub-domains are balanced. Not all algorithms in PUL-md take this into account, so in general we assume that `nParts` is a power of two and a balanced bisection is performed.

Next in the code fragment, we see the `bWidth` and `p2` fields in `partn` being set through the appropriate macros. Once this is done the actual bisection is performed. In this case the bisection is spectral and so `spectral_bisection` is called, but if, for example, an inertial bisection was required then a call to `inertial_bisection` would take its place here.

Following the bisection the resulting partition may be refined, if required, by either Mob or KL, as determined by the `case` statement.

Of particular importance are the two partition manager functions that are called before the recursive call to `__rsb__`. The first of these, `__partn_sort__` sorts the `vertexPtrs` array of the current partition, `partn`, according to the `index` field of each vertex. The actual positions of the vertices in the `vertices` array of the original partition are left unaltered. This function *must* be called before the subsequent calls to `__partn_split__`.

`__partn_split__` splits a sorted partition and returns a sub-partition corresponding to one or the other of the two halves (i.e. sub-domains) of the current partition. As the `vertexPtrs` array of the current partition has been sorted, the new partition's `vertexPtrs` array can point to the appropriate part of its

parent's array and its `vertices` can be `NULL`, no extra storage need be allocated. In this way we are able to implement this recursive process with only a very small additional memory overhead associated with the depth of recursion, which may be an important consideration if we are partitioning into a large number of sub-domains.

After the new partition, `p1`, has been split from the left side of its parent, it is then recursively partitioned by `__rsb__`. After the left side has been dealt with `p1` is then deallocated with `__partn_destroy__` and reused in the recursion down the right side in exactly the same manner.

Having introduced the concepts, design and data structures that form the framework in which the decomposition algorithms are implemented, we will now examine each of them in turn, starting with the global algorithms, then moving on to the local refinement techniques. In the course of doing so we will introduce the various tunable parameters that influence their behavior.

7.4 Implementation of Global Algorithms

There are two features of PUL-md's implementation of its global decomposition algorithms which are common to several of them, but which are not used by any of the local refinement techniques. Hence we shall introduce these features before proceeding.

7.4.1 Determining Layer Structures

Where the Cuthill-McKee algorithm is used in conjunction with lexicographic partitioning or recursive layered bisection, and also where we employ a similar process in the course of the greedy algorithm, we must compute a layer structure for the graph.

Although there are distinctions between these various instances, the basic algorithm used is the same, as shown in the pseudocode of figure 7.2.

The important feature to note is the use of the vertex queue, Q, which is implemented as a singly linked list. As the algorithm proceeds, vertices are added to the end of the queue (the tail of the list; line 15) when they are first encountered as neighbours, n, of the vertex, v, which has just been taken from the front of the

```
Pseudo: Layers
 1.  i = 0
 2.  layer^cur = 0
 3.  v = seed vertex
 4.  layer(v) = layer^cur
 5.  Vertex queue Q = v
 6.  While Q ≠ ∅
 7.     Pop vertex v from the head of Q
 8.     number(v) = i
 9.     i += 1
10.     If layer(v) ≠ layer^cur Then
11.        layer^cur += 1
12.     EndIf
13.     For All neighbours, n, of v not assigned a layer
14.        layer(n) = layer^cur + 1
15.        Add n to the tail of Q
16.     EndFor
17.  EndWhile
EndPseudo
```

Figure 7.2: Computing the layer structure.

queue (head of the list; line 7). It is clear from this that Q is of variable length and that we can make no *a priori* estimate of the maximum storage required, save to say that it must be smaller than the current graph itself, hence the use of a linked list data structure is to be preferred. When the list is empty ($Q \neq \emptyset$; line 6) we know that the entire graph has been considered if it is connected, or that at least a connected component has been found containing the seed vertex. In the latter case we must re-start the procedure with a new seed vertex in a part of the graph not yet visited. The numbering of the vertices (line 8) may be done before or after they are included in the list, but assigning the vertex to a layer (line 14) before it is added to the queue enables us to tell when an entire layer has been exhausted and increment the layer counter (lines 10 to 12) accordingly.

It can be seen that this pseudocode differs from that shown in figures 6.6 (Cuthill-McKee) and 6.12 (Farhat's greedy algorithm).

The process is actually equivalent to the Cuthill-McKee pseudocode, except that we are unconcerned with the sorting of vertices by vertex degree that occurs in line 11 of the pseudocode in figure 6.6. In point of fact, we prefer to keep the order in which vertices are encountered as neighbours, as we might reasonably expect there to be some coherence in the order that results *within* each layer. Thus, if a layer is split between two sub-domains we would expect the resulting boundary to be better than that which would result if the vertices were sorted by degree, which has no relation to adjacency.

The difference in the case of the Farhat's greedy algorithm is an evident result of the fact that the pseudocode in figure 6.12 is based on the mesh itself, not its dual graph. That said, the pseudocode presented here explores the entire mesh (or all of that connected part in which the seed vertex resides). If we wish to use this approach to implement the greedy algorithm we simply need to halt the process when sufficient vertices have been visited to fill an entire sub-domain and restart from another seed.

7.4.2 Separator Fields

Separator fields are already familiar to us from section 6.5.2, and `md_decompose` uses exactly the simple approach detailed the pseudocode of figure 6.22 in that section.

We are primarily concerned with bisection and so a simple optimisation opportunity is offered to us here. Rather than bisecting the current graph into two balanced sub-domains, we may allow some level of imbalance in return for a reduction in cut edges.

Before any decomposition of the current graph takes place, `md_decompose` considers all vertices to be initially in a single sub-domain determined by the `p1` field of the `Partition` data structure. It performs a bisection by moving the required vertices to sub-domain `p2`, using the partition manager function `__partn_move__`, which keeps track of various related data including the number of cut edges, which is recorded in the `bWidth` field of the `Partition` data structure.

Separator bisection is performed by sorting the `vertexPtrs` array on the key provided by the separator field and moving vertices in just this way. This means that we can keep track of `bWidth` in the course of this process and choose a bisection which minimizes it, subject to some maximum acceptable imbalance, at no significant extra cost. This sorting is unrelated to that performed by `__partn_sort__` which is only called later, after local refinement, as we have seen. The sorting here is done by a standard indexed implementation of the Quicksort algorithm [FPTV92].

We note for future reference that a vertex in the sorted `vertexPtrs` array that lies next to the bisection boundary is recorded in the `sep_vertex` field of the `Partition` data structure, so that the split point in the array may subsequently be identified.

If the user wishes to take this option, then the tunable parameter MD_SEP_IMBAL should be set to MD_TRUE using md_tune. If this is done then the maximum acceptable imbalance may be specified, via the the tunable parameter MD_SEP_MAX_IMBAL, which is considered as a percentage of the number of vertices in the current graph. The defaults for MD_SEP_IMBAL and MD_SEP_MAX_IMBAL are MD_FALSE and 5%, respectively. An upper limit of 30% is imposed on MD_SEP_MAX_IMBAL.

7.4.3 Simple Partitioning

Global algorithm selected by: MD_SR, MD_SC and MD_SL

Tunable parameters: MD_SL_CM_TIMES (MD_SL only)

Simple random, cyclic and lexicographic partitioning, may be selected by calling md_decompose with the symbolic constants MD_SR, MD_SC and MD_SL to specify the global decomposition algorithm, respectively. In order that any subsequent local refinement may proceed in a pair-wise fashion, all are implemented by means of successive bisections.

Random and cyclic, as we saw in sections 6.3.1 and 6.3.2, are not feasible decomposition algorithms in themselves, and their implementation is trivial. Hence they do not warrant further discussion, except to note that they are largely included for purposes of comparison and to give an arbitrary decomposition as the start of subsequent local refinement. There are no tunable parameters associated with these methods.

Lexicographic partitioning is again trivial to implement given some vertex numbering. This numbering may be either that implicit in the initial vertex ordering or may be provided by Cuthill-McKee renumbering, as we saw in section 6.3.4. If Cuthill-McKee is used then the method we have just detailed in 7.4.1 is employed. Whether this takes place or not is controlled by the tunable parameter MD_SL_CM_TIMES, which determines the number of Cuthill-McKee iterations. The default is 2 and selecting 0 iterations preserves the original vertex numbering.

7.4.4 Recursive Layered Bisection

Global algorithm selected by: MD_RLB

Tunable parameters: `MD_RLB_CM_TIMES` and `MD_RLB_CM_BEST`

As we have already described the basic algorithm in section 6.7.1, and have just detailed the mechanism used to determine the dual graph's layer structure in section 7.4.1, there are only a few point left to discuss concerning the implementation of RLB.

Two points we must consider are the choice of initial seed point and the possibility that the current graph may well be disconnected.

Looking back to the `Vertex` data structure, we see that is has a field, `position`, which may take either the value `MD_BOUNDARY` or `MD_INTERNAL`, depending on whether the vertex represents an element on the external mesh boundary or not. The initial seed point for the layer structure is found by cyclically traversing the `vertexPtrs` array from a random starting point and taking the first vertex found on the external boundary as the initial seed. If no such vertex is found then the first vertex found on an internal sub-domain boundary is used.

If the current graph is disconnected then each connected component is identified and considered in order of decreasing size. These components are associated, in that order, with the first of the two sub-domains in the bisection until the addition of another component would cause the sub-domain to exceed the required size; that connected graph component is then split between the two sub-domains according to its layer structure and all the remaining (smaller) components are assigned to the second sub-domain.

While we follow the pseudocode of figure 7.2 to determine the layer structure, in many cases we may terminate the process if sufficient vertices have been identified in a connected graph component to fill a sub-domain. We can not do this if we wish to use the process in a Cuthill-McKee like manner to find a maximally separated pair of vertices and take one of these as the seed point for the bisection, except on the last iteration.

The number of Cuthill-McKee iterations is controlled by the tunable parameter `MD_RLB_CM_TIMES`. If 0 iterations are selected the initial seed located on a border is used to determine the bisection, otherwise the required number of iterations are performed and the resulting last vertex found may be used.

We note that the bisection itself will occur along one of the layers we have found, and that which layer this is may be identified as we go along. If we assume that an approximate measure of the number of cut-edges that would result from a

bisection along this layer is given by the layer's size, then we may estimate the quality of the resulting bisection at no significant extra cost. This is useful because, although the Cuthill-McKee iterations tend to quickly settle down to alternating between a maximally separated pair of vertices, using one of these may result in a better bisection than the other and we may use our estimate to choose between them. Hence we implement this optimisation, which may be selected by setting `MD_RLB_CM_BEST` to the value `MD_TRUE`.

The default values of `MD_RLB_CM_TIMES` and `MD_RLB_CM_BEST` are 1 and `MD_FALSE`, respectively.

7.4.5 The Greedy Algorithm

Global algorithm selected by: `MD_GREEDY`

Tunable parameters: none

Like RLB, the greedy algorithm makes use of the method detailed in the pseudo-code of figure 7.2 and, as we said in section 7.4.1, we now halt the process when we have filled a sub-domain. Again we must consider the choice of seed vertex which will now determine the start of each new sub-domain; referring back to figures 6.13 to 6.15, these are indicated by the elements marked in black.

In dual graph terms, Farhat's algorithm takes as the seed of each new sub-domain a vertex with as few as possible neighbours that are not already part of an existing sub-domain. Our implementation does not take this into consideration, and simply uses the vertex that would have next been chosen after the last that was actually taken by the previous sub-domain. If this seed is not valid (i.e. is in an existing sub-domain) then an arbitrary one of its neighbours is used, or, if no neighbours are valid, the new seed is simply taken as the first vertex in the `vertexPtrs` array. As this is done after the call to `__partn_sort__` which defined the previous sub-domain the seed will always be valid. The very first seed is also chosen as the first vertex in the `vertexPtrs` array.

Our implementation is essentially a modified version of RLB, where a unbalanced bisection is used to partition the graph into one new sub-domain and the remainder of the, as yet unexplored, graph; hence the recursion is now single-sided. This is done subject to the choice of seeds just outlined and, of course, has none of the iterative aspects of Cuthill-McKee that RLB does. An additional consideration is what to do if a sub-domain becomes 'trapped' by being completely

surrounded by previously generated sub-domains and/or external boundaries before enough layers have been added to bring it up to its required size. In this case we must start from a new seed, in an as yet unexplored part of the graph, and claim as many more vertices as are needed to bring the sub-domain up to size. This results in a disconnected sub-domain being generated, but is unavoidable if balanced sub-domains are required. If this occurs then the new seed for the disconnected portion of the sub-domain is found by cyclically traversing the `vertexPtrs` array from a random starting point and taking the first valid vertex found on the external boundary. If no such vertex is found then the first valid vertex found on a internal sub-domain boundary is used. This is essentially the same process used by RLB.

We implemented the algorithm in this way so that we might use pair-wise refinement at each stage to improve the border between each new sub-domain formed and the remainder of the graph. A difficulty arises if we do this, in that the vertex on the boundary between the new sub-domain and the remainder of the graph that we would otherwise have used as the seed for the next sub-domain may no longer be on the border after local refinement. This issue is as yet unresolved, and so the current release of PUL-md does not support refinement in conjunction with this algorithm.

7.4.6 Recursive Coordinate Bisection

Global algorithm selected by: `MD_RCB`

Tunable parameters: `MD_RCB_CYCLE`, `MD_RCB_FIXED`, `MD_SEP_IMBAL` and `MD_SEP_MAX_IMBAL`

The implementation of RCB follow exactly that presented in section 6.6.1, where the algorithm was introduced.

The required coordinate information is extracted from the mesh by `__dual__` and placed in the `vertexCoords` array pointed to by the `Partition` data structure, and also by the `coords` field in the `Vertex` data structure. The coordinates of each vertex are taken to be the mean of the nodes making up the corresponding mesh element, but this information is only calculated and appropriate storage allocated if a geometric (i.e. RCB or RIB) algorithm is to be used (hence the `alg` argument to `__dual__`).

The chosen coordinate axis is then used as a separator field in the familiar manner, and all that is left to discuss is which axis is to be used. This is determined by the settings of the associated tunable parameters, as follows.

If MD_RCB_CYCLE takes the value MD_TRUE, the axis chosen cycles through the available dimensions, changing at each level of recursion. For example, in two dimensions, if the first bisection chooses the x-axis, then both of the next two bisections, which occur at the first level of recursion, will choose the y-axis. The next four bisections at the second level of recursion then cycle back to using the x-axis, and so on. This is option 3 presented in section 6.6.1, which we termed there orthogonal recursive bisection, and results in a decomposition like that illustrated in figure 6.26.

If, on the other hand, this parameter takes the value MD_FALSE, then the axis is chosen according to the shape of the mesh being bisected. Each axis is examined and the extent of the mesh in that direction found. The chosen axis is then that with maximum extent i.e. the long axis of the mesh. It is hoped that this results in a minimal cut surface area. This is option 2 presented in section 6.6.1, which we termed there (true) recursive coordinate bisection, and results in a decomposition like that illustrated in figure 6.25.

A further tunable parameter, MD_RCB_FIXED, if set to MD_TRUE results in option 1 from section 6.6.1 being taken. Now the direction of maximum extent is found only once, at the first level of recursion, and is always used. This we previously termed direct coordinate partitioning, and results in a decomposition like that illustrated in figure 6.24. It is an error to set both MD_RCB_CYCLE and MD_RCB_FIXED to MD_TRUE, for the obvious reasons.

7.4.7 Recursive Inertial Bisection

Global algorithm selected by: MD_RIB

Tunable parameters: MD_SEP_IMBAL and MD_SEP_MAX_IMBAL

Having just described RCB, and previously presented the inertial algorithm in section 6.6.2 all we really have to say concerning the implementation of RIB is how the eigensolution of the moment of inertia tensor is carried out. As this is a small real symmetric matrix, eigensolution is straight-forward; we use a standard implementation based on Jacobi transformations [FPTV92]. The

actual implementation of RIB will cope with an arbitrary number of dimensions, but we only expect to ever have to deal with the two and three dimensional cases.

There are no tunable parameters associated specifically with RIB and we do not implement the direct inertial partitioning referred to in section 6.6.2.

7.4.8 Recursive Spectral Bisection

Global algorithm selected by: `MD_RSB`

Tunable parameters: `MD_RSB_TOL`, `MD_RSB_ORTHOG`, `MD_SEP_IMBAL` and `MD_SEP_MAX_IMBAL`

In section 6.7.2 we showed how the basis of spectral partitioning is the evaluation of the Fiedler vector, that is the second eigenvector of the (possibly weighted) Laplacian matrix of the graph. We did not then go into detail as to how this may be carried out, so as to not distract from the derivation of the method. We stated there that this eigensolution could be implemented efficiently and we will now show this to be the case.

We will turn our attention first to what our requirements are, then see that the *Lanczos algorithm* is the method of choice to satisfy them. Subsequently we will look at the stability issues that need to be addressed in any practical Lanczos procedure, finally turning to the details of the PUL-md implementation.

7.4.8.1 Eigensolution

We are faced with the task of calculating a particular eigenvector of the large sparse symmetric matrix $\boldsymbol{L}^w$. If we take the eigenvalues to be ordered such that $\lambda_1 \leqslant \lambda_2 \leqslant \ldots \leqslant \lambda_{n_v}$, with corresponding orthonormal eigenvectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_{n_v}$, then the eigenvector we seek is $\boldsymbol{e}_2$, which is the Fiedler vector.

A complete eigensolution would be wasteful if a more efficient method could be found that would give us just the second eigenpair. Indeed, as $\boldsymbol{L}^w \in \mathbb{R}^{n_v \times n_v}$ and n_v may be in the millions, complete eigensolution is almost certainly out of the question. Additionally, we have the useful knowledge that $\boldsymbol{e}_1 = \boldsymbol{w}$, the vector of the square roots of the vertex weights, and would like to exploit this.

A method that satisfies these requirements is the Lanczos algorithm. The algorithm does not directly yield the eigensolution of $\boldsymbol{L}^w$, but rather a tridiagonal

matrix which is *similar* to it (i.e. has the same eigenvalues). However, the eigensolution of a tridiagonal matrix is straight-forward and may be carried out in linear time by standard methods. The eigenvectors of the tridiagonal matrix may then be mapped back to those of $\boldsymbol{L}^w$, so this an acceptable way to proceed.

For the early part of its history Lanczos was discounted in favour of Householder's tridiagonalisation when a complete eigensolution is required, the latter being more efficient even when the matrix is sparse and much more stable. It is in situations such as the partial eigensolution of $\boldsymbol{L}^w$ that Lanczos wins out, in that only a partial tridiagonalisation need be performed if it is the extreme eigenvalues that are required.

7.4.8.2 The Lanczos Algorithm

There are several ways to motivate the derivation of the Lanczos algorithm, but we shall follow [Par92] and take the Rayleigh-Ritz procedure as our starting point.

Say we have a symmetric matrix, $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, whose eigenpairs we wish to approximate from the m-dimensional subspace $\mathcal{S}^m \subset \mathbb{R}^n$. If a basis for $\mathcal{S}^m$ is given by the vectors $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_m$, so that $span\{\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_m\} = \mathcal{S}^m$, then the Rayleigh-Ritz procedure can be shown to give the best possible approximation to the eigenpairs from the given subspace.

The Rayleigh-Ritz procedure is as follows:

1. Orthonormalise the $\boldsymbol{s}_i$. Call the orthnormalised vectors $\boldsymbol{q}_i$ and write them as the columns of the matrix $\boldsymbol{Q} \in \mathbb{R}^{n \times m}$.
2. Form the (matrix) Rayleigh quotient[3] of $\boldsymbol{Q}$, $\rho(\boldsymbol{Q}) \equiv \boldsymbol{Q}^T \boldsymbol{A} \boldsymbol{Q} \in \mathbb{R}^{m \times m}$.
3. Compute the eigenpairs, $\{\theta_i, \boldsymbol{f}_i\}$, of $\rho(\boldsymbol{Q})$.
4. The approximate eigenvalues of $\boldsymbol{A}$ are then θ_i, with the approximate eigenvectors being given by $\boldsymbol{g}_i = \boldsymbol{Q}\boldsymbol{f}_i$.
5. Compute residual error bounds, which need not concern us here.

The approximate eigenpair $\{\theta_i, \boldsymbol{g}_i\}$ is known as the *Ritz pair*, where θ_i is the *Ritz value* and $\boldsymbol{g}_i$ the *Ritz vector*. As $\boldsymbol{Q}$ is orthogonal, $\rho(\boldsymbol{Q})$ is a similarity

[3]The Rayleigh quotient of any arbitrary $\boldsymbol{x}$, for the matrix $\boldsymbol{A}$, being $\rho(\boldsymbol{x}) \equiv \frac{\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}}{\boldsymbol{x}^T \boldsymbol{x}}$ (a scalar) as we saw in section 6.9.3.

transformation of $\boldsymbol{A}$ if $m = n$, which justifies taking the eigenvalues of $\rho(\boldsymbol{Q})$ as approximations of the eigenvalues of $\boldsymbol{A}$.

Now suppose that we have a subspace $\mathcal{S}^j$ and we wish to expand that subspace to another, $\mathcal{S}^{j+1} \supset \mathcal{S}^j$, in such a way that the new subspace contains estimates of the true extreme eigenvalues of $\boldsymbol{A}$, λ_1 and λ_n, which are improved as much as possible.

The best estimate for λ_n from $\mathcal{S}^j$ is $\theta_j = \rho(\boldsymbol{g}_j)$. We would therefore expect that expanding the subspace to include the gradient of the Rayleigh quotient, $\nabla\rho(\boldsymbol{g}_j)$, would maximise the *increase* in the estimate of λ_n, as $\rho(\boldsymbol{g}_j)$ increases most rapidly in that direction.

In other words, (remembering that $\boldsymbol{g}_j$ is calculated from $\mathcal{S}^j$) we should take

$$\mathcal{S}^{j+1} = \mathcal{S}^j \cup span\{\nabla\rho(\boldsymbol{g}_j)\}.$$

Similarly, the best estimate from the smaller subspace for λ_1 is $\theta_1 = \rho(\boldsymbol{g}_1)$, so we should expand the subspace to include $-\nabla\rho(\boldsymbol{g}_1)$ to maximise the *decrease* in the estimate of λ_1 subsequently calculated from $\mathcal{S}^{j+1}$.

Hence, (remembering that $\boldsymbol{g}_1$ is calculated from $\mathcal{S}^j$) we should have also

$$\mathcal{S}^{j+1} = \mathcal{S}^j \cup span\{\nabla\rho(\boldsymbol{g}_1)\}.$$

We now note that

$$\nabla\rho(\boldsymbol{x}) = \frac{2}{\boldsymbol{x}^T\boldsymbol{x}}(\boldsymbol{A}\boldsymbol{x} - \rho(\boldsymbol{x})\boldsymbol{x}) \in span\{\boldsymbol{x}, \boldsymbol{A}\boldsymbol{x}\},$$

so we may satisfy both these requirements if

$$\mathcal{S}^m = span\{\boldsymbol{s}_1, \boldsymbol{A}\boldsymbol{s}_1, \boldsymbol{A}^2\boldsymbol{s}_1, \ldots, \boldsymbol{A}^{m-1}\boldsymbol{s}_1\}, \tag{7.1}$$

so that

$$\mathcal{S}^{j+1} = \mathcal{S}^j \cup \boldsymbol{A}\mathcal{S}^j.$$

As it will now be the case that

$$\boldsymbol{g}_1, \boldsymbol{g}_j \in \mathcal{S}^j \subset \mathcal{S}^{j+1},$$

and

$$\boldsymbol{A}\boldsymbol{g}_1, \boldsymbol{A}\boldsymbol{g}_j \in \mathcal{S}^{j+1}.$$

We now recognise $\mathcal{S}^m$ as the *Krylov subspace*, $\mathcal{K}^m(\boldsymbol{A}, \boldsymbol{s}_1)$.

We conclude that if we are to apply the Rayleigh-Ritz procedure to a *sequence* of subspaces, such that we get increasingly better estimates for the extreme eigenvalues of $\boldsymbol{A}$, then that sequence should be the sequence of Krylov subspaces $\mathcal{K}^m(\boldsymbol{A}, \boldsymbol{s}_1)$. It is therefore stage (1.) of the Rayleigh-Ritz procedure that is the next obstacle to be overcome; namely the orthogonalisation of $\mathcal{K}^m$.

If we form the *Krylov matrix*, $\boldsymbol{K}_m(\boldsymbol{A}, \boldsymbol{s}_1) = [\boldsymbol{s}_1, \boldsymbol{A}\boldsymbol{s}_1, \boldsymbol{A}^2\boldsymbol{s}_1, \ldots, \boldsymbol{A}^{m-1}\boldsymbol{s}_1]$ and apply Gram-Schmidt orthogonalisation to the columns of $\boldsymbol{K}_m$ then the resulting vectors are know as the *Lanczos basis*. Moreover, if the Lanczos basis is written as the orthonormal matrix $\boldsymbol{Q}_m$ then we have the QR factorisation of the Krylov matrix

$$\boldsymbol{K}_m = \boldsymbol{Q}_m \boldsymbol{R}_m,$$

where $\boldsymbol{R}_m$ is upper triangular.

In general such a factorisation is burdensome, but for the Krylov sequence we may simplify the process considerably to yield a three term recurrence connecting the $\boldsymbol{q}_i$.

We first observe that $\rho(\boldsymbol{Q}_m)$ in the Rayleigh-Ritz procedure is tridiagonal. From the orthogonality of $\boldsymbol{Q}_m$ and the definition of $\mathcal{K}^m$ we know that $\boldsymbol{q}_i \perp \mathcal{K}^{i-1}$ and $\boldsymbol{A}\boldsymbol{q}_j \in \mathcal{K}^{j+1}$. Consequently

$$\boldsymbol{q}_i^T(\boldsymbol{A}\boldsymbol{q}_j) = 0 \quad \forall i > j+1.$$

By symmetry the same is true for $j < i-1$, and so $\rho(\boldsymbol{Q}_m) = \boldsymbol{Q}_m^T \boldsymbol{A} \boldsymbol{Q}_m \equiv \boldsymbol{T}_m$ is tridiagonal.

Let us now denote the elements of $\boldsymbol{T}_m$ as

$$\boldsymbol{T}_m = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \ddots & \ddots & \\ & & \ddots & \ddots & \beta_{m-1} \\ & & & \beta_{m-1} & \alpha_m \end{bmatrix},$$

so that from $\boldsymbol{AQ}_m = \boldsymbol{Q}_m \boldsymbol{T}_m$ we have

$$\boldsymbol{A}\boldsymbol{q}_j = \beta_{j-1}\boldsymbol{q}_{j-1} + \alpha_j \boldsymbol{q}_j + \beta_j \boldsymbol{q}_{j+1} \qquad \beta_0 \boldsymbol{q}_0 \equiv \boldsymbol{0}. \tag{7.2}$$

The orthogonality of the $\boldsymbol{q}_j$ allows us to determine that $\alpha_j = \boldsymbol{q}_j^T \boldsymbol{A}\boldsymbol{q}_j$ after which we can use the recurrence 7.2 to evaluate $\beta_j \boldsymbol{q}_{j+1}$. We may then take β_j as the appropriate normalisation factor and find that we have an iterative procedure for generating $\boldsymbol{q}_{j+1}$ from $\boldsymbol{q}_j$ and $\boldsymbol{q}_{j-1}$ without ever having to make reference to $\mathcal{K}^m$ at any stage.

This process is the *Lanczos algorithm* [Lan50] and is fully described by

$$\begin{aligned} \boldsymbol{v}_j &= \boldsymbol{A}\boldsymbol{q}_j - \beta_{j-1}\boldsymbol{q}_{j-1} \qquad (\beta_0 = 0) \\ \alpha_j &= \boldsymbol{q}_j^T \boldsymbol{v}_j \\ \boldsymbol{z}_j &= \boldsymbol{v}_j - \alpha_j \boldsymbol{q}_j \\ \beta_j &= |\boldsymbol{z}_j|_2 \\ \boldsymbol{q}_{j+1} &= (1/\beta_j)\boldsymbol{z}_j \end{aligned} \tag{7.3}$$

Clearly the algorithm must terminate if $\beta_j = 0$. In *exact* arithmetic this will occur when $j = rank(\boldsymbol{K}_n(\boldsymbol{A}, \boldsymbol{q}_1))$, so that we have found the smallest invariant subspace containing $\boldsymbol{q}_1$ [GL89].

As our application requires only an extreme eigenpairs to be found, and our derivation of the method would indicate that this information will emerge before this stage, we would like to find some better termination criteria.

Firstly, we note that our derivation does not conclusively prove the preferential emergence of extreme eigenvalues, after all $\nabla\rho(\boldsymbol{x})$ may be zero at the points used in the derivation. Nonetheless the convergence theory of Kaniel and Saad [Kan66, Saa80] shows that these values will emerge for j as small as $2\sqrt{n}$. But how do we know when this has indeed happened?

Let us first write 7.2 as

$$\boldsymbol{A}\boldsymbol{Q}_j - \boldsymbol{Q}_j \boldsymbol{T}_j = \beta_j \boldsymbol{q}_{j+1} \boldsymbol{u}_j^T, \tag{7.4}$$

where $\boldsymbol{u}_j = [0, \ldots, 0, 1] \in \mathbb{R}^j$.

If we now consider a Ritz pair $\{\theta, \boldsymbol{g}\}$, we find that we can get an error bound without having to calculate $\boldsymbol{g}$ by observing that

$$\begin{aligned} |\boldsymbol{A}\boldsymbol{g} - \theta\boldsymbol{g}|_2 &= |(\boldsymbol{A}\boldsymbol{Q}_j - \boldsymbol{Q}_j\boldsymbol{T}_j)\boldsymbol{f}|_2 \quad \text{by } \boldsymbol{g} = \boldsymbol{Q}_j\boldsymbol{f} \text{ and } \theta\boldsymbol{f} = \boldsymbol{T}_j\boldsymbol{f} \\ &= |(\beta_j \boldsymbol{q}_{j+1}\boldsymbol{u}_j^T)\boldsymbol{f}|_2 \quad \text{by 7.4} \\ &= \beta_j |\boldsymbol{u}_j^T\boldsymbol{f}| \quad \text{since } |\boldsymbol{q}_{j+1}| = 1. \end{aligned} \tag{7.5}$$

Hence, it is the product of β_j and $\boldsymbol{u}_j^T\boldsymbol{f}$, the bottom element of the normalised eigenvector of $\boldsymbol{T}_j$, which signals convergence for a particular eigenpair.

7.4.8.3 Stability Issues

We emphasised that 7.3 terminates for $\beta_j = 0$ in exact arithmetic, but in finite precision arithmetic this is almost never the case. The root cause of this is that round-off effects completely destroy the orthogonality of $\boldsymbol{Q}_j$. This was known to Lanczos when he first published the algorithm [Lan50], and was a large part of the reason that the algorithm was for so long discounted.

It might be thought that this breakdown of orthogonality would render the algorithm useless. Paige [Pai72], however, found that accurate results could be obtained nonetheless. It was observed that the breakdown of orthogonality is not simply due to the gradual build up of error in the course of the calculation, but was intimately liked to the convergence of a Ritz pair.

The behaviour of the Lanczos algorithm as formulated in 7.3 is to generate $\boldsymbol{q}_{j+1}$ with an unwanted non-trivial component in the direction of any converged Ritz vector. The loss of orthogonality therefore occurs at the same time as the first Ritz pairs start to emerge. If the algorithm is continued past this point it does not fail to produce the correct results, but rather begins to compute unwanted copies of the already converged Ritz pairs.

If this behaviour is to be prevented one solution is *complete orthogonalisation.* This may be done by explicitly orthogonalising each new $\boldsymbol{q}_{j+1}$ against *all* previous $\boldsymbol{q}_j$ using Gram-Schmidt. This is clearly a very costly operation, as the set of vectors we have to consider grows larger with each iteration, and one that we sought to avoid in the very derivation of Lanczos. It is possible to half the cost of complete re-orthogonalisation by storing $\boldsymbol{Q}_j$ factorised into a series of

Householder matrices [GUW72] but the cost remains large.

Parlett and Scott [PS79] made use of the singular way in which orthogonality deteriorates in a scheme known as *selective orthogonalisation.* Here each new $\boldsymbol{q}_{j+1}$ is orthogonalised only against the set of 'good' Ritz vectors, namely those that are close to convergence, as defined by the relation

$$|\boldsymbol{A}\boldsymbol{g} - \theta\boldsymbol{g}|_2 \approx \sqrt{(\text{unit round-off error})}|\boldsymbol{A}|_2.$$

As the set of good Ritz vectors will generally be small in comparison to the set of Lanczos vectors, orthogonalising only against the former is much more efficient. However, it should be noted that calculating the Ritz vectors themselves costs half as much as complete orthogonalisation at a given iteration. The major gain in efficiency of this method comes from the fact that an easily calculated measure of the overall loss of orthogonality may be employed. When, and only when, this measure indicates it is required, the algorithm 'pauses' to update the set of good Ritz vectors. This new set is then used for orthogonalisation at each subsequent iteration until the algorithm is forced by a fresh breakdown of orthogonality to pause once more. As this does not occur very often, selective orthogonalisation is considerably more efficient than complete orthogonalisation.

7.4.8.4 Implementation

As we stated at the beginning of this discussion, we would like to exploit the useful knowledge that the trivial eigenvector of the weighted Laplacian, $\boldsymbol{L}^w$, is $\boldsymbol{e}_1 = \boldsymbol{w}$.

If we are using the Lanczos method for eigensolution then we may effectively deflate the problem by explicitly orthogonalising each new Lanczos vector, $\boldsymbol{q}_{j+1}$ in 7.3, against $\boldsymbol{w}$ as it is generated. This will ensure that the subspace from which we are approximating $\boldsymbol{L}^w$ never contains any component of $\boldsymbol{w}$, so that $\mathcal{S}^{j+1} \perp \boldsymbol{e}_1$. We can therefore reasonably expect that convergence of the Lanczos algorithm at the left hand end of the spectrum will be preferentially to λ_2, allowing us to determine $\boldsymbol{e}_2$ with the minimum of calculation.

This is the approach taken in our implementation of the Lanczos algorithm for spectral bisection. As we only consider unweighted graphs, we simply orthogonalised against $\mathbf{1}$, the vector of all ones.

This leaves open the question of choosing an initial Lanczos vector, $\boldsymbol{q}_1$. In the

absence of any other information a random starting vector is appropriate, but it may well be the case that the vertex numbering in the graph contains useful information which we can also exploit. We follow the recommendation of [PSL90] in this respect and choose $i - (n_v + 1)/2$ as $\boldsymbol{q}_1$, which also ensures that $\boldsymbol{q}_1 \perp \mathbf{1}$, which it clearly must be.

Examining the Lanczos algorithm as presented in 7.3, we see that a major operation our implementation must perform is the product of $\boldsymbol{L}$ with $\boldsymbol{q}_j$. We can perform this operation without ever having to explicitly form $\boldsymbol{L}$, by making direct use of the connectivity information stored in the `neighbours` array given in each `Vertex` data structure. As vertices are only identified by their position in the partition's `vertexPtrs` array, we would not know which vertex we had reached if we only encountered it as the result of following a pointer in another vertex's `neighbours` array. For this reason our implementation of RSB initially records this information in the `vecIndex` field of each `Vertex` data structure that forms part of the current partition. In this way neighbours may subsequently be identified in the course of performing this sparse matrix-vector product.

We also need to consider that fact that, at all levels of recursion except the first, a vertex's `neighbours` array may point to vertices not in the current partition. We could simply check the `index` of each neighbour to see whether it is local to the current partition, but as we expect to perform this matrix-vector product for the Laplacian many times in the course of the Lanczos algorithm this could be expensive. A better option is to sort each vertex's `neighbours` array so that local neighbours are listed first and then only loop over that portion of the array. The number of local neighbours is stored in the `localnNeigh` field of the `Vertex` data structure, and may also be used to determine the diagonal entries in $\boldsymbol{L}$.

Of course, Lanczos does not give the entire eigensolution, only a tridiagonalisation obtained through a similarity transformation, so we still need to solve the resulting tridiagonal for its eigenvalues. To do this we use we use a standard implementation for tridiagonals based on QL factorisation with implicit shifts [FPTV92]. Once we have the eigenvalues, we take the lowest eigenvalue to be our estimate for λ_2 at the current Lanczos iteration and calculation of the corresponding eigenvector follows trivially. We can then use the product of the last element of this eigenvector and β_j to monitor convergence, exactly as detailed in equation 7.5.

We terminate the Lanczos iterations when this product falls below a specified

tolerance. This tolerance may be set via the tunable parameter MD_RSB_TOL, so that the tolerance used is $10^{-\texttt{MD\_RSB\_TOL}}$. We also calculate the final residual of the Ritz pair, which is output to give a true measure of the accuracy of the eigensolution.

Another tunable parameter associated with RSB is MD_RSB_ORTHOG, which controls the orthogonalisation method used. If it takes the value MD_TRUE then complete orthogonalisation by Gram-Schmidt is performed, but if it takes the value MD_FALSE then orthogonalisation is against the trivial eigenvector only. The latter is much faster and will, in most cases, converge satisfactorily. In either case, we store all the Lanczos vectors in memory, although this is not strictly necessary for the latter instance, where they could be placed in a backing store and only recalled after convergence when they are needed in order to calculate the Fiedler vector. We may allocate contiguous blocks of memory for each new Lanczos vector as it is generated, storing them as in simple linked list, and so need not make use of the PUL-md memory manager. It is, however, a very real possibility that we may run out of memory when dealing with very large problems.

The default values of MD_RSB_TOL and MD_RSB_ORTHOG are 5 (tolerance 10^{-5}) and MD_TRUE, respectively.

7.5 Implementation of Local Refinement Algorithms

We implement two local refinement algorithms, Mob and KL, both of which attempt to reduce cut edges while maintaining the existing load balance.

As we are concerned with pair-wise refinement between the two sub-domains resulting from a prior bisection, the gains used by these algorithms are a single integer per vertex which we store in the `gain` field of the `Vertex` data structure. Both algorithms make use of the same definition of vertex gain, namely that previously presented in equation 6.12 of section 6.8.1, during our discussion of KL.

The initialisation and update of vertex gains are handled by the partition manager and related functions. Gains are initialised whenever a new `Partition` data structure is created, either for the original graph by `__dual__` or by

__partn_split__ for subsequent sub-graphs, and are given the value minus the number of (local) neighbours; all vertices are considered to be in one sub-domain prior to bisection, the other sub-domain being initially empty. When a bisection is performed the required vertices are moved to the other sub-domain with the __partn_move__ function. As __partn_move__ updates vertex gains incrementally according to equation 6.13 and vertices are only ever moved using it, gains are always kept up-to-date. Thus all gains are already correct before local refinement commences. During the course of local refinement __partn_move__ is also used to keep track of gains as vertices are swapped from one sub-domain to the other. This process also keeps track of the maxGain field of the Partition data structure, so that $-\texttt{maxGain} \leqslant \texttt{gain} \leqslant \texttt{maxGain}$ for all vertices in the current partition.

7.5.1 Mob

Local refinement algorithm selected by: MD_REF_MOB

Tunable parameters: MD_MOBCOMPLETE, MD_MOBSIZE, MD_MOBSCHED,MD_MOBITERS, MD_MOBCLEANUP, MD_MOBCHOICE and MD_MOBVARY

Having considered how gains are initialised and updated the implementation of Mob largely follows the pseudocode of figure 6.40 in section 6.8.2. There is, however, one simple but significant algorithmic difference that may optionally be taken which leads to quite distinct results.

There are several tunable parameters associated with Mob that can be related to the pseudocode of figure 6.40 and our previous discussion of the algorithm. Firstly, MD_MOBITERS determines the number of iterations the algorithm should perform, as given in line 2 of the pseudocode. Secondly, the length of the mob schedule is determined by MD_MOBSCHED, as given by $len^{schedule}$ previously. Thirdly, the initial mob size in the schedule is determined by MD_MOBSIZE, which equates to $MOB_1^{schedule}$. These last two parameters completely determine the mob schedule, as the last entry in the schedule is always one and intermediate values linearly decrease.

We implement two versions of the algorithm, both of which have the previous parameters in common. The first version is exactly that presented in the pseudo-

code of figure 6.40. We note that in the inner loop of that pseudocode s (the index into the schedule) will only ever reach $len^{schedule}$ if a worse partition results from *every* exchange of mobs; an unlikely and unhelpful event. This means that the full schedule is not used and last mobs swapped will (depending on the schedule) probably be quite large. They are therefore likely to have contained at least some vertices whose movement was detrimental to the partition.

In view of this observation, we offer an alternative version of the algorithm that is always guaranteed to reach the end of its schedule, which we call *Mob Complete.* This version is selected by setting the tunable parameter `MD_MOBCOMPLETE` to the value `MD_TRUE` and follows the pseudocode of figure 7.3 below.

```
Pseudo: MOB
 1.  Create the mob schedule, MOB_s^schedule,
       where s = 1, len^schedule.
 2.  For required number of iterations
 3.     s = 1
 4.     Repeat
 5.        e = |E_cut|_e
 6.        MOB_0 = chooseMOB(MOB_s^schedule, S_0)
 7.        MOB_1 = chooseMOB(MOB_s^schedule, S_1)
 8.        Move MOB_0 to S_1 and MOB_1 to S_0.
 9.        Update gains.
10.        If |E_cut|_e ≥ e Then
11.           s += 1
12.        EndIf
13.     Until s = len^schedule
14.  EndFor
EndPseudo
```

Figure 7.3: The Mob Complete algorithm.

While the pseudocode of figure 6.40 was such that exactly `MD_MOBSCHED`×`MD_MOBITERS` mob exchanges always took place, now the number of exchanges is not explicitly limited. However, we now use $|E_{cut}|_e \geqslant e$, rather than $|E_{cut}|_e > e$, as the test to increment s. This prevents the cyclic swapping of vertices that result in no improvement and ensures that the inner loop always terminates.

The differences in the behaviour of the original Mob algorithm (which is selected when `MD_MOBCOMPLETE` takes the value `MD_FALSE`, its current default) and the Mob Complete version will be further detailed in chapter 8, where we come to evaluate the effects of tuning these parameters.

As well as the options that control the top-level structure of the algorithm, there are two tunable parameters which relate to the function *chooseMOB* in the

pseudocode, namely MD_MOBCHOICE and MD_MOBVARY.

Setting the tunable parameter MD_MOBCHOICE to MD_TRUE causes vertices in the Pre-Mob to be placed in the following categories:

Category 1: all edges cut (isolated vertices).

Category 2: some edges cut (border vertices).

Category 3: no edges cut (interior vertices).

Now, when we come to select vertices from the Pre-Mob to form the actual mob to be swapped, we first choose from those in category 1, then from those in category 2 and finally from those in category 3. As we only choose a sub-set of the Pre-Mob, this means that we are preferentially choosing the vertices in low-numbered categories, which we hope will be the best candidates for swapping.

If this option has been chosen then the MD_MOBVARY tunable parameter allows the mob size to vary according to the number of 'good' candidates available for swapping. By restricting the choice of vertices from the Pre-Mob to those vertices in categories 1 and 2 only, two mobs of less than the size specified in the schedule may be swapped. This is done in such a way that an equal mob size is still taken from each half of the bisection, so load balance is maintained.

The default values for MD_MOBCHOICE and MD_MOBVARY are both MD_FALSE. MD_MOBVARY has no effect unless MD_MOBCHOICE takes the value MD_TRUE.

The last tunable parameter associated with the Mob algorithm is MD_MOBCLEANUP. If set to MD_TRUE then - after Mob refinement - we check for isolated vertices (i.e. those category 1) in the entire partition and swap their sub-domain assignment. This may result in an unbalanced partition, so the default is MD_FALSE.

7.5.2 Kernighan and Lin

Local refinement algorithm selected by: MD_REF_KL

Tunable parameters: MD_KL_TERM_OBJ, MD_KL_TERM_FAILS, MD_KL_TERM_FAILS_MAX, MD_KL_BORDER_ONLY, MD_KL_BORDER_SIZE, MD_KL_RANDOM_RETRIES

Our implementation of KL retains the overall structure of the algorithm as we presented it in the pseudocode of figure 6.35 in section 6.8.1 but (unlike Mob)

differs in many of its details. As we saw in section 6.8.1, the inner loop runs through a *pass* of the algorithm, during which pairs of vertices are exchanged between the two sub-domains in the bisection based on their gains, and the best partition found is recorded. The best partition found in the previous pass is then used as the starting point of another, and so on, until no further improvement can be found. This basic structure is maintained in our implementation.

However, as it would obviously be too expensive an operation to 'record' the entire partition by simply copying it as we notionally did in line 10 of the previous pseudocode, we need to employ some sort of incremental scheme to keep track of the progress of the algorithm. In the previous section we also referred the Fiduccia and Mattheyses (FM) linear time implementation of KL which we make use of. The FM implementation is based, as we shall see, on a particular data structure in which vertices are stored according to their gain, and so we can make use of this to essentially keep a copy of the current state, distinct for that defined by the partition data structure.

Our implementation is detailed in the pseudocode of figure 7.4, where we see that the bisection defined by the partition data structure, P, lags behind that defined by the FM data structure, G^{FM}, and is only brought up-to-date with it when a better state is found (line 8 of the pseudocode). It may be helpful to look again at figure 6.37, which showed the progress of the KL algorithm; we may think of P as taking the path illustrated by the bold line in that graph, and of G^{FM} as taking the paths illustrated by the fine lines on each pass.

We shall detail the FM data structure shortly, but first we will look at how the *FM_update* function is employed in our implementation.

The *FM_update* function is called twice in the pseudocode, once for each sub-domain in the bisection. Each call selects a vertex in the specified sub-domain that has maximum gain, removes it from the FM data structure so that it is not reconsidered in the course of the current pass, and updates the gains of its neighbours in the usual manner. This means that the selection of the second vertex in the pair to be swapped takes into account the movement of the first, which will have altered the gain of the second if the pair are neighbours. This falls somewhere between the options of maximising $g_i + g_j$ or $g_i + g_j - 2w_e(e_{ij})$, which we examined in section 6.8.1. Our experience is that this is superior to simply selecting two vertices without considering how the movement of one effects the other, particularly if the vertices are presented to the algorithm in a highly

Pseudo: Kernighan and Lin

1. Let P be the initial `Partition` data structure.
2. **Repeat**
3. Initialise the `FM_Gain` data structure, G^{FM}, based on P.
4. **Repeat**
5. Call $FM\_update(G^{FM}, 0)$
6. Call $FM\_update(G^{FM}, 1)$
7. **If** $|E_{cut}|_e$ for $G^{FM} < |E_{cut}|_e$ for P **Then**
8. Bring P up-to-date with G^{FM}.
9. **EndIf**
10. **Until** G^{FM} is empty **or** termination criteria met.
11. **Until** No better partition found.

EndPseudo

Function: $FM\_update(G^{FM}, i)$

1. Choose v_i from sub-domain S_i induced by G^{FM} such that g_i is at a *maximum.*
2. Move v_i other sub-domain and remove it from G^{FM} vertex lists.
3. Update G^{FM} to reflect altered gains of neighbours of v_i.
4. **Return**

EndFunction

Figure 7.4: The Kernighan and Lin algorithm.

ordered manner, as they will be if they have previously been sorted by separator field, which increases the probability that a pair which are neighbours will be selected.

As we also saw in section 6.8.1, for the algorithm as a whole to run in linear time with respect to number of vertices, n_v, the selection of vertices and update to neighbours which we have now embodied in the FM_update function must run in constant time. Given that we expect there to be a limit on n_e^{max}, the maximum number of neighbour per vertex, imposed by the origin of the dual graph as a representation of an unstructured mesh, this can be achieved by use of the FM data structure[4].

Our version of the FM data structure has type `FM_Gain`, and is defined as follows:

```
typedef struct {
    DItem *item_loc;
    Item_Data *item_data_loc;
    int nBuckets;
    Bucket *bucket;
    int maxGain;
} FM_Gain;
```

[4]If there is no limit on n_e^{max}, then n_e is a more appropriate measure of the size of the problem than n_v as overall complexity will still be linear in that measure. The distinction need not concern us here.

```
typedef struct {
    int best_non_empty;
    DList *gain_list;
} Bucket;

typedef struct {
    Vertex *vertex;
    int gain;
} Item_Data;
```

How this structure is used is illustrated in figure 7.5. The basic idea originated by Fiduccia and Mattheyses is to maintain a set of lists of vertices which are bucket sorted according to their gain. This is done for both sub-domains in the bisection, so selecting a vertex with maximum gain from either one is simply a matter of taking an arbitrary vertex from the appropriate list, so long as we have kept track of the non-empty list which currently has highest gain.

In our data structure a `Bucket` is associated with each sub-domain and has a field, `best_non_empty`, which keeps track of this information. The `gain_list` field in each `Bucket` points to an array of doubly linked lists, of type `DList`, dimensioned from $-$`maxGain` to $+$`maxGain`, as defined by the `FM_Gain` structure. The `Bucket`'s themselves are given as an array pointed to by `FM_Gain`, so that we might easily extend the data structure for l-way refinement in the manner described in section 6.8.1.2 and reference [HL93a]. In that instance there are $l(l-1)$ types of move and we would therefore require $l(l-1)$ `Buckets`. For bisection we only require 2, as indicated by the `nBuckets` field in `FM_Gain`.

Selecting a vertex with maximum gain from a given `Bucket` is therefore simply a question of going to the `DList` indicated by `best_non_empty` and taking a `DItem` list member from it. We always remove items from the tail of a list, for reasons that will become clear. As we do not replace the selected vertex into `FM_Gain` subsequent to moving it, we never reconsider a vertex during the course of a pass.

The `DItem` points to the `Item_Data`, which specifies both a vertex and its gain and, as we can see from figure 7.5, the `gain` given in the `Vertex` need not be the same as that given in the `Item_Data`. Further, the `index` for the `Vertex` need not indicate that it is part of the same sub-domain that its assignment to a particular `Bucket` would indicate. In this way we are able to entirely define a bisection and related gains by the `FM_Gain` structure; this has no relation to the bisection defined through the `Vertex` items in the `Partition` structure.

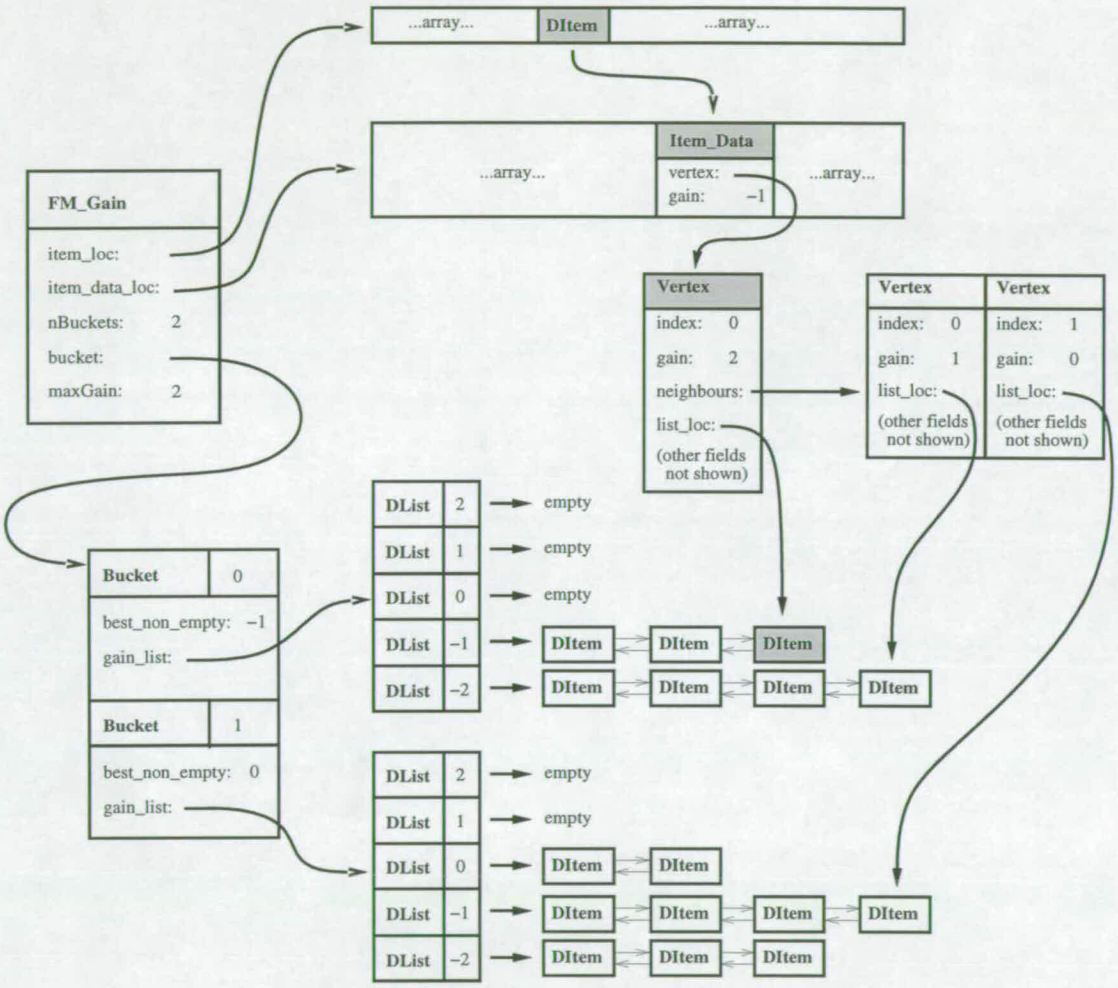


Figure 7.5: The FM_Gain data-structure. The data in the shaded boxes all relates to the same vertex.

Note that the `item_loc` and `item_data_loc` fields in `FM_Gain` are only used for allocation and deallocation, and that the data in the shaded boxes in the figure all relates to the same vertex. Thus, the two shaded `DItems` are in fact the same data.

Selecting and moving a vertex proceeds as we have just outlined, but we still need to update the gains of its neighbours and move them in the `gain_lists` to reflect any changes. If a vertex is moved we consider each of its `neighbours` in turn. As the `list_loc` for each neighbour gives the position of its associated `DItem` in the lists, and the `DItem` itself points to the `Item_Data` which in turn gives the neighbour's `gain` (as far as the `FM_Gain` structure is concerned) we know both where to find the appropriate list item and the list it is a member of. We may therefore remove the neighbour's `DItem` from its current location, recompute its `gain` and add it to the *tail* of the list associated with that gain. As a neighbour may be anywhere in a `DList`, we have to use a doubly-linked list, rather than singly-linked, or we would not be able to remove it without corrupting the list.

If at any point removing a vertex from a particular `Bucket` or moving its neighbours within the `Bucket` results in a change in the maximum gain, then the `Bucket`'s `best_non_empty` field is changed accordingly. Also, we always set a `Vertex`'s `list_loc` to `NULL` when it has been selected for movement and removed from the `FM_Gain` structure. This ensures that if we subsequently encounter it as the neighbour of another vertex we do not inadvertently return it to the data structure.

We emphasise that we always select a vertex for movement from the tail of the most favourable list and that, when we have removed neighbouring vertices in order to update their gain, we always append them to the tail of the list appropriate to their new gain. This means that when we next come to select a vertex for movement it is quite likely that we shall select a neighbour of the previous selection. Thus, the implementation tends to move clumps of adjacent vertices, even though this is not explicitly specified.

Having described our implementation, we are now in a position to describe the various tunable parameters associated with it and see how they affect its behaviour.

The first group of tunable parameters we shall look at affect the termination of a pass, as given in line 10 of the pseudocode of figure 7.4. Referring again to figure 6.37 in section 6.8.1, we see that the majority of vertex moves do not result

in productive work being done. If we can set some criteria that will determine when in a pass we think it unlikely that further improvement will occur, we can avoid excessive redundant computation taking place.

The first option we implement is selected by setting the tunable parameter `MD_KL_TERM_OBJ` to the value `MD_TRUE`. In this case we terminate a pass if the number of cut edges for the bisection defined by the `FM_Gain` structure rises above the value it took at the beginning of the pass plus `maxGain`.

The second option is selected by setting `MD_KL_TERM_FAILS` to the value `MD_TRUE`, in which case we terminate a pass if a specified number of consecutive (individual) vertex moves fails to produce any reduction in cut edges. The number of vertex moves is itself a tunable parameter, namely `MD_KL_TERM_FAILS_MAX`, and is given as a percentage of the size of the current `Partition`.

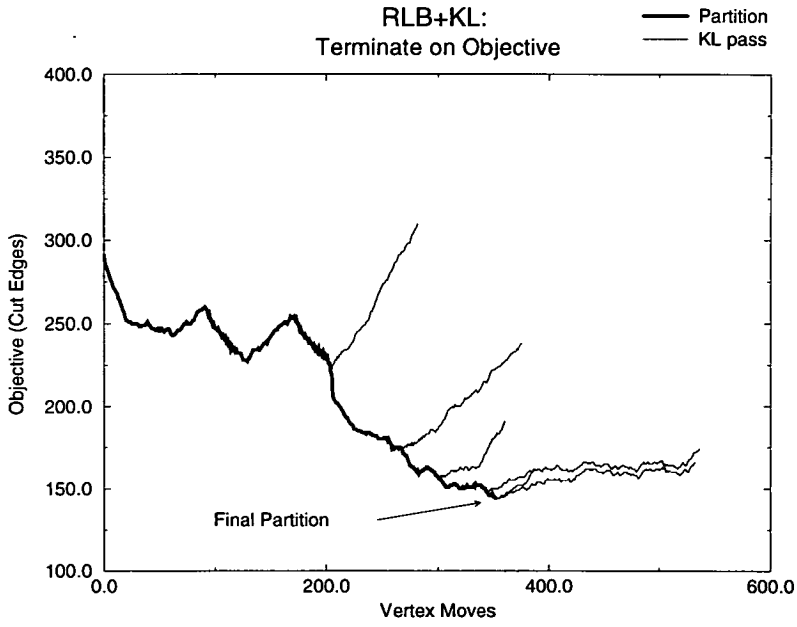


Figure 7.6: Progress of the KL algorithm from an initial layered (RLB) bisection. Each pass is terminated according to cut edges.

The effects of employing these termination criteria are illustrated in figures 7.6 and 7.7. The data-set and initial bisection is the same as was used in figure 6.37, and the same improvement in cut edges also results from KL refinement in these two new cases. Figure 7.6 shows the effect of setting `MD_KL_TERM_OBJ` to the value `MD_TRUE`, while figure 7.7 shows the effect of setting `MD_KL_TERM_FAILS` to the value `MD_TRUE` and `MD_KL_TERM_FAILS_MAX` to 5%. For robustness the default values of these parameters are `MD_FALSE` for `MD_KL_TERM_OBJ`, while

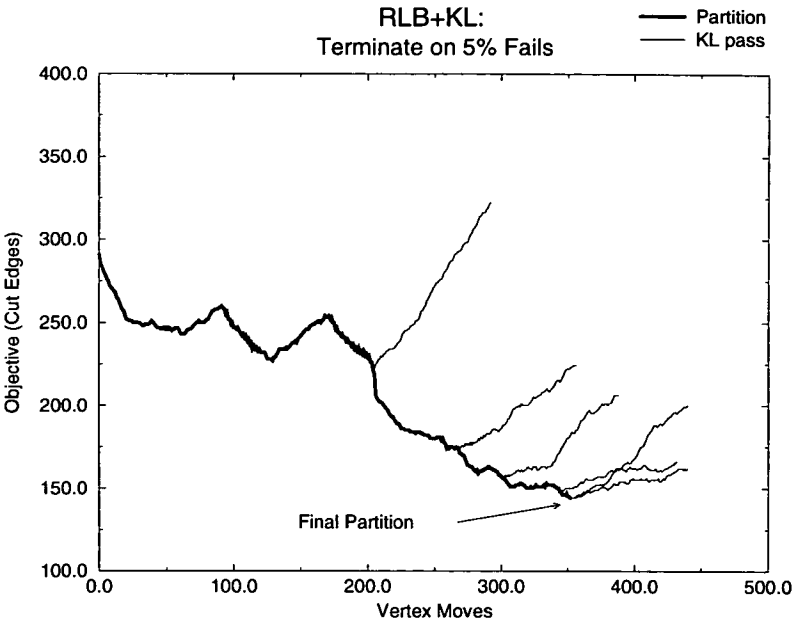


Figure 7.7: Progress of the KL algorithm from an initial layered (RLB) bisection. Each pass is terminated according to the number of consecutive vertex moves that failed to find an improvement.

`MD_KL_TERM_FAILS` defaults to `MD_TRUE` and `MD_KL_TERM_FAILS_MAX` to 20%. In most cases faster setting may be used.

An unrelated tunable parameter is `MD_KL_RANDOM_RETRIES`, which allows the introduction of a certain amount of randomisation into the order of the `gain_lists` so that the algorithm may attempt another pass even if the previous one resulted in no improvement, in violation of line 11 of the pseudocode of figure 7.4. As the order in which vertices are chosen is determined by their order in the `gain_lists`, which is totally arbitrary, it is quite possible for us to change this order and allow the algorithm to escape from a state which may only be locally optimal, potentially giving it the opportunity to find a better configuration.

We implement this by going through each `DList`, randomly alternating between the list's head and tail and removing a certain number of vertices. The order of the removed vertices is then randomised before they are placed back in the list by appending them to its tail. This is considerably more economic that randomising the entire list and, we hope, has a similar effect, as it is focusing the randomisation on the tail of the list which is precisely where vertex selection occurs. The number of vertices removed and randomised is fixed (i.e. not tunable with `md_tune`) at compile time by the symbolic constant `N_RAND_ITEMS` which

currently takes the value 40.

The tunable parameter MD_KL_RANDOM_RETRIES determines the number of consecutive times a pass may result in no improvement. If it takes the value 0 (its default) then no randomisation is ever performed and the algorithm terminates as normal. If it takes a value greater than zero then the gain_lists are always randomised before each pass and the specified number of consecutive unproductive passes are permitted.

The final two tunable parameters associated with KL make use of the information provided by prior separator field based bisection. If such a bisection was employed then the vertexPtrs array will have been sorted by separator field value and we can use this to determine the border region surrounding the initial bisection boundary.

This is illustrated in figures 7.8 and 7.9. These figures show the border region for the Widget mesh when partitioned by RCB and RSB, respectively. In both cases the border region contains 18% of the mesh elements. With coordinate bisection the border can be seen to be defined by two lines (planes in 3D), one to either side of the sub-domain boundary and parallel to it. This will also be the case for inertial bisection (not illustrated), although the lines (planes) will no longer be aligned with the coordinate axes. For spectral bisection the situation is less intuitive, but may be thought of as analogous to taking an isosurface through the separator field.

As we can expect the majority of improvements to the bisection made by KL to be in the vicinity of the initial bisection boundary, it may be beneficial to restrict the operation of the algorithm to the border region surrounding it. We can implement this easily by initialising the FM_Gain structure with vertices in the border region only. These may be identified by simply indexing the appropriate range in the sorted vertexPtrs array. Even if an unbalanced initial partition has been generated as a result of setting MD_SEP_IMBAL to MD_TRUE, we may still take this approach as the sep_vertex field of the Partition data structure records where the cut-off point between the two halves of the bisection lies in the vertexPtrs array.

The benefits of restricting KL to this border region are two-fold; a reduction in the storage required by the FM_Gain structure and a potential decrease in runtime. The former is self evident but we shall examine the latter in more detail in chapter 8.

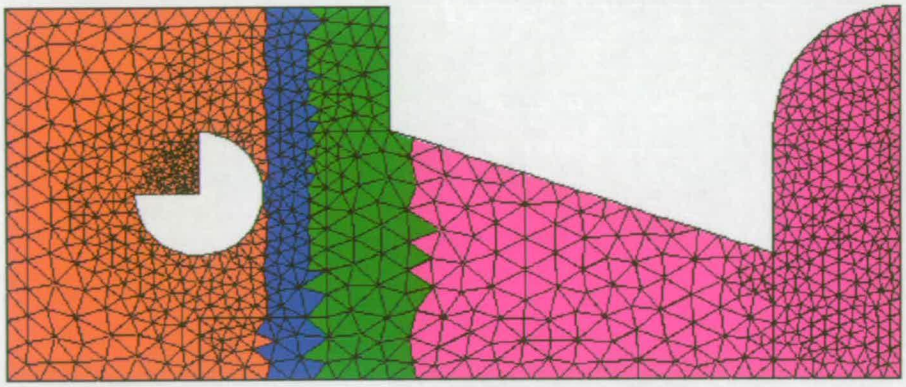


Figure 7.8: The KL border region defined by taking the x-coordinate (horizontal) as a separator field for the Widget data-set. The border region contains 18% of the mesh.

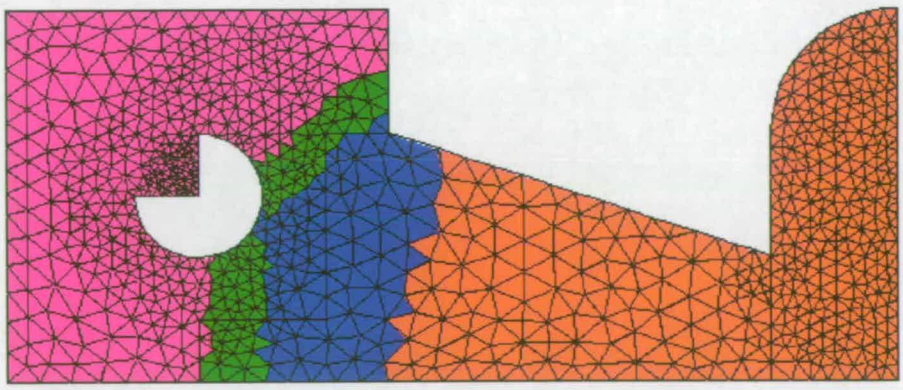


Figure 7.9: The KL border region defined by taking the Fiedler vector as a separator field for the Widget data-set. The border region contains 18% of the mesh.

If this option is to be taken then the tunable parameter `MD_KL_BORDER_ONLY` should be set to the value `MD_TRUE`. If this is done then the number of vertices in the border may be specified through the parameter `MD_KL_BORDER_SIZE`, where the number of vertices is given as a percentage of the size of the current `Partition`. The default values of these two parameters are `MD_FALSE` and 20%, respectively.

7.6 Visualisation

A utility has been included in PUL-md to provide a simple interface to the popular AVS visualisation package, allowing visualisation and analysis of both mesh decomposition and application data in either 2 or 3 dimensions. This utility consists of a file translation program, `mdesc2ucd`. It takes as input a mesh structure file, a mesh decomposition file and/or an application data file. As output it writes a file in AVS unstructured cell data (*ucd*) format.

The AVS file format includes a material type which may be used to indicate processor assignment for the mesh elements (*cells*, in AVS parlance). The mesh can then be 'exploded' into its component sub-domains, each defined by their different material type.

`mdesc2ucd` takes the following command line arguments (in order):

- `-nodata`, `-nodecomp`, `-order ijk`; options respectively indicating not to read data, not to read decomposition and to reorder element nodes, as specified. Other, application specific, options are also available.
- `baseFileName`, the mesh structure filename root.

Like the `md_test` program, it looks for a standard mesh structure file in `baseFileName.mdesc`, but additionally looks for a decomposition in `baseFileName.decomp` and/or application data in `baseFileName.data`. It then writes its output to `baseFileName.inp`, ready for input into AVS.

Example visualisations are shown in figures 7.10 to 7.14. These illustrate two data-sets; the Wedge3 data-set and the m6 data-set, both tetrahedral finite element meshes derived from the FLITE3D aerospace CFD project [BMT96]. We will use these or closely related meshes as examples in our subsequent evaluation of decomposition algorithms, and further details of their origin and structure may

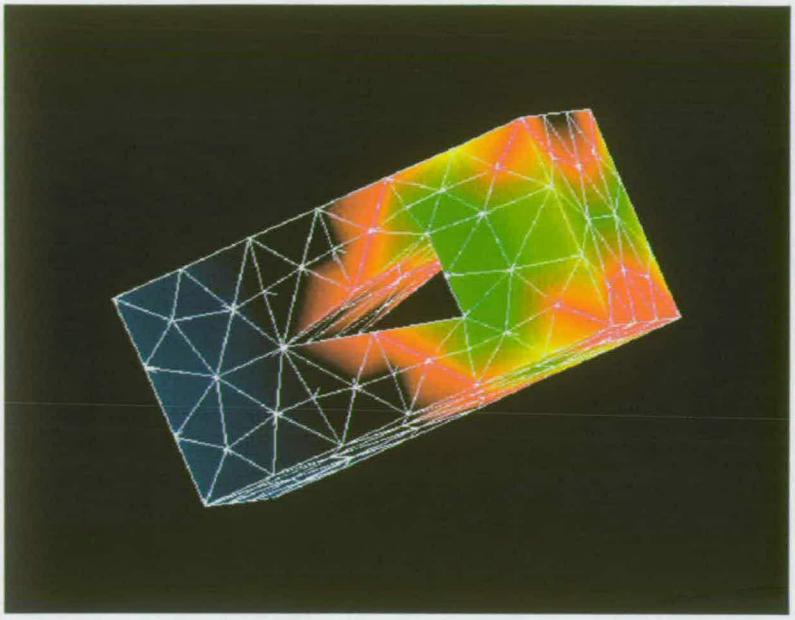


Figure 7.10: The Wedge3 data-set, showing surface mesh and flow solution.

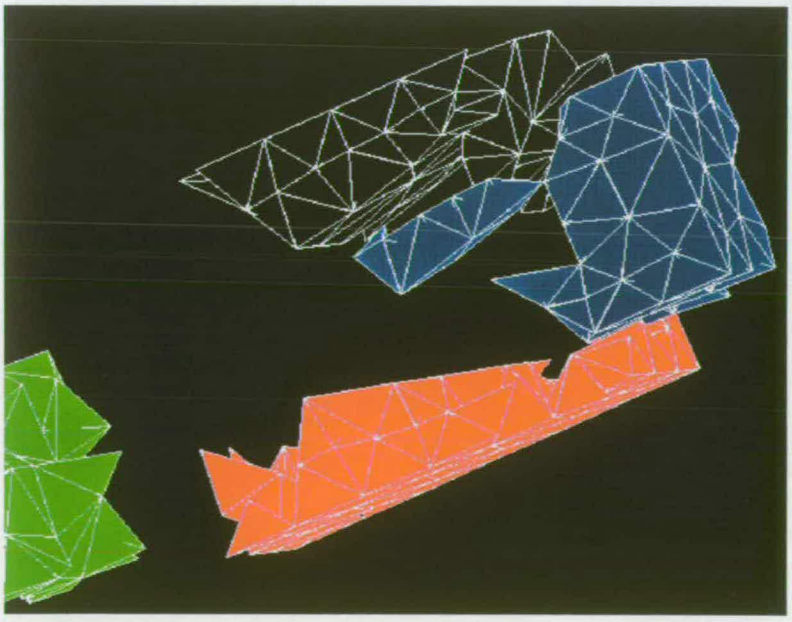


Figure 7.11: The Wedge3 data-set, showing decomposition into 4 sub-domains.

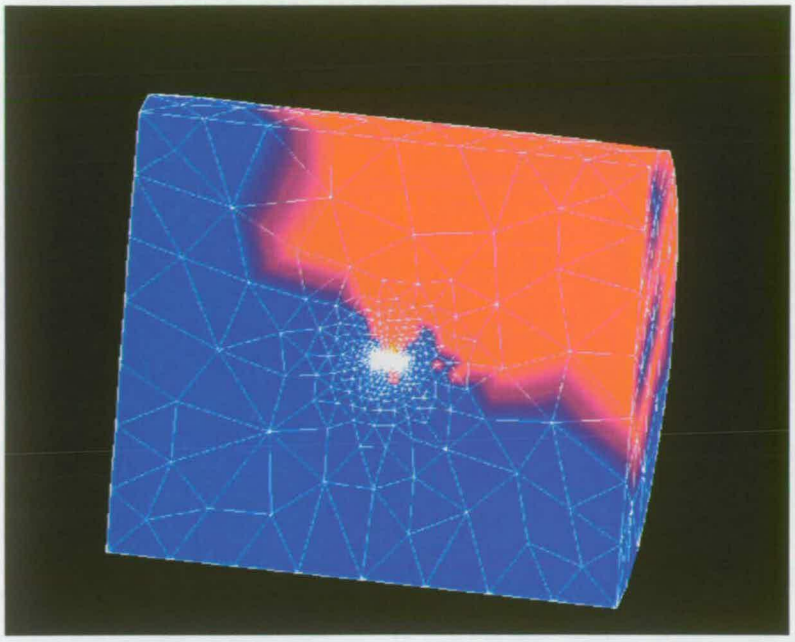


Figure 7.12: The m6 data-set, showing surface mesh of the entire simulation domain together with flow solution.

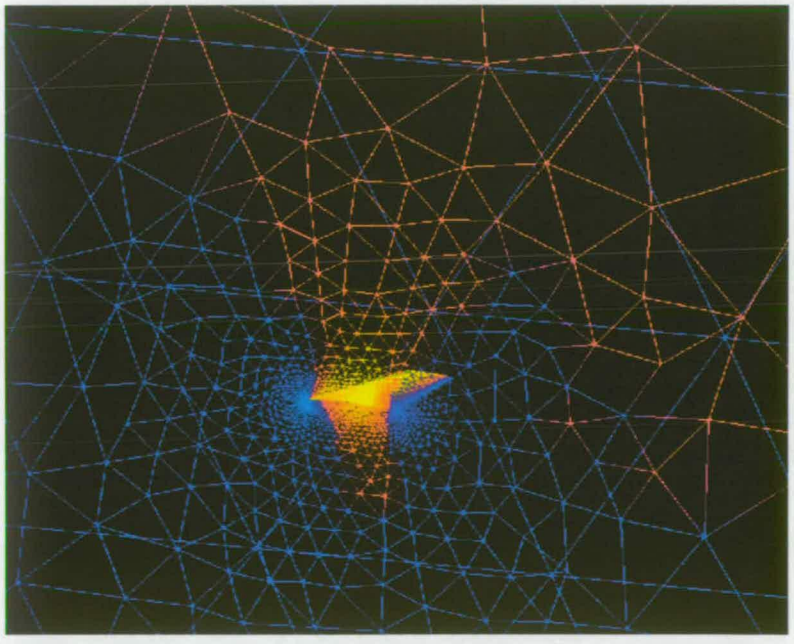


Figure 7.13: The m6 data-set, showing a wire-frame view of the m6-wing and surrounding surface mesh.

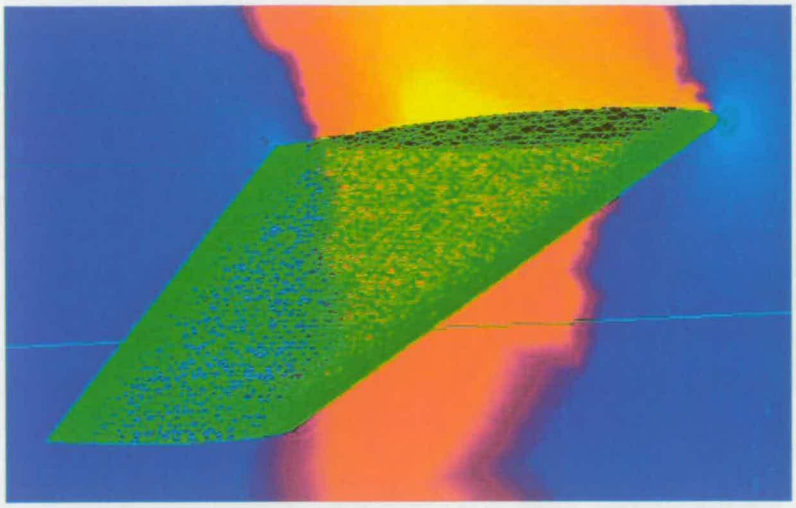


Figure 7.14: The m6 data-set, showing a close up of the m6-wing and a slice-plane perpendicular to it through the flow solution.

be found in appendix A, where we present statistics relating to the meshes, their dual-graphs and decomposition by a variety of algorithms. It should be noted that the earlier visualisations of the Widget data-set originate from a simple Open-GL based tool developed specifically for the HEAT2D demonstration code, not from AVS as (in two dimensions) this is somewhat more convenient.

Chapter 8

Evaluation and Discussion of Decomposition Algorithms

Having detailed the algorithms implemented in PUL-md together with the range of tunable parameters associated with them in the previous chapter, we now move on to evaluating their relative merits. We base this evaluation on a thorough exploration of the various combinations of algorithms (initial decomposition and subsequent refinement) and associated parameter settings, tabulating the resulting decomposition statistics in appendix A. Here we will discuss and evaluate these results, referring to the relevant tables in the appendix as necessary.

The results presented in appendix A are derived from three data-sets; the Widget data-set, the Wedge1 data-set and the m6 data-set. Each of these data-sets is described at the beginning of the relevant section of appendix A, although the Widget data-set is already familiar to us, while visualisations of the Wedge3 and m6 data-sets were presented in figures 7.10 to 7.14 in section 7.6 of the previous chapter.

Before discussing these results, we must first describe the data we have gathered (metrics of quality, execution times, etc.), its derivation and the form of its presentation in appendix A.

8.1 Collection and Presentation of Results

If required, the `md_decompose` function can provide statistics relating to decomposition quality; whether it does so or not is determined by the user's choice of

compile time options. The `md_test` program is compiled so as to provide these statistics, and it is from this that we derive our data.

A typical example of the output of the `md_test` program in this respect is as follows:

```
Dualgraph: total vertices 1746, total edges 10072
           neighbours: min 5, avg 11.54, max 16
Domain  Domain-Size   Bdry-Vertices   Bdry-Cuts   Adj-Domains (which)
   0         354             24          80      1 ( 1 )
   1         432             51         194      3 ( 0 2 3 )
   2         528             55         165      2 ( 1 3 )
   3         432             57         205      2 ( 1 2 )
=============================================================
tot:        1746            187         644      8
result was 1
```

Examining this example, we see that statistics relating to the dual graph are presented first, followed by statistics relating to the quality of its partition into 4 sub-domains.

The dual graph statistics, `total vertices` and `total edges`, equate to $n_v = |V|_v$ and $n_e = |E|_e$ in our notation[1]. The `neighbours` line then gives the minimum, mean and maximum number of neighbours for a vertex in the dual graph, which we shall denote n_e^{min}, n_e^{mean}, n_e^{max}, respectively. For each of the three data-sets these statistics are tabulated (tables A.1, A.16 and A.31), together with timings for the extraction of the dual from the mesh; we shall say more on the subject of timing figures shortly.

Turning to the decomposition statistics, we see that the sub-domains are identified as `0`, `1`, `2`, `3` under the first column, which is labelled `Domain`. The `Domain-Size` column then gives $|S_l|_v$ for each sub-domain S_l. The next column, `Bdry-Vertices`, gives $|\{v_i \in S_l : \exists e_{ij} \in E_{cut}\}|$, which is the number of vertices in the sub-domain which have at least one edge connecting them to a vertex in another sub-domain; i.e. those that are on the sub-domain boundary. Cut edges for each sub-domain are given under the `Bdry-Cuts` column, which is simply defined as $|\{e_{ij} \in E_{cut} : v_i \in S_l\}|_e$. Finally, the number of adjacent sub-domains is listed for each sub-domain under the `Adj-Domains` column, where sub-domain S_m is considered to be adjacent to S_l if $\exists e_{ij} \in E : v_i \in S_l$ and $v_j \in S_m$; the sub-domains in question are then listed in brackets at the end of the row.

[1]Note that, for the unweighted dual graphs used by PUL-md, $|V|_v = |V|$ (and similarly for edges) but we retain the sub-scripts for clarity.

The totals for these quantities are listed in the row beginning `tot`. Clearly the total under `Bdry-Cuts` will be $2|E_{cut}|_e$, as each cut edge will have been counted twice, once for each of the two sub-domains it connects.

From the `Domain-Size` column we may derive a measure of load imbalance, $\Delta_s \equiv max(|S_l|_v) - |V|_v/k$, by noting that run-time of an application using a given decomposition will depend on load balance through the sub-domain that requires most calculation, as measured by $max(|S_l|_v)$, and that for a perfectly balanced partition $max(|S_l|_v) = |V|_v/k$ and Δ_s is therefore zero.

We are now in position to define the quantities tabulated in the tables relating to decomposition presented in appendix A; these are Δ_s, which we have just defined; $|V_b|$, which is the total for the `Bdry-Vertices` column; $|E_{cut}|_e$, which requires no further explanation; and s_{adj}, the total for the `Adj-Domains` column.

It may now be helpful to look ahead to table A.8 in appendix A, as the example data presented above forms part of that table. In the example, we employed RSB with no subsequent refinement to partition a node-based dual graph derived from the Widget data set into 4 sub-domains. We have set `MD_SEP_IMBAL` to `MD_TRUE` and permitted a maximum imbalance of 5% in each bisection. Complete orthogonalisation was not employed, and Lanczos termination was set with `MD_RSB_TOL` $= -3$.

Examining the relevant table, we see that the algorithm used is listed at its head, while the associated parameters are listed at the left-hand side, where 'T' indicates that the parameter takes the value `MD_TRUE`, 'F' the value `MD_FALSE`, and '-' a setting which is not relevant. The metrics we have just described are then given for both node- and edge-based (and in three dimensions, face-based) dual graphs of the data-set, with the resulting data split up into blocks according to the number of sub-domains, k, in the partition.

Each row in the table is identified as a different 'case', as this determines the parameter settings used and number of sub-domains requested, and therefore completely determines the behaviour of the decomposition algorithm(s) in question. Comparing table A.8 with the description of the parameter setting for the example used here, we see that it forms case 9 of that table.

For convenience, where we henceforth require to refer to a set of statistics for a particular numerical experiment, we shall use an abbreviation of the form A.8:9, to refer to case 9 of table A.8, and further add the suffix 'n' for the node-based graph, 'e' for the edge-based graph, and (in three dimensions) 'f' for the face-

based graph. Thus our example would be A.8:9n.

As well as the graph and decomposition statistics, the tables in appendix A also include timing information. All runs of the program were performed on a Sun Ultra Enterprise 3000 server with 1024Mb of main memory. This system is based on the UltraSPARC-II 64-bit RISC processor, running at a clock rate of 250MHz. The code was compiled with the native SunSoft C compiler and optimised for this system[2]. The timings were made by inserting calls to the `getrusage` C library function at the relevant points in the `md_decompose` function and calculating amount of time spent executing in user mode to microsecond accuracy. These points are marked as `/* Timing point A */`, `B` and `C` of the code presented in section 7.3.1.

The timings for dual graph extraction presented in tables A.1, A.16 and A.31 are taken as the time elapsed between timing points `A` and `B` and therefore represent the time spent in the `__dual__` function. Examining these tables we see that times marked 'Coord,' 'Border,' and 'Other' are listed, which represent the slightly different requirements of the decomposition functions. RCB and RIB both require coordinate information which is not otherwise calculated and resulting timings are listed under 'Coord'; GREEDY and RLB both may require an initial seed vertex on the graph border, and an identification of such vertices is only made in this instance, as listed under 'Border'; finally 'Other' gives the timings when no additional information is required.

The timings for decomposition presented in the remaining tables are taken as the time elapsed between timing points `B` and `C`, and so represent the time spent in the chosen decomposition function. Note that, as the refinement routines are called from within the decomposition functions, this timing represents the sum of all initial bisections *and* subsequent refinement steps.

8.2 Analysis of Results

We will now consider each algorithm in turn, and see how the setting of the associated tunable parameters affects the quality of the resulting partition, also comparing the relative merits of the algorithms as we go along. Before doing so, we will discuss the dual graphs that may be extracted from the three example

[2]We found `cc -fast -xO4 -xdepend -fsimple=2 -xarch=v8plusa` most effective.

meshes we study and the consider the implications the process of graph extraction has for the performance of the library as a whole.

8.2.1 Dual Graph Statistics

As the time taken for dual graph extraction depends both on the *type* of graph required (nodes, edges or faces) and on the decomposition algorithm that we intend to use, we must discuss the cost of this process as well as that of the decomposition itself; indeed, the two should not be viewed in complete isolation.

For a given dual graph type, the timings in tables A.1, A.16 and A.31 show that the added cost of calculating coordinate or border information is small compared to the cost of basic graph extraction. However, there is a much more significant variation in the runtime between the different graph types, with (for the three dimensional data-sets) an edge-based graph taking almost twice as long to extract as faces, and nodes taking almost five times as long[3].

Previously we have discussed the computational complexity of partitioning primarily in terms of n_v, having used the justifiable claim that n_e is proportional to n_v for graphs extracted from unstructured meshes. Although this simplified the comparison of the asymptotic behaviour of algorithms with increasing problem size, it ignores the fact that many of the algorithms that we have discussed have runtimes dominated by n_e, and that the constant of proportionallity between n_v and n_e depends on the type of graph we are dealing with.

Examining the values of n_e in the relevant tables shows that there is a marked increase as we move from face- to edge- to node-based graphs; for example, the node-based graph for the m6 data-set has approximately 20 times the number of edges as does the face-based graph. Clearly this will make itself seen whenever we use any algorithm that exploits graph connectivity, which most algorithms do. This is shown graphically in figure 8.1 for several representative algorithms applied to the Wedge1 data-set. We see that there is generally a linear relationship between n_e and runtime, although this is not always the case, indeed for the RSB plot with tolerance set to -5 solution for the faces graph actually takes

[3]Although the PUL standard mesh structure file format may include a specification of edges and faces, the three data-sets studied here do not include this information. If this information is provided then `_dual_` makes use of it; otherwise it is calculated, as needed, according to the required dual graph type. Hence, the relative cost of dual graph extraction will differ if that information is available.

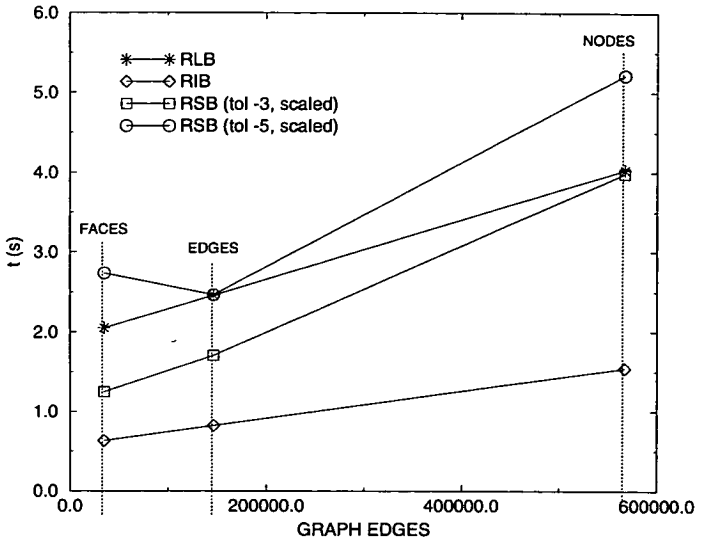


Figure 8.1: Runtime of sample decomposition algorithms as a function of n_e for the Wedge1 data-set. The algorithms used are as A.19:17, A.21:15, A.22:16 and A.22:18, all with $k = 16$. Note that times for the two RSB plots have been scaled by a factor of 0.1.

longer than for edges; we shall discuss this further in the subsequent sections appropriate to each of these algorithms.

8.2.2 Simple Algorithms

The results for the simple algorithms SR, SC and SL are tabulated in tables A.2 and A.17 for the Widget and Wedge1 data-sets, respectively.

As we noted in sections 6.3.1 and 6.3.2, neither SR nor SC are feasible decomposition algorithms in themselves, and this is made quite clear by comparing the results for those two algorithms in the bisection cases with even the worse results for SL, which is itself a rather naïve algorithm. Comparing A.2:1 and A.2:2 with A.2:3 we see that boundary vertices and cut edges are well over an order of magnitude worse for the first two cases, which confirms this for the widget data-set. Similarly, comparing A.17:1 and A.17:2 with A.17:3, we see that SL is far superior, although not to such a large degree, indicating that the element numbering for this data-set has less locality implicit in it than does the widget data-set, which is to be expected for a three dimensional mesh compared to one

in two dimensions.

It is only when we employ Cuthill-McKee renumbering with SL that we begin to see results of any quality. It is clear that after two Cuthill-McKee iterations the algorithm has settled down to alternating between two well separated vertices, as indicated by the alternating values of $|V_b|$ and $|E_{cut}|_e$ seen in A.2 for both graphs, and A.17 for the nodes-based graph. The edges and faces graphs for Wedge1 also show a broadly alternating behaviour with Cuthill-McKee iterations, although not with the same precision; in any event we may conclude that a few iterations is all that is needed to produce a considerable improvement in partition quality. This improvement does, however, come at the cost of significantly increased runtime, particularly for the larger Wedge1 data-set. However, even given one one Cuthill-McKee iteration, SL produces some of the lowest values of s_{adj} that we shall encounter.

As we noted in the previous section, runtime may be seen to depend on n_e, as determined by the dual graph type. Compare, for example, A.17:15n with A.17:15f; runtime for the former is about 2.6 times that for the latter.

As we shall see, even the best results for SL turn out to be rather poor in comparison to most other algorithms in PUL-md and we do not therefore present results for the m6 data-set, as we restrict ourselves to studying the better algorithms and parameter settings in the context of that mesh.

8.2.3 Greedy

If we compare the quality of results for GREEDY with SL applied to the Widget data-set (A.2 and A.3) we see that execution time are similar, but that the best results for SL are superior for $k = 2$ and $k = 4$, while GREEDY becomes competitive (in all but s_{adj}) when $k = 8$, particularly for the edge-based graph.

In general, we would expect this sort of behaviour; as k increases the sub-domains generated by SL become more elongated, with $|V_b|$ and $|E_{cut}|_e$ increasing accordingly, while each sub-domain should still have just two neighbours. GREEDY, on the other hand, generates quite compact sub-domains irrespective of k, and therefore will win out for higher k. This can be seen more clearly by making the same comparison for the Wedge1 data-set, where the highest value of k used is 16, rather than 8 for the Widget data-set. Here the values of $|V_b|$ and $|E_{cut}|_e$ for the better results of SL compared to GREEDY show that SL is superior for

$k = 2$, roughly equivalent for $k = 4$, but significantly worse for $k = 16$.

Although we are not able to make a similar comparison for the m6 data-set, A.32 does show one feature of our implementation quite clearly; the increase in runtime of GREEDY with k.

While we might expect the runtime of Farhat's greedy algorithm itself to be largely independent of k on the basis of the pseudocode of figure 6.12 which follows [Far88], as some authors have claimed (see, for example [DR96]) there are other factors to consider. The pseudocode, like the paper from which it was taken, says nothing about the action taken when a sub-domain becomes trapped and a new seed has to be found[4]. As k increases it is increasingly likely that this will occur, and new valid seeds also become harder and harder to find as successive sub-domains are claimed.

Our implementation searches for a new valid seed on a boundary, and so we would expect the cost of this operation to increase with k. Consider the case where the second to last sub-domain becomes trapped; if k is large then only a small number of valid vertices will exist compared to lower values of k, where (relatively speaking) more of the graph remains unclaimed in what will eventually form the later sub-domains. Therefore we would expect, as k increases, not only to have to make these searches more frequently, but also for the cost of each search to increase also.

If we did not perform this search, either by opting to produce unbalanced sub-domains and ignoring the whole issue, or by adopting some more sophisticated approach (perhaps by maintaining a list of good candidates as we go along), then this effect might be avoided. Of course, the recursive implementation we have chosen has added overheads associated with it, but we do not feel that they are significant in this respect, as such an unfavourable scaling with k is not seen in any of the other algorithms that have recursive implementations (compare with RLB, which we shall look at next, for example).

A closer examination of the execution times presented in table A.32 reveals a clear linear dependence on k for all three types of dual graph (not illustrated). While this is not particularly desirable, neither does it result in prohibitive runtimes and the motivation behind choosing this implementation was made clear in section 7.4.5. Overall, GREEDY may still be seen to be a reasonable choice for a simple

[4]Although [Far88] contains Fortran source for the original implementation, it is incomplete as regards this detail.

graph-based algorithm, even for large problems and large values of k.

Finally, we note that the runtimes for GREEDY do not include the initial determination of external border vertices, which adds approximately 0.01s, 0.23s and 2.12s to the time taken for the dual graph extraction of the Widget, Wedge1 and m6 data-sets, respectively. This also applies to RLB, which we study next.

8.2.4 RLB

RLB uses the essentially same exploration of the layer structure of the dual graph which SL employs, but with one significant additional feature; RLB has an option to estimate the quality of a bisection resulting from the choice of an initial seed point and, if Cuthill-McKee is used, to take the best seed (based on this estimate) found in the course of the Cuthill-McKee iterations as that which will define the bisection.

For the Widget and Wedge1 data-sets we examine a range of Cuthill-McKee iterations without choosing the best seed for bisection only (A.4:1 to A.4:6 and A.19:1 to A.19:6, respectively) and see exactly the same alternating behaviour as we did for SL. While the algorithm quickly settles down to alternating between two seed points, the quality of the bisection may be significantly better for one of these two seeds; for example, looking at the Wedge1 data-set, $|E_{cut}|_e$ alternates between 20068 and 17800, which represents a 12% increase of the larger figure over the smaller.

If we now examine the corresponding results for bisection where we choose the better seed (A.4:7 to A.4:9 and A.19:7 to A.19:9) we can see that this undesirable alternating behaviour is successfully avoided, and the better decomposition is found almost immediately. We therefore conclude that it is desirable for `MD_RLB_CM_BEST`, the parameter that controls this option, to be set to the value `MD_TRUE` in all cases. There is certainly little increase in runtime incurred by taking this option; at worst another Cuthill-McKee iteration will need to be performed, and, if the final seed was the best already found, then no action need be taken. This is may be seen by comparing the runtimes for an equivalent number of iterations in the two cases (either taking the best seed or not). Comparing A.19:8n with A.19:3n we see a small increase (0.80s as opposed to 0.68s, respectively) in runtime, while comparing A.19:9n with A.19:4n we see that runtimes are identical.

We take this advice to heart for all higher values of k (i.e. $k > 2$) studied for these two data-sets, and examine how the partition quality varies with iterations when we always choose the best seed. For the Widget data-set, two iterations are sufficient for the node-based graphs, while one is optimal for the edge-based graph. For the Wedge1 data-set values of either one or two are found to be optimal, depending on graph type and value of k.

We might expect the trend with increasing iterations to be as we have seen for the node-based graph of the Widget data-set when $k = 4$, where $|E_{cut}|_e$ takes the values 687, 483, 450 and 450, for 0, 1, 2, 3 iterations on choosing the best seed; in other words, a monotonic improvement with increasing iterations until a limit is reached. However, this is not always the case, as is particularly visible for the node-based graph of the Wedge1 data-set when $k = 16$, where the corresponding sequence in 154,803, 137,855, 136,258 and 137,710; the result for three iterations actually being inferior to that for two. We regard this as a largely coincidental occurrence, where we suspect a better bisection has been chosen at one level of recursion that may well have led to an inferior bisection (or bisections) being found at deeper levels of recursion. Moreover, we are only estimating which seed is preferable, and this estimate may not be sufficiently accurate in some cases.

As the results for $k = 2$ show a monotonic improvement with increasing iterations for both the Widget and Wedge1, we feel that opting for a larger number of iterations is likely to be beneficial if we are choosing the best seed, and so choose three iterations for the m6 data-set.

Comparing the results for the the better runs of RLB against GREEDY for all three data-sets produces no clear picture of which algorithm is preferable. For the Widget data-set RLB generally produces better quality results, and has an equivalent runtime to GREEDY; for the Wedge1 data-set results for RLB are of better quality only for $k = 2$ and 4 (particularly for 2) but take slightly longer on average; while for the m6 data-set RLB produces better results only for $k = 2$, being slightly worse for $k = 8$ and 32, here its runtime is slower in all cases other than $k = 32$ on the node- and edge-based graphs, being significantly faster for the former.

Finally, we note that the linear dependence of runtime for the algorithm with respect to n_e illustrated in 8.1 is unsurprising, as the determination of the layer structure of the graph requires the exploration of the neighbours of each vertex and so is directly dependent on the connectivity.

8.2.5 RCB

We now examine the first of the coordinate based algorithms we shall study, namely RCB. However, before looking at the results for RCB, we note that the runtimes presented in tables A.5, A.20 and A.34 do not include the initial calculation of vertex coordinates. This adds approximately 0.05s, 0.27s, 1.67s to the time taken for dual graph extraction of the Widget, Wedge1 and m6 data-sets, respectively, and also applies to RIB, which we study next.

There are three basic modes of operation for RCB; firstly to choose the coordinate axis along which the mesh has greatest extent *once*, at the first level of recursion, and always partition perpendicular to that axis (option 1, see A.5:1); to cycle through the axes, alternating at each level of recursion (option 3, see A.5:2); or to choose the axis of greatest extent at *every* level of recursion (option 2, see A.5:3). These three options are exactly those illustrated in figures 6.24, 6.26 and 6.25 of section 6.6.1, respectively.

We explore all three options for the Widget and Wedge1 data-sets, and see that for bisection options 1 and 2 produce identical results, as we would expect. At higher values of k, however, $|V_b|$ and $|E_{cut}|_e$ are significantly better for option 2. Comparing A.20:15 with A.20:17 provides a good illustration of this for $k = 16$ on the Wedge1 data-set, where $|E_{cut}|_e$ is a factor of 2.3 greater in the former case compared to the latter, when averaged over the three dual graph types.

It is only in terms of s_{adj} that the former case (using a fixed axis) is superior; here it behaves in a similar manner to SL, given at least one Cuthill-McKee iteration. This is due to each sub-domain generated by the versions of RCB and SL in question rarely possessing more than two neighbours.

We see that for bisection the remaining option, 3, produces worse results than either of the other two options for both Widget and Wedge1 data-sets. While this is entirely coincidental, in that performing the appropriate rotations on the two data-sets could produce identical results in all three cases, it is this sensitivity to orientation that is precisely the deficiency of this option. However, for higher values of k this option produces results of quality intermediate between the other two options in terms of $|V_b|$ and $|E_{cut}|_e$. Making a similar comparison to that made before for $k = 16$ on the Wedge1 data-set, we see that the values of $|E_{cut}|_e$ for A.20:16 are (again averaged over the three dual graph types) 1.3 times greater than those for A.20:17.

To evaluate the relative merits of GREEDY, RLB and RCB, it is useful to turn to the results for the m6 data-set, which clearly shows that RCB using option 2 produces vastly better results than either GREEDY or RLB and in a much shorter time too. Examining the relevant tables (A.32, A.33 and A.34) we see that there is no instance where this is not the case. RCB is particularly favourable for the higher values of k on the more densely connected graphs, especially in terms of runtime. A good example of this occurs for $k = 32$ on the nodes-based graph; examining $|E_{cut}|_e$, we see that GREEDY gives a value of 1,449,350 in 235s, RLB a value of 1,617,416 in 139s while RCB gives 1,257,892 in only 47s.

If we look for instances where this RCB option fares worse in such comparisons, they may be found in the smaller two data-sets, but these are the exception rather than the norm. For the Widget data-set, A.3:2e is a better partition than A.5:10e, but no other comparable set of results on those two tables shows GREEDY to be superior in any respect. RLB also shows a few instances where it is superior, for example A.4:11e and A.4:16n, but only marginally so. For the Wedge1 data-set, we find that GREEDY is slightly faster in some instances, such as A.18:2e and A.18:2f compared to A.20:10e and A.20:10f, but produces partitions of worse quality and, in any event, is slower in most other cases. The more favourable results for RLB on the Wedge1 data-set never manage to better RCB with this option, either in terms of quality or runtime.

A point of interest is the relative runtime of the three RCB options. For $k > 2$, we would expect option 3 to be faster than option 1, as it never evaluates which is the axis of greatest extent, and option 1 to similarly be faster than option 2, as the former only evaluates this information once, while the latter does so at every level of recursion. However, if we look at the runtimes for A.20:15, A.20:16 and A.20:17, we see that option 1 seems to take anomalously long. We suspect that this is due to the larger number of cut edges that result from this option; in our implementation we anticipate the subsequent use of a refinement algorithm by calculating gains as part of the bisection process, and so a larger number of cut edges may well influence runtime at this stage. More detailed profiling of the code would be required to ascertain if this is indeed the case; if so, then clearly it would be beneficial to ignore graph connectivity entirely for coordinate based algorithms when subsequent refinement is not required.

We have so far only discussed the behaviour of RCB for balanced partitions, but it will have been noticed that we also tabulate results where we set `MD_SEP_IMBAL` to `MD_TRUE` and vary `MD_SEP_MAX_IMBAL` to produce imbalanced partitions. We will

not discuss those results here, as they show broadly the same behaviour as we observe for RIB in this respect, and we explore a wider range of MD_SEP_MAX_IMBAL for that algorithm. Neither will we discuss the variation in runtime with n_e, as it is also similar to that observed for RIB, and so we refer the reader to the following section for a discussion of these topics.

8.2.6 RIB

Before examining the topics just alluded to, we will first compare RIB with the RCB, where we require a balanced partition and use option 2 for RCB. The best test of the relative merits of the two algorithms will be for the larger data-sets and for the larger values of k. If we compare A.34:3 and A.35:3, which show these results for the m6 data-set with $k = 32$, we see RIB produces slightly lower $|E_{cut}|_e$ for the node-based graph, but higher for edge- and face-based graphs, although only marginally so. For the bisection case RIB is uniformly superior in both $|V_b|$ and $|E_{cut}|_e$, but only to a relatively modest degree.

It is important to note that the results for RIB are invariant under rotations of the mesh, as we observed in section 6.6.2 and illustrated in figure 6.27; the similarity of the results for bisection are an indication that the mesh is strongly aligned with the coordinate axes. The Widget and Wedge1 data-sets also show the quality of partition produced by RIB to be generally similar to those for RCB, although sometimes better for higher values of k, particularly for the former data-set. This leads to the same conclusion for the Widget and Wedge1 data-sets regarding their alignment with the coordinate axes.

Looking at the runtimes for the the two algorithms on the m6 data-set, we see that RIB is only slightly slower; for example, 50.14s for A.35:3n, compared to 47.28s for A.34:3n. In view of this, we would tend to regard the cost of the extra runtime incurred by RIB relative to RCB as an acceptable price to pay for the added robustness of its rotational invariance. Of course, if it is known in advance that the mesh *is* strongly aligned with the coordinate axes, then the simpler algorithm may be employed.

Looking back to figure 8.1, we see that RIB exhibits a linear dependence between its runtime and n_e. The explanation for this is implicit in our comment in the previous section regarding the calculation of gains as part of the bisection process; clearly, as the number of neighbours a vertex has increases, so does the cost of

recalculating gains when it is moved by `__partn_move__`. As we discussed in section 7.5, when a vertex is moved, the gains of all its neighbours are updated incrementally according to equation 6.13, so a higher connectivity implies more calculation in `__partn_move__`. As can be seen from figure 8.1, the scaling of runtime with n_e is rather benign, so this should not concern us overmuch. Of course, exactly the same behaviour can be observed for RCB, although we do not present this graphically.

In figures 8.2 and 8.3 we study the effects of setting `MD_SEP_IMBAL` to `MD_TRUE` and varying `MD_SEP_MAX_IMBAL` to produce imbalanced partitions in the hope of reducing $|E_{cut}|_e$. This functionality is available with any of the separator field based decomposition algorithms implemented in PUL-md (RCB, RIB, RSB). For RIB the graphs plot two quantities, cut edges and load imbalance, against the chosen value of `MD_SEP_MAX_IMBAL`. Cut edges are plotted against the scale on the leftmost vertical axis, where they are given as as a percentage of $|E_{cut}|_e$ for the balanced partition. On the rightmost vertical axis, imbalance is given as simply Δ_s. The first figure shows the results for the Widget data-set, while the second those for the Wedge1 data-set (we do not explore this option for the m6 data-set); in both cases we illustrate bisection and the highest value of k tabulated (8 and 16, for the two data-sets, respectively). These graphs are based on the data in tables A.6 and A.21.

We see from the two figures that there is a clear improvement in $|E_{cut}|_e$ as `MD_SEP_MAX_IMBAL` is increased, although this evidently comes at the cost of degraded load balance, as the plots for Δ_s show. For both data-sets we see that allowing a maximum imbalance of 5% leads to a reduction in $|E_{cut}|_e$ of at least 10%, regardless of k. It is for bisection of the Widget data-set that we see the most marked improvements, where allowing a maximum imbalance of 10% leads to a reduction in $|E_{cut}|_e$ of over 60%; a very large improvement indeed.

To explain this it is helpful to examine figure 6.28 once more, where we depicted the results of RIB for the Widget data-set with $k = 8$. In that figure, the initial bisection is clearly visible as the vertical border just to left of centre. Now, if we allow some imbalance, then that border is free to move either to the left or the right along the principle axis of rotation of the mesh. Subject to `MD_SEP_MAX_IMBAL`, the border chosen will then be that which minimises $|E_{cut}|_e$. Hence the border will move progressively further to the right as we permit greater imbalance, for it is here that the length of the cut made through the mesh by the border will be minimised.

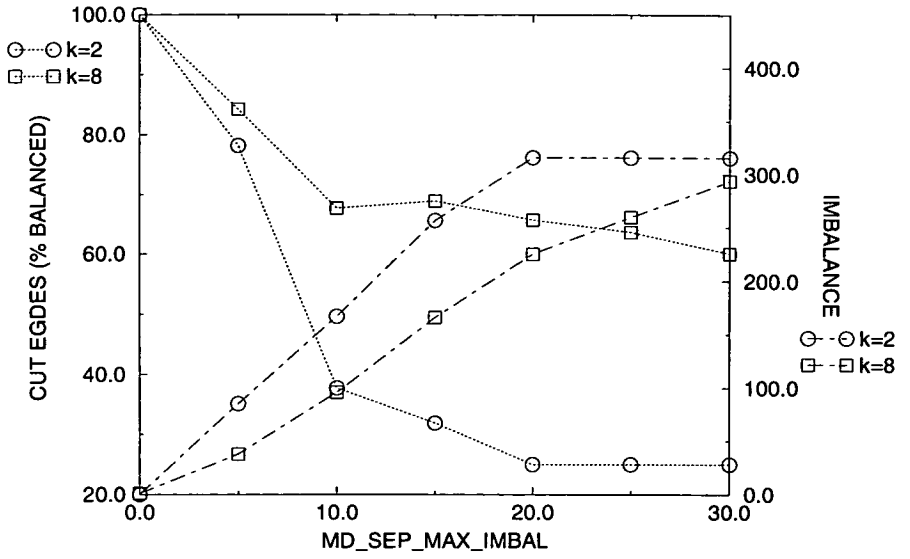


Figure 8.2: Imbalanced partition resulting from RIB with $k = 2$ and $k = 8$ for the Widget data-set, as tabulated in A.6.

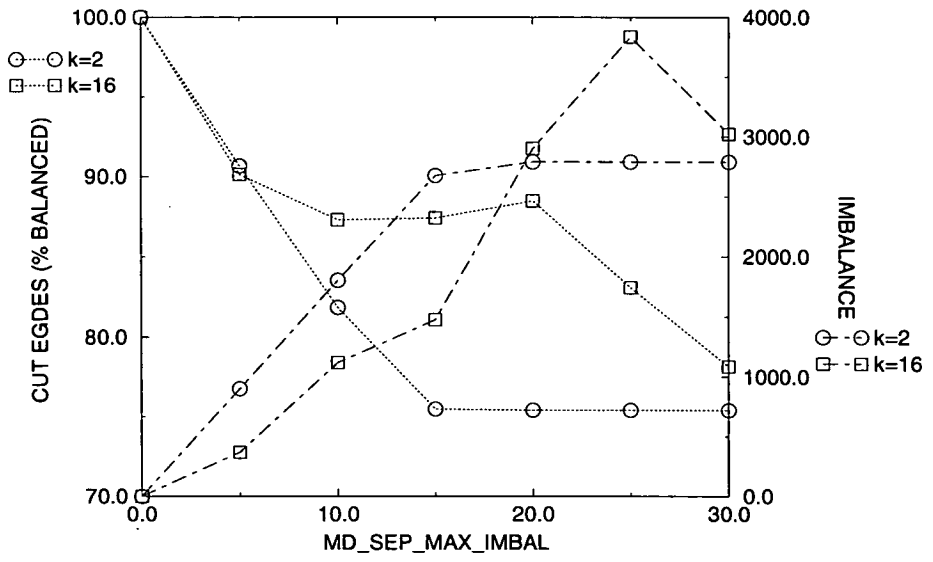


Figure 8.3: Imbalanced partition resulting from RIB with $k = 2$ and $k = 16$ for the Wedge1 data-set, as tabulated in A.21.

There are two obvious implications of this; firstly, that the benefits gained by this procedure are strongly dependent on the mesh geometry; and secondly, that we would expect an upper limit to be reached (for bisection of this data set), imposed by the sudden widening of the mesh towards the right. This latter effect can be seen quite clearly in figure 8.2, where no further improvement in $|E_{cut}|_e$ is gained past `MD_SEP_MAX_IMBAL` = 20%. Although this effect is a product of the mesh geometry, it is interesting to note that it is also clearly visible for bisection of the Wedge1 data-set, as shown in figure 8.3.

The influence of mesh geometry is likely to be lessened for higher values of k, where the essentially arbitrary shapes of the sub-domains found at intermediate levels of recursion lessen the influence of the overall geometry. For both data sets, the higher values of k show this procedure producing little benefit past `MD_SEP_MAX_IMBAL` = 10%.

We conclude our discussion of this matter by noting that the reduction in $|E_{cut}|_e$ (and any associated improvement in $|V_b|$ or s_{adj}) may or may not translate into a improvement in application runtime, as the corresponding increase in Δ_s will mitigate against this. Simply minimising $|E_{cut}|_e$ is a crude approach, and it would be better to minimising a more accurate objective function, say a weighted sum of $|E_{cut}|_e$, $|V_b|$, s_{adj} and Δ_s. Of course, the weights that should be used in such an objective function are dependent on both the application and the platform on which it is running, as so would have to be provided by the user and may be hard to determine.

8.2.7 RSB

RSB is the most sophisticated decomposition algorithm implemented in PUL-md and the quality of results it produces are significantly superior to any of the other algorithms we have thus far studied, as we shall see. In order to ascertain how to get the best performance from RSB, we first study the tunable parameters that influence the behaviour of the Lanczos eigensolution in isolation from the separator based parameters we have just discussed in the context of RIB, which also apply to RSB.

There are two aspects of the eigensolution that may be altered by the user; the convergence tolerance used, and the optional use of full orthogonalisation. For the Widget and Wedge1 data-sets we investigate the effects of varying the tunable

parameters associated with these aspects, tabulating the results in A.7 and A.22, respectively. In these two tables, we study the full range of values for MD_RSB_TOL between -5 and -1, with both full orthogonalisation (MD_RSB_ORTHOG = MD_TRUE), and partial orthogonalisation against the trivial eigenvector only (MD_FALSE).

Turning first to orthogonalisation, we see that for the Widget data set the quality of results with either full or partial orthogonalisation are, without exception, identical. This is almost true of the corresponding results for the Wedge1 data-set, where there are only a few exceptions and none that differ to a significant degree. Of those results that do differ, several actually show a small improvement through *not* using full orthogonalisation; compare A.22:13n with A.22:18n, A.22:11f with A.22:16f and A.22:14f with A.22:19f, for example. The only detrimental instance is seen in comparing A.22:13e with A.22:18e, where $|E_{cut}|_e$ takes the value 13,985 with full orthogonalisation and 13,993 if only partial orthogonalisation is used; hardly a cause for concern.

Regardless of orthogonalisation, if we examine the behaviour to RSB as we vary the convergence tolerance, we find it remarkably robust. In most cases, settings for MD_RSB_TOL between -5 and -2 produce results of equivalent quality to within a few percent. The worst exception to this is A.22:19f, where the result for a tolerance of -2 is approximately 19% worse than that given by A.22:18f, where a tolerance of -3 was used. We may conclude that generally a tolerance of -2 is sufficient, but -3 may be a safer choice.

If we examine the runtime of RSB in relation to orthogonalisation and convergence tolerance, we find a considerable benefit in avoiding full orthogonalisation, and in keeping the tolerance as loose as possible.

If we take a tolerance of -3 as representative and compare timings, respectively with full and only partial orthogonalisation, then we see the large increase in runtime associated with the former option clearly. The Widget data-set for $k = 8$ gives these times as 1.46s versus 0.65s for the node-based graph, and 2.47s versus 0.91s for edge-based. The Wedge1 data-set for $k = 16$ similarly gives; 58.62s versus 39.87s (nodes), 49.72s versus 17.04s (edges), and 75.00s versus 12.45s (faces). In view of the fact that there is no observed difference in the quality of results, we conclude that there is little need to employ full orthogonalisation. Nonetheless, it may be the case that graphs exist for which it may be necessary to the stability of the Lanczos algorithm to use full orthogonalisation, but the situation does not arise for any of the dual graphs derived from the data-sets we

have studied.

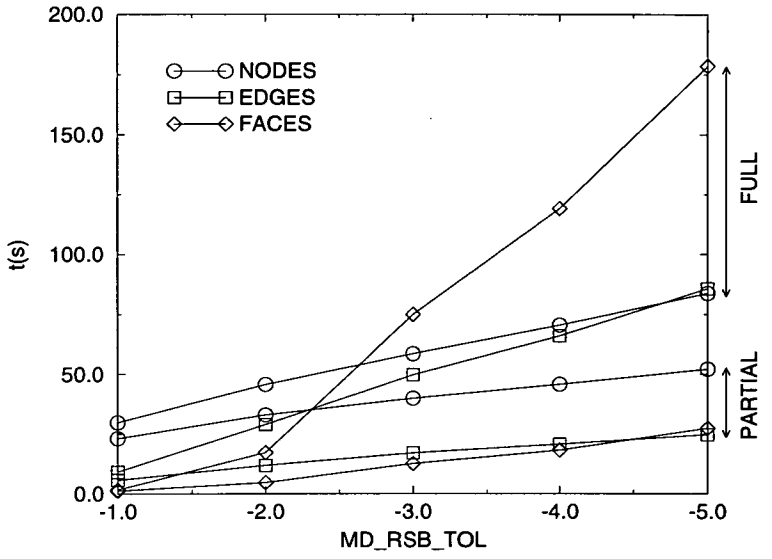


Figure 8.4: Runtime for RSB as a function of convergence tolerance for the Wedge1 data-set with $k = 16$ as tabulated in A.22.

Figure 8.4 also illustrates the effects of orthogonalisation on runtime, but is primarily a depiction of the effects of imposing increasingly strict convergence tolerance. From that graph we see a largely linear increase in runtime as we make the convergence tolerance stricter, although the greater runtimes are perhaps most noticeable for the less densely connected graphs. We see that, for full orthogonalisation, the runtimes for both edge- and face-based graphs have overtaken that for a node-based graph once the tolerance has reached -5. This is less noticeable when only partial orthogonalisation is used, where we see that the runtime for the face-based graph overtakes that for the edge-based, but not the node-based graph.

The primary computational cost involved in the Lanczos algorithm (disregarding orthogonalisation for the moment) is the matrix-vector product of the Laplacian, $\boldsymbol{L}$, with the Lanczos vector, $\boldsymbol{q}_j$, at each iteration. This cost is evidently proportional to the graph connectivity, as the number of non-zero entries in the row representing a given vertex is proportional to its number of neighbours, so it must be the case that the dual graph types with lower connectivities take more iterations to reach the specified tolerance if they have a longer runtime. If we add the extra cost of full orthogonalisation, this exaggerates the effect as the cost, at iteration j, is proportional to the number of Lanczos vectors so far generated, of

which there will be j. This goes some way to explaining the behaviour seen in figure 8.4, but does not explain the increase in iterations itself. Nonetheless, it is clear that it is uneconomic to use a needlessly strict convergence tolerance, as doing so increases runtime but may not produce a commensurate improvement in quality.

We conclude that setting convergence tolerance to -3 and employing only partial orthogonalisation is a reasonable choice of tunable parameters for RSB, and apply the algorithm to the m6 data-set in this manner, as tabulated in A.36. Comparing the quality of results produced by RSB against those produced by any of the other decomposition algorithms we have so far applied to the data-set, shows that RSB is superior in every case. A representative comparison is provided by examining the quality of partition for RIB and RSB with $k = 32$ (A.35:3 and A.36:3, respectively). We see an approximately uniform improvement of 30% in $|E_{cut}|_e$ as a result of using the more sophisticated algorithm. However, even using RSB with tunable parameter settings chosen for maximum efficiency, the runtime of the algorithm for a large problem such as this is considerable; while RIB for the node-based graph takes less than a minute (A.35:3n, 50.14s), RSB takes slightly over an hour (A.36:3n, 3721.42s).

Another consideration here is the memory requirements of RSB. Because we store the Lanczos vectors in main memory, even if we are only using partial orthogonalisation, the memory requirements of the algorithm may be very large; almost 400Mb for the face-based dual of this data-set, which is the worst behaved in this respect. Memory utilisation for this data set is shown in table 8.1 for both RIB and RSB. It should be noted that these figures include the memory requirements of `__dual__`, which accounts for the variation in the figures for RIB with graph type, as it is unlikely that the decomposition routine will exceed the requirements of the dual graph extraction routine.

MEMORY REQUIREMENTS			
	RSB		RIB
Dual	Iter's	Mb	Mb
NODES	154	311	86
EDGES	184	314	44
FACES	248	399	33

Table 8.1: Memory requirements for partitioning the m6 data-set with RSB and RIB, together with number of iteration to converge for RSB. Algorithm used are as A.35:1 and A.36:1.

Just as for RIB, we studied the effects of setting `MD_SEP_IMBAL` to `MD_TRUE`

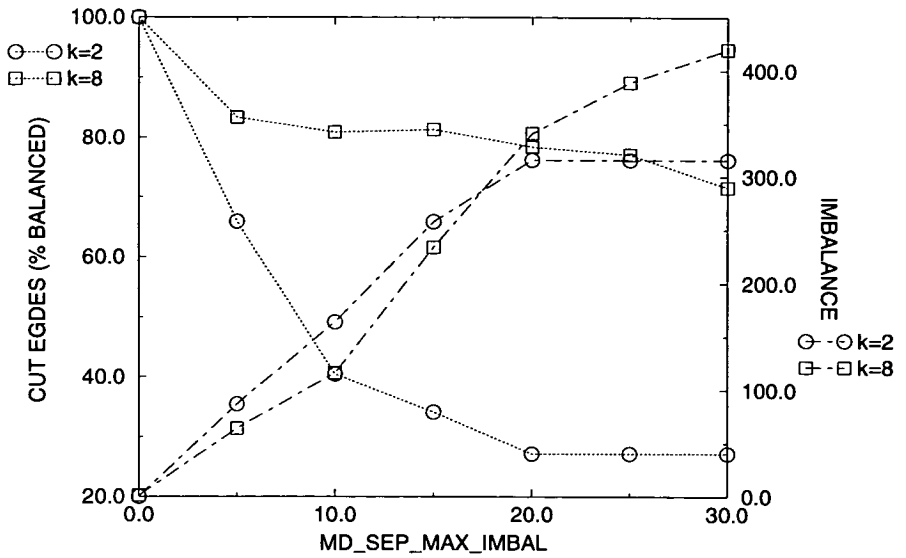


Figure 8.5: Imbalanced partition resulting from RSB with $k = 2$ and $k = 8$ for the Widget data-set, as tabulated in A.8.

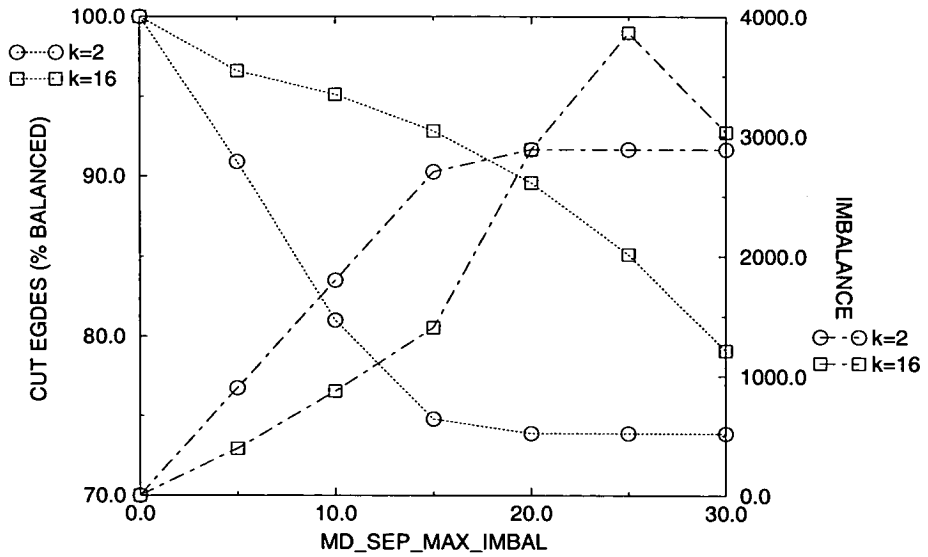


Figure 8.6: Imbalanced partition resulting from RSB with $k = 2$ and $k = 16$ for the Wedge1 data-set, as tabulated in A.23.

and varying MD_SEP_MAX_IMBAL to produce imbalanced partitions, we perform the same study for RSB, again on the Widget and Wedge1 data-sets. This is illustrated in figures 8.5 and 8.6, where the graphs shown are based on the data in tables A.8 and A.23, respectively. Again we see a marked improvement in $|E_{cut}|_e$ by allowing a 5% maximum imbalance for bisection, particularly for the Widget data-set, but now the benefits gained at higher values of k are not as noticeable as they were for RIB. Previously, at this level of imbalance, results for bisection were comparable to those for the higher values of k, but, in the case of RSB, we see only about half the improvement when partitioning into a larger number of sub-domains. Moreover, if we allow a maximum imbalance of greater than 5% at the higher values of k, then we see little further return for the increased load balance. This would seem to indicate that, for real problems where k is likely to be larger than 2, this technique of permitting some imbalance is less useful in combination with RSB than it is with RIB.

8.2.8 KL

As it is not feasible for us to examine the use of KL in combination with all of the global decomposition algorithms implemented in PUL-md, we attempt instead to examine its use in combination with three representative methods. We first examine using KL from an arbitrary initial configuration, in this case provided by SR so that the 'refinement' algorithm does all the work of partitioning, to see if it is competitive with any of the other algorithms we have discussed. We then look at refining the partitions provided by RIB and RSB, in the hope that KL, in combination with the former, may provide a faster route to good quality partitions than the time consuming spectral algorithm alone, and subsequently examine whether we can improve on RSB itself, in the hope of producing partitions of the very highest quality.

The results for SR refined by KL (SR+KL) are presented for bisection of the Widget and Wedge1 data-sets in tables A.9 and A.24, respectively. If we examine these results for the node-based graph of the Widget data-set, then we see that the best value of $|E_{cut}|_e$ achieved is 147, which is superior to the best corresponding result for RSB, which was 154 (although a value of ~173 is more representative, see A.7). However, the fastest time taken by SR+KL to achieve this promising result was 2.51s (A.9:1n), approximately five or more times that for RSB. Further, if we turn to the edge-based graph, we note that none of the

results for SR+KL are superior RSB, or even RIB, although runtimes are now sometimes more comparable.

Making a similar comparison for the Wedge1 data-set, firstly for its node-based graph, we see that the best value of $|E_{cut}|_e$ achieved by SR+KL is 11,032, while RSB gave 11,656. For this larger data-set the runtimes are no longer so disproportionate, with the former now taking only approximately twice as long as the latter. The results of SR+KL for the node-based graph are of quite uniform quality, but the other two graphs show more mixed results; for edge-based we do see many results significantly superior to RSB, while for face-based we see that all results are significantly inferior to both RSB and RIB.

For SR+KL we have no concept of a border region, as we would if the initial partition was provided by a separator field based technique, and so can not explore the `MD_KL_BORDER_ONLY` option, but we can study the effects of termination criteria and randomisation on the performance of KL. For the Widget data-set we see marked improvements in runtime as a result of setting the termination criteria so that a full KL pass is not performed at each iteration, but we also see that the quality of final partition is badly degraded. However, for the Wedge1 data-set we see that (up to a point) little or no degradation in quality occurs for the node-based graph and significant improvements in runtime are still evident. The quality of partition for the edge- and face-based graphs does seem to be more sensitive to early termination of KL, although runtime is nonetheless improved.

Finally, for the SR+KL combination, we note that some improvement in quality occasionally results from allowing a certain number of random retries, as determined by the `MD_KL_RANDOM_RETRIES` parameter. For the Widget data-set, comparing A.9:5n with A.9:15n we see a drop in $|E_{cut}|_e$ from 313 to 215, where we have allowed 3 retries. For the Wedge1 data-set this is much less marked, for example compare the same cases, A.24:5n and A.24:15n, where the drop is only from 11,043 to 11,032, which is hardly worth the cost of increased runtime.

We conclude that using KL from an arbitrary initial partition is a rather unpredictable and relatively inefficient course of action, considering that it only occasionally betters RSB alone, and is usually very much more time consuming.

Moving on to looking at refining an initial partition provided by RIB with KL (RIB+KL), we find we have a much more competitive combination. Looking first at table A.10, where we present results for bisection of the Widget data-set, we see that the quality of partition for RIB+KL is generally almost as good as

that resulting from RSB alone. While RSB gives $|E_{cut}|_e$ of 173 for the node based graph, RIB+KL generally gives a value of ~184, and in some cases as little as 144, but while RSB takes 0.32s, RIB+KL takes as little as 0.04s for the former result and 0.28s for the latter. For the edge-based graph RIB+KL uniformly gives $|E_{cut}|_e$ of 23, which is a similar figure to RSB which gives 21, but RIB+KL is again much faster.

It is interesting to note that the better results on the node-based graph ($|E_{cut}|_e$ of 144) for RIB+KL come as a result of allowing KL at least one random retry. Looking at A.10:4n and A.10:5n, we see that one random retry gives the same improvement as do five retries. It would therefore seem that one retry is sufficient, and we take this position in a more through examination of the contrast between no randomisation and one random retry for $k = 4$ on table A.11, where we explore a range of other KL tunable parameters both with and without randomisation. We see that, while the quality of partition for the edge-based graph is unaffected, the quality for the node-based graph, is improved by randomisation and to a degree which makes the difference between RIB+KL being superior or inferior to RSB. Looking at the same cases for the Wedge1 data-set, we do not see such a noticeable improvement with randomisation. In fact, for the bisection case there is no change in $|E_{cut}|_e$ as a result of randomisation, as shown on table A.25. For the higher value of k tabulated in A.26, there is a marginal improvement with randomisation, but not one which really justifies the associated increase in runtime.

This behaviour would seem to indicate that the benefits of the randomisation method we use are limited to smaller graphs. This may well be a result of our randomising only a small fixed number of vertices from each gain list; something which will not necessarily scale with problem size. This is, however, speculation as we have not had the opportunity to investigate this and so all our results are based on `N_RAND_ITEMS` = 40 (set at compile time).

The importance of setting sensible termination criteria for KL also are particularly apparent on table A.26. For the node based-graph, KL with a full pass at each iteration takes 223.52s (A.26:1n), while if we set termination criteria based on $|E_{cut}|_e$ then KL takes only 21.94s (A.26:2n) and termination based on a maximum number of consecutive unproductive vertex moves of 5% takes 26.81s (A.26:3n). The quality of results in these three cases is equivalent, with no early termination giving a value of 83,307, and both the faster options a similar figure of 83,370. As the two termination criteria generally behave similarly, we tend to

favour termination based on $|E_{cut}|_e$, as there is no associated percentage value that also needs to be tuned by the user.

For the m6 data-set we therefore use the $|E_{cut}|_e$ based termination criteria, but do not employ randomisation for the reasons previously stated. With these parameter settings we see that RIB+KL produces a similar quality of partition to RSB alone across a range of values of k, although it is on the more densely connected graphs that it fares best in this comparison. Comparing A.36:3n with A.37:11n, we see that RSB took 3721.42s to give a $|E_{cut}|_e$ of 886,179, while RIB+KL took only 487.93s to give a $|E_{cut}|_e$ of 854,272. Although not all results of RIB+KL are superior (see the corresponding results for the edge-based graph of m6, for example), runtimes are considerable better and always by a large factor, sometimes as much as an order of magnitude faster.

A feature of KL that we can investigate for RIB that we could not examine for SR, as we did not have a separator field to work with in that case, is the effect of restricting KL to the border region surrounding the initial bisection boundary. Although the results for $k = 2$ of the Widget and Wedge1 data-sets indicate that it is only when we take the defined border size as low as 5% that this procedure starts to influence the quality of partition for the worst (compare A.25:10n through A.25:13n, for example), for higher values of k on these data-sets the transition is less well defined. This is also true for any value of k to the m6 data-set. If we turn to this larger data-set to provide an illustration of the effect of restricting KL to the border region, we can see that reducing the size of this region reduces runtime, but that it also prevents KL from refining the partition to as great a degree as it otherwise would have been able to do. This is shown in figure 8.7, where we plot both $|E_{cut}|_e$ and runtime against the size of the border region. For reference, we also show the values of $|E_{cut}|_e$ for RIB and RSB alone, as the horizontal lines on that graph. We see that, while runtime is always reduced by restricting KL to a smaller region, even taking a border region as large as 60% still prevents KL from refining RIB enough to produce a better partition than RSB. That said, a significant improvement in $|E_{cut}|_e$ always occurs.

We close this discussion of KL by examining its behaviour when used in combination with RSB (RSB+KL). Here we are less concerned with runtime, as the additional cost of KL is rather small compared to the long runtimes typical of RSB. For the Widget data-set we see a good improvement in quality for RSB+KL on the node-based graph, but only a quite modest improvement for the

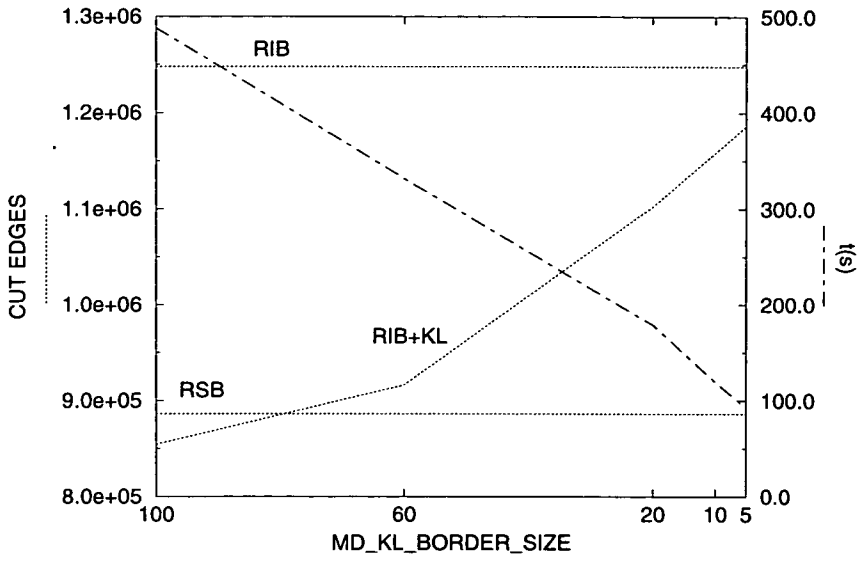


Figure 8.7: Influence of MD_KL_BORDER_SIZE on runtime and $|E_{cut}|_e$ for RIB+KL applied to the m6 data-set with $k = 32$, as tabulated in A.37:11n onward.

edge-based graph; this may well be an indication that RSB has already provided a near optimal partition on this very small graph, rather than being an indication of any failure on KL's part, however. For the Wedge1 data-set, making the same comparisons show that KL can always improve on RSB alone, regardless of dual graph type. For the m6 data-set we make this comparison more explicit by noting that KL has reduced $|E_{cut}|_e$ by $\sim$7% for the node- and edge-based graphs and by over 20% for face-based. These statistics are for $k = 32$ when comparing A.36:3 with A.39:13.

Finally, we note that the degradation of partition quality when KL is restricted to the border region is much less obvious when the initial partition originates from RSB, than RIB. This can be seen by comparing A.26:11 onwards with A.27:17 onwards for the Wedge1 data-set, and also by comparing A.37:11 onwards with A.39:11 onwards for the m6 data-set. For some graph types $|E_{cut}|_e$ is actually less when KL is restricted to a border region as small as 20%, although we suspect this to be largely coincidental. A possible explanation for the increased efficiency of using KL border restriction with RSB compared with RIB, might be that the border region is much more suitably defined by the Lanczos vector acting as separator field than the vertex coordinates in the inertial direction. As the Lanczos vector is derived from the graph structure directly it is likely to give a definition of a border region that is more closely allied to the region in which

KL prefers to operate, as KL is itself entirely dependent on graph structure. Unfortunately this is of largely academic interest, as the reduction in runtime gained by restricting KL to the border region is negligible compared to the overall runtime of RSB.

8.2.9 Mob

In order to be able to make a meaningful comparison with KL refinement, we study Mob applied to the same set of initial partitions, namely those provided by SR, RIB and RSB.

We look first at the results for SR refined by Mob (SR+MOB) tabulated in A.13 and A.28, for bisection of the Widget and Wedge1 data-sets, respectively. There we notice immediately a wide variation in both partition quality and runtime with the various tunable parameter settings explored. This variation is a significant one; if we consider the node-based graphs of the two data-sets, then we see that the best result in terms of $|E_{cut}|_e$ for Widget is 178 (A.13:17n) but that the worst is 1,997 (A.13:3n), while for Wedge1 the best is 11,103 (A.28:22n) but that the worst is 116,014 (A.28:3n). The fact that we see a variation of easily an order of magnitude would seem to indicate that Mob is rather sensitive to its parameter settings, but it should be noted that the poorer results are found when the `MD_MOBCOMPLETE` tunable parameter takes the value `MD_FALSE`. Depending on the setting of this parameter two quite different versions of the algorithm are executed; this is something we shall discuss in more depth shortly, when we come to look at refining partitions provided by RIB.

Overall, it is difficult to find evidence in A.13 and A.28 that SR+MOB fares any better in comparison with RSB than did SR+KL; there are occasions where SR+MOB is faster and occasions where it gives equivalent results, but it does not reliably do both together. We conclude, as we did for SR+KL, that SR+MOB is relatively inefficient, although we do not rule out the possibility that, given the correct choice of parameter settings, partitions of reasonable quality may still be found.

If we supply Mob with an initial partition produced by RIB (RIB+MOB), then the better results obtained are of a similar quality to RIB+KL. However, on examining the relevant tables (A.14 and A.29), we see that there is one important distinction between the two refinement algorithms, in that Mob may actually

make the partition *worse* if unfavourable parameter settings are used. The reason for this is that our implementation of Mob, unlike KL, does not record the best configuration found during its execution and will often be required to swap vertices with negative gain when there are insufficient vertices of positive gain available to fill a mob of the size determined by the current entry in the mob schedule. We can see this clearly in cases where larger mob sizes are used, for instance in A.14:1 or A.14:2 for the Widget data-set, and in A.29:1 or A.29:2, which are the same cases for the Wedge1 data-set.

If we examine the results in more detail however, then a question arises; if this undesirable behaviour is the result of a large mob size, then why do we only see the effect when MD_MOBCOMPLETE takes the value MD_FALSE? This parameter was false in all four of the cases we just referenced, but if we look at the corresponding cases where the mob size is just as large, but MD_MOBCOMPLETE takes the value MD_TRUE (A.14:10 or A.14:11 for the Widget data-set, and A.29:10 or A.29:11 for the Wedge1 data-set), we see much more reasonable results, and even some worthwhile improvement over RIB alone (although this may be subject to the number of iterations performed). As we mentioned before, the setting of MD_MOBCOMPLETE results in one of two quite different versions of the algorithm being executed. If the parameter is false, then we are not guaranteed to reach the end of the mob schedule, while if it is true, then we always will.

The effect of this on the progress of the algorithm is illustrated in figures 8.8 and 8.9 where we plot $|E_{cut}|_e$ against the number of mob exchanges for the Widget data-set. In each of these graphs we compare the progress of the algorithm for the two possible settings of MD_MOBCOMPLETE, using a fixed schedule length and initial mob size. The only other difference in parameter settings for the two plots on each graph is the larger number of iterations in the instance where we do not complete the mob schedule, which allows us to plot an equivalent total number of exchanges for the two versions of the algorithm.

From figure 8.8, we can see that the action of Mob is such that, early on in the schedule, a large amount of 'noise' is introduced into the state of the partition. As we only increment the counter into the mob schedule when the state becomes worse (or no better) then the resulting initial degradation in quality may be as large as the mob size itself. However, as we progress through the schedule and the mob size reduces, we see the algorithm settling back down towards a better configuration. It is here that the two versions of the algorithm depart; if we complete the schedule then we generally remove the noise that was previously

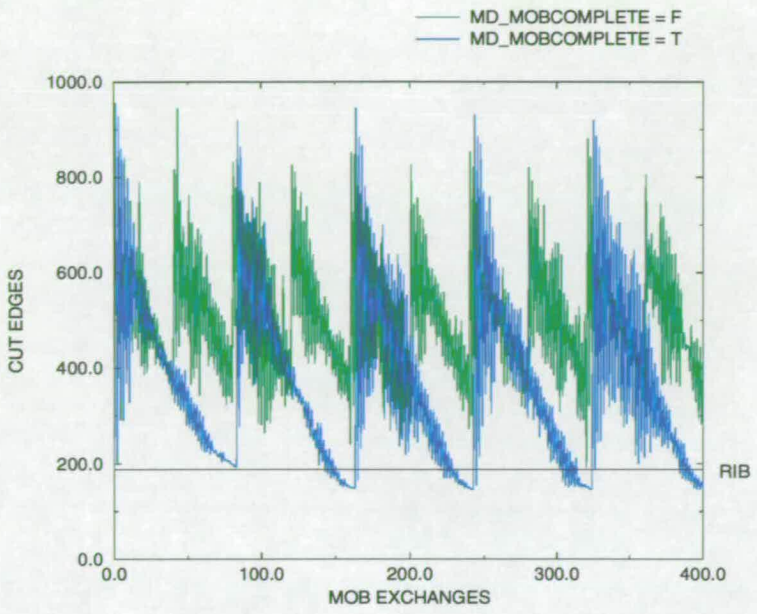


Figure 8.8: Progress of the Mob algorithm with `MD_MOBSIZE` = 10% subsequent to RIB. `MOB_ITERS` was 5 for `MD_MOBCOMPLETE` = `MD_FALSE`, and 10 for `MD_MOBCOMPLETE` = `MD_TRUE`. `MD_MOBSCHED` was 40 in both cases.

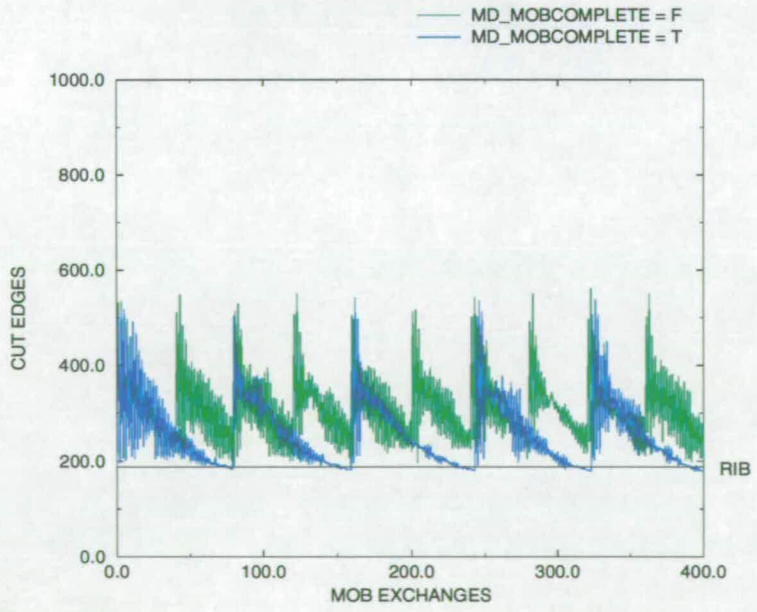


Figure 8.9: Progress of the Mob algorithm with `MD_MOBSIZE` = 5% subsequent to RIB. Other Mob parameters as for figure 8.8.

added and often find a better partition, but if we do not then the partition is invariably left in a worse state than when we started. In figure 8.9 we have reduced the initial mob size and see that the version of the algorithm where we do not complete the schedule performs better in comparison, but that the other version of the algorithm is not now given sufficient freedom to allow it to make any significant improvement over the initial RIB partition for this small data-set.

It is clear from this, and from the results for bisection in tables A.14 and A.29 where we make the appropriate comparisons, that the version of the algorithm where we insist that the mob schedule be completed is the preferred option. Hence, where we examine the behaviour of the algorithm for higher values of k, we generally set `MD_MOBCOMPLETE` to `MD_TRUE`.

Comparing the results for $k = 4$ for the Widget data-set with RIB+KL we see that similar improvements are made to partition quality by Mob and KL, and also that runtimes of the two algorithms are similar. For $k = 16$ and the Wedge1 data-set the same is also true, and here the results for the node-based graph provide a good illustration that not only is a smaller mob size preferable for this larger data-set (compare A.29:20 with A.29:22), but also that good results can sometimes be obtained in as few as two iterations (i.e. two complete cycles through the schedule; compare A.29:22 with A.29:23).

For the m6 data-set we find that it is necessary to restrict ourselves to one iteration of Mob if runtimes are to be competitive with KL, but that results are quite promising even so. While RIB+KL took 487.93s to give its best result with $|E_{cut}|_e$ of 854,272 when partitioning with $k = 32$ on the node-based graph (A.37:11n), RIB+MOB gave a $|E_{cut}|_e$ of 867,014 in just 207.64s (A.37:12n). It will also be seen that the mob sizes and schedule lengths are reduced compared to those we used on the smaller data sets in order to better KL on this large data-set; it should be noted that it required some experimentation to estimate the region of the parameter space in which the better settings lay.

Our final comments concern the use of Mob with RSB (RSB+MOB), as tabulated in A.15, A.30 and A.40, for the Widget, Wedge1 and m6 data-sets. Our conclusions here are largely the same as those for RSB+KL; that the additional runtime due to refinement is small in comparison to that of RSB itself and that improvements over the initial RSB partition of a similar degree also result. Turning again to the m6 data-set with $k = 32$, we see that RSB+MOB is marginally superior to RSB+KL for the node- and edge-based graphs, but that RSB+MOB

for the face-based graph gives rather erratic results which are generally inferior.

8.3 Summary

We now summarise our conclusions from this discussion, with particular reference to the individual characteristics of the algorithms and their preferred parameter settings.

For each algorithm we have observed the following:

SR and SC:

- Not feasible decomposition algorithms in themselves.

SL:

- Without Cuthill-McKee quality determined arbitrarily by element numbering.
- Cuthill-McKee results in great improvements in quality after even two iterations, thereafter exhibiting alternating behaviour.
- Overall quality nonetheless poor for large k, except in terms of s_{adj}.

GREEDY:

- Better $|V_b|$ and $|E_{cut}|_e$ than SL when k is large.
- Runtime increases linearly with k.

RLB:

- Without Cuthill-McKee quality determined arbitrarily by element numbering.
- Cuthill-McKee results in great improvements in quality after even two iterations, and any undesirable alternating behaviour may be avoided via `MD_RLB_CM_BEST`.
- Similar performance to GREEDY.
- Runtime increases linearly with n_e.

RCB:

- Added cost of calculating vertex coordinates negligible.

- Using a fixed axis gives similar behaviour to SL with Cuthill-McKee; overall quality poor for large k, except in terms of s_{adj}.
- Evaluating the axis of greatest extent at every level of recursion gives best performance.
- Reasonable quality partitions produced with very short runtime.
- Superior to SL, GREEDY and RLB.
- Many features in common with RIB (see below), but quality dependent on alignment of mesh with coordinate axes.

RIB:

- Similar performance to RCB for the data-sets analysed, but more robust in general, due to rotational invariance.
- RIB only marginally slower than RCB.
- Runtime increases linearly with n_e, but quite benignly.
- Allowing imbalanced partitions may reduce $|E_{cut}|_e$, but is counter productive if the imbalance is allowed to be too great.

RSB:

- Very high quality partitions reliably produced; superior to all previous algorithms above.
- May be prohibitively slow for large problems.
- Memory requirements very large.
- Loose convergence tolerances may be used to reduce runtime without compromising quality.
- Full orthogonalisation unnecessary.
- While runtime per iteration increases with n_e, number of iterations may decrease, *occasionally* leading to longer runtime for less densely connected graphs. However, for reasonable convergence tolerance runtime increases linearly with n_e.
- Allowing imbalanced partitions may reduce $|E_{cut}|_e$, but is counter productive if the imbalanced is allowed to be too great.

KL:

- SR+KL not competitive.
- RIB+KL can produce results as good as RSB with much shorter runtime.
- RSB+KL improves on RSB alone, and the additional runtime is small compared to that of RSB.
- Randomisation may produce better quality results, but only on smaller problems; we suspect this indicates our implementation should introduce more randomisation for larger problems than it does at present.
- Runtime greatly reduced by setting sensible termination criteria for each KL pass without compromising quality.
- Runtime may also be reduced by restricting KL to the border region (when used with separator field based techniques), but this may compromise quality. This compromise is more noticeable when used with RIB than RSB.
- KL never increases $|E_{cut}|_e$.

Mob:

- Mob may increases $|E_{cut}|_e$, particularly if `MD_MOBCOMPLETE` is false.
- It is almost always preferable for `MD_MOBCOMPLETE` to be true.
- SR+MOB not competitive.
- RIB+MOB can produce results as good as RSB with much shorter runtime.
- RSB+MOB may improve on RSB alone, and the additional runtime is small compared to that of RSB.
- Smaller mob size, reduced schedule length and number of iterations may be required for larger problems if runtime is to be competitive to KL.
- Overall, more erratic than KL, due to sensitivity to parameter settings, but may sometimes be superior if the correct settings can be found.

Chapter 9

A Demonstration Application

In the previous chapter we based our evaluation of the decomposition algorithms implemented in PUL-md on the metrics of partition quality Δ_s, $|E_{cut}|_e$, $|V_b|$ and s_{adj}, with particular emphasis falling on the first two of these metrics. It is evident from our discussions in section 5.3, where we examined the factors that affect the runtime of parallel unstructured mesh calculations, that these are an approximate abstraction at best, and that even a weighted sum of these metrics fails to capture the full complexity of the interaction between decomposition, application and hardware (for example, network distance does not figure in these metrics). While a full empirical exploration of the validity of these metrics would involve a survey of a variety of applications running on a variety of platforms, which is far outside the scope of this thesis, these metrics are often quoted in the literature and it is proper that we should examine their applicability in practice. To this end, we study the runtime of an example application as a function of these metrics.

The example application we use is `pheat2d`, a parallel version of the `HEAT2D` finite element heat transfer code due to Usmani and Huang [HU94]. The implementation of the parallel version of this code was used as a demonstration of the capabilities of PUL-md and PUL-sm as part of the collaboration between EPCC and Fujitsu Parallel Computing Centre. This demonstration showed that real gains are to be made by the use of a parallel platform, and that PUL-md and PUL-sm together vastly simplified the task of parallelisation, as documented in [BD96, BDH97]. These two publications discuss, amongst other topics, details of the parallelisation of the code and the resulting performance on several parallel platforms; the Fujitsu AP1000, the Meiko CS-2, and a cluster of Sun

IPX workstations connected by Ethernet. A worthwhile parallel speed-up was observed up to $k = 8$ for the test cases studied running on either the AP1000 or CS-2 (in fact, almost identical speed-ups were observed), but performance on the cluster of workstations was poor past $k = 4$. We refer the reader to these two publications for full details of these comparisons, as here we shall restrict ourselves to examining performance on the CS-2 as a function of decomposition only. However, before doing so, we shall introduce the finite element code itself, and say a little regarding some pertinent details of its parallelisation.

9.1 The Serial Code

The HEAT2D program offers a range of options for solving the two-dimensional heat conduction equation on unstructured meshes of 3 and 4-noded linear, and 6 or 9-noded quadratic elements. Both steady state and transient analysis may be performed for two dimensional or axisymmetric problems. Phase change, internal heat generation and forced convection may all be studied and non-linear material properties are permitted.

In order to analyse these phenomena, the code uses the finite element method to solve the differential equation

$$\frac{\partial}{\partial x_i}\left(k_{ij}\frac{\partial T}{\partial x_j}\right) + Q = \rho c \frac{\partial T}{\partial t} \tag{9.1}$$

where the unknown of interest is the temperature field T, Q is the heat-source term, ρ the density of the material in question, c its specific heat capacity and k_{ij} the conductivity tensor. This equation is the basic governing equation of heat conduction in a solid; HEAT2D solves both this and related equations incorporating convective and radiative flows[1]

If we restrict ourselves to looking for a steady-state solution, we can apply the condition $\partial T/\partial t = 0$. The approximating matrix equation can then be written as

$$\boldsymbol{K}\boldsymbol{\tau} = \boldsymbol{f} \tag{9.2}$$

where $\boldsymbol{K}$ is the global conductivity matrix, $\boldsymbol{\tau}$ is now the discretised temperature

[1]See Chapter 2 of [HU94] for further details.

field (which gives the solution for T) and $\boldsymbol{f}$ the load vector (incorporating the continuum source term Q). Solving 9.1 thus comes down to solving the above matrix equation.

The serial program takes as input a file specifying the problem mesh geometry, material properties and boundary conditions. It then calculates each element's contribution to the matrix $\boldsymbol{K}$, namely the element conductivity matrices $\boldsymbol{K}^e$, and similarly the element load vectors $\boldsymbol{f}^e$ and assembles these into a global form, finally applying a suitable matrix solver to the global matrix equation 9.2.

9.2 The Parallel Code

The tasks requiring the most significant amount of computation involved in solving 9.1 in this manner fall in calculating each element's $\boldsymbol{K}^e$ and in solving the matrix equation 9.2. As we saw in chapter 5, the individual $\boldsymbol{K}^e$ may be calculated in parallel without incurring any communication overheads, as they are independent. The resulting distributed global matrix, $\boldsymbol{K}$, which is their sum, may then be solved in parallel with communication occurring only for elements which are physically adjacent, but reside on different processors as we also saw in section 5.1.4 of that chapter. Thus the areas requiring attention in order to parallelise the serial code are the initial input phase and the matrix solver.

The use of the PUL-md and PUL-sm libraries enables the adoption of a relatively straightforward strategy in parallelising `HEAT2D`:

- Write a serial preprocessor, `heat2dpp`[2], to convert `HEAT2D` input files into parallel equivalents. This may be done by calling `md_decompose` to provide a decomposition of the mesh, then using PUL-md's output functions to write the data files accordingly.
- Replace the serial code input routine with a routine based on PUL-sm's mesh distribution functions which read the files written by `heat2dpp`.
- Replace the existing direct solver (a Cholesky profile solver) routine with a parallel iterative solver using PUL-sm's halo swapping functions.

The resulting parallel code is `pheat2d`.

[2] `heat2dpp` is a C program, while `pheat2d` is written in Fortran 77.

Some of the facilities of the serial code have been restricted in the parallel version; in particular the parallel meshes are restricted to 3-noded linear elements (i.e. triangles) only.

We chose one of the simplest iterative solvers to implement in parallel, the over-relaxed Gauss-Seidel algorithm. It was felt that this would suffice for the demonstration purposes, although for 'production' use of the code, particularly for large meshes, we would obviously prefer a more sophisticated algorithm (conjugate gradient or minimal residual, for example). As we saw in section 5.1.5, the key operation in any iterative matrix solution is the distributed matrix-vector product. Hence, although our Gauss-Seidel is admittedly crude, it should exhibit the same dependence on decomposition quality that the more sophisticated algorithms would be expected to display.

9.3 Effects of Decomposition Quality

In order to study the effects of decomposition quality on the runtime of the parallel code, we partition a single mesh by a variety of means and examine how the metrics of quality we have previously employed relate to the runtimes observed. The mesh we use for this comparison is the the familiar Widget data-set, for which we already have many example decompositions tabulated in appendix A, and which we have discussed in detail in the previous chapter.

We look first at balanced decompositions, so as to examine the other metrics in as much isolation from the effects of Δ_s as is possible. We then look at unbalanced decompositions where there may be a trade off between Δ_s and the other metrics which are related to communication costs.

The timings we present in the following sections are derived from a series of runs on four processors ($k = 4$) of the CS-2 where the overall execution time, the total time spent in solving 9.2 and the time spent in PUL-sm communication routines *during* solution were recorded. As the overall execution time is greater than the total solver time only by a constant factor (the difference being in I/O and calculation of individual $\boldsymbol{K}^e$ which are unaffected by decomposition), where we refer to total time henceforth we understand it to mean the latter. Similarly, where we refer to communication time, we understand it to mean communication during matrix solution.

For the Widget data-set, setting a convergence tolerance for the matrix solution of 1.0×10^{-6} gives solutions in agreement with the serial code to the order of 10^{-4}, which is quite satisfactory. Typically this level of accuracy requires approximately 700 iterations, but this figure is not entirely independent of decomposition. While this effect is discussed in [BD96, BDH97], we wish to avoid it influencing our results here and so use a fixed number of iterations. As the runtimes for the code are rather short, we use 5000 iterations so that fluctuations in timings due to other loads on the machine are averaged out.

In a finite element code such as `pheat2d`, elements are required to communicate wherever they share a mesh node, as it is at the nodes that the unknowns (the temperature in this case) reside. We therefore use the node-based dual graph of the Widget data-set in all cases.

9.3.1 Balanced Decompositions

In table 9.1, we present decomposition statistics for the Widget data-set together with corresponding total runtimes. The choice of parameter settings and combinations of algorithms have been based on the analysis of our results presented in the previous chapter, and we have tried to choose the most representative of settings and combinations. We summarise the parameter settings for each algorithm in parentheses after its name, while the second column indicates the corresponding table entry in appendix A, so that the precise settings used are available for reference. For SR+KL and SR+MOB there is no corresponding entry for $k = 4$, as indicated, but the parameter settings may still be determined from the $k = 2$ tables.

The first thing we note from table 9.1, is that the slowest runtime is found in the first entry, SL (0 Cuthill-McKee), and corresponds to the highest values of $|V_b|$ and $|E_{cut}|_e$, and the joint highest value of s_{adj}. However, we also see that the other algorithm with the same high value of s_{adj}, namely GREEDY, exhibits a comparatively reasonable runtime, implying that a high value of this metric *on its own* is not necessarily detrimental. While we do see some variation in s_{adj}, its scope is limited by the rather low value of k, so it may well be that a stronger dependence on this metric may be seen for higher k, but we do not investigate this here.

The fact that there is a notable correspondence between both $|V_b|$ and $|E_{cut}|_e$ with

PHEAT2D RUNTIMES					
Algorithm	Param's as	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
SL (0 Cuthill-McKee)	A.2:8	659	1273	12	131.219
SL (2 Cuthill-McKee)	A.2:10	262	479	6	67.368
GREEDY	A.3:2	298	542	12	64.126
RLB (0 Cuthill-McKee)	A.4:10	310	558	8	69.447
RLB (best of 3 Cuthill-McKee)	A.4:13	248	452	8	62.778
RCB (fixed axis)	A.5:8	342	576	6	70.833
RCB (best axis)	A.5:10	307	528	8	71.561
RIB	A.6:8	292	512	8	68.881
RSB (tol -3, no orthog.)	A.8:8	230	404	8	59.504
SR+KL (full pass, one retry)	A.9:8 ($k = 2$)	204	350	8	58.253
RIB+KL (full pass, one retry)	A.11:8	191	324	8	57.156
RSB+KL (full pass, one retry)	A.12:14	205	346	8	59.112
SR+MOB (complete schedule)	A.13:17 ($k = 2$)	265	427	8	64.531
RIB+MOB (complete schedule)	A.14:22	212	353	8	60.571
RSB+MOB (complete schedule)	A.15:22	207	358	8	58.647

Table 9.1: Runtimes of `pheat2d` for balanced decompositions of the Widget data-set with $k = 4$.

runtime for this first entry, where the observed time is almost 130% greater than that of the fastest run tabulated, indicates that these metrics are of some use in practice. Further, we note that the fastest run, that for RIB+KL, corresponds to the lowest values of $|V_b|$ and $|E_{cut}|_e$ (but interestingly, not of s_{adj}).

To investigate whether $|V_b|$ and $|E_{cut}|_e$ are capable of capturing the finer details of the behaviour of the code, we plot runtime against each of these two metrics in figures 9.1 and 9.2, respectively. In these two graphs, we have included the data for every algorithm appearing in table 9.1, with the one exception of the SL (0 Cuthill-McKee) entry, so as to focus on the more competitive algorithms. Losing this outlying data point does not preclude significant variations in runtime, as there is still a variation of 25% between the slowest remaining entry, RCB (best axis), and the fastest run tabulated.

Examining the two figures, we see that both show an overall increase in runtime with the metric plotted, but in neither case is the relationship clearly identifiable. Both show considerable departures from the linear regression plotted through the points, indicating that other factors are at work here. The 'other factors,' it would seem, do not include s_{adj}, as an identification of the data points of the two graphs with the data in table 9.1 shows no correspondence between s_{adj} and the variations seen in figures 9.1 and 9.2.

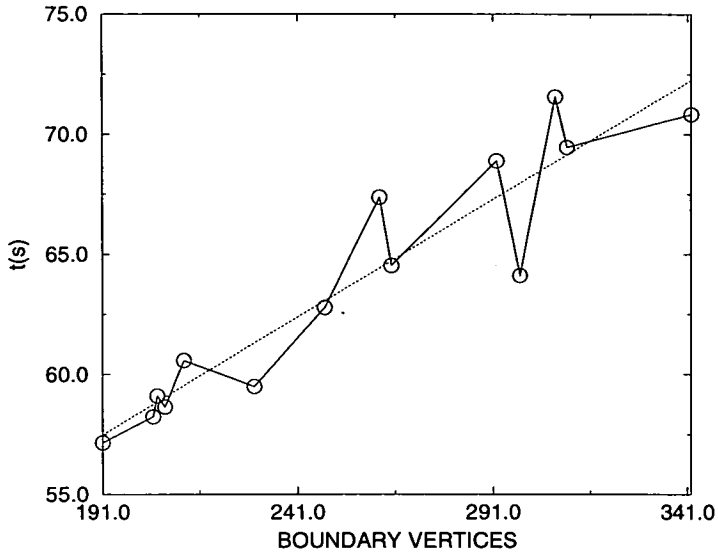


Figure 9.1: Variation in total time of `pheat2d` with $|V_b|$. Dotted line is a linear regression.

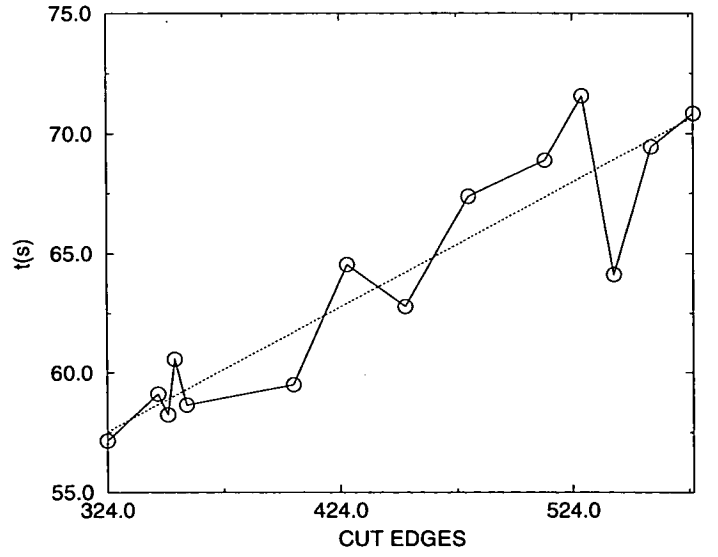


Figure 9.2: Variation in total time of `pheat2d` with $|E_{cut}|_e$. Dotted line is a linear regression.

9.3.2 Unbalanced Decompositions

In sections 8.2.6 and 8.2.7 of the previous chapter, for RIB and RSB respectively, we studied the effects of setting MD_SEP_IMBAL to MD_TRUE and varying MD_SEP_MAX_IMBAL to produce imbalanced partitions in the hope of reducing $|E_{cut}|_e$. We now examine whether this procedure can deliver an improvement in performance in practice for the application and data-set we are studying.

In figure 9.3, we plot $|E_{cut}|_e$, total and communication time against Δ_s for the Widget data-set decomposed using RIB with $k = 4$. We have obtained this data from a series of runs where MD_SEP_MAX_IMBAL was varied between 0%[3] and 10%.

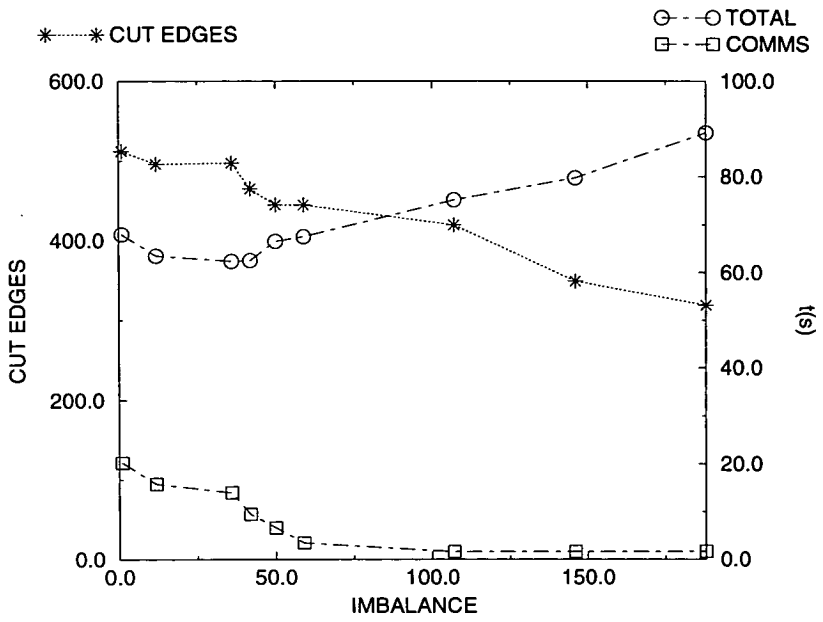


Figure 9.3: Variation in timings of pheat2d and $|E_{cut}|_e$ with Δ_s for RIB.

Examining that figure, we see a general reduction in $|E_{cut}|_e$ as Δ_s increases, with the plot for communication time following that for $|E_{cut}|_e$ to a remarkable degree. Communication time does, however, reach a point around $\Delta_s = 100$ beyond which no further reduction is seen (presumably due to communication latency rather than volume being most significant here), although by this point it has been reduced by over an order of magnitude relative to the balanced partition. As this reduction in communication time goes hand in hand with the

[3]Technically, MD_SEP_IMBAL set to MD_FALSE, but the implication that a balanced partition was specified is clear. The actual values of MD_SEP_MAX_IMBAL used were 1% to 6%, 8% and 10%.

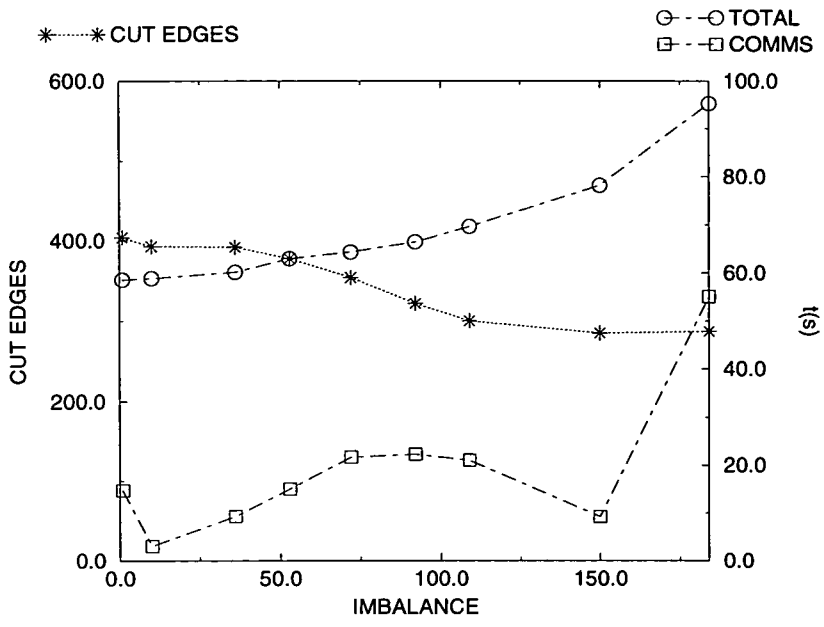


Figure 9.4: Variation in timings of `pheat2d` and $|E_{cut}|_e$ with Δ_s for RSB.

increase in Δ_s, we would expect that total time would be improved for some small level of imbalance, but eventually worsen as the inefficiency in unbalanced computational loads on the processors comes to dominate the overall behaviour. This is precisely what we observe, with the largest improvement occurring when `MD_SEP_MAX_IMBAL` is set to 2% (the third data point on the plot), where there is an 8% decrease in total time compared to the balanced partition.

While the observed behaviour for RIB is promising, and largely what we would have hoped to see, the situation for RSB is not so clear. Turning to figure 9.4, we see that, although $|E_{cut}|_e$ does still decrease as Δ_s increases, this does not translate into any improvement in total time. This, in itself, is easily explained, in that $|E_{cut}|_e$ starts at a much lower value for RSB compared to RIB, and so the cost associated with the unbalanced computational loads may dominate from the beginning. However, when we look at the plot for communication time, we see no relationship to $|E_{cut}|_e$ at all; moreover, the wide variations in communication time does not appear to have *any* noticeable impact on total time.

9.4 Summary

From our study of balanced partitions, we conclude that runtime, by and large, increases with both $|E_{cut}|_e$ and $|V_b|$, and that neither seems to be a superior metric to the other. It should be noted that the two metrics are particularly related for the node-based dual graph we have used, but this would be much less the case for edge- or face-based dual graphs. Certainly minimising either of these quantities for a balanced decomposition leads to clearly observable increases in application performance for the node-based graph we have studied.

For imbalanced partitions, we have observed that (as seen for RIB) it is indeed sometimes possible to improve application performance by permitting some level of imbalance in return for reduced communication costs. However, this is not always the case (as seen for RSB) and the resulting behaviour may be unexpected. Clearly, numerical experiments with other, preferably larger, data-sets would be necessary before we could go so far as to conclude that this procedure is generally beneficial.

Chapter 10

A Seed-Based Optimisation Approach to Partitioning

In the course of a Summer Scholarship project a novel approach to mesh decomposition originated at EPCC was investigated [Wen96]. The approach was to use optimisation techniques, in particular genetic algorithms, to find favourable seed vertices in the dual-graph whose positions would then determine the full partition.

10.1 Seed-Based Partitioning

This *seed-based* approach to partitioning is designed to alleviate some of the problems normally associated with the use of optimisation techniques for this purpose. If individual vertices are treated separately then, although the whole search space may be explored, fragmented or ill-formed sub-domains tend to dominate the procedure (statistically most possible partitions are poor, after all) and efficiency is impaired. Steps therefore need to be taken to restrict the search space to what we hope will be mostly 'reasonable' partitions (see [Wil91]).

In the seed-based approach each sub-domain is associated with a single seed vertex. Starting from these seeds, successive layers of adjacent vertices are built up around them in a deterministic manner, until the layers added to different sub-domains meet and form the sub-domain boundaries. This has the advantage that each sub-domain will always be a connected set of vertices and hopefully compact in shape.

This can be seen as akin to Farhat's greedy algorithm, but starting from each seed simultaneously, as it were. However, in Farhat's algorithm, growth of a sub-domain is not halted if it is trapped by neighbouring sub-domains in a region too small for it to reach its required size, as another seed is then sought from which growth continues. Thus balanced, but potentially disconnected sub-domains result. Our approach is to optimise the locations of the seeds for good load balance and minimal communication, but the partitioning options we explored (with one exception) do not guarantee either.

Several variations on the details of how best to grow sub-domains out from their seed vertices were studied:

1. Each sub-domain gains an entire layer at a time.
2. Each sub-domain gains a vertex at a time.
3. Each sub-domain gains a vertex at a time, but preferentially chooses new vertices neighbouring previous additions.
4. Trapped sub-domains may 'steal' vertices from their neighbours, thus guaranteeing load balance.
5. The smallest sub-domain gains an entire layer at a time.
6. Each sub-domain gains an entire layer at a time, but sub-domains that collide coordinate their growth to be at the same rate.

These partitioning options were compared qualitatively and statistically (by examining the quality of results for a number of random seed configurations) to determine if there was a bias that would favour a particular method. From this perspective, option 5 proved to be most competitive. Option 1 tended, statistically, to produce more imbalanced partitions; option 2 behaved reasonably, but was poor in fine detail; option 3 was an attempt to remedy the failings of 2, but performed little better; option 4 produced very ill-formed and often disconnected sub-domains; finally, option 6 failed in its attempt to improve load balance relative to option 5.

10.2 Optimisation

Given one of the deterministic methods for arriving at a partition from the seed vertices just outlined and a specified objective function, standard optimisation

techniques may then be employed. The objective function used to model application execution time was a linear combination of Δ_s, $|E_{cut}|_e$, $|V_b|$ and s_{adj}, although it would be a simple matter to substitute a more complex relation without affecting the actual optimisation technique employed.

The project examined three techniques for optimising the seed point locations:

- Gradient Descent
- Simulated Annealing
- Genetic Algorithms

Gradient descent was able to improve the seed locations, but produced widely differing results depending on their initial configuration, indicating that it was, as we would expect, prone to becoming trapped in local minima. Simulated annealing produced better results but was still somewhat subject to the initial configuration, although a different choice of cooling schedule might have alleviated this. The more promising partitioning options were compared when used with these two optimisation techniques, and the qualitative and statistical analysis they had previously been subject to was largely born out, with options 2 and 5 showing themselves to be superior. Genetic algorithms received particular attention, as they are much more amenable to parallel implementation than the other techniques, and, indeed proved to be the best approach.

10.3 Genetic Algorithms

The actual implementation was carried out using RPL2[1], which may easily be run in parallel and provides a variety of evolution and populations models.

10.3.1 Representation

A critical requirement for the efficient application of genetic algorithms is a good choice of representation. Clearly, if the genotype is a full specification of the partition then each individual will be quite large and this may impose a limit to the size of the population due to memory constraints. This, together

[1]The **R**eproductive **P**lan **L**anguage developed at EPCC to facilitate experimentation with genetic algorithms, and now marketed by Quadstone Ltd [Qua95]

with the potential impairment of efficiency due the unwanted exploration of unpromising regions of the search space mentioned previously, makes this a poor representation. The representation provided by the seed-based method is not subject of these deficiencies, as each genotype will only consist of a number of integers equal to the number of processors, and will be implicitly biased towards good solutions.

RPL2 provides a built-in set representation, suitable for our genotype, which is merely a set of integers (*not* ordered), but a bespoke library of operators for evaluation, mutation and recombination were needed to address the specifics of the partitioning problem.

10.3.2 Evaluation

The evaluation operator is required to return the value of the objective (fitness) function for a given configuration of seeds. Thus it is required to partition the graph according to our seed-based scheme[2] before this value can be calculated. Here we see the down side of our approach, in that going from the genotype (the seeds) to the phenotype (the partition) in order to evaluate the objective function is a computationally expensive operation.

10.3.3 Mutation

The mutation operator we choose simply moves a seed's location to a neighbouring vertex. This was implemented by considering each seed in turn and, with a specified probability, moving it to a randomly chosen neighbour.

10.3.4 Recombination

If the seed-based representation is to be used, then we would also like to ensure that the recombination operator does not unduly garble the good qualities of the parents. However, this is not straight-forward unless additional information is used.

We provide this additional information by dividing the graph into *segments* and only allow the exchange of two seeds between parents if those seeds are both

[2]Partitioning option 5 was the only one explored for GA's.

in the same segment, thus introducing some notion of locality to the recombination operator. Fortunately, we have a easy source from which to define these segments so that they do in fact reflect locality; namely the best partition found in a particular generation. The segments are initially defined from the starting population, and then are updated every tenth generation.

The recombination operator used was thus to select a certain number of seeds (defined by the *crossover rate*, typically one or two seeds) from one parent and exchange them with seeds taken from the other parent *only* if they fall in the same segment.

10.3.5 Population Models

The project implemented the following population models:

- The unstructured model, which is the basic form of GA outlined in section 6.4.5.
- The structured island model, where the population on each 'island' evolves independently, except for occasional migration of the fittest individuals to other islands.
- The fine grained structured model, where each individual has a spatial location.

Two types of migration were permitted for the island model; either to a root island that was populated with the fittest individuals from the other islands, or to neighbouring islands where we consider the islands to be arranged in a one-dimensional array. The fine grained structured model, on the other hand, used a two-dimensional array (in fact, a torus) upon which each individual was sited at a separate grid point and was only permitted to interact with individuals within four grid points of its site.

All of these models may, in theory, be implemented in parallel, with the island model being particularly suited to parallel execution, as each island can be mapped to a processor and very little communication is required. In practice, RPL2 provided very poor speed-up for the unstructured model (a factor of 1.1 on seven processors), and the fine grained model could not be run in parallel due to deficiencies in the release of RPL2 used. The island model, however, produced a speed-up of just under five running on seven processors, which is quite

encouraging.

10.4 Summary

The seed-based genetic algorithm developed in the course of this project proved itself to be a quite promising partitioning algorithm, superior to either of the other optimisation techniques explored. Due to the large computational cost involved in the execution of the evaluation operator, it is thought that the genetic algorithm is unlikely to be competitive compared to the more traditional algorithms employed for graph partitioning when implemented in serial. However, experience with the island model has demonstrated that the cost of the evaluation operator may be mitigated by the efficiency gained in parallel implementation.

While the algorithm has many attractive features (always produces connected sub-domains, amenable to parallel implementation, etc.) further study and development is required if it is to be shown that the seed-based genetic algorithm represents a useful new addition to the array of partitioning algorithms already available to us.

Chapter 11

Conclusions

In closing we review our conclusions, and also look at what issues remain outstanding regarding the development of PUL-md and related software so as to provide an outline of possible future work.

11.1 Review

We began this thesis by presenting motivating and background material relating to mesh decomposition. This entailed short studies both of unstructured mesh calculations, as typified by the finite element and finite volume methods, and of high performance computing with particular reference to large scale parallelism. An examination of general decomposition techniques for parallel computation and implementation details for parallel unstructured mesh calculations then lead us to a precise definition of the the task of mesh decomposition in terms of graph partitioning.

We have presented an exhaustive survey of algorithms for mesh decomposition and graph partitioning, and compared them qualitatively according to the consensus in the relevant literature. Based on this we have implemented a variety of algorithms in the PUL-md library, including both global methods and local refinement techniques. Many of these implemented algorithms contain considerable optimisations which may be controlled by associated user-tunable parameters.

From numerical experiments performed on three representative example datasets, where we explored a range of combinations of global and local techniques and also a range of tunable parameter settings, we were able to evaluate the

merits of the implemented algorithms based on several metrics of quality.

We concluded that it is most favourable to use a reasonable global technique together with subsequent refinement, rather than to use refinement from a random or arbitrary initial configuration. We saw that simple graph based techniques such as the Greedy algorithm or recursive layered bisection can produce serviceable, though far from optimal, partitions. Where Cuthill-McKee was used in the course of lexicographic or layered partitioning it was found to produce considerable improvements in a very few iterations, although its alternating behaviour necessitates an estimate of partition quality to ensure the best results. Recursive coordinate or inertial bisections proved superior to the simple graph based techniques, but may only be used where there is geometric information available. The inertial algorithm did not show itself to be significantly superior to coordinate for the data-sets studied, but is clearly the more robust method in general.

Recursive spectral bisection is superior in terms of partition quality to any of the other global techniques when used alone, but may exhibit unacceptable runtime or memory requirements. It was found that an equivalent quality could often be attained by a simpler algorithm together with subsequent refinement, but that the very best partitions are produced by a combination of the spectral algorithm *and* refinement. Further, the added cost of refinement is small compared to the latter global technique's runtime. For none of the data-sets studied did the Lanczos eigensolution employed show any signs of misconvergence in the absence of explicit orthogonalisation, nor did convergence criteria appear to be a critical factor, with good partitions being obtained at very low tolerances.

In comparing the two implemented refinement techniques, we found that the Kernighan and Lin algorithm was the more reliable option, but that Mob could sometimes prove superior given the correct parameter settings, although it is not entirely clear *a priori* what these settings may be. Kernighan and Lin has the advantage that it will never increase cut-edges, which is not the case for our implementation of Mob. Of the two variants of Mob explored, the version where the end of the mob-schedule is always reached is seen to be most predictable and beneficial. Our optimisations of Kernighan and Lin include tunable termination criteria for a pass of the algorithm, randomisation of the Fiduccia and Mattheyses gain data-structures and the restriction of the action of the algorithm to a border region defined by a separator field. We demonstrated that reasonable termination criteria greatly reduced runtime and had little impact on refined partition quality.

Randomisation permitted greater refinement, but this was less noticeable for large problems. Restriction to the border region was seen to reduce runtime and memory costs, although the degree of refinement may be compromised if this border region is made too small.

Our numerical experiments recorded metrics which sought to abstract partition quality away from application or platform dependence, and it is on these that the previous conclusions are based. In order to validate these metrics we examined how the runtime of a typical application depends on decomposition quality. The application used was the `pheat2d` parallel finite element code; a version of the serial HEAT2D program [HU94] parallelised using the PUL-sm runtime support library.

We first examined balanced partitions and conclude that runtime, by and large, increases with both cut edges and boundary vertices, and that neither seems to be a superior metric to the other, although they are not unrelated quantities. Turning to imbalanced partitions, we have observed that it is indeed possible to improve application performance by permitting some level of imbalance in return for reduced communication costs; a procedure PUL-md permits for separator field based recursive bisection algorithms.

The seed-based genetic algorithm detailed towards the end of this thesis showed itself to be a quite promising approach to partitioning, and was certainly superior to either of the other optimisation techniques explored in that context. Due to the large computational cost involved in the execution of the evaluation operator, it is thought that the genetic algorithm is unlikely to be competitive compared to the more traditional algorithms employed for graph partitioning when implemented in serial. However, the cost of the evaluation operator may be mitigated by the efficiency gained in parallel implementation.

In summary, we conclude from the results presented in this thesis, from experience gained during the FLITE3D project [BMT96], and from parallelisation of the HEAT2D code that the PUL-md decomposition and PUL-sm runtime support libraries together represent a well proven and powerful set of tools to support efficient parallel unstructured mesh calculations.

11.2 Future Work

In terms of development of the PUL-md library, we may divide possible future work into two areas; incremental development of the current algorithms and major changes.

Incremental development could include many minor improvements and performance optimisations. There are several instances when the library performs unnecessary computation, which could be avoided with little further work. For instance, dual graph extraction, which we have seen can be time consuming, is not required if a geometric algorithm is used alone. Similarly, the calculation of vertex gains is unnecessary if there is to be no subsequent refinement. A performance optimisation to the Kernighan and Lin algorithm is possible by 'unrolling' the changes made to the Fiduccia and Mattheyses gain data-structures which result from unproductive changes to the partition; in combination with good termination criteria this may be less time consuming than reinitialising the data-structures at the start of each pass. While these are simple performance optimisations, higher degrees of refinement may be possible if we scaled the degree of randomisation in the Kernighan and Lin algorithm with problem size. Also, if we kept track of the best partition found by our implementation of the Mob algorithm, then we could ensure that no degradation of partition quality results from its use as a refinement algorithm. However, it may well be the case that this adds significantly to the algorithm's runtime, as we do not have two copies of the partition effectively already in place as there are in our Kernighan and Lin implementation. The final incremental development we propose is to add an objective function to the evaluation of imbalanced partition quality, rather than imposing a crude upper limit on imbalance as we do now.

Major development of the library should clearly aim towards parallel, multi-level algorithms, as the consensus in the literature shows clearly that this is the most promising direction currently known. This would not only vastly improve the performance of the decomposition library as a static partitioning tool, but would also open the way to merging the decomposition and runtime libraries so that dynamic partitioning could be tackled. However, this would require significant redesign which may necessitate abandoning much of the current code, as it very much assumes that serial recursive bisection is the favoured approach and does not take into account graph weighting. Extension to allow the partitioning of weighted graphs is desirable in itself, but is a requirement for the introduction of

multi-level features into the library. If multi-level refinement is to be added then the current refinement algorithms must be extended to handle k-way partitions, rather than simply bisections as they do now; our implementation of Kernighan and Lin anticipates this, but Mob does not. A quick route to the addition of parallel partitioning would be to incorporate the currently unrelated Refine utility (parallel Jostle sub-domain heuristic) into PUL-md proper in such a way that the functionality of both may be accessed through a common interface, but without merging the code with that already in place.

Another avenue of research is to expand on the work done on the seed-based optimisation approach to partitioning. Possible routes include speeding the execution of the evaluation operator - perhaps by making updates to the partition due to the movement of a single seed a local procedure involving only the sub-domain concerned and its immediate neighbours - and also parallelisation of this stage. A variant of the algorithm that could be explored would be to allow the seeds to exert a repulsive short-range 'force' on each other via the layer structure they impose. This would ensure migration of the seeds to a well distributed configuration, without any of the overheads associated with optimisation. As it stands, further study and development is required if it is to be shown that these seed-based algorithms represent a useful new addition to the array of partitioning algorithms already available to us.

Bibliography

[Akl89] S. A. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[All93] M. Allen. Parallel methods for static mesh decomposition. Summer Scholarship Report EPCC-SS93-01, Edinburgh Parallel Computing Centre, 1993.

[Bar82] E. R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM J. Algebraic Discete Methods*, 3(4):541–550, 1982.

[Bar95] S. Barnard. PMRSB: Parallel Multilevel Recursive Spectral Bisection. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, 1995. Published on CD-ROM by ACM (ACM Press order #415952) and IEEE (IEEE Computer Society Press order #FW07435) also on http://www.supercomp.org/sc95/proceedings/.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994. Available from http://www.netlib.org/linalg/html_templates/Templates.html.

[BD96] R. M. Baxter and R. A. Davey. Unstructured mesh libraries for the AP1000. *PCW '96, Proceedings of the Sixth Parallel Computing Workshop*, pages S–A–1 to S–A–14, 1996. Fujitsu Parallel Computing Research Center.

[BDH97] R. M. Baxter, R. A. Davey, and D. S. Henty. Unstructured mesh applications at Edinburgh Parallel Computing Centre: Libraries, applications and interactive learning. In B.H.V. Topping, editor, *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 81–95. Civil-Comp Press, 1997. Proceedings of

Euro-Conference on Parallel and Distributed Computing for Computational Mechanics 1997, Pre-Processing and Solution Procedures: EURO-CM-PAR97.

[BDT96] R.M. Baxter, R.A. Davey, and S.M. Trewin. *PUL-md Prototype User Guide.* Technical Report EPCC-KTP-PUL-MD-PROT-UG 1.0, Edinburgh Parallel Computing Centre, October 1996.

[BJ93] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorisation. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[BJW90] C.F. Baille, D.A. Johnston, and R.D. Williams. Computational aspects of simulating dynamically triangulated random surfaces. *Computer Physics Communications*, 58(105), 1990.

[BMT96] R. M. Baxter, K. D. Murphy, and S. M. Trewin. Experiences in parallelising FLITE3D on the Cray T3D. *Concurrency: Practice and Experience*, 8(10):741–755, December 1996.

[Boo96] S. Booth. Lattice QCD simulation programs on the Cray T3D. Technical Report EPCC-TR96-03, EPCC, 1996.

[Boy94] R. Boyd. Echinodome design curves using a task farm. Summer Scholarship Report EPCC-SS94-14, EPCC, 1994.

[Brä93] T. Bräunl. *Parallel Programming* an introduction. Prentice Hall, 1993.

[BS92] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames, 1992.

[BS95] S. Barnard and H. Simon. A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 627–632, 1995.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM Proceedings of 24th National Conference*, New York, 1969.

[Cyb89] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Par. Dist. Comput.*, 7:279–301, 1989.

[Dah90] E.D. Dahl. Mapping and compiled communication on the connection machine system. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 756–766. IEEE, 1990.

[DER86] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, 1986.

[DH73] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM J. Res. Develop.*, 17:420–425, 1973.

[DK85] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI ciruits. *IEEE Trans. CAD*, CAD-4:92–98, 1985.

[DM91] R.A. Davey and G.R. Marion. Monte carlo investigations of random surfaces. Master's thesis, Dept. of Mathematics, University of Edinburgh, 1991.

[DPPS95] R. Davey, J. Parker, M. Parsons, and M. Sawyer. Unstructured mesh partitioning and improvement on the AP1000. In *PCW '95, Proceedings of the Fourth International Parallel Computing Workshop*, pages 219–223. Imperial College/Fujitsu Parallel Computing Research Center, 1995.

[DR94] R. Van Driesche and D. Roose. A spectral algorithm for constrained graph partitioning 1: the bisection case. TW Report 216, Dept. Computer Science, Katholieke Universiteit Leuven, Belgium, 1994.

[DR95] R. Van Driessche and D. Roose. A graph contraction algorithm for the fast calculation of the Fiedler vector of a graph. In D. H. Bailey et al., editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 621–626. SIAM, 1995.

[DR96] R. Van Driesche and D. Roose. Load balancing computational fluid dynamics calcualtions on unstructured grids. *Preprint*, 1996. Dept. Computer Science, Katholieke Universiteit Leuven.

[DS83] J. J. Dennis and R. Schnabel. *Numerical Methods for Unconstrained Optimisation and Non-Linear Equations.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[Duf96] I.S. Duff. A review of frontal methods for solving linear-systems. *Computer Physics Communications*, 97(1-2):45–52, 1996.

[Ego92] T. Alan Egolf. Computational performance of CFD codes on the Connection Machine. *Parallel Computational Fluid Dynamics*, pages 271–280, 1992. Horst D. Simon, Editor.

[Far88] C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[Fie73] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematics Journal*, 23(98):298–305, 1973.

[Fie75] M. Fiedler. A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czechoslovak Mathematics Journal*, 25(100):619–633, 1975.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[FL93] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for parallel solution of problems in computational mechanics. *Internat. J. Numer. Meth. Eng.*, 36(5):745–764, 1993.

[Fle76] R. Fletcher. Conjugate gradient methods for indefinite systems. In G.A. Watson, editor, *Proc. Dundee Conf. on Num. Anal.*, pages 73–89. Springer-Verlag, 1976.

[FLS93] C. Farhat, S. Lanteri, and H. D. Simon. TOP/DOMDEC - a software tool for mesh partitioning and parallel processing. Technical Report RNR-93-011, NAS, NASA Ames, 1993.

[Fly66] M. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54:1901–1905, 1966.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. *Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[FPTV92] B.P. Flannery, W.H. Press, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipies in C.* Cambridge University Press, second edition, 1992.

[FR94] N. Floros and J. Reeve. *Domain decomposition tool, an abridged user's guide.* University of Southampton, Dept. of Electronics and Computer Science, 1994.

[FWM94] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., 1994.

[Gaz93] H. Gazit. Randomised parallel connectivity. In J. H. Reif, editor, *Synthesis of Parallel Applications*, pages 197–214. Morgan Kaufman Inc., 1993.

[GJS76] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[GL89] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[GUW72] G. H. Golub, R. Underwood, and J. H. Wilkinson. The Lanczos algorithm for the symmetric $Ax = \lambda Bx$ problem. Technical Report STAN-CS-72-270, Dept. Computer Science, Stanford University, Stanford, Califonia, 1972.

[Gwi95] C. S. Gwilliam. The OCCAM global ocean model. *Coming of Age (The Proceedings of the Sixth ECMWF Workshop on the use of Parallel Processors in Meteorology)*, pages 446–454, 1995.

[GZ87] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message passing multiprocessor. *International Journal of Parallel Programming*, 16(6):427–449, 1987.

[Haj88] B. Hajek. Cooling schedules for optimal annealing. *Math. Oper. Res.*, 13:311, 1988.

[Ham92] S. Hammond. *Mapping unstructured grid computations to massively parallel computers.* PhD thesis, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 1992.

[Hig93] High Performance Fortran Forum. High performance fortran language specification. *Scientific Programming*, 2(1/2), 1993.

[HL92] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical

Report SAND92-1460, Sandia National Laboratories, Albuquerque, New Mexico 87185, 1992.

[HL93a] B. Hendrickson and R. Leland. Multidimensional spectral load balancing. Technical Report SAND93-0074, Sandia National Laboratories, Albuquerque, New Mexico 87185, 1993.

[HL93b] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, Albuquerque, New Mexico 87185, 1993.

[HL94] B. Hendrickson and R. Leland. An empirical study of static load balancing algorithms. In *Proc. Scalable High-Perf. Comput. Conf.*, pages 682–685. IEEE, 1994.

[HL95] B. Hendrickson and R. Leland. *The Chaco user's guide Version 2.0.* Sandia National Laboratories, Albuquerque, New Mexico 87185, 1995.

[Hol75] J. H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[HU94] H-C. Huang and A.S. Usmani. *Finite Element Analysis for Heat Transfer.* Springer-Verlag, 1994.

[JaJ92] J. JaJa. *An introduction to parallel algorithms.* Addison-Wesley Pub. Co., 1992.

[JAMS89] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimisation by simulated annealing: an experimental evaluation; part 1: graph partitioning. *Opns. Res.*, 37:865–892, 1989.

[Jim97] P. Jimack. An overview of dynamic load-balancing for parallel adaptive computational mechanics codes. *Parallel and Distributed Processing for Computational Mechanics*, 1997. Pre-print acquired at EURO-CM-PAR97.

[JM92] A. Jennings and J. J. McKeown. *Matrix Computation.* John Wiley, 1992.

[JMJH93] Z. Johan, K. K. Mathur, S. L. Johnsson, and T. J. R. Hughes. An efficient communication strategy for finite element methods on the Connection Machine CM-5 system. Technical Report Series 256, Thinking Machines Corporation, Cambridge, Massachusetts, 1993.

[JMJH94] Z. Johan, K. K. Mathur, S. L. Johnsson, and T. J. R. Hughes. Mesh decomposition and communication procedure for finite element applications on the Connection Machine CM-5 system. Technical Report TR-08-94, Harvard Univ. Center for Research in Computing Technology, apr 1994.

[JMJH95] Z. Johan, K. K. Mathur, S. L. Johnsson, and T. J. R. Hughes. Parallel implementation of recursive spectral bisection on the connection machine CM-5 system. *Parallel Computational Fluid Dynamics: New Trends and Advances*, pages 451–459, 1995. A Ecer et al, editors.

[Jon92] C. Jones. *Vertex and edge partitions of graphs.* PhD thesis, Penn. State, Dept. Computer Science, State College, PA, USA, 1992.

[Kan66] S. Kaniel. Estimates for some computational techniques in linear algebra. *Math. Comp.*, 20:369–378, 1966.

[Ker69] B. W. Kernighan. *Some Graph Partitioning Problems Related to Program Segmentation.* PhD thesis, Princeton University, January 1969.

[KJV83] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimisation by simulated annealing. *Science*, 220:671–680, 1983.

[KK95a] G. Karypis and V. Kumar. *MeTiS: Unstructured Graph Partitioning and Sparse Matrix Ordering System Version 2.0.* Department of Computer Science, University of Minnesota, Minneapolis, MN, USA, 1995.

[KK95b] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report 95-064, Department of Computer Science, University of Minnesota, Minneapolis, MN, USA, 1995.

[KK97] G. Karypis and V. Kumar. A coarse-grain parallel formulation of a multilevel k-way graph partitioning algorithm. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(1):291–307, 1970.

[KLS+94] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* MIT Press,

1994.

[KR92] J. De Keyser and D. Roose. Grid partitioning by inertial recursive bisection. Technical Report TW 174, K.U. Leuven, Deptartment of Computer Science, Belgium, July 1992.

[KT93] A. I. Kahn and B. H. V. Topping. Subdomain generation for parallel finite element analysis. *Computing Systems in Engineering*, 4(4-6):473–488, 1993.

[Lan50] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Standards*, 45:255–280, 1950.

[LL96] S. Lanteri and M. Loriot. Large-scale solutions of three-dimensional compressible flows using the parallel N3S-MUSCL solver. *Concurrency: Practice and Experience*, 8(10):769–798, December 1996.

[Lub86] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4), 1986.

[Mat90] K. Mathur. On the use of randomised address maps in unstructured three-dimensional finite element simulations. Technical Report Series CS90-4, Thinking Machines Corporation, Cambridge, Massachusetts, 1990.

[Mat92] Kapil K. Mathur. Unstructured three dimensional finite element simulations on data parallel architectures. *Unstructured scientific computation on scalable multiprocessors*, pages 65–79, 1992. P. Mehrotra, J. Saltz and R. Voigt, Editors.

[Mic96] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, third edition, 1996.

[MJ90a] K. Mathur and S.L. Johnsson. Data parallel algorithms for the finite element method. Technical Report Series CS90-2, Thinking Machines Corporation, Cambridge, Massachusetts, 1990.

[MJ90b] K. Mathur and S.L. Johnsson. Data structures and algorithms for the finite element method on a data parallel supercomputer. *International Journal for Numerical Methods in Engineering*, 29:881–908, 1990.

[MJ92] K. Mathur and S.L. Johnsson. Communication primitives for unstructured finite element simulations on data parallel architectures. *Computing Systems in Engineering*, 3(1-4):63–71, 1992.

[MO94] O. C. Martin and S. W. Otto. Combining simulated annealing with local search heuristics. Preprint submitted to Metaheuristics in Combinatoric Optimisation, 1994.

[MO95] O. C. Martin and S. W. Otto. Partitioning of unstructured meshes for load balancing. *Concurrency: Practice and Experience*, 7(4):303–314, 1995.

[MOF91] O. C. Martin, S. W. Otto, and E. W. Felten. Large-step markov chains for the travelling salesman problem. *J. Complex Syst.*, 5(3):299, 1991.

[Moh88] B. Mohar. The Laplacian spectrum of graphs. In *6th Intl. Conf. Theory and Applications of Graphs*, Kalamazoo, MI, 1988.

[MPPC97] T. MacFarland, J. Pichlmeier, F. Pearce, and H. Couchman. MP Hydra: A parallel P^3M code for very large scale cosmological simulations. In *Proceedings of the Third European CRAY-SGI MPP Workshop*, 1997.

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Keller. Equation of state calculations for fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[MV95] W. Malalasekera and H.K. Versteeg. *An introduction to computational fluid dynamics : the finite volume method.* Longman Scientific and Technical, 1995.

[MW97] E. Minty and M. Westhead. MPI on Line: A Teaching Environment for MPI. In *Supercomputing '97*, 1997.

[NORL86] 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*, pages 209–227, New York, 1986. American Soc. Mech. Eng.

[Pai72] C. C. Paige. Computational variants of the Lanczos method for the eigenproblem. *J. Inst. Maths. Applics.*, 10:373–381, 1972.

[Par92] B. N. Parlett. *The Symmetric Eigenvalue Problem.* Computational Mathematics Series. Prentice-Hall, 1992.

[PD97] 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, 1997. Proceeding of EURO-CM-PAR97, Lochinver, Scotland.

[PS74] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Num. Anal.*, 12:617–629, 1974.

[PS79] B. Parlett and D. Scott. The lanczos algorithm with selective orthogonalisation. *Math. Comp.*, 33:217–238, 1979.

[PSL90] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix. Anal. Appl.*, 11(3):430–452, July 1990.

[PVMZ87] J. Peraire, M. Vahdati, K. Morgan, and O. Zienkiewicz. Adaptive remeshing for compressible flow calculations. *Journal of Computational Physics*, 72:449–466, 1987.

[Qua95] Quadstone. *The Reproductive Plan Language RPL2, Documentation for Version 1.0A, Issue 1.* Quadstone Ltd., 16 Chester Street, Edinburgh, EH3 7RA, 1995.

[Ree84] A. P. Reeves. Parallel pascal: An extended pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1:64–80, 1984.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[Rot85] J.M. Rotter. Bending theory of shells for bins and silos. In J.M. Rotter, editor, *Design of Steel Bins for the Storage of Bulk Solids.* University of Sydney, School of Civil and Mining Engineering, 1985. Workshop on Loading, Analysis and Stability of Thin-Shell Bins, Tanks and Silos.

[RS91] V. Rao and Y. Saab. Combinatorial optimisation by stocastic evolution. *IEEE Tran. on CAD, I.U.*, pages 525–535, April 1991.

[Saa80] Y. Saad. Error bounds on the interior Rayleigh-Ritz approximations from Krylov subspaces. *Soc. Ind. Appl. Math. J. Num. Anal.*, 17,

1980.

[Sar93] W. Sarle. Kangaroos. Article posted on *comp.ai.neural-nets*, September 1993.

[SGDM94] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system. *Parallel Computing*, 20(4):531–545, 1994.

[Sim91] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[Son94] J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Computing*, 20:853–868, 1994.

[SS86] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[ST93] H. D. Simon and S-H. Teng. How good is recursive bisection? Technical Report RNR-93-012, NAS, NASA Ames, 1993.

[ST97] C. Seale and B. H. V. Topping. Towards three-dimensional subdomain generation. In B.H.V. Topping, editor, *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 53–61. Civil-Comp Press, 1997. Proceeding of EURO-CM-PAR97, Lochinver, Scotland.

[SW91] J. E. Savage and M. G. Wloka. Parallelism in graph-partitioning. *Journal of Parallel and Distributed Computing*, 13:257–272, 1991.

[SW93] J.E. Savage and M.G. Wloka. Mob — a parallel heuristic for graph-embedding. Technical Report CS-93-01, Brown University, Computer Science Department, Providence, Rhode Island 02912 USA, January 1993. http://www.cs.brown.edu/publications/techreports/reports/CS-93-01.html.

[TB96] S.M. Trewin and R.M. Baxter. *PUL-sm (Halo) User Guide.* Technical Report EPCC-PG-SM-HALO-UG 0.3, Edinburgh Parallel Computing Centre, October 1996.

[Thi93a] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide*, 1993.

[Thi93b] Thinking Machines Corporation, Cambridge, Massachusetts. *CMSSL for CM Fortran, CM-200 Edition*, version 3.1 edition, 1993.

[Thi94] Thinking Machines Corporation, Cambridge, Massachusetts. *CMSSL for CM Fortran, CM-5 Edition*, version 3.2 edition, 1994.

[TN91] T. Tokuyama and J. Nakano. Geometric algorithms for a minimum cost assignment problem. In *Proc. 7th Annual Symposium on Computational Geometry*, pages 262–271. ACM, 1991.

[TR89a] J.G. Teng and J.M. Rotter. Elastic plastic large deflection analysis of axisymmetric shells. *Computers and Structures*, 31(2):211–233, 1989.

[TR89b] J.G. Teng and J.M. Rotter. Non-symmetric bifurcation of geometrically nonlinear elastic-plastic axisymmetric shells under combined loads including torsion. *Computers and Structures*, 32(2):453–475, 1989.

[Tre95a] S.M. Trewin. *PUL-md Prototype Design Description.* Technical Report EPCC-KTG-FUJITSU93-MD-PROT-DD 1.2, Edinburgh Parallel Computing Centre, 1995.

[Tre95b] S.M. Trewin. *PUL-sm Prototype User Guide.* Technical Report EPCC-KTP-PUL-SM-PROT-UG 0.2, Edinburgh Parallel Computing Centre, Feb 1995.

[WCE95a] C. Walshaw, M. Cross, and M. Everett. A localised algorithm for optimising unstructured mesh partitions. *Int. J. Supercomputer Appl.*, 9(4):280–295, 1995.

[WCE95b] C. Walshaw, M. Cross, and M. Everett. A parallelisable algorithm for optimising unstructured mesh partitions. Technical Report 95/IM/03, University of Greenwich, London SE18 6PF, UK, 1995.

[WCE97] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph-partitioning for unstructured meshes. Technical Report 97/IM/20, University of Greenwich, London SE18 6PF, UK, 1997.

[Wen96] C. Wendl. Domain decomposition using parallel genetic algorithms. Summer Scholarship Report EPCC-SS96-01, Edinburgh Parallel Computing Centre, 1996.

[Wes96] M. Westhead. EPIC: Building a structured learning environment. In *WebNet'96 (World Conference of the Web Society).* AACE, October

1996.

[Wil90] R. D. Williams. DIME: A User's Manual. Concurrent Computation Report C3P 861, Caltech, Feb. 1990.

[Wil91] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481, 1991.

[Wil94] R. Williams. Unification of spectral and inertial bisection. http://www.ccsf.caltech.edu/~roy/papers.html, 1994.

[Zie89] O.C. Zienkiewicz. *The finite element method*, volume 1-2. McGraw-Hill, 4th edition, 1989.

Appendix A

Decomposition Statistics

A.1 Widget Data-Set

In this section we present decomposition statistics for the *Widget* data-set. This data-set is a two dimensional finite element mesh with triangular elements, which originates from the HEAT2D heat transfer code [HU94] detailed in chapter 9.

The physical geometry of the mesh is already familiar to us, as it has been extensively used as an example in this thesis, being initially introduced in figure 6.1 of chapter 6, and featuring frequently in subsequent discussions.

We present decomposition statistics for all the most significant combinations of decomposition and refinement algorithms implemented in PUL-md. For each combination we attempt, so far as the size of this document permits, to explore as much of the space of settings of the tunable parameters that control the algorithm's behaviour as possible. However, even restricting ourselves to a small sub-set of this parameter space yields a large number of options, so we have tried to focus on the most pertinent of parameter settings and hope that personal bias and expectation have not influenced the choices made unduly.

DUAL GRAPH								
	t(s)			Statistics				
Dual	Coord	Border	Other	n_e	n_e^{min}	n_e^{mean}	n_e^{max}	n_v
NODES	0.19	0.16	0.15	10072	5	11.5	16	1746
EDGES	0.14	0.10	0.09	2527	2	2.8	3	:

Table A.1:

SIMPLE											
case	MD_SL_CM_TIMES	NODES					EDGES				
		Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
SR											
$k = 2$											
1	-	0	1745	5054	2	0.01	0	1512	1269	2	0.01
SC											
$k = 2$											
2	-	0	1746	5303	2	0.01	0	1614	1473	2	0.01
SL											
$k = 2$											
3	0	0	167	322	2	0.01	0	79	44	2	0.00
4	1	0	152	269	2	0.02	0	69	45	2	0.02
5	2	0	107	198	2	0.04	0	50	31	2	0.03
6	3	0	116	220	2	0.06	0	57	36	2	0.04
7	4	0	107	198	2	0.08	0	50	31	2	0.05
$k = 4$											
8	0	1	659	1273	12	0.02	1	338	221	12	0.01
9	1	1	517	944	6	0.04	1	224	144	6	0.02
10	2	1	262	479	6	0.06	1	125	74	6	0.04
11	3	1	285	531	6	0.07	1	138	87	6	0.05
12	4	1	262	479	6	0.09	1	125	74	6	0.06
$k = 8$											
13	0	1	1269	2831	46	0.02	1	751	543	40	0.02
14	1	1	1055	1938	14	0.04	1	467	307	14	0.03
15	2	1	553	1014	14	0.06	1	264	161	14	0.04
16	3	1	606	1121	14	0.08	1	291	195	14	0.06
17	4	1	553	1014	14	0.10	1	264	161	14	0.07

Table A.2:

GREEDY										
case	NODES					EDGES				
	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$										
1	0	168	298	2	0.01	0	69	45	2	0.01
$k = 4$										
2	1	298	542	12	0.03	1	121	76	8	0.02
$k = 8$										
3	1	564	1053	28	0.06	1	214	139	28	0.04

Table A.3:

RLB													
			NODES					EDGES					
case	MD_RLB_CM_TIMES	MD_RLB_CM_BEST	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)
$k = 2$													
1	0	-	0	106	194	2	0.01	0	52	34	2	0.01
2	1	F	0	153	279	2	0.02	0	59	36	2	0.02
3	2	F	0	106	194	2	0.04	0	52	34	2	0.03
4	3	F	0	153	279	2	0.05	0	59	36	2	0.05
5	4	F	0	106	194	2	0.06	0	52	34	2	0.06
6	5	F	0	153	279	2	0.08	0	59	36	2	0.07
7	1	T	0	106	194	2	0.03	0	52	34	2	0.03
8	2	T	0	106	194	2	0.04	0	52	34	2	0.03
9	3	T	0	106	194	2	0.06	0	52	34	2	0.06
$k = 4$													
10	0	-	1	381	687	6	0.03	1	169	111	6	0.02
11	1	T	1	259	483	8	0.06	1	119	74	8	0.05
12	2	T	1	247	450	8	0.08	1	120	79	8	0.07
13	3	T	1	247	450	8	0.12	1	114	76	8	0.10
$k = 8$													
14	0	-	1	720	1343	24	0.04	1	330	214	22	0.03
15	1	T	1	486	898	22	0.09	1	215	141	22	0.08
16	2	T	1	454	839	22	0.13	1	209	143	22	0.11
17	3	T	1	454	839	22	0.17	1	208	142	22	0.15

Table A.4:

RCB															
					NODES					EDGES					
case	MD_RCB_CYCLE	MD_RCB_FIXED	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$															
1	F	T	F	-	0	116	188	2	0.01	0	49	32	2	0.01
2	T	F	F	-	0	147	245	2	0.01	0	67	40	2	0.00
3	F	F	F	-	0	116	188	2	0.01	0	49	32	2	0.01
4	F	F	T	5	84	82	143	2	0.01	87	34	19	2	0.01
5	F	F	T	10	171	39	71	2	0.01	171	19	10	2	0.01
6	F	F	T	15	239	36	63	2	0.01	223	17	9	2	0.01
7	F	F	T	20	315	27	48	2	0.01	315	12	7	2	0.01
$k = 4$															
8	F	T	F	-	1	342	576	6	0.02	1	148	92	6	0.01
9	T	F	F	-	1	298	531	10	0.01	1	136	80	10	0.01
10	F	F	F	-	1	307	528	8	0.02	1	135	82	8	0.01
11	F	F	T	5	45	279	498	8	0.02	45	126	67	8	0.01
12	F	F	T	10	98	218	392	6	0.02	98	103	54	6	0.01
13	F	F	T	15	166	159	279	6	0.02	150	101	53	6	0.02
14	F	F	T	20	242	148	256	6	0.02	242	70	38	6	0.02
$k = 8$															
15	F	T	F	-	1	713	1222	14	0.03	1	313	190	14	0.02
16	T	F	F	-	1	566	1032	28	0.02	1	264	154	24	0.02
17	F	F	F	-	1	499	862	22	0.03	1	222	136	22	0.02
18	F	F	T	5	41	426	755	22	0.03	43	191	103	22	0.02
19	F	F	T	10	88	368	647	20	0.03	101	178	96	22	0.02
20	F	F	T	15	151	338	576	20	0.03	123	179	97	24	0.02
21	F	F	T	20	255	358	594	18	0.04	234	159	85	18	0.03

Table A.5:

RIB												
			NODES					EDGES				
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$												
1	F	-	0	111	188	2	0.01	0	47	31	2	0.01
2	T	5	85	80	147	2	0.01	85	34	19	2	0.01
3	T	10	167	45	71	2	0.01	170	19	10	2	0.01
4	T	15	257	32	60	2	0.01	221	16	9	2	0.01
5	T	20	316	25	47	2	0.02	313	12	7	2	0.01
6	T	25	316	25	47	2	0.02	313	12	7	2	0.01
7	T	30	316	25	47	2	0.02	313	12	7	2	0.01
$k = 4$												
8	F	-	1	292	512	8	0.03	1	130	79	8	0.02
9	T	5	59	258	445	8	0.03	86	111	62	8	0.03
10	T	10	188	195	319	8	0.03	177	90	48	8	0.03
11	T	15	176	169	283	6	0.03	115	75	40	6	0.03
12	T	20	205	157	262	6	0.03	229	66	36	6	0.03
13	T	25	205	157	262	6	0.04	229	66	36	6	0.03
14	T	30	205	155	257	6	0.04	229	66	36	6	0.03
$k = 8$												
15	F	-	1	480	838	24	0.04	1	216	136	20	0.04
16	T	5	38	416	706	20	0.05	51	176	98	20	0.05
17	T	10	96	339	568	18	0.05	93	158	85	18	0.04
18	T	15	166	344	578	16	0.05	134	151	82	18	0.04
19	T	20	225	320	551	16	0.05	230	142	77	18	0.04
20	T	25	260	314	534	16	0.06	262	139	75	18	0.05
21	T	30	294	299	504	18	0.06	300	137	74	18	0.05

Table A.6:

RSB															
					NODES					EDGES					
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$															
1	-5	T	F	-	0	100	172	2	1.91	0	46	26	2	4.84
2	-4	T	F	-	0	100	172	2	2.41	0	46	26	2	3.59
3	-3	T	F	-	0	101	173	2	0.84	0	41	21	2	1.32
4	-2	T	F	-	0	100	175	2	0.40	0	38	22	2	0.23
5	-1	T	F	-	0	91	154	2	0.10	0	65	39	2	0.02
6	-5	F	F	-	0	100	172	2	0.65	0	46	26	2	1.59
7	-4	F	F	-	0	100	172	2	0.43	0	46	26	2	1.09
8	-3	F	F	-	0	101	173	2	0.32	0	41	21	2	0.39
9	-2	F	F	-	0	100	175	2	0.17	0	38	22	2	0.08
10	-1	F	F	-	0	91	154	2	0.06	0	65	39	2	0.01
$k = 8$															
11	-5	T	F	-	1	397	687	16	3.20	1	175	107	16	8.86
12	-4	T	F	-	1	395	688	16	2.16	1	175	105	16	5.64
13	-3	T	F	-	1	396	689	16	1.46	1	171	90	16	2.47
14	-2	T	F	-	1	393	691	16	0.75	1	200	116	20	0.49
15	-1	T	F	-	1	453	792	22	0.25	1	409	253	34	0.05
16	-5	F	F	-	1	397	687	16	1.37	1	175	107	16	3.64
17	-4	F	F	-	1	395	688	16	0.95	1	175	105	16	2.15
18	-3	F	F	-	1	396	689	16	0.65	1	171	90	16	0.91
19	-2	F	F	-	1	393	691	16	0.39	1	200	116	20	0.22
20	-1	F	F	-	1	453	792	22	0.18	1	409	253	34	0.04

Table A.7:

RSB															
					NODES					EDGES					
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$															
1	-3	F	F	-	0	101	173	2	0.32	0	41	21	2	0.39
2	-3	F	T	5	87	63	114	2	1.06	81	29	16	2	0.39
3	-3	F	T	10	164	39	70	2	0.32	162	19	10	2	0.39
4	-3	F	T	15	258	34	59	2	0.32	212	16	9	2	0.39
5	-3	F	T	20	316	25	47	2	0.32	313	12	7	2	0.39
6	-3	F	T	25	316	25	47	2	0.32	313	12	7	2	0.39
7	-3	F	T	30	316	25	47	2	0.32	313	12	7	2	0.39
$k = 4$															
8	-3	F	F	-	1	230	404	8	0.53	1	97	50	8	0.75
9	-3	F	T	5	92	187	322	8	0.54	79	89	47	8	0.74
10	-3	F	T	10	184	173	287	6	0.56	179	75	40	6	0.50
11	-3	F	T	15	268	161	263	6	0.56	263	71	38	6	0.54
12	-3	F	T	20	383	142	248	6	0.51	384	66	36	6	0.86
13	-3	F	T	25	383	142	248	6	0.51	453	67	36	6	0.86
14	-3	F	T	30	515	134	238	6	0.52	453	67	36	6	0.86
$k = 8$															
15	-3	F	F	-	1	396	689	16	0.65	1	171	90	16	0.92
16	-3	F	T	5	64	337	574	18	1.43	48	157	85	18	0.95
17	-3	F	T	10	116	329	557	18	0.73	106	150	81	18	0.84
18	-3	F	T	15	234	339	560	16	0.82	211	143	78	16	0.98
19	-3	F	T	20	341	308	540	16	0.79	317	139	75	16	1.43
20	-3	F	T	25	389	310	531	16	0.79	406	134	73	20	1.66
21	-3	F	T	30	420	279	493	18	0.89	493	130	72	22	1.78

Table A.8:

SR+KL																	
							NODES						EDGES				
case	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)
$k = 2$																	
1	F	F	-	-	-	0	0	94	147	2	2.51	0	92	47	2	0.91
2	T	F	-	-	-	0	0	183	313	2	0.65	0	172	86	2	0.37
3	F	T	50	-	-	0	0	110	185	2	1.32	0	118	59	2	0.61
4	F	T	20	-	-	0	0	183	313	2	0.60	0	166	83	2	0.40
5	F	T	10	-	-	0	0	183	313	2	0.50	0	172	86	2	0.33
6	F	T	5	-	-	0	0	236	405	2	0.47	0	215	108	2	0.28
7	F	T	1	-	-	0	0	276	474	2	0.27	0	544	272	2	0.13
8	F	F	-	-	-	1	0	94	147	2	2.77	0	84	42	2	1.41
9	F	F	-	-	-	3	0	94	147	2	3.27	0	84	42	2	1.68
10	F	F	-	-	-	5	0	94	147	2	3.76	0	80	40	2	2.82
11	T	F	-	-	-	1	0	183	313	2	0.69	0	172	86	2	0.40
12	T	F	-	-	-	3	0	183	313	2	0.76	0	172	86	2	0.46
13	T	F	-	-	-	5	0	183	313	2	0.84	0	172	86	2	0.51
14	F	T	10	-	-	1	0	183	313	2	0.55	0	172	86	2	0.32
15	F	T	10	-	-	3	0	122	215	2	0.95	0	172	86	2	0.40
16	F	T	10	-	-	5	0	122	215	2	1.06	0	172	86	2	0.47
17	F	T	5	-	-	1	0	236	405	2	0.51	0	182	91	2	0.67
18	F	T	5	-	-	3	0	241	403	2	0.83	0	182	91	2	0.75
19	F	T	5	-	-	5	0	241	403	2	0.92	0	182	91	2	0.80

Table A.9:

RIB+KL																			
									NODES					EDGES					
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																			
1	F	-	F	F	-	F	-	0	0	99	184	2	0.51	0	46	23	2	0.27
2	F	-	T	F	-	F	-	0	0	99	184	2	0.09	0	46	23	2	0.06
3	F	-	F	T	5	F	-	0	0	99	184	2	0.09	0	46	23	2	0.07
4	F	-	T	F	-	F	-	1	0	86	144	2	0.28	0	46	23	2	0.09
5	F	-	T	F	-	F	-	5	0	86	144	2	0.41	0	46	23	2	0.19
6	F	-	F	F	-	T	60	0	0	99	184	2	0.31	0	46	23	2	0.17
7	F	-	F	F	-	T	20	0	0	99	184	2	0.11	0	46	23	2	0.07
8	F	-	F	F	-	T	10	0	0	99	184	2	0.06	0	46	23	2	0.04
9	F	-	F	F	-	T	5	0	0	103	186	2	0.03	0	46	23	2	0.02
10	F	-	T	F	-	T	60	0	0	99	184	2	0.07	0	46	23	2	0.04
11	F	-	T	F	-	T	20	0	0	99	184	2	0.05	0	46	23	2	0.03
12	F	-	T	F	-	T	10	0	0	99	184	2	0.04	0	46	23	2	0.02
13	F	-	T	F	-	T	5	0	0	103	186	2	0.03	0	46	23	2	0.02

Table A.10:

RIB+KL																			
									NODES					EDGES					
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 4$																			
1	F	-	F	F	-	F	-	0	1	214	375	8	1.89	1	121	61	8	0.54
2	F	-	T	F	-	F	-	0	1	241	416	8	0.26	1	121	61	8	0.13
3	F	-	F	T	5	F	-	0	1	241	416	8	0.24	1	121	61	8	0.15
4	F	-	T	F	-	T	60	0	1	241	416	8	0.20	1	121	61	8	0.09
5	F	-	T	F	-	T	20	0	1	239	418	8	0.13	1	121	61	8	0.06
6	F	-	T	F	-	T	10	0	1	249	436	8	0.08	1	121	61	8	0.05
7	F	-	T	F	-	T	5	0	1	275	484	8	0.06	1	129	65	8	0.04
8	F	-	F	F	-	F	-	1	1	191	324	8	2.78	1	121	61	8	0.80
9	F	-	T	F	-	F	-	1	1	207	351	8	0.49	1	121	61	8	0.18
10	F	-	F	T	5	F	-	1	1	210	356	8	0.46	1	121	61	8	0.21
11	F	-	T	F	-	T	60	1	1	203	353	8	0.45	1	121	61	8	0.13
12	F	-	T	F	-	T	20	1	1	211	377	8	0.25	1	121	61	8	0.07
13	F	-	T	F	-	T	10	1	1	278	477	8	0.20	1	121	61	8	0.06
14	F	-	T	F	-	T	5	1	1	275	484	8	0.07	1	129	65	8	0.04

Table A.11:

RSB+KL																					
											NODES					EDGES				
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k = 2$																					
1	-3	F	F	-	F	F	-	F	-	0	0	89	148	2	1.08	0	40	20	2	0.65
2	-3	F	F	-	T	F	-	F	-	0	0	89	148	2	0.44	0	40	20	2	0.45
3	-3	F	F	-	F	T	5	F	-	0	0	89	148	2	0.44	0	40	20	2	0.45
4	-3	F	F	-	T	F	-	F	-	1	0	94	147	2	0.68	0	40	20	2	0.47
5	-3	F	F	-	T	F	-	F	-	5	0	94	147	2	0.68	0	40	20	2	0.57
6	-3	F	F	-	F	F	-	T	60	0	0	89	148	2	0.77	0	40	20	2	0.54
7	-3	F	F	-	F	F	-	T	20	0	0	89	148	2	0.47	0	40	20	2	0.44
8	-3	F	F	-	F	F	-	T	10	0	0	89	148	2	0.40	0	36	18	2	0.44
9	-3	F	F	-	F	F	-	T	5	0	0	85	149	2	0.34	0	40	20	2	0.40
10	-3	F	F	-	T	F	-	T	60	0	0	89	148	2	0.41	0	40	20	2	0.42
11	-3	F	F	-	T	F	-	T	20	0	0	89	148	2	0.38	0	40	20	2	0.40
12	-3	F	F	-	T	F	-	T	10	0	0	89	148	2	0.36	0	36	18	2	0.41
13	-3	F	F	-	T	F	-	T	5	0	0	85	149	2	0.33	0	40	20	2	0.40
$k = 4$																					
14	-3	F	F	-	F	F	-	F	-	1	1	205	346	8	3.12	1	95	48	8	1.55
15	-3	F	F	-	T	F	-	F	-	1	1	205	346	8	0.97	1	95	48	8	0.93
16	-3	F	F	-	F	T	5	F	-	1	1	205	346	8	0.98	1	95	48	8	0.95
17	-3	F	F	-	T	F	-	T	60	1	1	204	342	8	0.84	1	94	47	8	0.89
18	-3	F	F	-	T	F	-	T	20	1	1	230	364	8	0.68	1	95	48	8	0.81
19	-3	F	F	-	T	F	-	T	10	1	1	213	356	8	0.67	1	90	45	8	0.75
20	-3	F	F	-	T	F	-	T	5	1	1	221	367	8	0.57	1	94	47	8	0.79

Table A.12:

SR+MOB																		
								NODES						EDGES				
case	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																		
1	F	50	40	20	F	F	-	0	594	1553	2	3.27	0	894	733	2	2.51
2	F	50	40	10	F	F	-	0	585	1470	2	1.66	0	762	727	2	1.28
3	F	50	40	1	F	F	-	0	899	1997	2	0.17	0	699	632	2	0.14
4	F	10	40	20	F	F	-	0	115	182	2	0.91	0	192	144	2	0.76
5	F	10	40	10	F	F	-	0	274	457	2	0.46	0	206	141	2	0.39
6	F	10	40	1	F	F	-	0	407	659	2	0.06	0	362	221	2	0.05
7	F	5	40	20	F	F	-	0	202	322	2	0.63	0	170	119	2	0.54
8	F	5	40	10	F	F	-	0	251	398	2	0.32	0	147	94	2	0.28
9	F	5	40	1	F	F	-	0	347	568	2	0.04	0	399	211	2	0.04
10	F	10	40	20	F	T	F	0	231	429	2	1.37	0	181	140	2	1.29
11	F	10	40	20	F	T	T	0	157	278	2	1.38	0	186	144	2	1.29
12	F	10	40	20	T	T	T	0	157	278	2	1.37	4	126	74	2	1.28
13	T	50	40	20	F	F	-	0	108	193	2	4.69	0	184	92	2	3.81
14	T	50	40	10	F	F	-	0	131	205	2	2.37	0	228	114	2	1.93
15	T	50	40	1	F	F	-	0	328	529	2	0.25	0	162	81	2	0.20
16	T	10	40	20	F	F	-	0	111	182	2	1.40	0	64	33	2	1.26
17	T	10	40	10	F	F	-	0	104	178	2	0.73	0	100	50	2	0.62
18	T	10	40	1	F	F	-	0	269	454	2	0.10	0	282	141	2	0.08
19	T	5	40	20	F	F	-	0	198	324	2	1.07	0	138	69	2	0.95
20	T	5	40	10	F	F	-	0	189	325	2	0.55	0	154	77	2	0.49
21	T	5	40	1	F	F	-	0	252	420	2	0.09	0	340	170	2	0.07
22	T	10	40	20	F	T	F	0	146	226	2	2.05	0	92	46	2	1.98
23	T	10	40	20	F	T	T	0	132	224	2	2.08	0	104	52	2	1.96
24	T	10	40	20	T	T	T	0	132	224	2	2.08	0	104	52	2	3.38

Table A.13:

RIB+MOB																				
										NODES					EDGES				
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)
$k = 2$																				
1	F	-	F	50	40	20	F	F	-	0	597	1535	2	3.45	0	564	486	2	2.60
2	F	-	F	50	40	1	F	F	-	0	719	1557	2	0.17	0	451	374	2	0.15
3	F	-	F	10	40	20	F	F	-	0	182	291	2	0.85	0	147	121	2	0.75
4	F	-	F	10	40	1	F	F	-	0	222	403	2	0.05	0	234	153	2	0.05
5	F	-	F	5	40	20	F	F	-	0	129	214	2	0.58	0	99	67	2	0.52
6	F	-	F	5	40	1	F	F	-	0	139	218	2	0.04	0	100	60	2	0.04
7	F	-	F	5	40	20	F	T	F	0	133	230	2	0.82	0	102	72	2	0.79
8	F	-	F	5	40	20	F	T	T	0	123	206	2	0.82	0	86	61	2	0.79
9	F	-	F	5	40	20	T	T	T	0	123	206	2	0.82	3	76	47	2	0.79
10	F	-	T	50	40	5	F	F	-	0	89	152	2	1.14	0	156	78	2	0.84
11	F	-	T	50	40	1	F	F	-	0	184	307	2	0.23	0	186	93	2	0.17
12	F	-	T	10	40	5	F	F	-	0	99	149	2	0.34	0	89	46	2	0.33
13	F	-	T	10	40	1	F	F	-	0	126	194	2	0.08	0	126	63	2	0.08
14	F	-	T	5	40	5	F	F	-	0	105	177	2	0.26	0	51	27	2	0.24
15	F	-	T	5	40	1	F	F	-	0	109	185	2	0.06	0	69	35	2	0.06
16	F	-	T	10	40	5	F	T	F	0	107	180	2	0.48	0	44	23	2	0.47
17	F	-	T	10	40	5	F	T	T	0	109	179	2	0.50	0	76	38	2	0.48
18	F	-	T	10	40	5	T	T	T	0	109	179	2	0.50	0	76	38	2	0.54
$k = 4$																				
19	F	-	F	5	40	20	F	F	-	1	255	436	12	1.16	1	224	146	12	1.04
20	F	-	T	10	40	5	F	F	-	1	227	354	8	0.69	1	179	92	8	0.67
21	F	-	T	10	40	2	F	F	-	1	211	361	8	0.29	1	175	91	10	0.28
22	F	-	T	5	40	5	F	F	-	1	212	353	8	0.51	1	127	66	8	0.48
23	F	-	T	5	40	2	F	F	-	1	218	374	8	0.22	1	156	81	8	0.21

Table A.14:

RSB+MOB																						
												NODES					EDGES				
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k = 2$																						
1	-3	F	F	-	F	50	40	20	F	F	-	0	660	1384	2	3.55	0	352	203	2	3.07
2	-3	F	F	-	F	50	40	1	F	F	-	0	776	1810	2	0.50	0	446	395	2	0.58
3	-3	F	F	-	F	10	40	20	F	F	-	0	193	342	2	1.17	0	168	135	2	1.13
4	-3	F	F	-	F	10	40	1	F	F	-	0	158	253	2	0.36	0	176	122	2	0.43
5	-3	F	F	-	F	5	40	20	F	F	-	0	142	251	2	0.89	0	98	63	2	0.91
6	-3	F	F	-	F	5	40	1	F	F	-	0	100	159	2	0.35	0	99	71	2	0.42
7	-3	F	F	-	F	5	40	20	F	T	F	0	140	264	2	1.15	0	81	71	2	1.16
8	-3	F	F	-	F	5	40	20	F	T	T	0	97	158	2	1.14	0	104	79	2	2.80
9	-3	F	F	-	F	5	40	20	T	T	T	0	97	158	2	1.15	13	76	45	2	1.15
10	-3	F	F	-	T	50	40	5	F	F	-	0	160	277	2	1.44	0	138	69	2	1.23
11	-3	F	F	-	T	50	40	1	F	F	-	0	238	407	2	0.54	0	138	69	2	0.56
12	-3	F	F	-	T	10	40	5	F	F	-	0	84	149	2	0.65	0	81	42	2	0.71
13	-3	F	F	-	T	10	40	1	F	F	-	0	87	149	2	0.39	0	106	53	2	0.46
14	-3	F	F	-	T	5	40	5	F	F	-	0	83	151	2	0.56	0	76	38	2	0.63
15	-3	F	F	-	T	5	40	1	F	F	-	0	88	151	2	0.37	0	57	29	2	0.44
16	-3	F	F	-	T	10	40	5	F	T	F	0	85	148	2	0.80	0	49	25	2	0.82
17	-3	F	F	-	T	10	40	5	F	T	T	0	95	150	2	0.81	0	74	38	2	0.88
18	-3	F	F	-	T	10	40	5	T	T	T	0	95	150	2	0.81	0	74	38	2	0.88
$k = 4$																						
19	-3	F	F	-	F	5	40	20	F	F	-	1	315	552	10	1.75	1	190	124	12	2.26
20	-3	F	F	-	T	10	40	5	F	F	-	1	211	357	8	1.19	1	154	80	12	2.29
21	-3	F	F	-	T	10	40	2	F	F	-	1	225	371	8	0.80	1	204	106	12	1.44
22	-3	F	F	-	T	5	40	5	F	F	-	1	207	358	8	1.02	1	172	88	10	1.80
23	-3	F	F	-	T	5	40	2	F	F	-	1	218	358	8	0.71	1	148	77	10	1.11

Table A.15:

A.2 Wedge1 Data-Set

In this section we present decomposition statistics for the *Wedge1* data-set. This data-set is a three dimensional finite element mesh with tetrahedral elements, which originates from the FLITE3D project [BMT96]. This was an EPCC industrial consultancy project to parallelise a British Aerospace unstructured mesh Euler-solver used in aircraft design.

The physical geometry which the mesh models is that of a rectilinear region, spanned by a solid wedge-shaped intrusion. This mesh forms the most dense mesh in a series of three multigrid meshes; Wedge3, Wedge2, Wedge1 (in ascending order of mesh density). The Wedge3 mesh is that previously illustrated in figures 7.10 to 7.11 of section 7.6. Each of the three meshes have elements of approximately uniform size within themselves, and differ only in their mesh density relative to one and other.

Here we explore essentially the same set of algorithms and parameter settings as we did for the Widget data-set, with the one exception of table A.27, where `MD_KL_RANDOM_RETRIES` takes the value zero, compared to its value of one in the corresponding table in the previous section (table A.12). Further, the highest values of k used is now 16, rather than 8 as it was for the Widget data-set.

DUAL GRAPH								
	t(s)			Statistics				
Dual	Coord	Border	Other	n_e	n_e^{min}	n_e^{mean}	n_e^{max}	n_v
NODES	5.79	5.76	5.53	567104	15	62.8	99	:
EDGES	2.58	2.56	2.33	146419	6	16.2	22	18037
FACES	1.65	1.57	1.35	34788	2	3.8	4	:

Table A.16:

SIMPLE																
		NODES					EDGES					FACES				
case	MD_SL_CM_TIMES	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
SR																
$k=2$																
1	-	1	18037	283481	2	0.19	1	18035	73225	2	0.10	1	16753	17454	2	0.07
SC																
$k=2$																
2	-	1	18037	288042	2	0.17	1	18037	77100	2	0.08	1	17825	21687	2	0.06
SL																
$k=2$																
3	0	1	13943	119336	2	0.15	1	8997	19679	2	0.07	1	4427	2658	2	0.05
4	1	1	4079	32433	2	0.69	1	2341	5071	2	0.30	1	1269	923	2	0.23
5	2	1	2204	17175	2	1.21	1	1362	2927	2	0.53	1	630	488	2	0.39
6	3	1	2022	15786	2	1.74	1	1261	2649	2	0.79	1	583	437	2	0.48
7	4	1	2204	17175	2	2.22	1	1370	2917	2	0.97	1	646	485	2	0.62
$k=4$																
8	0	1	16799	205747	12	0.88	1	12877	34832	12	0.64	1	7349	4814	12	0.57
9	1	1	10516	84573	6	1.45	1	6728	14563	6	0.88	1	3400	2560	6	0.71
10	2	1	7077	58014	6	1.96	1	4568	9913	6	1.10	1	2072	1582	6	0.86
11	3	1	6954	56914	6	2.47	1	4326	9304	6	1.32	1	2015	1504	6	0.99
12	4	1	7077	58014	6	3.01	1	4575	9979	6	1.65	1	2084	1562	6	1.13
$k=16$																
13	0	1	17768	295649	240	1.59	1	15607	52055	240	1.03	1	10378	7452	240	0.87
14	1	1	17088	277775	56	2.20	1	16589	62023	48	1.30	1	13906	11388	30	1.03
15	2	1	16940	244197	44	2.79	1	15962	44493	30	1.53	1	9136	6848	30	1.18
16	3	1	16853	246474	44	3.37	1	16064	43094	30	1.74	1	9156	6871	30	1.33
17	4	1	16940	244197	44	3.76	1	15799	43277	30	1.96	1	9270	6995	30	1.45

Table A.17:

GREEDY															
case	NODES					EDGES					FACES				
	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$															
1	1	5313	41439	2	0.25	1	2551	5530	2	0.14	1	1258	906	2	0.10
$k = 4$															
2	1	7354	62169	12	0.77	1	4628	10281	12	0.34	1	2008	1515	12	0.22
$k = 16$															
3	1	13388	137969	136	3.47	1	9363	23688	122	1.34	1	4245	3372	112	0.78

Table A.18:

RLB																	
			NODES					EDGES					FACES				
case	MD_RLB_CM_TIMES	MD_RLB_CM_BEST	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																	
1	0	-	1	4419	33591	2	0.25	1	2337	5066	2	0.14	1	972	729	2	0.10
2	1	F	1	2300	17800	2	0.47	1	1595	3386	2	0.27	1	649	482	2	0.21
3	2	F	1	2588	20068	2	0.68	1	1450	3182	2	0.41	1	565	430	2	0.33
4	3	F	1	2300	17800	2	0.89	1	1595	3386	2	0.55	1	649	482	2	0.44
5	4	F	1	2588	20068	2	1.11	1	1450	3182	2	0.68	1	565	430	2	0.55
6	5	F	1	2300	17800	2	1.33	1	1595	3386	2	0.81	1	649	482	2	0.67
7	1	T	1	2300	17800	2	0.49	1	1595	3386	2	0.27	1	649	482	2	0.21
8	2	T	1	2300	17800	2	0.80	1	1450	3182	2	0.41	1	565	430	2	0.33
9	3	T	1	2300	17800	2	0.89	1	1450	3182	2	0.66	1	565	430	2	0.50
$k = 4$																	
10	0	-	1	8566	71407	12	0.65	1	5797	12679	8	0.31	1	2178	1683	12	0.20
11	1	T	1	6940	56401	10	1.08	1	4288	9495	10	0.61	1	1877	1349	10	0.44
12	2	T	1	6940	56401	10	1.72	1	4342	9766	10	0.91	1	1836	1390	10	0.68
13	3	T	1	6940	56401	10	1.93	1	4342	9766	10	1.22	1	1836	1390	10	1.01
$k = 16$																	
14	0	-	1	14637	154803	136	1.42	1	10755	27392	114	0.64	1	4868	3909	114	0.44
15	1	T	1	13856	137855	104	2.28	1	9272	22688	94	1.23	1	4564	3592	108	0.94
16	2	T	1	13838	136258	104	3.47	1	8893	22091	110	1.82	1	4318	3372	104	1.43
17	3	T	1	14064	137710	102	4.03	1	9146	22735	108	2.46	1	4339	3392	92	2.05

Table A.19:

RCB																				
					NODES					EDGES					FACES					
case	MD_RCB_CYCLE	MD_RCB_FIXED	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																				
1	F	T	F	-	1	1708	11816	2	0.18	1	993	2024	2	0.09	1	478	318	2	0.06
2	T	F	F	-	1	3775	28145	2	0.18	1	2150	4621	2	0.09	1	1035	694	2	0.06
3	F	F	F	-	1	1708	11816	2	0.18	1	993	2024	2	0.09	1	478	318	2	0.06
4	F	F	T	5	900	1555	10722	2	0.20	827	903	1829	2	0.10	731	443	267	2	0.07
5	F	F	T	10	1802	1427	9672	2	0.23	1802	816	1683	2	0.10	1356	392	243	2	0.07
6	F	F	T	15	2684	1281	8912	2	0.23	2676	736	1476	2	0.11	2594	354	214	2	0.08
7	F	F	T	20	2792	1383	8890	2	0.25	3079	726	1447	2	0.12	3079	340	206	2	0.08
$k = 4$																				
8	F	T	F	-	1	5783	42716	6	0.98	1	3311	6946	6	0.67	1	1597	1059	6	0.58
9	T	F	F	-	1	5794	46324	12	0.55	1	3420	7532	12	0.23	1	1670	1142	12	0.14
10	F	F	F	-	1	5460	41320	8	0.76	1	3162	6782	8	0.45	1	1531	1038	8	0.36
11	F	F	T	5	868	5017	37705	8	0.75	585	2912	6123	8	0.43	606	1440	890	8	0.33
12	F	F	T	10	1954	4786	34291	8	0.81	1700	2718	5706	8	0.46	750	1357	844	8	0.37
13	F	F	T	15	3097	4171	31284	8	0.88	3039	2487	5148	8	0.52	3034	1208	756	8	0.40
14	F	F	T	20	3758	4245	30551	8	0.94	3924	2337	4845	8	0.55	3900	1132	701	8	0.44
$k = 16$																				
15	F	T	F	-	1	17886	211045	54	1.83	1	15671	34862	30	1.10	1	7902	5458	30	0.92
16	T	F	F	-	1	12975	111075	104	1.19	1	8056	18423	104	0.47	1	4088	2839	96	0.27
17	F	F	F	-	1	9682	85626	126	1.44	1	6099	14156	110	0.72	1	3113	2181	96	0.51
18	F	F	T	5	491	9434	82929	126	1.50	397	5812	13272	112	0.72	406	3014	1939	84	0.51
19	F	F	T	10	989	9526	81527	114	1.59	751	5818	13155	94	0.78	618	2954	1883	90	0.55
20	F	F	T	15	1182	9405	79759	102	1.72	1325	5791	12888	100	0.85	1343	2949	1876	82	0.59
21	F	F	T	20	1140	9438	79124	98	1.83	1672	5577	12448	86	0.91	1653	2859	1792	78	0.64

Table A.20:

RIB																	
			NODES					EDGES					FACES				
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_c$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_c$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_c$	s_{adj}	t(s)
$k = 2$																	
1	F	-	1	1720	11813	2	0.26	1	996	2040	2	0.17	1	478	327	2	0.15
2	T	5	901	1556	10711	2	0.28	833	898	1825	2	0.18	696	443	266	2	0.15
3	T	10	1802	1414	9666	2	0.29	1799	813	1674	2	0.18	1362	393	245	2	0.15
4	T	15	2676	1264	8916	2	0.31	2687	726	1478	2	0.19	2597	357	219	2	0.18
5	T	20	2790	1389	8908	2	0.33	3079	722	1450	2	0.20	3165	338	204	2	0.19
6	T	25	2790	1389	8908	2	0.37	3079	722	1450	2	0.21	3165	338	204	2	0.17
7	T	30	2790	1389	8908	2	0.40	3079	722	1450	2	0.22	3165	338	204	2	0.18
$k = 4$																	
8	F	-	1	4932	36529	12	0.71	1	2860	6123	12	0.41	1	1395	957	12	0.31
9	T	5	545	4734	34228	8	0.76	584	2687	5531	8	0.42	569	1474	914	8	0.33
10	T	10	1973	4676	32349	8	0.79	1844	2597	5324	8	0.44	1554	1268	786	8	0.33
11	T	15	3063	4397	31863	8	0.83	3091	2525	5190	8	0.44	2316	1234	764	8	0.35
12	T	20	3756	4452	31980	8	0.89	3868	2442	5113	8	0.46	3825	1174	734	8	0.35
13	T	25	4347	4229	30097	8	0.95	4223	2354	4848	8	0.48	3825	1157	718	8	0.38
14	T	30	4911	4057	28510	8	1.00	5168	2241	4610	8	0.50	5169	1099	674	8	0.37
$k = 16$																	
15	F	-	1	10434	93577	130	1.54	1	6638	15462	112	0.82	1	3425	2388	102	0.63
16	T	5	368	9615	84349	120	1.65	343	5985	13623	104	0.86	348	3597	2327	98	0.65
17	T	10	1120	9559	81704	112	1.76	781	5874	13207	92	0.90	777	3035	1950	82	0.66
18	T	15	1476	9545	81791	92	1.86	1426	5892	13214	92	0.95	973	2988	1888	78	0.70
19	T	20	2900	9589	82795	94	1.91	2973	5958	13496	90	0.96	2679	3057	1979	88	0.71
20	T	25	3835	9206	77740	96	2.14	3772	5609	12573	90	0.99	3316	2929	1866	90	0.76
21	T	30	3022	8643	73142	98	2.22	3700	5398	11970	88	1.06	4548	2722	1723	78	0.76

Table A.21:

RSB																				
					NODES					EDGES					FACES					
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																				
1	-5	T	F	-	1	1641	11659	2	31.08	1	958	1954	2	31.66	1	457	294	2	68.73
2	-4	T	F	-	1	1641	11659	2	22.42	1	957	1953	2	24.31	1	456	295	2	50.00
3	-3	T	F	-	1	1646	11656	2	18.71	1	955	1950	2	18.56	1	454	311	2	30.21
4	-2	T	F	-	1	1643	11665	2	14.21	1	959	1981	2	11.71	1	421	297	2	7.01
5	-1	T	F	-	1	1662	11736	2	10.22	1	924	1891	2	3.32	1	2217	1504	2	0.42
6	-5	F	F	-	1	1641	11659	2	15.75	1	958	1954	2	7.53	1	457	294	2	7.50
7	-4	F	F	-	1	1641	11659	2	13.25	1	957	1953	2	6.43	1	456	295	2	5.84
8	-3	F	F	-	1	1646	11656	2	11.55	1	955	1950	2	5.31	1	454	311	2	4.05
9	-2	F	F	-	1	1643	11665	2	9.51	1	959	1981	2	3.98	1	421	297	2	1.58
10	-1	F	F	-	1	1662	11736	2	7.27	1	924	1891	2	1.82	1	2217	1504	2	0.30
$k = 16$																				
11	-5	T	F	-	1	9574	84641	124	83.72	1	6080	13963	106	85.71	1	3026	2093	90	178.59
12	-4	T	F	-	1	9564	84637	124	70.49	1	6076	13957	106	66.04	1	3021	2086	88	119.10
13	-3	T	F	-	1	9547	84643	124	58.62	1	6082	13985	106	49.72	1	3031	2074	94	75.00
14	-2	T	F	-	1	9589	84692	124	45.78	1	6087	13955	106	28.98	1	3529	2469	94	17.07
15	-1	T	F	-	1	9645	85224	116	29.84	1	7323	17036	104	9.14	1	7646	5685	200	1.58
16	-5	F	F	-	1	9574	84641	124	52.17	1	6080	13963	106	24.70	1	3026	2091	90	27.37
17	-4	F	F	-	1	9564	84637	124	45.90	1	6076	13957	106	20.69	1	3020	2086	88	18.18
18	-3	F	F	-	1	9545	84640	124	39.87	1	6081	13993	104	17.04	1	3031	2074	94	12.45
19	-2	F	F	-	1	9589	84692	124	32.97	1	6087	13955	106	11.73	1	3529	2467	94	4.68
20	-1	F	F	-	1	9645	85224	116	23.05	1	7330	17051	104	5.62	1	7646	5685	200	1.16

Table A.22:

RSB																				
					NODES					EDGES					FACES					
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																				
1	-3	F	F	-	1	1646	11656	2	11.43	1	955	1950	2	7.13	1	454	311	2	3.93
2	-3	F	T	5	899	1549	10596	2	11.60	864	872	1793	2	5.40	715	428	269	2	3.96
3	-3	F	T	10	1800	1391	9440	2	11.68	1682	800	1611	2	5.38	1774	383	238	2	3.94
4	-3	F	T	15	2703	1252	8717	2	11.55	2675	707	1413	2	5.22	2527	347	208	2	3.91
5	-3	F	T	20	2887	1346	8613	2	11.70	2675	707	1413	2	5.31	3487	332	193	2	3.90
6	-3	F	T	25	2887	1346	8613	2	11.62	2675	707	1413	2	5.27	3487	332	193	2	4.06
7	-3	F	T	30	2887	1346	8613	2	11.70	2675	707	1413	2	5.32	3487	332	193	2	4.01
$k = 4$																				
8	-3	F	F	-	1	4176	30324	8	24.95	1	2379	4925	8	11.30	1	1109	728	8	8.54
9	-3	F	T	5	623	4054	28800	8	24.91	677	2217	4562	8	10.94	589	1285	801	8	6.00
10	-3	F	T	10	1101	3904	27640	8	25.96	1067	2144	4366	8	11.32	1754	1230	766	8	6.61
11	-3	F	T	15	1491	3732	26967	8	26.03	1523	2053	4168	8	11.21	2462	1201	745	8	5.89
12	-3	F	T	20	3824	3721	26088	8	25.92	1523	2053	4168	8	11.21	3937	1068	647	8	5.54
13	-3	F	T	25	4395	3522	24530	8	26.04	4247	1940	3910	8	11.25	4665	1070	645	8	5.59
14	-3	F	T	30	5011	3271	23572	10	25.99	4673	1903	3849	8	11.48	5458	1022	628	8	5.61
$k = 16$																				
15	-3	F	F	-	1	9545	84640	124	39.33	1	6081	13993	104	16.75	1	3031	2074	94	12.52
16	-3	F	T	5	391	9429	81742	112	40.35	359	5908	13347	98	17.32	232	2981	1894	82	9.88
17	-3	F	T	10	873	9331	80493	106	41.15	731	5716	12784	102	17.17	827	3182	2038	88	11.22
18	-3	F	T	15	1405	9337	78568	90	40.53	1040	5629	12494	88	16.61	1447	3245	2068	92	10.56
19	-3	F	T	20	2889	9103	75826	78	42.30	1472	5517	12162	80	16.90	1574	2846	1767	80	12.78
20	-3	F	T	25	3862	8691	72048	82	42.23	3191	5417	11864	80	18.09	2452	2859	1791	80	11.32
21	-3	F	T	30	3031	7943	66943	94	44.54	2570	5196	11247	66	17.99	5124	2475	1552	76	11.96

Table A.23:

SR+KL																					
							NODES					EDGES					FACES				
case	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k=2$																					
1	F	F	-	-	-	0	1	1631	11043	2	72.45	1	926	1806	2	26.79	1	2099	1081	2	26.68
2	T	F	-	-	-	0	1	1631	11043	2	16.69	1	926	1807	2	13.39	1	4739	2463	2	3.96
3	F	T	50	-	-	0	1	1631	11043	2	45.42	1	926	1807	2	17.13	1	4739	2463	2	5.38
4	F	T	20	-	-	0	1	1631	11043	2	25.39	1	926	1807	2	11.17	1	4739	2463	2	3.71
5	F	T	10	-	-	0	1	1631	11043	2	18.52	1	926	1807	2	9.21	1	4739	2463	2	3.02
6	F	T	5	-	-	0	1	1631	11043	2	15.24	1	1691	3365	2	7.22	1	4739	2463	2	2.71
7	F	T	1	-	-	0	1	1598	11271	2	10.85	1	3990	8032	2	5.24	1	4735	2465	2	2.46
8	F	F	-	-	-	1	1	1667	11033	2	154.66	1	922	1803	2	39.91	1	2060	1060	2	38.59
9	F	F	-	-	-	3	1	1673	11032	2	525.58	1	913	1792	2	70.08	1	2060	1060	2	41.52
10	F	F	-	-	-	5	1	1673	11032	2	236.72	1	913	1792	2	76.43	1	2060	1060	2	44.35
11	T	F	-	-	-	1	1	1667	11033	2	19.76	1	926	1807	2	13.70	1	4734	2459	2	4.84
12	T	F	-	-	-	3	1	1673	11032	2	22.14	1	926	1807	2	14.26	1	4734	2459	2	5.56
13	T	F	-	-	-	5	1	1673	11032	2	22.79	1	926	1807	2	14.86	1	4737	2458	2	7.67
14	F	T	10	-	-	1	1	1667	11033	2	29.92	1	926	1807	2	9.77	1	4734	2459	2	4.17
15	F	T	10	-	-	3	1	1673	11032	2	87.82	1	926	1807	2	11.00	1	4734	2459	2	5.69
16	F	T	10	-	-	5	1	1673	11032	2	41.39	1	926	1807	2	12.19	1	4737	2458	2	9.50
17	F	T	5	-	-	1	1	1667	11033	2	22.35	1	1691	3365	2	7.71	1	4734	2459	2	4.36
18	F	T	5	-	-	3	1	1673	11032	2	27.79	1	1691	3365	2	8.56	1	4734	2459	2	4.32
19	F	T	5	-	-	5	1	1673	11032	2	29.67	1	1691	3365	2	9.45	1	4737	2458	2	6.95

Table A.24:

RIB+KL																									
									NODES					EDGES						FACES					
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)
$k = 2$																									
1	F	-	F	F	-	F	-	0	1	1547	11243	2	51.15	1	931	1845	2	13.37	1	478	246	2	4.56
2	F	-	T	F	-	F	-	0	1	1547	11243	2	2.70	1	931	1845	2	1.31	1	478	246	2	0.93
3	F	-	F	T	5	F	-	0	1	1547	11243	2	5.02	1	931	1845	2	1.92	1	478	246	2	1.53
4	F	-	T	F	-	F	-	1	1	1547	11243	2	3.16	1	931	1845	2	1.58	1	478	246	2	1.21
5	F	-	T	F	-	F	-	5	1	1547	11243	2	4.97	1	931	1845	2	2.62	1	478	246	2	2.18
6	F	-	F	F	-	T	60	0	1	1547	11243	2	29.57	1	931	1845	2	7.91	1	478	246	2	2.80
7	F	-	F	F	-	T	20	0	1	1547	11243	2	9.32	1	931	1845	2	2.68	1	478	246	2	1.02
8	F	-	F	F	-	T	10	0	1	1547	11243	2	4.33	1	931	1845	2	1.37	1	478	246	2	0.58
9	F	-	F	F	-	T	5	0	1	1658	11512	2	1.24	1	934	1861	2	0.84	1	478	246	2	0.36
10	F	-	T	F	-	T	60	0	1	1547	11243	2	2.22	1	931	1845	2	0.91	1	478	246	2	0.64
11	F	-	T	F	-	T	20	0	1	1547	11243	2	1.72	1	931	1845	2	0.52	1	478	246	2	0.34
12	F	-	T	F	-	T	10	0	1	1547	11243	2	1.41	1	931	1845	2	0.41	1	478	246	2	0.26
13	F	-	T	F	-	T	5	0	1	1658	11512	2	0.76	1	934	1861	2	0.36	1	478	246	2	0.22

Table A.25:

RIB+KL																					
									NODES					EDGES					FACES				
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)
$k = 16$																							
1	F	-	F	F	-	F	-	0	1	9905	83307	106	223.52	1	6170	13658	98	57.72	1	3455	1880	94	19.46
2	F	-	T	F	-	F	-	0	1	9882	83370	106	21.94	1	6173	13642	98	6.95	1	3455	1880	94	4.07
3	F	-	F	T	5	F	-	0	1	9882	83370	106	26.81	1	6173	13642	98	8.57	1	3455	1880	94	4.72
4	F	-	T	F	-	T	60	0	1	10039	84989	104	17.58	1	6173	13642	98	5.32	1	3443	1866	96	2.88
5	F	-	T	F	-	T	20	0	1	10884	94047	124	9.30	1	6424	14422	110	2.85	1	3443	1866	96	1.56
6	F	-	T	F	-	T	10	0	1	11234	99870	128	5.59	1	6434	14539	106	2.09	1	3423	1859	90	1.24
7	F	-	T	F	-	T	5	0	1	10716	93302	126	3.30	1	6391	14421	112	1.48	1	3465	1902	88	0.98
8	F	-	F	F	-	F	-	1	1	9909	83305	106	266.65	1	6065	13364	96	81.22	1	3429	1856	90	29.52
9	F	-	T	F	-	F	-	1	1	9873	83373	106	25.02	1	6133	13524	96	9.43	1	3429	1856	90	5.84
10	F	-	F	T	5	F	-	1	1	9873	83373	106	31.92	1	6151	13624	94	11.14	1	3429	1856	90	6.89
11	F	-	T	F	-	T	60	1	1	10039	84989	104	19.56	1	6138	13586	100	6.43	1	3443	1866	96	3.52
12	F	-	T	F	-	T	20	1	1	10880	93940	124	11.15	1	6339	14176	106	3.73	1	3443	1866	96	1.81
13	F	-	T	F	-	T	10	1	1	11347	100649	122	6.56	1	6435	14512	108	2.45	1	3423	1859	90	1.37
14	F	-	T	F	-	T	5	1	1	10710	93301	126	3.75	1	6378	14392	116	1.68	1	3461	1901	90	1.10

Table A.26:

RSB+KL																										
											NODES					EDGES					FACES				
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k = 2$																										
1	-3	F	F	-	F	F	-	F	-	0	1	1556	11227	2	53.67	1	923	1832	2	22.20	1	463	238	2	12.99
2	-3	F	F	-	T	F	-	F	-	0	1	1556	11227	2	13.74	1	923	1832	2	6.80	1	463	238	2	4.73
3	-3	F	F	-	F	T	5	F	-	0	1	1556	11227	2	15.48	1	923	1832	2	7.61	1	463	238	2	4.91
4	-3	F	F	-	T	F	-	F	-	1	1	1556	11227	2	14.14	1	923	1832	2	7.06	1	463	238	2	5.09
5	-3	F	F	-	T	F	-	F	-	5	1	1556	11227	2	15.61	1	921	1831	2	9.51	1	463	238	2	5.98
6	-3	F	F	-	F	F	-	T	60	0	1	1556	11227	2	35.70	1	923	1832	2	15.28	1	463	238	2	6.66
7	-3	F	F	-	F	F	-	T	20	0	1	1556	11227	2	19.08	1	923	1832	2	8.55	1	463	238	2	4.85
8	-3	F	F	-	F	F	-	T	10	0	1	1556	11227	2	15.04	1	923	1832	2	6.91	1	463	238	2	4.42
9	-3	F	F	-	F	F	-	T	5	0	1	1582	11277	2	13.62	1	922	1834	2	5.97	1	463	238	2	4.20
10	-3	F	F	-	T	F	-	T	60	0	1	1556	11227	2	13.38	1	923	1832	2	6.46	1	463	238	2	4.47
11	-3	F	F	-	T	F	-	T	20	0	1	1556	11227	2	13.06	1	923	1832	2	5.80	1	463	238	2	4.14
12	-3	F	F	-	T	F	-	T	10	0	1	1556	11227	2	12.75	1	923	1832	2	5.67	1	463	238	2	4.09
13	-3	F	F	-	T	F	-	T	5	0	1	1582	11277	2	12.45	1	922	1834	2	5.58	1	463	238	2	4.00
$k = 16$																										
14	-3	F	F	-	F	F	-	F	-	0	1	9784	82191	108	186.93	1	5856	12915	94	68.56	1	3122	1695	88	30.79
15	-3	F	F	-	T	F	-	F	-	0	1	9830	82871	108	48.71	1	5856	12915	94	21.25	1	3122	1695	88	15.51
16	-3	F	F	-	F	T	5	F	-	0	1	9830	82871	108	53.63	1	5856	12915	94	23.51	1	3122	1695	88	16.25
17	-3	F	F	-	T	F	-	T	60	0	1	9830	82871	108	47.26	1	5856	12915	94	19.74	1	3122	1695	88	14.31
18	-3	F	F	-	T	F	-	T	20	0	1	9839	82847	108	44.86	1	5856	12915	94	18.23	1	3122	1695	88	13.10
19	-3	F	F	-	T	F	-	T	10	0	1	9860	82875	108	42.52	1	5866	12963	96	17.27	1	3117	1686	88	12.76
20	-3	F	F	-	T	F	-	T	5	0	1	9793	82794	108	41.02	1	5850	13004	96	17.31	1	3103	1689	86	12.59

Table A.27:

SR+MOB																					
								NODES					EDGES					FACES				
case	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k=2$																					
1	F	50	40	20	F	F	-	1	6091	58833	2	95.76	1	5674	20949	2	47.50	1	7734	8708	2	31.86
2	F	50	40	10	F	F	-	1	6351	59104	2	48.54	1	5928	22119	2	24.06	1	8532	10028	2	15.92
3	F	50	40	1	F	F	-	1	12425	116014	2	5.60	1	8931	27544	2	2.46	1	8954	10486	2	1.75
4	F	10	40	20	F	F	-	1	1696	11775	2	21.52	1	1880	3995	2	12.73	1	2126	1989	2	10.44
5	F	10	40	10	F	F	-	1	1874	11908	2	11.42	1	1930	4175	2	6.54	1	2663	2295	2	5.37
6	F	10	40	1	F	F	-	1	5025	35180	2	1.60	1	5377	11484	2	0.82	1	5596	4122	2	0.65
7	F	5	40	20	F	F	-	1	3371	22246	2	14.25	1	1957	4003	2	8.79	1	2431	1759	2	7.47
8	F	5	40	10	F	F	-	1	3306	22220	2	7.40	1	2515	5187	2	4.52	1	3298	2103	2	3.77
9	F	5	40	1	F	F	-	1	4841	31744	2	1.07	1	5934	12209	2	0.61	1	5292	3385	2	0.49
10	F	10	40	20	F	T	F	1	2679	17432	2	32.67	1	2322	6031	2	19.33	1	2320	2074	2	15.54
11	F	10	40	20	F	T	T	1	1727	11942	2	32.07	1	1982	4286	2	19.49	1	2320	2074	2	15.55
12	F	10	40	20	T	T	T	1	1707	11923	2	32.08	4	1944	4234	2	19.39	23	2020	1552	2	15.60
13	T	50	40	20	F	F	-	1	1634	11204	2	131.64	1	948	1893	2	72.54	1	2883	1587	2	59.17
14	T	50	40	10	F	F	-	1	1668	11250	2	66.23	1	970	1921	2	36.76	1	3028	1672	2	29.61
15	T	50	40	1	F	F	-	1	2431	16536	2	8.05	1	3091	6245	2	4.23	1	3518	1919	2	2.89
16	T	10	40	20	F	F	-	1	1679	11403	2	34.62	1	887	1767	2	21.31	1	949	521	2	20.41
17	T	10	40	10	F	F	-	1	1697	11411	2	18.86	1	1204	2406	2	11.04	1	1426	763	2	11.01
18	T	10	40	1	F	F	-	1	2980	20464	2	2.96	1	2626	5242	2	1.63	1	4830	2547	2	1.40
19	T	5	40	20	F	F	-	1	1719	11518	2	23.28	1	908	1791	2	15.59	1	1608	848	2	16.40
20	T	5	40	10	F	F	-	1	2904	19409	2	12.40	1	1333	2649	2	8.06	1	2080	1073	2	8.33
21	T	5	40	1	F	F	-	1	3269	22237	2	2.28	1	4749	9636	2	1.17	1	5188	2731	2	1.15
22	T	10	40	20	F	T	F	1	1725	11103	2	48.79	1	886	1759	2	29.54	1	752	419	2	25.85
23	T	10	40	20	F	T	T	1	1556	11223	2	48.14	1	913	1810	2	30.19	1	729	393	2	26.40
24	T	10	40	20	T	T	T	1	1556	11223	2	48.15	1	913	1810	2	30.42	1	729	393	2	39.80

Table A.28:

RIB+MOB																									
										NODES					EDGES					FACES				
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k = 2$																									
1	F	-	F	50	40	20	F	F	-	1	9450	89524	2	89.93	1	6524	21716	2	42.40	1	4811	5738	2	26.38
2	F	-	F	50	40	1	F	F	-	1	6868	63729	2	4.92	1	8462	25940	2	2.28	1	4811	5738	2	1.46
3	F	-	F	10	40	20	F	F	-	1	1736	11619	2	18.26	1	1685	3924	2	11.30	1	1748	1680	2	8.17
4	F	-	F	10	40	1	F	F	-	1	1754	11974	2	1.19	1	1754	3990	2	0.69	1	1745	1739	2	0.54
5	F	-	F	5	40	20	F	F	-	1	1703	11540	2	11.69	1	1208	2556	2	6.84	1	956	927	2	5.67
6	F	-	F	5	40	1	F	F	-	1	1704	11651	2	0.87	1	1115	2356	2	0.50	1	1013	928	2	0.42
7	F	-	F	5	40	20	F	T	F	1	1741	11617	2	16.81	1	1236	2703	2	9.90	1	706	726	2	8.37
8	F	-	F	5	40	20	F	T	T	1	1665	11278	2	16.28	1	1067	2267	2	9.77	1	802	806	2	8.09
9	F	-	F	5	40	20	T	T	T	1	1665	11278	2	16.42	2	1059	2260	2	9.81	25	581	447	2	8.48
10	F	-	T	50	40	5	F	F	-	1	1656	11240	2	31.63	1	1664	3381	2	15.55	1	2966	1560	2	10.00
11	F	-	T	50	40	1	F	F	-	1	1591	11245	2	6.41	1	1413	2817	2	3.20	1	1515	831	2	1.97
12	F	-	T	10	40	5	F	F	-	1	1642	11205	2	7.26	1	923	1828	2	4.37	1	1119	591	2	3.62
13	F	-	T	10	40	1	F	F	-	1	1596	11263	2	1.65	1	935	1861	2	0.99	1	802	425	2	0.85
14	F	-	T	5	40	5	F	F	-	1	1708	11416	2	4.67	1	917	1826	2	3.01	1	990	516	2	2.75
15	F	-	T	5	40	1	F	F	-	1	1691	11406	2	1.15	1	930	1870	2	0.73	1	579	296	2	0.68
16	F	-	T	10	40	5	F	T	F	1	1664	11204	2	10.23	1	926	1848	2	6.27	1	493	263	2	4.74
17	F	-	T	10	40	5	F	T	T	1	1644	11189	2	10.17	1	936	1849	2	6.29	1	701	369	2	5.26
18	F	-	T	10	40	5	T	T	T	1	1644	11189	2	10.14	1	936	1849	2	6.25	1	701	369	2	5.20
$k = 16$																									
19	F	-	F	5	40	20	F	F	-	1	9999	83578	98	46.73	1	6651	15193	132	27.40	1	4603	3334	238	22.92
20	F	-	T	10	40	5	F	F	-	1	10515	85498	100	28.86	1	6486	14167	94	16.89	1	4410	2485	134	14.37
21	F	-	T	10	40	2	F	F	-	1	10249	84724	102	12.67	1	6456	14114	100	7.22	1	4514	2462	122	6.22
22	F	-	T	5	40	5	F	F	-	1	10054	83975	100	18.71	1	5839	12858	94	11.98	1	4299	2364	102	10.91
23	F	-	T	5	40	2	F	F	-	1	9955	83576	100	8.45	1	6398	14229	110	5.33	1	3784	2078	108	4.72

Table A.29:

RSB+MOB																										
												NODES					EDGES					FACES				
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k = 2$																										
1	-3	F	F	-	F	50	40	20	F	F	-	1	6183	61223	2	100.22	1	7516	24978	2	47.71	1	5384	6458	2	32.11
2	-3	F	F	-	F	50	40	1	F	F	-	1	10152	91879	2	16.08	1	8169	25116	2	7.54	1	5384	6458	2	5.35
3	-3	F	F	-	F	10	40	20	F	F	-	1	1761	11857	2	29.73	1	1698	3945	2	15.62	1	1688	1663	2	11.88
4	-3	F	F	-	F	10	40	1	F	F	-	1	1911	12071	2	12.42	1	1747	4036	2	5.79	1	1752	1763	2	4.35
5	-3	F	F	-	F	5	40	20	F	F	-	1	1683	11372	2	22.42	1	1209	2534	2	12.17	1	1050	969	2	9.41
6	-3	F	F	-	F	5	40	1	F	F	-	1	1673	11510	2	12.14	1	1122	2366	2	5.74	1	972	877	2	4.26
7	-3	F	F	-	F	5	40	20	F	T	F	1	1862	12383	2	27.78	1	1421	3574	2	15.14	1	742	754	2	14.38
8	-3	F	F	-	F	5	40	20	F	T	T	1	1653	11273	2	27.71	1	1069	2263	2	16.83	1	886	872	2	11.89
9	-3	F	F	-	F	5	40	20	T	T	T	1	1653	11273	2	27.73	2	1061	2256	2	16.87	72	609	456	2	12.54
10	-3	F	F	-	T	50	40	5	F	F	-	1	1690	11232	2	43.28	1	1451	2915	2	20.92	1	2151	1136	2	13.95
11	-3	F	F	-	T	50	40	1	F	F	-	1	1544	11279	2	17.68	1	1488	2972	2	8.50	1	1620	887	2	5.93
12	-3	F	F	-	T	10	40	5	F	F	-	1	1614	11187	2	18.19	1	938	1843	2	9.44	1	849	449	2	7.61
13	-3	F	F	-	T	10	40	1	F	F	-	1	1671	11318	2	12.98	1	931	1849	2	6.20	1	875	451	2	4.65
14	-3	F	F	-	T	5	40	5	F	F	-	1	1628	11204	2	15.71	1	918	1812	2	8.15	1	752	390	2	6.77
15	-3	F	F	-	T	5	40	1	F	F	-	1	1611	11242	2	12.35	1	926	1853	2	5.94	1	528	273	2	4.52
16	-3	F	F	-	T	10	40	5	F	T	F	1	1663	11235	2	21.60	1	915	1813	2	11.54	1	510	280	2	8.51
17	-3	F	F	-	T	10	40	5	F	T	T	1	1638	11181	2	21.40	1	930	1853	2	11.52	1	686	356	2	8.98
18	-3	F	F	-	T	10	40	5	T	T	T	1	1638	11181	2	21.21	1	930	1853	2	11.50	1	686	356	2	8.92
$k = 16$																										
19	-3	F	F	-	F	5	40	20	F	F	-	1	10170	83920	104	82.26	1	8024	18173	134	47.79	1	5100	3682	236	35.10
20	-3	F	F	-	T	10	40	5	F	F	-	1	9991	82998	104	65.05	1	5884	12793	84	31.88	1	4623	2593	154	27.21
21	-3	F	F	-	T	10	40	2	F	F	-	1	9936	83239	108	49.02	1	5813	12793	94	23.03	1	4503	2522	144	16.55
22	-3	F	F	-	T	5	40	5	F	F	-	1	9916	82992	104	55.21	1	5875	12888	98	27.09	1	3852	2111	110	21.00
23	-3	F	F	-	T	5	40	2	F	F	-	1	9997	83320	106	44.68	1	5843	12952	100	21.57	1	3862	2149	108	16.39

Table A.30:

A.3 m6 Data-Set

In this section we present decomposition statistics for the *m6* data-set. This data-set is a three dimensional finite element mesh with tetrahedral elements, which again originates from the FLITE3D project [BMT96].

The physical geometry which the mesh models is that of the ONERA m6-wing, a standard test case used in aerospace CFD. The mesh is that previously illustrated in figures 7.12 to 7.14 of section 7.6, and forms the most dense mesh in a series of three multigrid meshes (the smaller two meshes are not detailed in this thesis).

This is the largest and most realistic of the data-sets we study, being typical of the sort of mesh encountered in medium to large scale aerodynamics calculations. Of note is the large variation in element size for this mesh compared to the Widget and Wedge1 data-sets; here element dimensions vary over almost three orders of magnitude.

We do not explore the wide range of algorithms and tunable parameters that we studied for the previous two data-sets in this appendix, but rather focus on those algorithms and parameter setting that our previous studies have indicated might be most suitable for 'production' use of the decomposition library.

DUAL GRAPH								
	t(s)			Statistics				
Dual	Coord	Border	Other	n_e	n_e^{min}	n_e^{mean}	n_e^{max}	n_v
NODES	77.89	79.13	76.83	7366374	11	74.4	120	:
EDGES	33.19	33.38	31.38	1831873	6	18.5	29	197797
FACES	19.89	19.83	17.75	387413	2	3.9	4	:

Table A.31:

GREEDY															
case	NODES					EDGES					FACES				
	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$															
1	1	44377	407301	2	6.50	1	25309	63992	2	2.42	1	9275	7198	2	1.53
$k = 8$															
2	1	86900	877106	50	55.21	1	54056	144431	50	17.08	1	20150	15926	46	7.27
$k = 32$															
3	1	131472	1449350	414	235.17	1	84846	238850	390	70.32	1	34350	27582	316	28.03

Table A.32:

RLB																
			NODES					EDGES					FACES			
case	MD_RLB_CM_TIMES	MD_RLB_CM_BEST	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																	
1	3	T	1	30857	284950	2	23.73	1	20058	49309	2	10.04	1	7830	6104	2	6.98
$k = 8$																	
2	3	T	1	89679	904155	44	82.99	1	57319	150669	44	38.51	1	23245	18519	42	30.32
$k = 32$																	
3	3	T	1	140734	1617416	386	139.10	1	91581	257693	382	63.63	1	38361	30999	380	53.16

Table A.33:

RCB																		
					NODES					EDGES					FACES			
case	MD_RCB_CYCLE	MD_RCB_FIXED	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																		
1	F	F	F	-	1	13992	117256	2	4.30	1	8002	18711	2	1.82	1	3494	2425	2	1.10
$k = 8$																		
2	F	F	F	-	1	60379	545462	48	24.63	1	35980	87878	46	8.81	1	16138	11220	44	4.78
$k = 32$																		
3	F	F	F	-	1	119337	1257892	466	47.28	1	76757	201979	406	19.84	1	36276	25783	314	12.58

Table A.34:

RIB																
			NODES					EDGES					FACES			
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																	
1	F	-	1	12227	104886	2	5.16	1	7040	16557	2	2.70	1	3093	2106	2	2.09
$k = 8$																	
2	F	-	1	66488	628358	56	27.83	1	40656	101073	56	12.41	1	18651	12992	42	8.49
$k = 32$																	
3	F	-	1	118207	1247893	596	50.14	1	77219	204244	478	21.86	1	37334	26806	350	14.73

Table A.35:

RSB																	
					NODES					EDGES					FACES			
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																			
1	-3	F	F	-	1	11911	100628	2	1757.06	1	6717	15571	2	553.49	1	2737	1856	2	219.74
$k = 8$																			
2	-3	F	F	-	1	43752	393446	44	2988.66	1	25213	60569	38	987.07	1	10863	7393	42	399.44
$k = 32$																			
3	-3	F	F	-	1	91494	886179	304	3721.42	1	55261	138207	278	1275.80	1	26909	18672	302	554.13

Table A.36:

RIB+KL									NODES					EDGES					FACES				
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																							
1	F	-	T	F	-	F	-	0	1	11641	94205	2	51.40	1	6661	15031	2	19.24	1	3096	1654	2	13.97
2	F	-	T	F	-	T	60	0	1	12103	97707	2	35.05	1	6665	15128	2	13.58	1	3100	1657	2	9.52
3	F	-	T	F	-	T	20	0	1	12160	98756	2	22.65	1	6678	15138	2	7.77	1	3100	1660	2	4.93
4	F	-	T	F	-	T	10	0	1	12228	99782	2	16.95	1	6678	15196	2	6.85	1	3104	1669	2	3.79
5	F	-	T	F	-	T	5	0	1	12213	100559	2	13.98	1	6707	15285	2	5.38	1	3092	1674	2	3.21
$k = 8$																							
6	F	-	T	F	-	F	-	0	1	42357	369865	46	266.84	1	29497	68062	40	96.44	1	18246	9962	42	47.39
7	F	-	T	F	-	T	60	0	1	47837	428547	54	189.39	1	33150	78198	52	59.17	1	18350	10048	46	33.81
8	F	-	T	F	-	T	20	0	1	57335	516793	56	102.19	1	37002	88160	56	35.01	1	18399	10148	42	17.87
9	F	-	T	F	-	T	10	0	1	62578	574138	56	69.17	1	37848	90653	54	25.68	1	18408	10241	42	14.27
10	F	-	T	F	-	T	5	0	1	64348	595289	56	55.70	1	38527	92861	54	20.86	1	18546	10508	42	12.37
$k = 32$																							
11	F	-	T	F	-	F	-	0	1	90517	854272	334	487.93	1	60056	144894	330	240.91	1	36409	20311	360	77.20
12	F	-	T	F	-	T	60	0	1	94305	916124	462	331.10	1	65851	163015	414	106.09	1	36753	20601	384	56.35
13	F	-	T	F	-	T	20	0	1	109602	1102009	508	178.66	1	70341	176776	470	61.37	1	36375	20575	364	31.19
14	F	-	T	F	-	T	10	0	1	113070	1158129	602	119.73	1	71063	180320	476	44.79	1	36470	20859	360	25.19
15	F	-	T	F	-	T	5	0	1	115578	1186380	572	92.27	1	73098	187389	474	35.88	1	36984	21626	354	21.48

Table A.37:

RIB+MOB																									
										NODES					EDGES					FACES					
case	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b \rvert$	$\lvert E_{cut} \rvert_e$	s_{adj}	t(s)
$k = 2$																									
1	F	-	T	5	40	1	F	F	F	1	11663	95406	2	54.30	1	6608	14780	2	39.15	1	3928	2104	2	27.32
2	F	-	T	2	40	1	F	F	F	1	12170	97333	2	36.37	1	6533	14565	2	30.12	1	3806	2053	2	24.61
3	F	-	T	2	20	1	F	F	F	1	12141	97837	2	38.86	1	6648	14885	2	25.17	1	3490	1923	2	18.09
4	F	-	T	1	20	1	F	F	F	1	12337	97964	2	28.97	1	6661	15001	2	17.11	1	3113	1727	2	16.27
5	F	-	T	1	10	1	F	F	F	1	12167	98322	2	32.99	1	6673	15086	2	23.97	1	3183	1795	2	15.20
$k = 8$																									
6	F	-	T	5	40	1	F	F	F	1	45352	400423	40	193.34	1	30467	70207	44	106.49	1	20699	11573	52	82.85
7	F	-	T	2	40	1	F	F	F	1	45259	398057	46	138.37	1	33814	77897	44	85.97	1	17674	9848	50	77.26
8	F	-	T	2	20	1	F	F	F	1	45978	402389	46	124.93	1	36413	84620	42	65.76	1	17350	9746	50	51.60
9	F	-	T	1	20	1	F	F	F	1	45895	402993	48	111.46	1	36347	84619	46	56.95	1	16646	9283	48	46.16
10	F	-	T	1	10	1	F	F	F	1	46291	406340	44	146.98	1	36605	85539	46	59.22	1	18554	10511	50	39.67
$k = 32$																									
11	F	-	T	5	40	1	F	F	F	1	93413	879009	322	276.34	1	64894	157293	336	154.04	1	38677	22175	468	133.11
12	F	-	T	2	40	1	F	F	F	1	92140	867014	316	207.64	1	62875	151762	330	131.49	1	35905	20490	402	117.05
13	F	-	T	2	20	1	F	F	F	1	92682	873781	308	179.33	1	67000	162880	352	95.79	1	37498	21545	378	76.94
14	F	-	T	1	20	1	F	F	F	1	92007	869796	326	180.81	1	68158	165481	348	82.79	1	34050	19540	364	70.37
15	F	-	T	1	10	1	F	F	F	1	92603	873494	332	209.75	1	67823	164770	340	83.57	1	36682	21203	374	55.53

Table A.38:

RSB+KL																										
											NODES					EDGES					FACES				
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_KL_TERM_OBJ	MD_KL_TERM_FAILS	MD_KL_TERM_FAILS_MAX	MD_KL_BORDER_ONLY	MD_KL_BORDER_SIZE	MD_KL_RANDOM_RETRIES	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)	Δ_s	$\lvert V_b\rvert$	$\lvert E_{cut}\rvert_e$	s_{adj}	t(s)
$k=2$																										
1	-3	F	F	-	T	F	-	F	-	0	1	11324	91470	2	1811.59	1	6401	14312	2	574.99	1	2819	1526	2	232.26
2	-3	F	F	-	T	F	-	T	60	0	1	11324	91470	2	1810.57	1	6401	14312	2	571.27	1	2819	1526	2	227.55
3	-3	F	F	-	T	F	-	T	20	0	1	11662	94275	2	1770.55	1	6401	14312	2	572.24	1	2819	1526	2	222.81
4	-3	F	F	-	T	F	-	T	10	0	1	11559	95105	2	1753.43	1	6407	14313	2	551.89	1	2819	1526	2	221.26
5	-3	F	F	-	T	F	-	T	5	0	1	11880	96365	2	1753.44	1	6430	14383	2	552.73	1	2818	1526	2	221.17
$k=8$																										
6	-3	F	F	-	T	F	-	F	-	0	1	39734	349361	44	3080.57	1	24201	56050	36	1037.56	1	11053	6054	42	434.52
7	-3	F	F	-	T	F	-	T	60	0	1	39734	349361	44	3058.75	1	24201	56050	36	1023.15	1	11053	6054	42	420.92
8	-3	F	F	-	T	F	-	T	20	0	1	41637	363561	42	2997.45	1	24188	55992	36	1003.22	1	11053	6054	42	407.75
9	-3	F	F	-	T	F	-	T	10	0	1	42760	373875	44	3056.25	1	24226	56129	38	1000.13	1	11049	6040	42	404.84
10	-3	F	F	-	T	F	-	T	5	0	1	43902	384412	44	3021.87	1	24280	56234	38	1002.80	1	11037	6047	42	402.91
$k=32$																										
11	-3	F	F	-	T	F	-	F	-	0	1	89390	833906	312	3910.86	1	53655	129242	268	1342.53	1	26412	14680	268	598.26
12	-3	F	F	-	T	F	-	T	60	0	1	89024	831386	306	3876.10	1	53655	129242	268	1318.04	1	26412	14680	268	576.54
13	-3	F	F	-	T	F	-	T	20	0	1	88550	826477	294	3748.94	1	53405	128322	266	1294.03	1	26407	14679	268	554.43
14	-3	F	F	-	T	F	-	T	10	0	1	90156	842661	300	3754.63	1	53406	128720	278	1293.31	1	26415	14668	268	549.30
15	-3	F	F	-	T	F	-	T	5	0	1	91651	859351	292	3740.39	1	53510	129245	282	1294.03	1	26456	14741	266	551.17

Table A.39:

RSB+MOB													NODES					EDGES					FACES				
case	MD_RSB_TOL	MD_RSB_ORTHOG	MD_SEP_IMBAL	MD_SEP_MAX_IMBAL	MD_MOBCOMPLETE	MD_MOBSIZE	MD_MOBSCHED	MD_MOBITERS	MD_MOBCLEANUP	MD_MOBCHOICE	MD_MOBVARY	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)	Δ_s	$\|V_b\|$	$\|E_{cut}\|_e$	s_{adj}	t(s)
$k = 2$																											
1	-3	F	F	-	T	5	40	1	F	F	F	1	11717	93507	2	1808.74	1	6324	14122	2	596.84	1	4684	2554	2	253.36
2	-3	F	F	-	T	2	40	1	F	F	F	1	11741	94999	2	1822.82	1	6380	14198	2	582.88	1	4109	2209	2	245.93
3	-3	F	F	-	T	2	20	1	F	F	F	1	11778	95173	2	1792.90	1	6395	14355	2	578.90	1	3256	1831	2	235.90
4	-3	F	F	-	T	1	20	1	F	F	F	1	11889	95385	2	1778.32	1	6416	14386	2	566.74	1	2839	1584	2	235.97
5	-3	F	F	-	T	1	10	1	F	F	F	1	11982	95549	2	1794.30	1	6431	14418	2	574.33	1	2840	1604	2	234.54
$k = 8$																											
6	-3	F	F	-	T	5	40	1	F	F	F	1	41478	357115	36	3119.73	1	23946	55076	42	1076.34	1	19198	10790	54	598.51
7	-3	F	F	-	T	2	40	1	F	F	F	1	41773	360765	36	3045.22	1	23863	54933	40	1057.46	1	21121	11801	54	630.57
8	-3	F	F	-	T	2	20	1	F	F	F	1	41930	362874	36	3036.71	1	23964	55475	40	1036.93	1	14426	8143	46	528.56
9	-3	F	F	-	T	1	20	1	F	F	F	1	41976	363735	36	3025.20	1	24097	55703	38	1025.28	1	11098	6245	42	437.32
10	-3	F	F	-	T	1	10	1	F	F	F	1	41989	363736	38	3040.06	1	24256	56013	38	1026.06	1	11165	6379	42	431.28
$k = 32$																											
11	-3	F	F	-	T	5	40	1	F	F	F	1	88925	825183	272	3841.48	1	53270	127787	288	1396.56	1	36530	21127	508	823.96
12	-3	F	F	-	T	2	40	1	F	F	F	1	89366	826261	280	3773.62	1	53229	127202	272	1373.69	1	34046	19462	422	811.30
13	-3	F	F	-	T	2	20	1	F	F	F	1	89524	827417	278	3761.43	1	53278	127887	278	1346.72	1	30388	17536	330	682.49
14	-3	F	F	-	T	1	20	1	F	F	F	1	89582	832975	288	3749.62	1	53397	128238	274	1333.57	1	28468	16309	278	611.78
15	-3	F	F	-	T	1	10	1	F	F	F	1	88885	832493	282	3771.26	1	53384	128585	272	1339.72	1	27150	15729	282	616.54

Table A.40: